





# The Julia Language

The Julia Project

November 24, 2017

## Contents

Contents	i
I Home	1
II Julia Documentation	3
1 매뉴얼	5
2 Standard Library	9
3 Developer Documentation	11
III Manual	13
4 소개글	15

## 6 시작하기

## CONTENTS

5.1 읽을거리 . . . . .	23
<b>6 Variables</b>	<b>25</b>
6.1 Allowed Variable Names . . . . .	27
6.2 Stylistic Conventions . . . . .	29
<b>7 Integers and Floating-Point Numbers</b>	<b>31</b>
7.1 Integers . . . . .	32
Overflow behavior . . . . .	36
Division errors . . . . .	37
7.2 Floating-Point Numbers . . . . .	37
Floating-point zero . . . . .	39
Special floating-point values . . . . .	40
Machine epsilon . . . . .	42
Rounding modes . . . . .	44
Background and References . . . . .	45
7.3 Arbitrary Precision Arithmetic . . . . .	46
7.4 Numeric Literal Coefficients . . . . .	48
Syntax Conflicts . . . . .	50
7.5 Literal zero and one . . . . .	50
<b>8 Mathematical Operations and Elementary Functions</b>	<b>53</b>
8.1 Arithmetic Operators . . . . .	53
8.2 Bitwise Operators . . . . .	54
8.3 Updating operators . . . . .	55
8.4 Vectorized "dot" operators . . . . .	56
8.5 Numeric Comparisons . . . . .	57
Chaining comparisons . . . . .	61
Elementary Functions . . . . .	62
8.6 Operator Precedence and Associativity . . . . .	62

CONTENTS	64
7 Numerical Conversions	64
Rounding functions	66
Division functions	66
Sign and absolute value functions	66
Powers, logs and roots	66
Trigonometric and hyperbolic functions	67
Special functions	68
9 Complex and Rational Numbers	69
9.1 Complex Numbers	69
9.2 Rational Numbers	74
10 Strings	79
10.1 Characters	81
10.2 String Basics	84
10.3 Unicode and UTF-8	86
10.4 Concatenation	90
10.5 Interpolation	91
10.6 Triple-Quoted String Literals	92
10.7 Common Operations	94
10.8 Non-Standard String Literals	96
10.9 Regular Expressions	96
10.10 Byte Array Literals	103
10.11 Version Number Literals	105
10.12 Raw String Literals	106
11 함수	109
11.1 Argument Passing Behavior	110
11.2 The <code>return</code> Keyword	111
11.3 Operators Are Functions	113
11.4 Operators With Special Names	114

	CONTENTS
iv 11.5Anonymous Functions . . . . .	114
11.6Tuples . . . . .	116
11.7Named Tuples . . . . .	117
11.8Multiple Return Values . . . . .	117
11.9Argument destructuring . . . . .	118
11.10Varargs Functions . . . . .	119
11.11Optional Arguments . . . . .	122
11.12Keyword Arguments . . . . .	123
11.13Evaluation Scope of Default Values . . . . .	125
11.14Do-Block Syntax for Function Arguments . . . . .	125
11.15Dot Syntax for Vectorizing Functions . . . . .	127
11.16Further Reading . . . . .	130
 12Control Flow . . . . .	131
12.1복합 표현 . . . . .	131
12.2조건부 평가 . . . . .	133
12.3단락 평가 . . . . .	139
12.4반복 평가: 루프 . . . . .	143
12.5예외 처리 . . . . .	148
기본 제공 '예외' . . . . .	148
throw 함수 . . . . .	148
오류 . . . . .	151
경고 및 정보 메시지 . . . . .	152
try/catch문 . . . . .	153
finally문 . . . . .	155
12.6태스크(일명 코루틴) . . . . .	156
코어 태스크 연산 . . . . .	159
태스크와 이벤트 . . . . .	160
태스크 상태 . . . . .	161

CONTENTS	Variables	163
14		165
14.1	Global Scope	166
14.2	Local Scope	168
Let Blocks		175
For Loops and Comprehensions		178
14.3	Constants	180
15	Types	181
15.1	Type Declarations	183
15.2	Abstract Types	185
15.3	Primitive Types	188
15.4	Composite Types	190
15.5	Mutable Composite Types	194
15.6	Declared Types	195
15.7	Type Unions	196
15.8	Parametric Types	197
Parametric Composite Types		198
Parametric Abstract Types		203
Tuple Types		207
Vararg Tuple Types		208
Named Tuple Types		209
Parametric Primitive Types		211
15.9	UnionAll Types	212
15.10	Type Aliases	214
15.10	Operations on Types	215
15.10	Custom pretty-printing	217
15.10	Value types	221
15.10	Mullable Types: Representing Missing Values	223

vi	Constructing <code>Nullable</code> objects . . . . .	CONTENTS
	Checking if a <code>Nullable</code> object has a value . . . . .	225
	Safely accessing the value of a <code>Nullable</code> object . . . . .	225
	Performing operations on <code>Nullable</code> objects . . . . .	226
16	메소드 . . . . .	229
16.1	메소드 정의 . . . . .	230
16.2	방법 모호성 . . . . .	235
16.3	파라 메트릭 메소드 . . . . .	236
16.4	#재정의 메소드 . . . . .	240
16.5	파라미터 방법을 사용한 디자인 패턴 . . . . .	243
	super-type에서 type 매개 변수 추출 . . . . .	243
	다른 형식 매개 변수를 사용하여 비슷한 유형 만들기 . . . . .	245
	반복 디스패치 . . . . .	245
	Trait-based 디스패치 . . . . .	246
	출력 유형 계산 . . . . .	247
	별도의 변환과 귀巢로직 . . . . .	249
16.6	매개 변수 적으로 제한된 Varargs 메소드 . . . . .	249
16.7	키워드 인수 선택 사항에 대한 참고 사항 . . . . .	250
16.8	함수같은 객체 . . . . .	251
16.9	빈 일반 함수 . . . . .	253
16.10	방법 설계 및 모호한 방지 . . . . .	253
	튜플 및 NTuple 인수 . . . . .	253
	디자인 직교화 . . . . .	254
	한 번에 하나의 인수로 디스패치 . . . . .	255
	추상 컨테이너 및 요소 유형 . . . . .	256
	복잡한 인수 "cascades"와 기본 인수 . . . . .	256
17	Constructors . . . . .	259
17.1	Outer Constructor Methods . . . . .	260

CONTENTS	
17.1Empty Constructor Methods	261
17.3Incomplete Initialization	264
17.4Parametric Constructors	268
17.5Case Study: Rational	272
17.6Constructors and Conversion	276
17.7Outer-only constructors	277
18Conversion and Promotion	281
18.1Conversion	283
Defining New Conversions	284
Case Study: Rational Conversions	286
18.2Promotion	287
Defining Promotion Rules	290
Case Study: Rational Promotions	291
19Interfaces	293
19.1Iteration	293
19.2Indexing	298
19.3Abstract Arrays	299
20Modules	307
20.1Summary of module usage	309
Modules and files	309
Standard modules	311
Default top-level definitions and bare modules	311
Relative and absolute module paths	312
Module file paths	313
Namespace miscellanea	313
Module initialization and precompilation	314
21Documentation	321

viii	21.1 Accessing Documentation . . . . .	CONTENTS
	21.2 Functions & Methods . . . . .	327
	21.3 Advanced Usage . . . . .	329
	Dynamic documentation . . . . .	330
	21.4 Syntax Guide . . . . .	330
	Functions and Methods . . . . .	331
	Macros . . . . .	332
	Types . . . . .	332
	Modules . . . . .	333
	Global Variables . . . . .	334
	Multiple Objects . . . . .	335
	Macro-generated code . . . . .	336
	21.5 Markdown syntax . . . . .	337
	Inline elements . . . . .	337
	Toplevel elements . . . . .	340
	21.6 Markdown Syntax Extensions . . . . .	347
22	Metaprogramming . . . . .	349
	22.1 Program representation . . . . .	349
	Symbols . . . . .	351
	22.2 Expressions and evaluation . . . . .	353
	Quoting . . . . .	353
	Interpolation . . . . .	354
	Splatting interpolation . . . . .	355
	Nested quote . . . . .	356
	QuoteNode . . . . .	357
	eval and effects . . . . .	358
	Functions on Expressions . . . . .	360
	22.3 Macros . . . . .	362

<b>CONTENTS</b>	<b>362</b>
<b>Basics</b>	
Hold up: why macros?	363
Macro invocation	364
Building an advanced macro	367
Hygiene	370
Macros and dispatch	374
<b>22.4 Code Generation</b>	375
<b>22.5 Non-Standard String Literals</b>	377
<b>22.6 Generated functions</b>	380
An advanced example	387
Optionally-generated functions	391
<b>23 Multi-dimensional Arrays</b>	393
<b>23.1 Arrays</b>	394
Basic Functions	394
Construction and Initialization	394
Concatenation	395
Typed array initializers	395
Comprehensions	396
Generator Expressions	397
Indexing	398
Assignment	400
Supported index types	401
Iteration	406
Array traits	407
Array and Vectorized Operators and Functions	407
Broadcasting	408
Implementation	410
<b>23.2 Sparse Vectors and Matrices</b>	413

x	Compressed Sparse Column (CSC) Sparse Matrix Storage ..	CONTENTS
	Sparse Vector Storage .....	415
	Sparse Vector and Matrix Constructors .....	416
	Sparse matrix operations .....	418
	Correspondence of dense and sparse methods .....	418
24	Linear algebra	423
24.1	Special matrices .....	427
	Elementary operations .....	428
	Matrix factorizations .....	428
	The uniform scaling operator .....	429
24.2	Matrix factorizations .....	431
25	Networking and Streams	433
25.1	Basic Stream I/O .....	433
25.2	Text I/O .....	435
25.3	O Output Contextual Properties .....	436
25.4	Working with Files .....	436
25.5	A simple TCP example .....	438
25.6	Resolving IP Addresses .....	442
26	Parallel Computing	443
26.1	Code Availability and Loading Packages .....	446
26.2	Data Movement .....	449
27	Global variables	451
27.1	Parallel Map and Loops .....	453
27.2	Synchronization With Remote References .....	457
27.3	Scheduling .....	457
27.4	Channels .....	459
27.5	Remote References and AbstractChannels .....	465

<b>CONTENTS</b>	
27.6Channels and RemoteChannels . . . . .	466
27.7Remote References and Distributed Garbage Collection . . . . .	469
27.8Shared Arrays . . . . .	470
27.9Shared Arrays and Distributed Garbage Collection . . . . .	477
27.10ClusterManagers . . . . .	477
27.11Cluster Managers with Custom Transports . . . . .	483
27.12Network Requirements for LocalManager and SSHManager . . . . .	485
27.13Cluster Cookie . . . . .	487
27.14Specifying Network Topology (Experimental) . . . . .	488
27.15Multi- Threading (Experimental) . . . . .	488
Setup . . . . .	489
The @threads Macro . . . . .	489
27.16@threadcall (Experimental) . . . . .	491
28Date and DateTime	493
28.1Constructors . . . . .	494
28.2Durations/Comparisons . . . . .	497
28.3Accessor Functions . . . . .	499
28.4Query Functions . . . . .	501
28.5TimeType- Period Arithmetic . . . . .	503
28.6Adjuster Functions . . . . .	506
28.7Period Types . . . . .	509
28.8Rounding . . . . .	510
Rounding Epoch . . . . .	511
29Interacting With Julia	513
29.1The different prompt modes . . . . .	514
The Julian mode . . . . .	514
Help mode . . . . .	515
Shell mode . . . . .	516

xii	Search modes . . . . .	CONTENTS
29.2	Key bindings . . . . .	517
	Customizing keybindings . . . . .	517
29.3	Tab completion . . . . .	518
29.4	Customizing Colors . . . . .	522
30	Running External Programs	525
30.1	Interpolation . . . . .	527
30.2	Quoting . . . . .	530
30.3	Pipelines . . . . .	532
	Avoiding Deadlock in Pipelines . . . . .	534
	Complex Example . . . . .	534
31	Calling C and Fortran Code	537
31.1	Creating C-Compatible Julia Function Pointers . . . . .	541
31.2	Mapping C Types to Julia . . . . .	544
	Auto-conversion: . . . . .	544
	Type Correspondences: . . . . .	545
	Bits Types: . . . . .	545
	Struct Type correspondences . . . . .	549
	Type Parameters . . . . .	551
	SIMD Values . . . . .	551
	Memory Ownership . . . . .	552
	When to use T, Ptr{T} and Ref{T} . . . . .	553
31.3	Mapping C Functions to Julia . . . . .	553
	ccall/cfunction argument translation guide . . . . .	553
	ccall/cfunction return type translation guide . . . . .	555
	Passing Pointers for Modifying Inputs . . . . .	557
	Special Reference Syntax for ccall (deprecated): . . . . .	557
31.4	Some Examples of C Wrappers . . . . .	558

CONTENTS	
31Usage Collection Safety	562
31.6Non-constant Function Specifications	562
31.7Indirect Calls	563
31.8Closing a Library	564
31.9Calling Convention	564
31.10Accessing Global Variables	565
31.11Accessing Data through a Pointer	566
31.12Thread-safety	567
31.13More About Callbacks	568
31.14C++	568
32Handling Operating System Variation	571
33Environment Variables	573
33.1File locations	573
JULIA_HOME	573
JULIA_LOAD_PATH	574
JULIA_PKGDIR	575
JULIA_HISTORY	575
JULIA_PKGRESOLVE_ACCURACY	576
33.2External applications	576
JULIA_SHELL	576
JULIA_EDITOR	576
33.3Parallelization	577
JULIA_CPU_CORES	577
JULIA_WORKER_TIMEOUT	577
JULIA_NUM_THREADS	577
JULIA_THREAD_SLEEP_THRESHOLD	577
JULIA_EXCLUSIVE	578
33.4REPL formatting	578

	CONTENTS
xiv JULIA_ERROR_COLOR . . . . .	578
JULIA_WARN_COLOR . . . . .	578
JULIA_INFO_COLOR . . . . .	578
JULIA_INPUT_COLOR . . . . .	578
JULIA_ANSWER_COLOR . . . . .	579
JULIA_STACKFRAME_LINEINFO_COLOR . . . . .	579
JULIA_STACKFRAME_FUNCTION_COLOR . . . . .	579
33.5 Debugging and profiling . . . . .	579
JULIA_GC_ALLOC_POOL, JULIA_GC_ALLOC_OTHER, JULIA_GC_AL- LOC_PRINT . . . . .	579
JULIA_GC_NO_GENERATIONAL . . . . .	580
JULIA_GC_WAIT_FOR_DEBUGGER . . . . .	580
ENABLE_JITPROFILING . . . . .	581
JULIA_LLVM_ARGS . . . . .	581
JULIA_DEBUG_LOADING . . . . .	581
34 Embedding Julia . . . . .	583
34.1 High-Level Embedding . . . . .	583
Using julia-config to automatically determine build parameters . .	585
34.2 Converting Types . . . . .	587
34.3 Calling Julia Functions . . . . .	588
34.4 Memory Management . . . . .	589
Manipulating the Garbage Collector . . . . .	591
34.5 Working with Arrays . . . . .	591
Accessing Returned Arrays . . . . .	592
Multidimensional Arrays . . . . .	592
34.6 Exceptions . . . . .	593
Throwing Julia Exceptions . . . . .	594
35 Packages . . . . .	595

<b>CONTENTS</b>	
35.Package Status . . . . .	595
35.2Adding and Removing Packages . . . . .	596
35.3Offline Installation of Packages . . . . .	600
35.4Installing Unregistered Packages . . . . .	601
35.5Updating Packages . . . . .	602
35.6Checkout, Pin and Free . . . . .	604
35.7Custom METADATA Repository . . . . .	608
36.Package Development	609
36.1Initial Setup . . . . .	609
36.2Making changes to an existing package . . . . .	610
Documentation changes . . . . .	610
Code changes . . . . .	611
Dirty packages . . . . .	614
Making a branch post hoc . . . . .	615
Squashing and rebasing . . . . .	615
36.3Creating a new Package . . . . .	617
REQUIRE speaks for itself . . . . .	617
Guidelines for naming a package . . . . .	618
Generating the package . . . . .	620
Loading Static Non-Julia Files . . . . .	622
Making Your Package Available . . . . .	622
Tagging and Publishing Your Package . . . . .	623
36.4Fixing Package Requirements . . . . .	627
36.5Requirements Specification . . . . .	628
37.Profiling	631
37.1Basic usage . . . . .	632
37.2Accumulation and clearing . . . . .	637
37.3Options for controlling the display of profile results . . . . .	637

xvi	CONTENTS	639
37.4 Configuration . . . . .		639
38 Memory allocation analysis . . . . .	641	
39 Stack Traces . . . . .	643	
39.1 Viewing a stack trace . . . . .	643	
39.2 Extracting useful information . . . . .	645	
39.3 Error handling . . . . .	646	
39.4 Comparison with backtrace . . . . .	649	
40 Performance Tips . . . . .	653	
40.1 Avoid global variables . . . . .	653	
40.2 Measure performance with @time and pay attention to memory allocation . . . . .	654	
40.3 Tools . . . . .	656	
40.4 Avoid containers with abstract type parameters . . . . .	657	
40.5 Type declarations . . . . .	657	
Avoid fields with abstract type . . . . .	658	
Avoid fields with abstract containers . . . . .	661	
Annotate values taken from untyped locations . . . . .	667	
Declare types of keyword arguments . . . . .	669	
40.6 Break functions into multiple definitions . . . . .	669	
40.7 Write "type-stable" functions . . . . .	670	
40.8 Avoid changing the type of a variable . . . . .	671	
40.9 Separate kernel functions (aka, function barriers) . . . . .	671	
40.10 Types with values-as-parameters . . . . .	674	
40.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters) . . . . .	676	
40.12 Access arrays in memory order, along columns . . . . .	678	
40.13 Pre-allocating outputs . . . . .	681	
40.14 More dots: Fuse vectorized operations . . . . .	683	

<b>CONTENTS</b>	
<b>40 Consider using views for slices . . . . .</b>	<b>684</b>
40.1 Copying data is not always bad . . . . .	685
40.1 Avoid string interpolation for I/O . . . . .	687
40.1 Optimize network I/O during parallel execution . . . . .	687
40.19x deprecation warnings . . . . .	688
40.20weak . . . . .	688
40.21Performance Annotations . . . . .	689
40.20Treat Subnormal Numbers as Zeros . . . . .	694
40.20code_warntype . . . . .	696
<b>41 Workflow Tips . . . . .</b>	<b>701</b>
41.1 REPL-based workflow . . . . .	701
A basic editor/REPL workflow . . . . .	701
Simplify initialization . . . . .	702
41.2 Browser-based workflow . . . . .	703
<b>42 Style Guide . . . . .</b>	<b>705</b>
42.1 Write functions, not just scripts . . . . .	705
42.2 Avoid writing overly-specific types . . . . .	705
42.3 Handle excess argument diversity in the caller . . . . .	707
42.4 Append ! to names of functions that modify their arguments . . . . .	707
42.5 Avoid strange type Unions . . . . .	708
42.6 Avoid type Unions in fields . . . . .	708
42.7 Avoid elaborate container types . . . . .	709
42.8 Use naming conventions consistent with Julia's <code>base/</code> . . . . .	709
42.9 Don't overuse try-catch . . . . .	710
42.10 Don't parenthesize conditions . . . . .	710
42.11 Don't overuse . . . . .	710
42.12 Don't use unnecessary static parameters . . . . .	710
42.13 Avoid confusion about whether something is an instance or a type . . . . .	711

xvii	CONTENTS	
42.1	Don't overuse macros . . . . .	712
42.1.1	Don't expose unsafe operations at the interface level . . . . .	712
42.1.2	Don't overload methods of base container types . . . . .	712
42.1.3	Avoid type piracy . . . . .	713
42.1.4	Be careful with type equality . . . . .	714
42.1.5	Do not write <code>x-&gt;f(x)</code> . . . . .	714
42.2	Avoid using floats for numeric literals in generic code when possible	714
43	Frequently Asked Questions	717
43.1	Sessions and the REPL . . . . .	717
	How do I delete an object in memory? . . . . .	717
	How can I modify the declaration of a type in my session? . . . . .	717
43.2	Functions . . . . .	718
	I passed an argument <code>x</code> to a function, modified it inside that function, but on the outside, the variable <code>x</code> is still unchanged. Why?	718
	Can I use <code>using</code> or <code>import</code> inside a function? . . . . .	720
	What does the <code>...</code> operator do? . . . . .	721
	The two uses of the <code>...</code> operator: slurping and splatting . . . . .	721
	. . . combines many arguments into one argument in function definitions . . . . .	721
	. . . splits one argument into many different arguments in function calls . . . . .	722
	What is the return value of an assignment? . . . . .	723
43.3	Types, type declarations, and constructors . . . . .	725
	What does "type-stable" mean? . . . . .	725
	Why does Julia give a <code>DomainError</code> for certain seemingly-sensible operations? . . . . .	726
	Why does Julia use native machine integer arithmetic? . . . . .	726

<b>CONTENTS</b>	xix
<b>What are the possible causes of an <code>UndefVarError</code> during remote execution?</b>	734
<b>43.4 Packages and Modules</b>	737
<b>What is the difference between "using" and "import"?</b>	737
<b>43.5 Nothingness and missing values</b>	737
<b>How does "null" or "nothingness" work in Julia?</b>	737
<b>43.6 Memory</b>	738
<b>Why does <code>x += y</code> allocate memory when <code>x</code> and <code>y</code> are arrays?</b>	738
<b>43.7 Asynchronous IO and concurrent synchronous writes</b>	740
<b>Why do concurrent writes to the same stream result in inter-mixed output?</b>	740
<b>43.8 Julia Releases</b>	741
<b>Do I want to use a release, beta, or nightly version of Julia?</b>	741
<b>When are deprecated functions removed?</b>	742
<b>44 Noteworthy Differences from other Languages</b>	743
<b>44.1 Noteworthy differences from MATLAB</b>	743
<b>44.2 Noteworthy differences from R</b>	747
<b>44.3 Noteworthy differences from Python</b>	752
<b>44.4 Noteworthy differences from C/C++</b>	753
<b>45 Unicode Input</b>	759
<b>IV Standard Library</b>	761
<b>46 Essentials</b>	763
<b>46.1 Introduction</b>	763
<b>46.2 Getting Around</b>	763
<b>46.3 Keywords</b>	765
<b>46.4 Base Modules</b>	782
<b>46.5 All Objects</b>	783

xx	46.6 Dealing with Types . . . . .	CONTENTS
46.7	Special Types . . . . .	808
46.8	Generic Functions . . . . .	813
46.9	Syntax . . . . .	817
46.10	Nullables . . . . .	823
46.11	System . . . . .	826
46.12	Errors . . . . .	839
46.13	Events . . . . .	848
46.14	Reflection . . . . .	850
46.15	Internals . . . . .	856
47	Collections and Data Structures . . . . .	865
47.1	Iteration . . . . .	865
47.2	General Collections . . . . .	869
47.3	Iterable Collections . . . . .	871
47.4	Indexable Collections . . . . .	913
47.5	Associative Collections . . . . .	914
47.6	Set-Like Collections . . . . .	930
47.7	Deques . . . . .	935
47.8	Utility Collections . . . . .	945
48	Mathematics . . . . .	947
48.1	Mathematical Operators . . . . .	947
48.2	Mathematical Functions . . . . .	976
49	Examples . . . . .	993
49.1	Statistics . . . . .	1018
50	Numbers . . . . .	1027
50.1	Standard Numeric Types . . . . .	1027
	Abstract number types . . . . .	1027

CONTENTS		1028
Concrete number types . . . . .		1028
50.2Data Formats . . . . .		1032
50.3General Number Functions and Constants . . . . .		1042
Integers . . . . .		1054
50.4BigFloats . . . . .		1057
50.5Random Numbers . . . . .		1059
51Strings . . . . .		1069
52Arrays . . . . .		1111
52.1Constructors and Types . . . . .		1111
52.2Basic functions . . . . .		1127
52.3Broadcast and vectorization . . . . .		1135
52.4Indexing and assignment . . . . .		1141
52.5Views (SubArrays and other view types) . . . . .		1147
52.6Concatenation and permutation . . . . .		1155
52.7Array functions . . . . .		1173
52.8Combinatorics . . . . .		1186
52.9BitArrays . . . . .		1195
52.10Sparse Vectors and Matrices . . . . .		1196
53Tasks and Parallel Computing . . . . .		1211
53.1Tasks . . . . .		1211
53.2General Parallel Computing Support . . . . .		1222
53.3Multi-Threading . . . . .		1244
53.4ccall using a threadpool (Experimental) . . . . .		1252
53.5Synchronization Primitives . . . . .		1252
53.6Cluster Manager Interface . . . . .		1256
54Linear Algebra . . . . .		1261
54.1Standard Functions . . . . .		1261

xxi	54.2 Low-level matrix operations . . . . .	CONTENTS	373
54.3 BLAS Functions . . . . .	1378		
BLAS Character Arguments . . . . .	1379		
54.4 LAPACK Functions . . . . .	1391		
55 Constants		1421	
56 Filesystem		1425	
57 Delimited Files		1443	
58 I/O and Network		1447	
58.1 General I/O . . . . .	1447		
58.2 Text I/O . . . . .	1465		
58.3 Multimedia I/O . . . . .	1475		
58.4 Network I/O . . . . .	1481		
59 Punctuation		1491	
60 Sorting and Related Functions		1493	
60.1 Sorting Functions . . . . .	1496		
60.2 Order-Related Functions . . . . .	1502		
60.3 Sorting Algorithms . . . . .	1507		
61 Package Manager Functions		1511	
62 Dates and Time		1519	
62.1 Dates and Time Types . . . . .	1519		
62.2 Dates Functions . . . . .	1521		
Accessor Functions . . . . .	1530		
Query Functions . . . . .	1535		
Adjuster Functions . . . . .	1540		
Periods . . . . .	1544		

CONTENTS	1546
Founding Functions	1546
Conversion Functions	1552
Constants	1553
Iteration utilities	1555
Unit Testing	1565
64.1Testing Base Julia	1565
64.2Basic Unit Tests	1566
64.3Working with Test Sets	1568
64.4Other Test Macros	1572
64.5Broken Tests	1575
64.6Creating Custom AbstractTestSet Types	1576
C Interface	1579
LLVM Interface	1591
C Standard Library	1593
Dynamic Linker	1597
Profiling	1601
StackTraces	1605
SIMD Support	1609
Base64	1611
Memory-mapped I/O	1615
Shared Arrays	1619
File Events	1623
CRC32c	1625

77	Reflection and introspection	1629
77.1	Module bindings	1629
77.2	DataType fields	1629
77.3	Subtypes	1630
77.4	DataType layout	1631
77.5	Function methods	1631
77.6	Expansion and lowering	1631
77.7	Intermediate and compiled representations	1632
78	Documentation of Julia's Internals	1635
78.1	Initialization of the Julia runtime	1635
main()		1635
julia_init()		1635
true_main()		1639
Base._start		1639
Base.eval		1639
jl_atexit_hook()		1640
julia_save()		1640
78.2	Julia ASTs	1640
Lowered form		1640
Surface syntax AST		1649
78.3	More about types	1653
Types and sets (and Any and Union{}/Bottom)		1653
UnionAll types		1655
Free variables		1657
TypeNames		1657
Tuple types		1660
Diagonal types		1662

CONTENTS		1664
Subtyping diagonal variables	.....	1664
Introduction to the internal machinery	.....	1665
Subtyping and method sorting	.....	1666
78.4 Memory layout of Julia Objects	.....	1667
Object layout (jl_value_t)	.....	1667
Garbage collector mark bits	.....	1669
Object allocation	.....	1669
78.5 Eval of Julia code	.....	1672
Julia Execution	.....	1672
Parsing	.....	1674
Macro Expansion	.....	1674
Type Inference	.....	1675
JIT Code Generation	.....	1676
System Image	.....	1678
78.6 Calling Conventions	.....	1678
Julia Native Calling Convention	.....	1678
JL Call Convention	.....	1679
C ABI	.....	1679
78.7 High-level Overview of the Native-Code Generation Process	....	1679
Representation of Pointers	.....	1679
Representation of Intermediate Values	.....	1680
Union representation	.....	1681
Specialized Calling Convention Signature Representation	.....	1682
78.8 Julia Functions	.....	1683
Method Tables	.....	1683
Function calls	.....	1683
Adding methods	.....	1684
Creating generic functions	.....	1685
Closures	.....	1685

xxvi	Constructors . . . . .	CONTENT 686
	Builtins . . . . .	1686
	Keyword arguments . . . . .	1687
	Compiler efficiency issues . . . . .	1689
78.9	Base.Cartesian . . . . .	1691
	Principles of usage . . . . .	1691
	Basic syntax . . . . .	1691
78.10	Talking to the compiler (the :meta mechanism) . . . . .	1698
78.11	SubArrays . . . . .	1700
	Indexing: cartesian vs. linear indexing . . . . .	1700
	Index replacement . . . . .	1701
	SubArray design . . . . .	1702
78.12	System Image Building . . . . .	1708
	Building the Julia system image . . . . .	1708
	System image optimized for multiple microarchitectures . . . . .	1710
78.13	Working with LLVM . . . . .	1712
	Overview of Julia to LLVM Interface . . . . .	1712
	Building Julia with a different version of LLVM . . . . .	1713
	Passing options to LLVM . . . . .	1713
	Debugging LLVM transformations in isolation . . . . .	1714
	Improving LLVM optimizations for Julia . . . . .	1715
	The jlcall calling convention . . . . .	1715
	GC root placement . . . . .	1716
78.14	printf() and stdio in the Julia runtime . . . . .	1722
	Libuv wrappers for stdio . . . . .	1722
	Interface between JL_STD* and Julia code . . . . .	1723
	printf() during initialization . . . . .	1724
	Legacy <code>ios.c</code> library . . . . .	1724
78.15	Bounds checking . . . . .	1725

CONTENTS	xvii
77. Ending bounds checks . . . . .	1726
Propagating inbounds . . . . .	1726
The bounds checking call hierarchy . . . . .	1727
78. Proper maintenance and care of multi-threading locks . . . . .	1728
Locks . . . . .	1728
Broken Locks . . . . .	1731
Shared Global Data Structures . . . . .	1731
78.1 Arrays with custom indices . . . . .	1733
Generalizing existing code . . . . .	1733
Writing custom array types with non-1 indexing . . . . .	1737
Summary . . . . .	1740
78.18 Base.LibGit2 . . . . .	1740
78.19 Module loading . . . . .	1811
Experimental features . . . . .	1812
78.20 Inference . . . . .	1812
How inference works . . . . .	1812
Debugging inference.jl . . . . .	1813
The inlining algorithm (inline_worthy) . . . . .	1814
79 Developing/debugging Julia's C code . . . . .	1819
79.1 Reporting and analyzing crashes (segfaults) . . . . .	1819
Version/Environment info . . . . .	1820
Segfaults during bootstrap ( <code>sysimg.jl</code> ) . . . . .	1820
Segfaults when running a script . . . . .	1821
Errors during Julia startup . . . . .	1822
Glossary . . . . .	1823
79.2 gdb debugging tips . . . . .	1823
Displaying Julia variables . . . . .	1823
Useful Julia variables for Inspecting . . . . .	1823

xxviii	Useful Julia functions for Inspecting those variables . . . . .	CONTENT	1824
	Inserting breakpoints for inspection from gdb . . . . .		1825
	Inserting breakpoints upon certain conditions . . . . .		1825
	Dealing with signals . . . . .		1826
	Debugging during Julia's build process (bootstrap) . . . . .		1826
	Debugging precompilation errors . . . . .		1829
	Mozilla's Record and Replay Framework (rr) . . . . .		1829
79.3	Using Valgrind with Julia . . . . .		1829
	General considerations . . . . .		1830
	Suppressions . . . . .		1830
	Running the Julia test suite under Valgrind . . . . .		1831
	Caveats . . . . .		1831
79.4	Sanitizer support . . . . .		1831
	General considerations . . . . .		1831
	Address Sanitizer (ASAN) . . . . .		1832
	Memory Sanitizer (MSAN) . . . . .		1832

Part I

Home



Part II

Julia Documentation



# Chapter 1

## 매뉴얼

Introduction

Getting Started

Variables

Integers and Floating-Point Numbers

Mathematical Operations and Elementary Functions

Complex and Rational Numbers

Strings

Functions

Control Flow

Scope of Variables

Types

Methods

Constructors

Conversion and Promotion

Modules

Documentation

Metaprogramming

Multi-dimensional Arrays

Linear Algebra

Networking and Streams

Parallel Computing

Date and DateTime

Running External Programs

Calling C and Fortran Code

Handling Operating System Variation

Environment Variables

Interacting With Julia

Embedding Julia

Packages

Profiling

Stack Traces

Performance Tips

Workflow Tips

Style Guide

Frequently Asked Questions

Unicode Input



# Chapter 2

## Standard Library

Essentials

Collections and Data Structures

Mathematics

Numbers

Strings

Arrays

Tasks and Parallel Computing

Linear Algebra

Constants

Filesystem

Delimited Files

I/O and Network

Punctuation

Sorting and Related Functions

Dates and Time

Iteration utilities

Unit Testing

C Interface

C Standard Library

Dynamic Linker

StackTraces

SIMD Support

Profiling

Memory-mapped I/O

Shared Arrays

Base64

File Events

# Chapter 3

## Developer Documentation

[Reflection and introspection](#)

[Documentation of Julia's Internals](#)

- [Initialization of the Julia runtime](#)
- [Julia ASTs](#)
- [More about types](#)
- [Memory layout of Julia Objects](#)
- [Eval of Julia code](#)
- [Calling Conventions](#)
- [High-level Overview of the Native-Code Generation Process](#)
- [Julia Functions](#)
- [Base.Cartesian](#)
- [Talking to the compiler \(the :meta mechanism\)](#)
- [SubArrays](#)
- [System Image Building](#)
- [Working with LLVM](#)
- [printf\(\) and stdio in the Julia runtime](#)

- [Bounds checking](#)
- [Proper maintenance and care of multi-threading locks](#)
- [Arrays with custom indices](#)
- [Base.LibGit2](#)
- [Module loading](#)
- [Inference](#)

## Developing/debugging Julia's C code

- [Reporting and analyzing crashes \(segfaults\)](#)
- [gdb debugging tips](#)
- [Using Valgrind with Julia](#)
- [Sanitizer support](#)

Part III

Manual



# Chapter 4

## 소개글

과학 분야 컴퓨팅은 고성능의 많은 수학적 계산 처리를 필요로 한다. 하지만 당사자인 전문 연구자들은 속도가 느리더라도 동적인 언어로서 그들의 업무를 처리한다. 동적인 언어를 즐겨쓰는 나름의 이유들로 보아, 이러한 추세는 쉽게 사그러들지 않아 보인다. 다행히 근래의 언어 디자인과 컴파일러 기법의 발달은 미뤄뒀던 성능 부분을 해결함으로서 프로토타이핑 작업시 개별 환경의 생산성을, 성능이 중요한 애플리케이션 구축시 그 효용성을 충분히 발휘한다. 줄리아 프로그래밍 언어는 다음과 같은 역할을 수행한다: 과학과 수학 분야의 컴퓨팅에 적합한 기존의 정적 타입 언어에 견줄만한 성능을 갖춘 유연한 동적 언어.

줄리아 컴파일러는 파이썬, R에서의 인터프리터 방식과 다르다. 그래서 줄리아의 성능을 처음 접하면 아마도 의아할 것이다. 헌데 막상 작성한 코드가 느리다면 [Performance Tips](#)을 읽어보길 권한다. 줄리아가 어떤 식으로 작동하는지 이해했다면, C로 짠거마냥 빠른 코드를 쉽게 작성할 수 있을 것이다.

줄리아는 타입 추론과 [LLVM](#)으로 구현한 [적시 컴파일 \(JIT\)](#), implemented using [LLVM](#)을 사용해 선택적 타입, 멀티플 디스패치, 좋은 성능을 이뤄내고 있다. 그리고 명령형, 함수형, 객체 지향 프로그래밍의 특징을 포괄하는 다양한 패러다임을 추구한다. 줄리아는 고급 단계의 수치 계산에 있어 R, 매트랩, 파이썬처럼 간편하고 표현력이 우수하다. 뿐만 아니라 일반적인 형태의 프로그래밍 또한 가능하다. 이를 위해 줄리아는 수학 프로그래밍 언어를 근간으로 구축하였고 [리프스](#), [펄](#), [파이썬](#), [루아](#), [루비](#)와 같은 대중적인 동적 언어의 특징을 가져와 취합하고 있다.

기준에 있는 동적 언어와 비교해 줄리아만의 독특한 점은:

## CHAPTER 4. 소개글

핵심 언어는 최소한으로 꾸린다; 정수를 다루는 프리미티브 연산자(+ - \* 같은)를 비롯하여 기본 라이브러리는 줄리아 자체로 작성되었다.

객체를 구성하고 서술하는데 쓸 타입을 언어에서 풍부히 지원하며, 타입 선언을 할 때에도 선택적으로 사용할 수 있다.

인자 타입을 조합함으로서 함수의 작동 행위를 정의하는 [multiple dispatch](#)

인자 타입에 따라 효율적이고 특화된 코드를 자동으로 생성한다

C처럼 정적으로 컴파일되는 언어에 근접하는 훌륭한 성능

종종 동적 언어에 대해 "타입이 없다"는 식으로 말하는데 실은 그렇지 않다: 프리미티브(숫자와 같은 기본 타입의)이거나 별도 정의를 통틀어 모든 객체는 타입을 가진다. 그러나 대부분의 동적 언어는 타입 선언이 부족해 컴파일러가 해당 값의 타입을 인지하지 못하거나 종종 타입에 대해 무엇인지 명시적으로 밝힐 수 없는 상태가 되곤 한다. 한편 정적 언어는 타입 정보를 - 보통 반드시 - 컴파일러용으로 달기에 타입은 오직 컴파일 시점에만 존재해 실행시에는 이를 다루거나 표현할 수가 없다. 줄리아에서 타입은 그 자체로 런타임 객체이며 컴파일러가 요하는 정보를 알려주는 데에도 쓰인다.

보통의 프로그래머라면 개의치 않을 타입과 멀티플 디스패치는 줄리아의 핵심 개념이다: 함수들은 서로 다른 인자 타입들을 조합함으로서 정의되고 가장 그 정의와 맞물리는 타입을 찾아 디스패치하여 실행된다. 이 모델은 수학 프로그래밍과 잘 맞는데, 객체 지향에선 연산자가 첫번째 인자를 "갖는" 방식이기에 자연스럽지가 않다. 연산자는 단지 특별히 표기한 함수일 뿐이다 + 함수에 위일 새로운 데이터 타입을 정의하려면 해당하는 메서드 정의만 추가하면 된다. 기존 코드는 새로운 데이터 타입과 더불어 원활하게 작동한다.

런타임 타입 추론(타입 지정을 점진적으로 늘려가며)을 이유로 또 이 프로젝트를 시작할 때 무엇보다 성능을 강조하였기에 줄리아의 계산 효율은 다른 동적 언어들에 비해 우월하며 심지어 정적으로 컴파일하는 경쟁 언어들마저 능가한다. 거대한 규모의 수치 해석 문제에 있어 속도는 매번 그리고 앞으로도, 아마 항상 결정적 요소일 것이다: 처리되는 데이터의 양이 지난 수십년간 무어의 법칙을 따르고 있으니 말이다.

사용하기 편하면서도 강력하고 효율적인 언어를 줄리아는 목표하고 있다. 다른 시스템과  
견주어 줄리아를 쓰으로 얻는 이득은 다음과 같다:

자유롭게 사용 가능하며 오픈 소스이다 ([MIT licensed](#))

사용자가 정의한 타입 또한 내장한 타입처럼 빠르고 간결하다

성능을 위해 코드를 벡터화할 필요가 없다; 벡터화하지 않은 코드도 빠르다

병렬과 분산 처리를 위해 고안되었다

가벼운 “그린” 쓰레딩 ([코루틴](#))

거슬리지 않는 강력한 타입 시스템

숫자와 다른 타입을 위한 우아하고 확장 가능한 컨버전 및 프로모션(타입 변환)

효율적인 [Unicode](#) 와 [UTF-8](#) 지원

C 함수 직접 호출(별도의 래퍼나 특정한 API가 필요하지 않음)

다른 프로세스를 관리하는 쉘과 비슷한 강력한 기능

리스트와 비슷한 매크로, 메타프로그래밍을 위한 장치들



# Chapter 5

## 시작하기

줄리아의 설치는 어렵지 않다. 미리 컴파일된 실행파일을 이용하거나, 아니면 스스로부터 직접 컴파일하는 두가지 방법이 존재한다. <https://julialang.org/downloads/>에서 알려주는 방법에 따라 Julia를 다운로드하고 설치하면 된다.

Julia를 처음 접할 때는 대화형 실행 환경을 통해서 시작하는 것이 가장 쉽게 Julia를 익힐 수 있는 방법이다. 대화형 실행 환경은 단순히 Julia 실행파일을 더블 클릭하거나, 명령창에서 `julia` 명령어를 입력하여 실행할 수 있다.

```
$ julia

         _ _(_)_ | A fresh approach to technical computing
(._) | (._) (._) | Documentation: https://docs.julialang.
org
          _ -| |- -- -| Type "?help" for help.
| | | | | | / _` | |
| | | -| | | | (._| | | Version 0.5.0-dev+2440 (2016-02-01 02:22
UTC)
-/ | \__'__|_-|_| \__'__| Commit 2bb94d6 (11 days old master)
|__/_ | x86_64-apple-darwin13.1.0

julia> 1 + 2
3
```

```
julia> ans
3
```

대화형 실행 환경을 종료하기 위해서는 ^D(컨트롤 끄와 d 키를 함께 누른다.) 를 입력하거나 `quit()`를 대화형 실행 환경 입력창에 타이핑한다. 대화형 실행 환경을 실행하면, 위와 같이 `julia` 배너가 보여지고, 커서창이 사용자의 입력을 기다리며 깜빡이고 있다. 사용자가 `1 + 2`, 와 같은 표현식을 입력한 뒤, 엔터 버튼을 누르는 순간, Julia는 표현식을 평가하고 그 결과를 보여준다. 만약 사용자가 입력한 표현식이 세미콜론(:)으로 끝난다면, 대화형 실행 환경은 결과를 바로 보여주지 않는다. 대신에 `ans`라는 변수가 결과를 보여주든 보여주지 않든 가장 마지막으로 계산된 표현식의 결과를 저장하고 있다 `ans` 변수는 대화형 실행 환경에서만 존재하며, 다른 방식으로 동작하는 Julia 코드 상에서는 나타나지 않는다.

`file.jl`라는 소스 파일에 저장되어 있는 표현식을 계산하기 위해서는, `include("file.jl")`와 같이 입력한다.

대화형 실행 환경을 이용하지 않고 파일에 저장되어 있는 소스 코드를 실행하기 위해서는, 소스 코드 파일 이름을 `julia` 명령어의 첫번째 매개 변수로 넣어서 실행한다.

명령어:

```
$ julia script.jl arg1 arg2...
```

예제와 같이 `julia` 실행 명령 뒤에 오는 매개변수들은 전역 상수 `ARGS`라고 불리우는 `script.jl`라는 프로그램의 명령줄 인자로 작동한다. 이 프로그램의 이름은 전역 상수 `PROGRAM_FILE`에도 설정된다. 또한 `ARGS`는 이 뿐만이 아니라 `-e` 옵션을 통해서 `julia` 스크립트를 실행할 때도 설정할 수 있음을 알 필요가 있다. 그러나 이 경우에는 `PROGRAM_FILE`은 아무것도 설정되지 않은 채로 실행될 것이다.(아래의 `julia` 도움말을 보도록 하자.) 예를 들어, 단순히 스크립트에 주어진 명령줄 인자를 출력할 때는 다음과 같이 입력하면 된다.

```
$ julia -e 'println(PROGRAM_FILE); for x in ARGS; println(x); end'
foo bar
```

```
| foo
| bar
```

아니면 저 코드를 스크립트 파일에 넣고 실행시켜도 가능하다.

```
$ echo 'println(PROGRAM_FILE); for x in ARGS; println(x); end' >
    script.jl
$ julia script.jl foo bar
script.jl
foo
bar
```

-- 구분자는 명령어와 줄리아에 넘겨줄 인자를 구분하는데 사용한다.

```
$ julia --color=yes -O -- foo.jl arg1 arg2..
```

Julia는 -p 옵션이나 --machinefile 옵션을 이용하여 병렬 환경에서 실행시킬 수 있다. -p n 옵션은 n개의 worker 프로세스를 생성하지만, --machinefile file 옵션은 file의 각 행에 지정된 노드마다 worker를 생성한다. file에 지정된 노드(machine)들은 ssh 로그인을 통해 패스워드가 필요없이 실행할 수 있어야 하며, Julia는 현재 호스트와 같은 경로에 설치가 되어 있어야 한다. file에 작성되는 노드는 [count\*][user@]host[:port] [bind\_addr[:port]]와 같은 형식으로 작성한다. user는 현재 user id를 나타내고, port는 기본 ssh port, count는 각 노드당 생성하는 worker의 개수 (기본값 : 1) bin-to bind\_addr[:port]은 선택적인 옵션으로 다른 worker들이 현재의 worker로 연결하기 위해 필요한 특정 ip 주소와 포트를 지정한다.

만약 Julia가 실행할 때마다 실행되는 코드가 있다면, 그 코드를 ~/.juliarc.jl에 넣으면 된다.

```
$ echo 'println("Greetings! ! ?")' > ~/.juliarc.jl
$ julia
Greetings! ! ?
...
...
```

`perl` 과 `ruby` 와 같이, Julia 코드를 실행하고 옵션을 지정하는 방법은 다음과 같다.  
여러가지가 있다.

```
julia [switches] -- [programfile] [args...]  
-v, --version .  
-h, --help .  
  
-J, --sysimage <file> <file> .  
-H, --home <dir> julia .  
--startup-file={yes|no} ~/.juliarc.jl .  
--handle-signals={yes|no} Julia .  
--sysimage-native-code={yes|no}  
/ .  
--compiled-modules={yes|no}  
/ .  
  
-e, --eval <expr> <expr>  
-E, --print <expr> <expr> .  
-L, --load <file> <file> .  
  
-p, --procs {N|auto} N worker . "auto" Julia worker  
. .  
--machinefile <file> <file> worker .  
  
-i ; PEPL ininteractive() true.  
-q, --quiet , REPL .  
--banner={yes|no} / .  
--color={yes|no} .  
--history-file={yes|no} .  
  
--depwarn={yes|no|error} / .("error" .)
```

```
-C, --cpu-target <target> <target> CUPU ()      "help" )
-O, --optimize={0,1,2,3}      .( 2 ,       3 )
-g, -g <level>             / .( 1,       2)
--inline={yes|no}           @inline     , inlining .
--check-bounds={yes|no}     /. ( )
--math-mode={ieee,fast}     IEEE   ( )     .

--code-coverage={none|user|all}, --code-coverage
                           . (: "user")
--track-allocation={none|user|all}, --track-allocation
```

## 5.1 읽을거리

이 매뉴얼 뿐만 아니라 Julia를 처음 접하는 사용자들에게 도움을 줄 수 있는 다른 문서를 소개한다.

Julia and IJulia cheatsheet

Learn Julia in a few minutes

Learn Julia the Hard Way

Julia by Example

Hands-on Julia

Tutorial for Homer Reid's numerical analysis class

An introductory presentation

Videos from the Julia tutorial at MIT

YouTube videos from the JuliaCons



# Chapter 6

## Variables

A variable, in Julia, is a name associated (or bound) to a value. It's useful when you want to store a value (that you obtained after some math, for example) for later use. For example:

```
# Assign the value 10 to the variable x
julia> x = 10
10

# Doing math with x's value
julia> x + 1
11

# Reassign x's value
julia> x = 1 + 1
2

# You can assign values of other types, like strings of text
julia> x = "Hello World!"
"Hello World!"
```

Julia provides an extremely flexible system for naming variables. Variable

Names are case-sensitive, and have no semantic meaning (the language will not treat variables differently based on their names).

```
julia> x = 1.0
1.0

julia> y = -3
-3

julia> Z = "My string"
"My string"

julia> customary_phrase = "Hello world!"
"Hello world!"

julia> UniversalDeclarationOfHumanRightsStart = """
"
```

Unicode names (in UTF-8 encoding) are allowed:

```
julia> δ = 0.00001
1.0e-5

julia> ℨ = "Hello"
"Hello"
```

In the Julia REPL and several other Julia editing environments, you can type many Unicode math symbols by typing the backslashed LaTeX symbol name followed by tab. For example, the variable name  $\delta$  can be entered by typing `\delta-tab`, or even  $\alpha$  by `\alpha-tab-\hat{-}tab-\_2-tab`. (If you find a symbol somewhere, e.g. in someone else's code, that you don't know how to type, the REPL help will tell you: just type `?` and then paste the symbol.)

Julia ~~ALLOWS~~ **ALLOWS VARIABLE NAMES** built-in constants and functions (or though this is not recommended to avoid potential confusions):

```
julia> pi = 3
```

```
3
```

```
julia> pi
```

```
3
```

```
julia> sqrt = 4
```

```
4
```

However, if you try to redefine a built-in constant or function already in use, Julia will give you an error:

```
julia> pi
```

```
π = 3.1415926535897...
```

```
julia> pi = 3
```

```
ERROR: cannot assign variable MathConstants.pi from module Main
```

```
julia> sqrt(100)
```

```
10.0
```

```
julia> sqrt = 4
```

```
ERROR: cannot assign variable Base.sqrt from module Main
```

## 6.1 Allowed Variable Names

Variable names must begin with a letter (A-Z or a-z), underscore, or a subset of Unicode code points greater than 00A0; in particular, [Unicode character categories](#) Lu/LI/Lt/Lm/Lo/Nl (letters), Sc/So (currency and other symbols),

and a few other letter-like characters (e.g. a subset of the Latin alphabet) are allowed. Subsequent characters may also include ! and digits (0-9) and other characters in categories Nd/No), as well as other Unicode code points: diacritics and other modifying marks (categories Mn/Mc/Me/Sk), some punctuation connectors (category Pc), primes, and a few other characters.

Operators like + are also valid identifiers, but are parsed specially. In some contexts, operators can be used just like variables; for example (+) refers to the addition function, and (+) = f will reassign it. Most of the Unicode infix operators (in category Sm), such as , are parsed as infix operators and are available for user-defined methods (e.g. you can use const = kron to define as an infix Kronecker product). Operators can also be suffixed with modifying marks, primes, and sub/superscripts, e.g. +" is parsed as an infix operator with the same precedence as +.

The only explicitly disallowed names for variables are the names of built-in statements:

```
julia> else = false
ERROR: syntax: unexpected "else"
```

```
julia> try = "No"
ERROR: syntax: unexpected "="
```

Some Unicode characters are considered to be equivalent in identifiers. Different ways of entering Unicode combining characters (e.g., accents) are treated as equivalent (specifically, Julia identifiers are NFC-normalized). The Unicode characters (U+025B: Latin small letter open e) and μ (U+00B5: micro sign) are treated as equivalent to the corresponding Greek letters, because the former are easily accessible via some input methods.

While Julia imposes few restrictions on valid names, it has become useful to adopt the following conventions:

Names of variables are in lower case.

Word separation can be indicated by underscores (' \_ '), but use of underscores is discouraged unless the name would be hard to read otherwise.

Names of **Types** and **Modules** begin with a capital letter and word separation is shown with upper camel case instead of underscores.

Names of **functions** and **macros** are in lower case, without underscores.

Functions that write to their arguments have names that end in !. These are sometimes called "mutating" or "in-place" functions because they are intended to produce changes in their arguments after the function is called, not just return a value.

For more information about stylistic conventions, see the [Style Guide](#).



# Chapter 7

## Integers and Floating - Point Numbers

Integers and floating-point values are the basic building blocks of arithmetic and computation. Built-in representations of such values are called numeric primitives, while representations of integers and floating-point numbers as immediate values in code are known as numeric literals. For example, 1 is an integer literal, while 1.0 is a floating-point literal; their binary in-memory representations as objects are numeric primitives.

Julia provides a broad range of primitive numeric types, and a full complement of arithmetic and bitwise operators as well as standard mathematical functions are defined over them. These map directly onto numeric types and operations that are natively supported on modern computers, thus allowing Julia to take full advantage of computational resources. Additionally, Julia provides software support for [Arbitrary Precision Arithmetic](#), which can handle operations on numeric values that cannot be represented effectively in native hardware representations, but at the cost of relatively slower performance.

The following are Julia's primitive numeric types:

Integer types:

Floating-point types:

Type	Signed?	Number of Bits	Range of Integers and Validating Integers	Range of Rational Numbers
Int8		8	-2^7	2^7 - 1
UInt8		8	0	2^8 - 1
Int16		16	-2^15	2^15 - 1
UInt16		16	0	2^16 - 1
Int32		32	-2^31	2^31 - 1
UInt32		32	0	2^32 - 1
Int64		64	-2^63	2^63 - 1
UInt64		64	0	2^64 - 1
Int128		128	-2^127	2^127 - 1
UInt128		128	0	2^128 - 1
Bool	N/A	8	false (0)	true (1)

Type	Precision	Number of bits
Float16	half	16
Float32	single	32
Float64	double	64

Additionally, full support for [Complex and Rational Numbers](#) is built on top of these primitive numeric types. All numeric types interoperate naturally without explicit casting, thanks to a flexible, user-extensible [type promotion system](#).

## 7.1 Integers

Literal integers are represented in the standard manner:

```
julia> 1
1

julia> 1234
1234
```

The default type for an integer literal depends on whether the target system has a 32-bit architecture or a 64-bit architecture:

```
# 32-bit system:
julia> typeof(1)
Int32
```

```
# 64-bit system:  
julia> typeof(1)  
Int64
```

The Julia internal variable `Sys.WORD_SIZE` indicates whether the target system is 32-bit or 64-bit:

```
# 32-bit system:  
julia> Sys.WORD_SIZE  
32  
  
# 64-bit system:  
julia> Sys.WORD_SIZE  
64
```

Julia also defines the types `Int` and `UInt`, which are aliases for the system's signed and unsigned native integer types respectively:

```
# 32-bit system:  
julia> Int  
Int32  
julia> UInt  
UInt32  
  
# 64-bit system:  
julia> Int  
Int64  
julia> UInt  
UInt64
```

CHAPTER 7 AND NUMBERS AND FLOATING-POINT NUMBERS  
Bigger integer literals be represented in 64 bits always create 64-bit integers, regardless of the system type:

```
# 32-bit or 64-bit system:  
julia> typeof(3000000000)  
Int64
```

Unsigned integers are input and output using the `0x` prefix and hexadecimal (base 16) digits `0-9a-f` (the capitalized digits A-F also work for input). The size of the unsigned value is determined by the number of hex digits used:

```
julia> 0x1  
0x01  
  
julia> typeof(ans)  
UInt8  
  
julia> 0x123  
0x0123  
  
julia> typeof(ans)  
UInt16  
  
julia> 0x1234567  
0x01234567  
  
julia> typeof(ans)  
UInt32  
  
julia> 0x123456789abcdef  
0x0123456789abcdef
```

```
julia> typeof(ans)  
UInt64
```

This behavior is based on the observation that when one uses unsigned hex literals for integer values, one typically is using them to represent a fixed numeric byte sequence, rather than just an integer value.

Recall that the variable `ans` is set to the value of the last expression evaluated in an interactive session. This does not occur when Julia code is run in other ways.

Binary and octal literals are also supported:

```
julia> 0b10  
0x02  
  
julia> typeof(ans)  
UInt8  
  
julia> 0o10  
0x08  
  
julia> typeof(ans)  
UInt8
```

The minimum and maximum representable values of primitive numeric types such as integers are given by the `typemin` and `typemax` functions:

```
julia> (typemin(Int32), typemax(Int32))  
(-2147483648, 2147483647)  
  
julia> for T in  
→   [Int8, Int16, Int32, Int64, Int128, UInt8, UInt16, UInt32, UInt64, UInt128]
```

```

    println("$(lpad(T,7)): [$(typemin(T)), $(typemax(T))]")
end

Int8: [-128, 127]
Int16: [-32768, 32767]
Int32: [-2147483648, 2147483647]
Int64: [-9223372036854775808, 9223372036854775807]
Int128: [-
→ 170141183460469231731687303715884105728, 170141183460469231731687303715884
UInt8: [0, 255]
UInt16: [0, 65535]
UInt32: [0, 4294967295]
UInt64: [0, 18446744073709551615]
UInt128: [0, 340282366920938463463374607431768211455]

```

The values returned by `typemin` and `typemax` are always of the given argument type. (The above expression uses several features we have yet to introduce, including [for loops](#), [Strings](#), and [Interpolation](#), but should be easy enough to understand for users with some existing programming experience.)

## Overflow behavior

In Julia, exceeding the maximum representable value of a given type results in a wraparound behavior:

```

julia> x = typemax(Int64)
9223372036854775807

julia> x + 1
-9223372036854775808

```

```
julia> x + 1 == typemin(Int64)
```

```
true
```

Thus, arithmetic with Julia integers is actually a form of [modular arithmetic](#). This reflects the characteristics of the underlying arithmetic of integers as implemented on modern computers. In applications where overflow is possible, explicit checking for wraparound produced by overflow is essential; otherwise, the [BigInt](#) type in [Arbitrary Precision Arithmetic](#) is recommended instead.

## Division errors

Integer division (the `div` function) has two exceptional cases: dividing by zero, and dividing the lowest negative number (`typemin`) by -1. Both of these cases throw a [DivideError](#). The remainder and modulus functions (`rem` and `mod`) throw a [DivideError](#) when their second argument is zero.

## 7.2 Floating-Point Numbers

Literal floating-point numbers are represented in the standard formats, using [E-notation](#) when necessary:

```
julia> 1.0
```

```
1.0
```

```
julia> 1.
```

```
1.0
```

```
julia> 0.5
```

```
0.5
```

```
julia> .5
```

```
0.5
```

```
julia> -1.23
```

```
-1.23
```

```
julia> 1e10
```

```
1.0e10
```

```
julia> 2.5e-4
```

```
0.00025
```

The above results are all `Float64` values. Literal `Float32` values can be entered by writing an `f` in place of `e`:

```
julia> 0.5f0
```

```
0.5f0
```

```
julia> typeof(ans)
```

```
Float32
```

```
julia> 2.5f-4
```

```
0.00025f0
```

Values can be converted to `Float32` easily:

```
julia> Float32(-1.5)
```

```
-1.5f0
```

```
julia> typeof(ans)
```

```
Float32
```

Hexadecimal floating-point literals are also valid, but only as `Float64` values, with `p` preceding the base-2 exponent:

```
julia> 0x1p0
```

```
1.0
```

```
julia> 0x1.8p3
```

```
12.0
```

```
julia> 0x.4p-1
```

```
0.125
```

```
julia> typeof(ans)
```

```
Float64
```

Half-precision floating-point numbers are also supported ([Float16](#)), but they are implemented in software and use [Float32](#) for calculations.

```
julia> sizeof(Float16(4.))
```

```
2
```

```
julia> 2*Float16(4.)
```

```
Float16(8.0)
```

The underscore `_` can be used as digit separator:

```
julia> 10_000, 0.000_000_005, 0xdead_beef, 0b1011_0010
```

```
(10000, 5.0e-9, 0xdeadbeef, 0xb2)
```

## Floating-point zero

Floating-point numbers have [two zeros](#), positive zero and negative zero. They are equal to each other but have different binary representations, as can be seen using the `bits` function:

```
julia> 0.0 == -0.0
```

```
true
```

## Special floating-point values

There are three specified standard floating-point values that do not correspond to any point on the real number line:

Float16	Float32	Float64	Name	Description
Inf16	Inf32	Inf	positive infinity	a value greater than all finite floating-point values
-Inf16	-Inf32	-Inf	negative infinity	a value less than all finite floating-point values
NaN16	NaN32	NaN	not a number	a value not == to any floating-point value (including itself)

For further discussion of how these non-finite floating-point values are ordered with respect to each other and other floats, see [Numeric Comparisons](#). By the [IEEE 754 standard](#), these floating-point values are the results of certain arithmetic operations:

```
julia> 1/Inf  
0.0  
  
julia> 1/0  
Inf  
  
julia> -5/0  
-Inf  
  
julia> 0.000001/0
```

```
julia> 0/0
```

NaN

```
julia> 500 + Inf
```

Inf

```
julia> 500 - Inf
```

-Inf

```
julia> Inf + Inf
```

Inf

```
julia> Inf - Inf
```

NaN

```
julia> Inf * Inf
```

Inf

```
julia> Inf / Inf
```

NaN

```
julia> 0 * Inf
```

NaN

The `typemin` and `typemax` functions also apply to floating-point types:

```
julia> (typemin(Float16), typemax(Float16))  
(-Inf16, Inf16)
```

```
julia> (typemin(Float32), typemax(Float32))
```

```
julia> (typemin(Float64), typemax(Float64))  
(-Inf, Inf)
```

## Machine epsilon

Most real numbers cannot be represented exactly with floating-point numbers, and so for many purposes it is important to know the distance between two adjacent representable floating-point numbers, which is often known as **machine epsilon**.

Julia provides `eps`, which gives the distance between `1.0` and the next larger representable floating-point value:

```
julia> eps(Float32)  
1.1920929f-7  
  
julia> eps(Float64)  
2.220446049250313e-16  
  
julia> eps() # same as eps(Float64)  
2.220446049250313e-16
```

These values are  $2.0^{-23}$  and  $2.0^{-52}$  as `Float32` and `Float64` values, respectively. The `eps` function can also take a floating-point value as an argument, and gives the absolute difference between that value and the next representable floating point value. That is, `eps(x)` yields a value of the same type as `x` such that `x + eps(x)` is the next representable floating-point value larger than `x`:

```
julia> eps(1.0)  
2.220446049250313e-16
```

```
julia> eps(1000.)  
1.1368683772161603e-13
```

```
julia> eps(1e-27)  
1.793662034335766e-43
```

```
julia> eps(0.0)  
5.0e-324
```

The distance between two adjacent representable floating-point numbers is not constant, but is smaller for smaller values and larger for larger values. In other words, the representable floating-point numbers are densest in the real number line near zero, and grow sparser exponentially as one moves farther away from zero. By definition, `eps(1.0)` is the same as `eps(Float64)` since `1.0` is a 64-bit floating-point value.

Julia also provides the `nextfloat` and `prevfloat` functions which return the next largest or smallest representable floating-point number to the argument respectively:

```
julia> x = 1.25f0  
1.25f0
```

```
julia> nextfloat(x)  
1.2500001f0
```

```
julia> prevfloat(x)  
1.2499999f0
```

```
julia> bitstring(prevfloat(x))  
"00111111001111111111111111111111"
```

```
julia> bitstring(x)  
"00111111010000000000000000000000"  
  
julia> bitstring(nextfloat(x))  
"00111111010000000000000000000001"
```

This example highlights the general principle that the adjacent representable floating-point numbers also have adjacent binary integer representations.

## Rounding modes

If a number doesn't have an exact floating-point representation, it must be rounded to an appropriate representable value, however, if wanted, the manner in which this rounding is done can be changed according to the rounding modes presented in the [IEEE 754 standard](#).

```
julia> x = 1.1; y = 0.1;  
  
julia> x + y  
1.2000000000000002  
  
julia> setrounding(Float64, RoundDown) do  
    x + y  
end  
1.2
```

The default mode used is always `RoundNearest`, which rounds to the nearest representable value, with ties rounded towards the nearest value with an even least significant bit.

Rounding is generally only correct for basic arithmetic functions (`+`, `-`, `*`, `/` and `sqrt`) and type conversion operations. Many other functions assume the default `RoundNearest` mode is set, and can give erroneous results when operating under other rounding modes.

## Background and References

Floating-point arithmetic entails many subtleties which can be surprising to users who are unfamiliar with the low-level implementation details. However, these subtleties are described in detail in most books on scientific computation, and also in the following references:

The definitive guide to floating point arithmetic is the [IEEE 754-2008 Standard](#); however, it is not available for free online.

For a brief but lucid presentation of how floating-point numbers are represented, see John D. Cook's [article](#) on the subject as well as his [introduction](#) to some of the issues arising from how this representation differs in behavior from the idealized abstraction of real numbers.

Also recommended is Bruce Dawson's [series of blog posts on floating-point numbers](#).

For an excellent, in-depth discussion of floating-point numbers and issues of numerical accuracy encountered when computing with them, see David Goldberg's paper [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#).

For even more extensive documentation of the history of, rationale for, and issues with floating-point numbers, as well as discussion of many other topics in numerical computing, see the [collected writings](#) of William

interest may be [An Interview with the Old Man of Floating-Point](#).

## 7.3 Arbitrary Precision Arithmetic

To allow computations with arbitrary-precision integers and floating point numbers, Julia wraps the [GNU Multiple Precision Arithmetic Library \(GMP\)](#) and the [GNU MPFR Library](#), respectively. The `BigInt` and `BigFloat` types are available in Julia for arbitrary precision integer and floating point numbers respectively.

Constructors exist to create these types from primitive numerical types, and `parse` can be used to construct them from `AbstractStrings`. Once created, they participate in arithmetic with all other numeric types thanks to Julia's [type promotion and conversion mechanism](#):

```
julia> BigInt(typemax(Int64)) + 1  
9223372036854775808
```

```
julia> parse(BigInt, "123456789012345678901234567890") + 1  
123456789012345678901234567891
```

```
julia> parse(BigFloat, "1.23456789012345678901")
```

```
julia> BigFloat(2.0^66) / 3
```

```
julia> factorial(BigInt(40))
```

81591528324789773434561126959611589427200000000

However, type promotion between the primitive types above and [BigInt](#)/[BigFloat](#) is not automatic and must be explicitly stated.

-9223372036854775808

```
julia> x = x - 1  
9223372036854775807
```

```
julia> typeof(x)  
Int64
```

```
julia> y = BigInt(typemin(Int64))  
-9223372036854775808
```

```
julia> y = y - 1  
-9223372036854775809
```

```
julia> typeof(y)  
BigInt
```

The default precision (in number of bits of the significand) and rounding mode of `BigFloat` operations can be changed globally by calling `setprecision` and `setrounding`, and all further calculations will take these changes in account. Alternatively, the precision or the rounding can be changed only within the execution of a particular block of code by using the same functions with a `do` block:

```
julia> setrounding(BigFloat, RoundUp) do
```

```
BigFloat(1) + parse(BigFloat, "0.1")
```

end

## 7.4 Numeric Literal Coefficients

To make common numeric formulas and expressions clearer, Julia allows variables to be immediately preceded by a numeric literal, implying multiplication. This makes writing polynomial expressions much cleaner:

```
julia> x = 3  
3  
  
julia> 2x^2 - 3x + 1  
10  
  
julia> 1.5x^2 - .5x + 1  
13.0
```

It also makes writing exponential functions more elegant:

```
julia> 2^2x  
64
```

The precedence of **NUMERIC LITERAL COEFFICIENTS** is the same as that of unary operators such as negation. So  $2^3x$  is parsed as  $2^{(3x)}$ , and  $2x^3$  is parsed as  $2*(x^3)$ .<sup>49</sup>

Numeric literals also work as coefficients to parenthesized expressions:

```
julia> 2(x-1)^2 - 3(x-1) + 1  
3
```

#### Note

The precedence of numeric literal coefficients used for implicit multiplication is higher than other binary operators such as multiplication ( $*$ ), and division ( $/$ ,  $\backslash$ , and  $//$ ). This means, for example, that  $1 / 2im$  equals  $-0.5im$  and  $6 // 2(2 + 1)$  equals  $1 // 1$ .

Additionally, parenthesized expressions can be used as coefficients to variables, implying multiplication of the expression by the variable:

```
julia> (x-1)x  
6
```

Neither juxtaposition of two parenthesized expressions, nor placing a variable before a parenthesized expression, however, can be used to imply multiplication:

```
julia> (x-1)(x+1)  
ERROR: MethodError: objects of type Int64 are not callable
```

```
julia> x(x+1)  
ERROR: MethodError: objects of type Int64 are not callable
```

Both expressions are interpreted as function application: any expression that is not a numeric literal, when immediately followed by a parenthetical, is interpreted as a function applied to the values in parentheses (see [Functions](#)

For more about functions, see [CHAPTER 7, FUNCTIONS](#), [CHAPTER 8, INTEGERS AND FLOATING-POINT NUMBERS](#)

the left-hand value is not a function.

The above syntactic enhancements significantly reduce the visual noise incurred when writing common mathematical formulae. Note that no whitespace may come between a numeric literal coefficient and the identifier or parenthesized expression which it multiplies.

## Syntax Conflicts

Juxtaposed literal coefficient syntax may conflict with two numeric literal syntaxes: hexadecimal integer literals and engineering notation for floating-point literals. Here are some situations where syntactic conflicts arise:

The hexadecimal integer literal expression `0xff` could be interpreted as the numeric literal `0` multiplied by the variable `xff`.

The floating-point literal expression `1e10` could be interpreted as the numeric literal `1` multiplied by the variable `e10`, and similarly with the equivalent E form.

In both cases, we resolve the ambiguity in favor of interpretation as a numeric literals:

Expressions starting with `0x` are always hexadecimal literals.

Expressions starting with a numeric literal followed by e or E are always floating-point literals.

## 7.5 Literal zero and one

Julia provides functions which return literal 0 and 1 corresponding to a specified type or the type of a given variable.

## FUNCTORIALE ZERO AND ONE

51

<code>zero(x)</code>	Literal zero of type x or type of variable x
<code>one(x)</code>	Literal one of type x or type of variable x

These functions are useful in [Numeric Comparisons](#) to avoid overhead from unnecessary [type conversion](#).

Examples:

```
julia> zero(Float32)
```

```
0.0f0
```

```
julia> zero(1.0)
```

```
0.0
```

```
julia> one(Int32)
```

```
1
```

```
julia> one(BigFloat)
```

```
1.0
```



# Chapter 8

## Mathematical Operations and Elementary Functions

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types, as well as providing portable, efficient implementations of a comprehensive collection of standard mathematical functions.

### 8.1 Arithmetic Operators

The following [arithmetic operators](#) are supported on all primitive numeric types:

Expression	Name	Description
<code>+x</code>	unary plus	the identity operation
<code>-x</code>	unary minus	maps values to their additive inverses
<code>x + y</code>	binary plus	performs addition
<code>x - y</code>	binary minus	performs subtraction
<code>x * y</code>	times	performs multiplication
<code>x / y</code>	divide	performs division
<code>x \ y</code>	inverse divide	equivalent to <code>y / x</code>
<code>x ^ y</code>	power	raises <code>x</code> to the <code>y</code> th power
<code>x % y</code>	remainder	equivalent to <code>rem(x,y)</code>

as well as the negation on [Bool](#) types:

Expression	Name	Description
<code>!x</code>	negation	changes <code>true</code> to <code>false</code> and vice versa

types “just work” naturally and automatically. See [Conversion and Promotion](#) for details of the promotion system.

Here are some simple examples using arithmetic operators:

```
julia> 1 + 2 + 3
```

```
6
```

```
julia> 1 - 2
```

```
-1
```

```
julia> 3*2/12
```

```
0.5
```

(By convention, we tend to space operators more tightly if they get applied before other nearby operators. For instance, we would generally write  $-x + 2$  to reflect that first  $x$  gets negated, and then 2 is added to that result.)

## 8.2 Bitwise Operators

The following [bitwise operators](#) are supported on all primitive integer types:

Expression	Name
$\sim x$	bitwise not
$x \& y$	bitwise and
$x \mid y$	bitwise or
$x \text{ } y$	bitwise xor (exclusive or)
$x \ggg y$	logical shift right
$x \gg y$	arithmetic shift right
$x \ll y$	logical/arithmetic shift left

Here are some examples with bitwise operators:

```
julia> ~123
```

```
-124
```

106

**julia>** 123 | 234

251

**julia>** 123 & 234

145

**julia>** xor(123, 234)

145

**julia>** ~UInt32(123)

0xffffffff84

**julia>** ~UInt8(123)

0x84

## 8.3 Updating operators

Every binary arithmetic and bitwise operator also has an updating version that assigns the result of the operation back into its left operand. The updating version of the binary operator is formed by placing `a =` immediately after the operator. For example, writing `x += 3` is equivalent to writing `x = x + 3`:

**julia>** x = 1

1

**julia>** x += 3

4

**Julia>** x

4

The updating versions of all the binary arithmetic and bitwise operators are:

```
+= -= *= /= \= ÷= %= ^= &= |= = >>>= >>= <<=
```

Note

An updating operator rebinds the variable on the left-hand side. As a result, the type of the variable may change.

```
julia> x = 0x01; typeof(x)
UInt8

julia> x *= 2 # Same as x = x * 2
2

julia> typeof(x)
Int64
```

## 8.4 Vectorized "dot" operators

For every binary operation like `^`, there is a corresponding "dot" operation `.^` that is automatically defined to perform `^` element-by-element on arrays. For example, `[1, 2, 3] ^ 3` is not defined, since there is no standard mathematical meaning to "cubing" a (non-square) array, but `[1, 2, 3] .^ 3` is defined as computing the elementwise (or "vectorized") result `[1^3, 2^3, 3^3]`. Similarly for unary operators like `!` or `√`, there is a corresponding `.√` that applies the operator elementwise.

```
julia> [1, 2, 3] .^ 3
3-element Array{Int64, 1}:
```

8

27

More specifically, `a .^ b` is parsed as the “dot” call `(^).(a, b)`, which performs a [broadcast](#) operation: it can combine arrays and scalars, arrays of the same size (performing the operation elementwise), and even arrays of different shapes (e.g. combining row and column vectors to produce a matrix). Moreover, like all vectorized “dot calls,” these “dot operators” are fusing. For example, if you compute `2 .* A.^2 .+ sin.(A)` (or equivalently `@. 2A^2 + sin(A)`, using the `@.` macro) for an array `A`, it performs a single loop over `A`, computing `2a^2 + sin(a)` for each element of `A`. In particular, nested dot calls like `f.(g.(x))` are fused, and “adjacent” binary operators like `x .+ 3 .* x.^2` are equivalent to nested dot calls `(+).(x, (*).(3, (^).(x, 2))).`

Furthermore, “dotted” updating operators like `a .+= b` (or `@. a += b`) are parsed as `a .= a .+ b`, where `.=` is a fused in-place assignment operation (see the [dot syntax documentation](#)).

Note the dot syntax is also applicable to user-defined operators. For example, if you define `(A, B) = kron(A, B)` to give a convenient infix syntax `A B` for Kronecker products ([kron](#)), then `[A, B] . [C, D]` will compute `[AC, BD]` with no additional coding.

Combining dot operators with numeric literals can be ambiguous. For example, it is not clear whether `1.+x` means `1. + x` or `1 .+ x`. Therefore this syntax is disallowed, and spaces must be used around the operator in such cases.

## 8.5 Numeric Comparisons

Standard comparison operations are defined for all the primitive numeric types:

## CHAPTER 8: MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

<code>==</code>	equality
<code>!=, ≠</code>	inequality
<code>&lt;</code>	less than
<code>&lt;=, ≤</code>	less than or equal to
<code>&gt;</code>	greater than
<code>&gt;=, ≥</code>	greater than or equal to

Here are some simple examples:

```
julia> 1 == 1
```

```
true
```

```
julia> 1 == 2
```

```
false
```

```
julia> 1 != 2
```

```
true
```

```
julia> 1 == 1.0
```

```
true
```

```
julia> 1 < 2
```

```
true
```

```
julia> 1.0 > 3
```

```
false
```

```
julia> 1 >= 1.0
```

```
true
```

```
julia> -1 <= 1
```

```
true
```

```
julia> -1 <= 1
```

true

```
julia> -1 <= -2
```

false

```
julia> 3 < -0.5
```

false

Integers are compared in the standard manner – by comparison of bits.

Floating-point numbers are compared according to the [IEEE 754 standard](#):

Finite numbers are ordered in the usual manner.

Positive zero is equal but not greater than negative zero.

`Inf` is equal to itself and greater than everything else except `NaN`.

`-Inf` is equal to itself and less than everything else except `NaN`.

`NaN` is not equal to, not less than, and not greater than anything, including itself.

The last point is potentially surprising and thus worth noting:

```
julia> NaN == NaN
```

false

```
julia> NaN != NaN
```

true

```
julia> NaN < NaN
```

false

60 CHAPTER 8. MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

```
julia> NaN > NaN
```

```
false
```

and can cause especial headaches with [Arrays](#):

```
julia> [1 NaN] == [1 NaN]
```

```
false
```

Julia provides additional functions to test numbers for special values, which can be useful in situations like hash key comparisons:

Function	Tests if
<code>isequal(x, y)</code>	x and y are identical
<code>isfinite(x)</code>	x is a finite number
<code>isinf(x)</code>	x is infinite
<code>isnan(x)</code>	x is not a number

`isequal` considers NaNs equal to each other:

```
julia> isequal(NaN, NaN)
```

```
true
```

```
julia> isequal([1 NaN], [1 NaN])
```

```
true
```

```
julia> isequal(NaN, NaN32)
```

```
true
```

`isequal` can also be used to distinguish signed zeros:

```
julia> -0.0 == 0.0
```

```
true
```

```
julia> isequal(-0.0, 0.0)
```

```
false
```

## Mixed numeric comparisons

Comparisons between signed integers, unsigned integers, and floats can be tricky. A great deal of care has been taken to ensure that Julia does them correctly.

For other types, `isequal` defaults to calling `==`, so if you want to define equality for your own types then you only need to add a `==` method. If you define your own equality function, you should probably define a corresponding `hash` method to ensure that `isequal(x, y)` implies `hash(x) == hash(y)`.

## Chaining comparisons

Unlike most languages, with the [notable exception of Python](#), comparisons can be arbitrarily chained:

```
julia> 1 < 2 <= 2 < 3 == 3 > 2 >= 1 == 1 < 3 != 5
true
```

Chaining comparisons is often quite convenient in numerical code. Chained comparisons use the `&&` operator for scalar comparisons, and the `&` operator for elementwise comparisons, which allows them to work on arrays. For example, `0 .< A .< 1` gives a boolean array whose entries are true where the corresponding elements of `A` are between 0 and 1.

Note the evaluation behavior of chained comparisons:

```
julia> v(x) = (println(x); x)
v (generic function with 1 method)

julia> v(1) < v(2) <= v(3)
2
1
3
true
```

## 62CHAPTER 8. MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

```
julia> v(1) > v(2) <= v(3)  
2  
1  
false
```

The middle expression is only evaluated once, rather than twice as it would be if the expression were written as `v(1) < v(2) && v(2) <= v(3)`. However, the order of evaluations in a chained comparison is undefined. It is strongly recommended not to use expressions with side effects (such as printing) in chained comparisons. If side effects are required, the short-circuit `&&` operator should be used explicitly (see [Short-Circuit Evaluation](#)).

### Elementary Functions

Julia provides a comprehensive collection of mathematical functions and operators. These mathematical operations are defined over as broad a class of numerical values as permit sensible definitions, including integers, floating-point numbers, rationals, and complex numbers, wherever such definitions make sense.

Moreover, these functions (like any Julia function) can be applied in "vectorized" fashion to arrays and other collections with the [dot syntax](#) `f.(A)`, e.g. `sin.(A)` will compute the sine of each element of an array `A`.

## 8.6 Operator Precedence and Associativity

Julia applies the following order and associativity of operations, from highest precedence to lowest:

---

<sup>1</sup>The unary operators `+` and `-` require explicit parentheses around their argument to disambiguate them from the operator `++`, etc. Other compositions of unary operators are parsed with right-associativity, e. g., `\sqrt{-a}` as `\sqrt{(-a)}`.

<sup>2</sup>The operators `+`, `++` and `*` are non-associative. `a + b + c` is parsed as `+(a, b, c)` not `+(+a, b), c`. However, the fallback methods for `+(a, b, c, d...)` and `*(a, b, c, d...)` are associative.

Category	Operator Precedence and Associativity	Associativity
Syntax	. followed by ::	Left
Exponentiation	$^$	Right
Unary	$+ - \sqrt{\cdot}$	Right <sup>1</sup>
Fractions	$//$	Left
Multiplication	$* / \% \& \backslash$	Left <sup>2</sup>
Bitshifts	$<< >> >>>$	Left
Addition	$+ -  $	Left <sup>2</sup>
Syntax	$\vdots \dots$	Left
Syntax	$ >$	Left
Syntax	$< $	Right
Comparisons	$> < >= <= == === != !== <:$	Non-associative
Control flow	$\&\&$ followed by    followed by ?	Right
Assignments	$= += -= *= /= //=\backslash= ^= ÷=%=  = &= =$ $<<= >>= >>>=$	Right

For a complete list of every Julia operator's precedence, see the top of this file: [src/julia-parser.scm](#)

You can also find the numerical precedence for any given operator via the built-in function `Base.operator_precedence`, where higher numbers take precedence:

```
julia> Base.operator_precedence(:+), Base.operator_precedence(:*),
   ↳ Base.operator_precedence(:..)
(9, 11, 15)

julia> Base.operator_precedence(:sin),
   ↳ Base.operator_precedence(:+=), Base.operator_precedence(:(=))
   ↳ # (Note the necessary parens on `:(=)` )
(0, 1, 1)
```

A symbol representing the operator associativity can also be found by calling the built-in function `Base.operator_associativity`:

---

d...) both default to left-associative evaluation.

## 64 CHAPTER 8. MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

```
Julia> Base.operator_associativity(:)
↳ Base.operator_associativity(:+),
↳ Base.operator_associativity(:^)
(:left, :none, :right)

julia> Base.operator_associativity(:),
↳ Base.operator_associativity(:sin),
↳ Base.operator_associativity(:)
(:left, :none, :right)
```

Note that symbols such as `:sin` return precedence 0. This value represents invalid operators and not operators of lowest precedence. Similarly, such operators are assigned associativity `:none`.

## 8.7 Numerical Conversions

Julia supports three forms of numerical conversion, which differ in their handling of inexact conversions.

The notation `T(x)` or `convert(T, x)` converts `x` to a value of type `T`.

- If `T` is a floating-point type, the result is the nearest representable value, which could be positive or negative infinity.
- If `T` is an integer type, an `InexactError` is raised if `x` is not representable by `T`.

`x % T` converts an integer `x` to a value of integer type `T` congruent to `x` modulo  $2^n$ , where `n` is the number of bits in `T`. In other words, the binary representation is truncated to fit.

The [Rounding functions](#) take a type `T` as an optional argument. For example, `round(Int, x)` is a shorthand for `Int(round(x))`.

```
julia> Int8(127)
```

```
127
```

```
julia> Int8(128)
```

```
ERROR: InexactError: trunc(Int8, 128)
```

```
Stacktrace:
```

```
[1] throw_inexacterror(::Symbol, ::Type{Int8}, ::Int64) at  
→ ./int.jl:34  
[2] checked_trunc_sint at ./int.jl:438 [inlined]  
[3] convert at ./int.jl:458 [inlined]  
[4] Int8(::Int64) at ./sysimg.jl:114
```

```
julia> Int8(127.0)
```

```
127
```

```
julia> Int8(3.14)
```

```
ERROR: InexactError: convert(Int8, 3.14)
```

```
Stacktrace:
```

```
[1] convert at ./float.jl:682 [inlined]  
[2] Int8(::Float64) at ./sysimg.jl:114
```

```
julia> Int8(128.0)
```

```
ERROR: InexactError: convert(Int8, 128.0)
```

```
Stacktrace:
```

```
[1] convert at ./float.jl:682 [inlined]  
[2] Int8(::Float64) at ./sysimg.jl:114
```

```
julia> 127 % Int8
```

```
127
```

## 66 CHAPTER 8. MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

Julia> 128 % Int8

-128

julia> round(Int8, 127.4)

127

julia> round(Int8, 127.6)

ERROR: InexactError: trunc(Int8, 128.0)

Stacktrace:

[1] trunc at ./float.jl:675 [inlined]

[2] round(::Type{Int8}, ::Float64) at ./float.jl:353

See [Conversion and Promotion](#) for how to define your own conversions and promotions.

### Rounding functions

Function	Description	Return type
round(x)	round x to the nearest integer	typeof(x)
round(T, x)	round x to the nearest integer	T
floor(x)	round x towards -Inf	typeof(x)
floor(T, x)	round x towards -Inf	T
ceil(x)	round x towards +Inf	typeof(x)
ceil(T, x)	round x towards +Inf	T
trunc(x)	round x towards zero	typeof(x)
trunc(T, x)	round x towards zero	T

### Division functions

### Sign and absolute value functions

### Powers, logs and roots

For an overview of why functions like [hypot](#), [expm1](#), and [log1p](#) are necessary and useful, see John D. Cook's excellent pair of blog posts on the subject: [expm1](#), [log1p](#), [erfc](#), and [hypot](#).

	Description
<code>div(x,y)</code>	truncated division; quotient rounded towards zero
<code>fld(x,y)</code>	floored division; quotient rounded towards <code>-Inf</code>
<code>cld(x,y)</code>	ceiling division; quotient rounded towards <code>+Inf</code>
<code>rem(x,y)</code>	remainder; satisfies $x == \text{div}(x,y)*y + \text{rem}(x,y)$ ; sign matches $x$
<code>mod(x,y)</code>	modulus; satisfies $x == \text{fld}(x,y)*y + \text{mod}(x,y)$ ; sign matches $y$
<code>mod1(x,y)</code>	mod with offset 1; returns $r(0,y]$ for $y>0$ or $r[y,0)$ for $y<0$ , where $\text{mod}(r,y) == \text{mod}(x,y)$
<code>mod2pi(x)</code>	modulus with respect to 2pi; $0 \leq \text{mod2pi}(x) < 2\pi$
<code>divrem(x,y)</code>	returns $(\text{div}(x,y), \text{rem}(x,y))$
<code>fld-mod(x,y)</code>	returns $(\text{fld}(x,y), \text{mod}(x,y))$
<code>gcd(x,y...)</code>	greatest positive common divisor of $x, y, \dots$
<code>lcm(x,y...)</code>	least positive common multiple of $x, y, \dots$

Function	Description
<code>abs(x)</code>	a positive value with the magnitude of $x$
<code>abs2(x)</code>	the squared magnitude of $x$
<code>sign(x)</code>	indicates the sign of $x$ , returning $-1, 0$ , or $+1$
<code>signbit(x)</code>	indicates whether the sign bit is on (true) or off (false)
<code>copysign(x,y)</code>	a value with the magnitude of $x$ and the sign of $y$
<code>flipsign(x,y)</code>	a value with the magnitude of $x$ and the sign of $x*y$

## Trigonometric and hyperbolic functions

All the standard trigonometric and hyperbolic functions are also defined:

<code>sin</code>	<code>cos</code>	<code>tan</code>	<code>cot</code>	<code>sec</code>	<code>csc</code>
<code>sinh</code>	<code>cosh</code>	<code>tanh</code>	<code>coth</code>	<code>sech</code>	<code>csch</code>
<code>asin</code>	<code>acos</code>	<code>atan</code>	<code>acot</code>	<code>asec</code>	<code>acsc</code>
<code>asinh</code>	<code>acosh</code>	<code>atanh</code>	<code>acoth</code>	<code>asech</code>	<code>acsch</code>
<code>sinc</code>	<code>cosc</code>	<code>atan2</code>			

These are all single-argument functions, with the exception of `atan2`, which gives the angle in `radians` between the  $x$ -axis and the point specified by its arguments, interpreted as  $x$  and  $y$  coordinates.

Additionally, `sinpi(x)` and `cospi(x)` are provided for more accurate computations of `sin(pi*x)` and `cos(pi*x)` respectively.

## 68 CHAPTER 8. MATHEMATICAL OPERATIONS AND ELEMENTARY FUNCTIONS

<code>sqrt(x)</code> , $\sqrt{x}$	square root of $x$
<code>cbrt(x)</code> , $\sqrt[3]{x}$	cube root of $x$
<code>hypot(x,y)</code>	hypotenuse of right-angled triangle with other sides of length $x$ and $y$
<code>exp(x)</code>	natural exponential function at $x$
<code>expm1(x)</code>	accurate $\exp(x) - 1$ for $x$ near zero
<code>ldexp(x,n)</code>	$x \cdot 2^n$ computed efficiently for integer values of $n$
<code>log(x)</code>	natural logarithm of $x$
<code>log(b,x)</code>	base $b$ logarithm of $x$
<code>log2(x)</code>	base 2 logarithm of $x$
<code>log10(x)</code>	base 10 logarithm of $x$
<code>log1p(x)</code>	accurate $\log(1+x)$ for $x$ near zero
<code>exponent(x)</code>	binary exponent of $x$
<code>significand(x)</code>	binary significand (a.k.a. mantissa) of a floating-point number $x$

In order to compute trigonometric functions with degrees instead of radians, suffix the function with d. For example, `sind(x)` computes the sine of  $x$  where  $x$  is specified in degrees. The complete list of trigonometric functions with degree variants is:

<code>sind</code>	<code>cosd</code>	<code>tand</code>	<code>cotd</code>	<code>secd</code>	<code>cscd</code>
<code>asind</code>	<code>acosd</code>	<code>atand</code>	<code>acotd</code>	<code>asecd</code>	<code>acscd</code>

## Special functions

Function	Description
<code>gamma(x)</code>	gamma function at $x$
<code>lgamma(x)</code>	accurate $\log(\text{gamma}(x))$ for large $x$
<code>lfact(x)</code>	accurate $\log(\text{factorial}(x))$ for large $x$ ; same as <code>lgamma(x+1)</code> for $x > 1$ , zero otherwise
<code>beta(x,y)</code>	beta function at $x,y$
<code>lbeta(x,y)</code>	accurate $\log(\text{beta}(x,y))$ for large $x$ or $y$

# Chapter 9

## Complex and Rational Numbers

Julia ships with predefined types representing both complex and rational numbers, and supports all standard [Mathematical Operations and Elementary Functions](#) on them. [Conversion and Promotion](#) are defined so that operations on any combination of predefined numeric types, whether primitive or composite, behave as expected.

### 9.1 Complex Numbers

The global constant `im` is bound to the complex number  $i$ , representing the principal square root of  $-1$ . It was deemed harmful to co-opt the name `i` for a global constant, since it is such a popular index variable name. Since Julia allows numeric literals to be [juxtaposed with identifiers as coefficients](#), this binding suffices to provide convenient syntax for complex numbers, similar to the traditional mathematical notation:

```
julia> 1 + 2im  
1 + 2im
```

You can perform all the standard arithmetic operations with complex numbers:

70 CHAPTER 9. COMPLEX AND RATIONAL NUMBERS

```
julia> (1 + 2im)*(2 - 3im)
```

8 + 1im

```
julia> (1 + 2im)/(1 - 2im)
```

-0.6 + 0.8im

```
julia> (1 + 2im) + (1 - 2im)
```

2 + 0im

```
julia> (-3 + 2im) - (5 - 1im)
```

-8 + 3im

```
julia> (-1 + 2im)^2
```

-3 - 4im

```
julia> (-1 + 2im)^2.5
```

2.7296244647840084 - 6.960664459571898im

```
julia> (-1 + 2im)^(1 + 1im)
```

-0.27910381075826657 + 0.08708053414102428im

```
julia> 3(2 - 5im)
```

6 - 15im

```
julia> 3(2 - 5im)^2
```

-63 - 60im

```
julia> 3(2 - 5im)^{-1.0}
```

0.20689655172413796 + 0.5172413793103449im

The promotion mechanism ensures that combinations of operands of different types just work:

```
julia> 2 + 1im
```

```
2 + 2im
```

```
julia> (2 + 3im) - 1
```

```
1 + 3im
```

```
julia> (1 + 2im) + 0.5
```

```
1.5 + 2.0im
```

```
julia> (2 + 3im) - 0.5im
```

```
2.0 + 2.5im
```

```
julia> 0.75(1 + 2im)
```

```
0.75 + 1.5im
```

```
julia> (2 + 3im) / 2
```

```
1.0 + 1.5im
```

```
julia> (1 - 3im) / (2 + 2im)
```

```
-0.5 - 1.0im
```

```
julia> 2im^2
```

```
-2 + 0im
```

```
julia> 1 + 3/4im
```

```
1.0 - 0.75im
```

Note that  $3/4im == 3/(4*im) == -(3/4*im)$ , since a literal coefficient binds more tightly than division.

Standard functions to manipulate complex values are provided:

```
julia> z = 1 + 2im
```

```
72 + 2im
```

## CHAPTER 9. COMPLEX AND RATIONAL NUMBERS

```
julia> real(1 + 2im) # real part of z
1

julia> imag(1 + 2im) # imaginary part of z
2

julia> conj(1 + 2im) # complex conjugate of z
1 - 2im

julia> abs(1 + 2im) # absolute value of z
2.23606797749979

julia> abs2(1 + 2im) # squared absolute value
5

julia> angle(1 + 2im) # phase angle in radians
1.1071487177940904
```

As usual, the absolute value (`abs`) of a complex number is its distance from zero. `abs2` gives the square of the absolute value, and is of particular use for complex numbers where it avoids taking a square root. `angle` returns the phase angle in radians (also known as the argument or `arg` function). The full gamut of other [Elementary Functions](#) is also defined for complex numbers:

```
julia> sqrt(1im)
0.7071067811865476 + 0.7071067811865475im

julia> sqrt(1 + 2im)
1.272019649514069 + 0.7861513777574233im
```

```
julia> cos(1 + 2im)
```

```
2.0327230070196656 - 3.0518977991518im
```

```
julia> exp(1 + 2im)
```

```
-1.1312043837568135 + 2.4717266720048188im
```

```
julia> sinh(1 + 2im)
```

```
-0.4890562590412937 + 1.4031192506220405im
```

Note that mathematical functions typically return real values when applied to real numbers and complex values when applied to complex numbers. For example, `sqrt` behaves differently when applied to `-1` versus `-1 + 0im` even though `-1 == -1 + 0im`:

```
julia> sqrt(-1)
```

```
ERROR: DomainError with -1.0:
```

```
sqrt will only return a complex result if called with a complex
→ argument. Try sqrt(Complex(x)).
```

```
Stacktrace:
```

```
[...]
```

```
julia> sqrt(-1 + 0im)
```

```
0.0 + 1.0im
```

The [literal numeric coefficient notation](#) does not work when constructing a complex number from variables. Instead, the multiplication must be explicitly written out:

```
julia> a = 1; b = 2; a + b*im
```

```
1 + 2im
```

However, this is not recommended; Use the `complex` function instead to construct a complex value directly from its real and imaginary parts:

```
74 julia> a = 1; b = 2; complex(a, b) CHAPTER 9: COMPLEX AND RATIONAL NUMBERS  
1 + 2im
```

This construction avoids the multiplication and addition operations.

`Inf` and `NaN` propagate through complex numbers in the real and imaginary parts of a complex number as described in the [Special floating-point values](#) section:

```
julia> 1 + Inf*im  
1.0 + Inf*im
```

```
julia> 1 + NaN*im  
1.0 + NaN*im
```

## 9.2 Rational Numbers

Julia has a rational number type to represent exact ratios of integers. Ratios are constructed using the `//` operator:

```
julia> 2//3  
2//3
```

If the numerator and denominator of a rational have common factors, they are reduced to lowest terms such that the denominator is non-negative:

```
julia> 6//9  
2//3
```

```
julia> -4//8  
-1//2
```

```
julia> 5//-15
```

```
julia> -4//-12  
1//3
```

This normalized form for a ratio of integers is unique, so equality of rational values can be tested by checking for equality of the numerator and denominator. The standardized numerator and denominator of a rational value can be extracted using the `numerator` and `denominator` functions:

```
julia> numerator(2//3)  
2  
  
julia> denominator(2//3)  
3
```

Direct comparison of the numerator and denominator is generally not necessary, since the standard arithmetic and comparison operations are defined for rational values:

```
julia> 2//3 == 6//9  
true  
  
julia> 2//3 == 9//27  
false  
  
julia> 3//7 < 1//2  
true  
  
julia> 3//4 > 2//3  
true
```

76 **julia>**  $2//4 + 1//6$

$2//3$

## CHAPTER 9. COMPLEX AND RATIONAL NUMBERS

**julia>**  $5//12 - 1//4$

$1//6$

**julia>**  $5//8 * 3//12$

$5//32$

**julia>**  $6//5 / 10//7$

$21//25$

Rationals can be easily converted to floating-point numbers:

**julia>** `float(3//4)`

0.75

Conversion from rational to floating-point respects the following identity for any integral values of  $a$  and  $b$ , with the exception of the case  $a == 0$  and  $b == 0$ :

**julia>** `a = 1; b = 2;`

**julia>** `isequal(float(a//b), a/b)`

true

Constructing infinite rational values is acceptable:

**julia>**  $5//0$

$1//0$

**julia>**  $-3//0$

$-1//0$

```
julia> typeof(ans)  
Rational{Int64}
```

Trying to construct a `NaN` rational value, however, is not:

```
julia> 0//0  
ERROR: ArgumentError: invalid rational: zero(Int64)//zero(Int64)  
Stacktrace:  
[...]
```

As usual, the promotion system makes interactions with other numeric types effortless:

```
julia> 3//5 + 1  
8//5  
  
julia> 3//5 - 0.5  
0.0999999999999998  
  
julia> 2//7 * (1 + 2im)  
2//7 + 4//7*im  
  
julia> 2//7 * (1.5 + 2im)  
0.42857142857142855 + 0.5714285714285714im  
  
julia> 3//2 / (1 + 2im)  
3//10 - 3//5*im  
  
julia> 1//2 + 2im  
1//2 + 2//1*im
```

78

**julia>** `1 + 2//3im`

`1//1 - 2//3*im`

## CHAPTER 9. COMPLEX AND RATIONAL NUMBERS

**julia>** `0.5 == 1//2`

`true`

**julia>** `0.33 == 1//3`

`false`

**julia>** `0.33 < 1//3`

`true`

**julia>** `1//3 - 0.33`

`0.003333333333332993`

# Chapter 10

## Strings

Strings are finite sequences of characters. Of course, the real trouble comes when one asks what a character is. The characters that English speakers are familiar with are the letters A, B, C, etc., together with numerals and common punctuation symbols. These characters are standardized together with a mapping to integer values between 0 and 127 by the [ASCII](#) standard. There are, of course, many other characters used in non-English languages, including variants of the ASCII characters with accents and other modifications, related scripts such as Cyrillic and Greek, and scripts completely unrelated to ASCII and English, including Arabic, Chinese, Hebrew, Hindi, Japanese, and Korean. The [Unicode](#) standard tackles the complexities of what exactly a character is, and is generally accepted as the definitive standard addressing this problem. Depending on your needs, you can either ignore these complexities entirely and just pretend that only ASCII characters exist, or you can write code that can handle any of the characters or encodings that one may encounter when handling non-ASCII text. Julia makes dealing with plain ASCII text simple and efficient, and handling Unicode is as simple and efficient as possible. In particular, you can write C-style string code to process ASCII strings, and they will work as expected, both in terms of performance and semantics. If such code encounters non-ASCII text, it will gracefully fail with a

8 Clear error message, rather than silently introducing corruption. This happens, modifying the code to handle non-ASCII data is straightforward.

There are a few noteworthy high-level features about Julia's strings:

The built-in concrete type used for strings (and string literals) in Julia is `String`. This supports the full range of `Unicode` characters via the `UTF-8` encoding. (A `transcode` function is provided to convert to/from other Unicode encodings.)

All string types are subtypes of the abstract type `AbstractString`, and external packages define additional `AbstractString` subtypes (e.g. for other encodings). If you define a function expecting a string argument, you should declare the type as `AbstractString` in order to accept any string type.

Like C and Java, but unlike most dynamic languages, Julia has a first-class type representing a single character, called `Char`. This is just a special kind of 32-bit primitive type whose numeric value represents a Unicode code point.

As in Java, strings are immutable: the value of an `AbstractString` object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.

Conceptually, a string is a partial function from indices to characters: for some index values, no character value is returned, and instead an exception is thrown. This allows for efficient indexing into strings by the byte index of an encoded representation rather than by a character index, which cannot be implemented both efficiently and simply for variable-width encodings of Unicode strings.

A `Char` value represents a single character: it is just a 32-bit primitive type with a special literal representation and appropriate arithmetic behaviors, whose numeric value is interpreted as a [Unicode code point](#). Here is how `Char` values are input and shown:

```
julia> 'x'  
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)  
  
julia> typeof(ans)  
Char
```

You can convert a `Char` to its integer value, i.e. code point, easily:

```
julia> Int('x')  
120  
  
julia> typeof(ans)  
Int64
```

On 32-bit architectures, `typeof(ans)` will be `Int32`. You can convert an integer value back to a `Char` just as easily:

```
julia> Char(120)  
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

Not all integer values are valid Unicode code points, but for performance, the `Char` conversion does not check that every character value is valid. If you want to check that each converted value is a valid code point, use the `isvalid` function:

```
julia> Char(0x110000)
'\U110000': Unicode U+110000 (category Cn: Other, not assigned)

julia> isvalid(Char, 0x110000)
false
```

As of this writing, the valid Unicode code points are **U+00** through **U+d7ff** and **U+e000** through **U+10ffff**. These have not all been assigned intelligible meanings yet, nor are they necessarily interpretable by applications, but all of these values are considered to be valid Unicode characters.

You can input any Unicode character in single quotes using \u followed by up to four hexadecimal digits or \U followed by up to eight hexadecimal digits (the longest valid value only requires six):

```
julia> '\u0'
'\0': ASCII/Unicode U+0000 (category Cc: Other, control)

julia> '\u78'
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)

julia> '\u2200'
' ': Unicode U+2200 (category Sm: Symbol, math)

julia> '\U10ffff'
'\U10ffff': Unicode U+10ffff (category Cn: Other, not assigned)
```

Julia uses your system's locale and language settings to determine which characters can be printed as-is and which must be output using the generic, escaped \u or \U input forms. In addition to these Unicode escape forms, all of C's traditional escaped input forms can also be used:

10.1 CHARACTERS  
**julia> Int('\'0')**

0

**julia> Int('\'t')**

9

**julia> Int('\'n')**

10

**julia> Int('\'e')**

27

**julia> Int('\'x7f')**

127

**julia> Int('\'177')**

127

**julia> Int('\'xff')**

255

You can do comparisons and a limited amount of arithmetic with Char values:

**julia> 'A' < 'a'**

true

**julia> 'A' <= 'a' <= 'Z'**

false

**julia> 'A' <= 'X' <= 'Z'**

true

84  
julia> 'x' - 'a'

23

CHAPTER 10. STRINGS

julia> 'A' + 1

'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)

## 10.2 String Basics

String literals are delimited by double quotes or triple double quotes:

julia> str = "Hello, world.\n"

"Hello, world.\n"

julia> """Contains "quote" characters"""

"Contains \"quote\" characters"

If you want to extract a character from a string, you index into it:

julia> str[1]

'H': ASCII/Unicode U+0048 (category Lu: Letter, uppercase)

julia> str[6]

', ': ASCII/Unicode U+002c (category Po: Punctuation, other)

julia> str[end]

'\n': ASCII/Unicode U+000a (category Cc: Other, control)

All indexing in Julia is 1-based: the first element of any integer-indexed object is found at index 1. (As we will see below, this does not necessarily mean that the last element is found at index `n`, where `n` is the length of the string.)

In any STRING BASIC expression, the keyword `end` can be used as a shorthand for the last index (computed by `endof(str)`). You can perform arithmetic and other operations with `end`, just like a normal value:

```
julia> str[end-1]
'.': ASCII/Unicode U+002e (category Po: Punctuation, other)

julia> str[end÷2]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

Using an index less than 1 or greater than `end` raises an error:

```
julia> str[0]
ERROR: BoundsError: attempt to access "Hello, world.\n"
    at index [0]
[...]

julia> str[end+1]
ERROR: BoundsError: attempt to access "Hello, world.\n"
    at index [15]
Stacktrace:
[...]
```

You can also extract a substring using range indexing:

```
julia> str[4:9]
"lo, wo"
```

Notice that the expressions `str[k]` and `str[k:k]` do not give the same result:

```
julia> str[6]
', ': ASCII/Unicode U+002c (category Po: Punctuation, other)
```

```
julia> str[6:6]
" "
```

The former is a single character value of type `Char`, while the latter is a string value that happens to contain only a single character. In Julia these are very different things.

Range indexing makes a copy of the selected part of the original string. Alternatively, it is possible to create a view into a string using the type `SubString`, for example:

```
julia> str = "long string"
"long string"

julia> substr = SubString(str, 1, 4)
"long"

julia> typeof(substr)
SubString{String}
```

Several standard functions like `chop`, `chomp` or `strip` return a `SubString`.

### 10.3 Unicode and UTF-8

Julia fully supports Unicode characters and strings. As [discussed above](#), in character literals, Unicode code points can be represented using Unicode \u and \U escape sequences, as well as all the standard C escape sequences. These can likewise be used to write string literals:

```
julia> s = "\u2200 x \u2203 y"
" x y"
```

~~What the INTCODE AND THE UTF~~ 8 characters are displayed as escapes or shown ~~as~~ special characters depends on your terminal's locale settings and its support for Unicode. String literals are encoded using the UTF-8 encoding. UTF-8 is a variable-width encoding, meaning that not all characters are encoded in the same number of bytes. In UTF-8, ASCII characters – i.e. those with code points less than 0x80 (128) – are encoded as they are in ASCII, using a single byte, while code points 0x80 and above are encoded using multiple bytes – up to four per character. This means that not every byte index into a UTF-8 string is necessarily a valid index for a character. If you index into a string at such an invalid byte index, an error is thrown:

```
julia> s[1]
': Unicode U+2200 (category Sm: Symbol, math)

julia> s[2]
ERROR: UnicodeError: invalid character index
[...]

julia> s[3]
ERROR: UnicodeError: invalid character index
[...]

julia> s[4]
' ': ASCII/Unicode U+0020 (category Zs: Separator, space)
```

In this case, the character is a three-byte character, so the indices 2 and 3 are invalid and the next character's index is 4; this next valid index can be computed by `nextind(s, 1)`, and the next index after that by `nextind(s, 4)` and so on.

Extraction of a substring using range indexing also expects valid byte indices or an error is thrown:

89 **julia>** s[1:1]

" "

**julia>** s[1:2]

ERROR: UnicodeError: invalid character index  
[ ... ]

**julia>** s[1:4]

" "

## CHAPTER 10. STRINGS

Because of variable-length encodings, the number of characters in a string (given by `length(s)`) is not always the same as the last index. If you iterate through the indices 1 through `endof(s)` and index into `s`, the sequence of characters returned when errors aren't thrown is the sequence of characters comprising the string `s`. Thus we have the identity that `length(s) <= endof(s)`, since each character in a string must have its own index. The following is an inefficient and verbose way to iterate through the characters of `s`:

```
julia> for i = 1:endof(s)
    try
        println(s[i])
    catch
        # ignore the index error
    end
```

```
x
```

```
y
```

The blank lines actually have spaces on them. Fortunately, the above awkward idiom is unnecessary for iterating through the characters in a string, since you can just use the string as an iterable object, no exception handling required:

```
julia> for c in s  
           println(c)  
       end
```

```
x
```

```
y
```

Julia uses the UTF-8 encoding by default, and support for new encodings can be added by packages. For example, the [LegacyStrings.jl](#) package implements `UTF16String` and `UTF32String` types. Additional discussion of other encodings and how to implement support for them is beyond the scope of this document for the time being. For further discussion of UTF-8 encoding issues, see the section below on [byte array literals](#). The `transcode` function

CHAPTER 11: CHARCODINGS, STRINGS

is provided to convert data between the various UTF-8 encodings, strings, for working with external data and libraries.

## 10.4 Concatenation

One of the most common and useful string operations is concatenation:

```
julia> greet = "Hello"  
"Hello"  
  
julia> whom = "world"  
"world"  
  
julia> string(greet, ", ", whom, ".\n")  
"Hello, world.\n"
```

Julia also provides `*` for string concatenation:

```
julia> greet * ", " * whom * ".\n"  
"Hello, world.\n"
```

While `*` may seem like a surprising choice to users of languages that provide `+` for string concatenation, this use of `*` has precedent in mathematics, particularly in abstract algebra.

In mathematics, `+` usually denotes a commutative operation, where the order of the operands does not matter. An example of this is matrix addition, where  $A + B == B + A$  for any matrices  $A$  and  $B$  that have the same shape. In contrast, `*` typically denotes a noncommutative operation, where the order of the operands does matter. An example of this is matrix multiplication, where in general  $A * B != B * A$ . As with matrix multiplication, string concatenation is noncommutative: `greet * whom != whom * greet`. As such, `*` is a

~~more INTERPOLATION~~ an infix string concatenation operator, consistent with common mathematical use.

More precisely, the set of all finite-length strings  $S$  together with the string concatenation operator  $*$  forms a **free monoid**  $(S, *)$ . The identity element of this set is the empty string, `" "`. Whenever a free monoid is not commutative, the operation is typically represented as `\cdot`, `*`, or a similar symbol, rather than `+`, which as stated usually implies commutativity.

## 10.5 Interpolation

Constructing strings using concatenation can become a bit cumbersome, however. To reduce the need for these verbose calls to `string` or repeated multiplications, Julia allows interpolation into string literals using `$`, as in Perl:

```
julia> "$greet, $whom.\n"
"Hello, world.\n"
```

This is more readable and convenient and equivalent to the above string concatenation – the system rewrites this apparent single string literal into a concatenation of string literals with variables.

The shortest complete expression after the `$` is taken as the expression whose value is to be interpolated into the string. Thus, you can interpolate any expression into a string using parentheses:

```
julia> "1 + 2 = $(1 + 2)"
"1 + 2 = 3"
```

Both concatenation and string interpolation call `string` to convert objects into string form. Most non-`AbstractString` objects are converted to strings closely corresponding to how they are entered as literal expressions:

```
92 julia> v = [1,2,3]
```

## CHAPTER 10. STRINGS

```
3-element Array{Int64,1}:
```

```
1
```

```
2
```

```
3
```

```
julia> "v: $v"
```

```
"v: [1, 2, 3]"
```

`string` is the identity for `AbstractString` and `Char` values, so these are interpolated into strings as themselves, unquoted and unescaped:

```
julia> c = 'x'
```

```
'x': ASCII/Unicode U+0078 (category Ll: Letter, lowercase)
```

```
julia> "hi, $c"
```

```
"hi, x"
```

To include a literal `$` in a string literal, escape it with a backslash:

```
julia> print("I have \$100 in my account.\n")
```

```
I have $100 in my account.
```

## 10.6 Triple-Quoted String Literals

When strings are created using triple-quotes (""""..."""") they have some special behavior that can be useful for creating longer blocks of text. First, if the opening """ is followed by a newline, the newline is stripped from the resulting string.

```
"""hello"""
```

is equivalent to

```
hello"""
```

but

```
"""
```

```
hello"""
```

will contain a literal newline at the beginning. Trailing whitespace is left unaltered. They can contain " symbols without escaping. Triple-quoted strings are also dedented to the level of the least-indented line. This is useful for defining strings within code that is indented. For example:

```
julia> str = """  
Hello,  
world.  
"""  
" Hello,\n world.\n"
```

In this case the final (empty) line before the closing """" sets the indentation level.

Note that line breaks in literal strings, whether single- or triple-quoted, result in a newline (LF) character \n in the string, even if your editor uses a carriage return \r (CR) or CRLF combination to end lines. To include a CR in a string, use an explicit escape \r; for example, you can enter the literal string "a CRLF line ending\r\n".

You can lexicographically compare strings using the standard comparison operators:

```
julia> "abracadabra" < "xylophone"
true

julia> "abracadabra" == "xylophone"
false

julia> "Hello, world." != "Goodbye, world."
true

julia> "1 + 2 = 3" == "1 + 2 = $(1 + 2)"
true
```

You can search for the index of a particular character using the `search` function:

```
julia> search("xylophone", 'x')
1

julia> search("xylophone", 'p')
5

julia> search("xylophone", 'z')
0
```

You can start the search for a character at a given offset by providing a third argument:

```
julia> search("xylophone", 'o')
4
```

```
julia> search("xylophone", 'o', 5)
```

```
7
```

```
julia> search("xylophone", 'o', 8)
```

```
0
```

You can use the `contains` function to check if a substring is contained in a string:

```
julia> contains("Hello, world.", "world")
```

```
true
```

```
julia> contains("Xylophon", "o")
```

```
true
```

```
julia> contains("Xylophon", "a")
```

```
false
```

```
julia> contains("Xylophon", 'o')
```

```
true
```

The last example shows that `contains` can also look for a character literal.

Two other handy string functions are `repeat` and `join`:

```
julia> repeat(".:Z:.", 10)
```

```
".:Z::::Z::::Z::::Z::::Z::::Z::::Z::::Z::::Z::::Z:."
```

```
julia> join(["apples", "bananas", "pineapples"], ", ", " and ")
```

```
"apples, bananas and pineapples"
```

Some other useful functions include:

96 `endof(str)` gives the maximal (byte) index that can be used into `str`.

`length(str)` the number of characters in `str`.

`i = start(str)` gives the first valid index at which a character can be found in `str` (typically 1).

`c, j = next(str,i)` returns next character at or after the index `i` and the next valid character index following that. With `start` and `endof`, can be used to iterate through the characters in `str`.

`ind2chr(str,i)` gives the number of characters in `str` up to and including any at index `i`.

`chr2ind(str,j)` gives the index at which the `j`th character in `str` occurs.

## 10.8 Non-Standard String Literals

There are situations when you want to construct a string or use string semantics, but the behavior of the standard string construct is not quite what is needed. For these kinds of situations, Julia provides [non-standard string literals](#). A non-standard string literal looks like a regular double-quoted string literal, but is immediately prefixed by an identifier, and doesn't behave quite like a normal string literal. Regular expressions, byte array literals and version number literals, as described below, are some examples of non-standard string literals. Other examples are given in the [Metaprogramming](#) section.

## 10.9 Regular Expressions

Julia has Perl-compatible regular expressions (regexes), as provided by the [PCRE](#) library. Regular expressions are related to strings in two ways: the obvious connection is that regular expressions are used to find regular patterns

10 Strings REGULAR EXPRESSIONS 107  
The other function is that regular expressions are themselves input as strings, which are parsed into a state machine that can be used to efficiently search for patterns in strings. In Julia, regular expressions are input using non-standard string literals prefixed with various identifiers beginning with `r`. The most basic regular expression literal without any options turned on just uses `r"..."`:

```
julia> r"^\s*(?:#|$)"  
r"^\s*(?:#|$)"
```

```
julia> typeof(ans)  
Regex
```

To check if a regex matches a string, use `ismatch`:

```
julia> ismatch(r"^\s*(?:#|$)", "not a comment")  
false
```

```
julia> ismatch(r"^\s*(?:#|$)", "# a comment")  
true
```

As one can see here, `ismatch` simply returns true or false, indicating whether the given regex matches the string or not. Commonly, however, one wants to know not just whether a string matched, but also how it matched. To capture this information about a match, use the `match` function instead:

```
julia> match(r"^\s*(?:#|$)", "not a comment")
```

```
julia> match(r"^\s*(?:#|$)", "# a comment")  
RegexMatch("#")
```

If the regular expression does not match the given string, `match` returns `nothing` – a special value that does not print anything at the interactive prompt.

Other than not printing, it is a completely normal value CHAPTER 10. STRINGS programmatically:

```
m = match(r"\s*(?:#|$)", line)
if m === nothing
    println("not a comment")
else
    println("blank or comment")
end
```

If a regular expression does match, the value returned by `match` is a **RegexMatch** object. These objects record how the expression matches, including the substring that the pattern matches and any captured substrings, if there are any. This example only captures the portion of the substring that matches, but perhaps we want to capture any non-blank text after the comment character. We could do the following:

```
julia> m = match(r"\s*(?:#\s*(.*))\s*$", "# a comment ")
RegexMatch("# a comment ", 1="a comment")
```

When calling `match`, you have the option to specify an index at which to start the search. For example:

```
julia> m = match(r"[0-9]", "aaaa1aaaa2aaaa3", 1)
RegexMatch("1")
```

```
julia> m = match(r"[0-9]", "aaaa1aaaa2aaaa3", 6)
RegexMatch("2")
```

```
julia> m = match(r"[0-9]", "aaaa1aaaa2aaaa3", 11)
RegexMatch("3")
```

You can extract the following info from a **RegexMatch** object:

the captured substrings as an array of strings: `m.captures`

the offset at which the whole match begins: `m.offset`

the offsets of the captured substrings as a vector: `m.offsets`

For when a capture doesn't match, instead of a substring, `m.captures` contains `nothing` in that position, and `m.offsets` has a zero offset (recall that indices in Julia are 1-based, so a zero offset into a string is invalid). Here is a pair of somewhat contrived examples:

```
julia> m = match(r"(a|b)(c)?(d)", "acd")
RegexMatch("acd", 1="a", 2="c", 3="d")

julia> m.match
"acd"

julia> m.captures
3-element Array{Union{Void, SubString{String}},1}:
 "a"
 "c"
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> m = match(r"(a|b)(c)?(d)", "ad")
RegexMatch("ad", 1="a", 2=nothing, 3="d")

julia> m.match
"ad"

julia> m.captures
3-element Array{Union{Void, SubString{String}},1}:
 "a"
 nothing
 "d"

julia> m.offset
1

julia> m.offsets
3-element Array{Int64,1}:
 1
 0
 2
```

It is convenient to have captures returned as an array so that one can use destructuring syntax to bind them to local variables:

```
julia> first, second, third = m.captures; first
"a"
```

Captures can also be accessed by indexing the `RegexMatch` object with the number or name of the capture group:

```
julia> m=match(r"(?<hour>\d+):(?<minute>\d+)", "12:45")
RegexMatch("12:45", hour="12", minute="45")
```

```
julia> m[:minute]
```

```
"45"
```

```
julia> m[2]
```

```
"45"
```

Captures can be referenced in a substitution string when using `replace` by using `\n` to refer to the nth capture group and prefixing the substitution string with `s`. Capture group 0 refers to the entire match object. Named capture groups can be referenced in the substitution with `g<groupname>`. For example:

```
julia> replace("first second", r"(\w+) (?<agroup>\w+)" ,  
    → s"\g<agroup> \1")  
"second first"
```

Numbered capture groups can also be referenced as `\g<n>` for disambiguation, as in:

```
julia> replace("a", r".", s"\g<0>1")  
"a1"
```

You can modify the behavior of regular expressions by some combination of the flags `i`, `m`, `s`, and `x` after the closing double quote mark. These flags have the same meaning as they do in Perl, as explained in this excerpt from the [perlre manpage](#):

i Do case-insensitive pattern matching.

If locale matching rules are in effect, the case map is taken from the current locale for code points less than 255, and

102 CHAPTER 10. STRINGS  
from Unicode rules for larger code points. However, matches that would cross the Unicode rules/non-Unicode rules boundary (ords 255/256) will not succeed.

- m Treat string as multiple lines. That is, change "^" and "\$" from matching the start or end of the string to matching the start or end of any line anywhere within the string.
- s Treat string as single line. That is, change "." to match any character whatsoever, even a newline, which normally it would not match.

Used together, as r""ms, they let the "." match any character whatsoever, while still allowing "^" and "\$" to match, respectively, just after and just before newlines within the string.

- x Tells the regular expression parser to ignore most whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The '#' character is also treated as a metacharacter introducing a comment, just as in ordinary code.

For example, the following regex has all three flags turned on:

```
julia> r"a+.*b+.*?d$"ism
r"a+.*b+.*?d$"ims

julia> match(r"a+.*b+.*?d$"ism, "Goodbye,\nOh, angry,\nBad
→ world\n")
RegexMatch("angry,\nBad world")
```

Todo - [Byte Array Literals](#) Of the form `r"""\n..."""`, are also supported (and may be convenient for regular expressions containing quotation marks or newlines).

## 10.10 Byte Array Literals

Another useful non-standard string literal is the byte-array string literal: `b"\n..."`. This form lets you use string notation to express literal byte arrays – i.e. arrays of `UInt8` values. The rules for byte array literals are the following:

ASCII characters and ASCII escapes produce a single byte.

`\x` and octal escape sequences produce the byte corresponding to the escape value.

Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

There is some overlap between these rules since the behavior of `\x` and octal escapes less than 0x80 (128) are covered by both of the first two rules, but here these rules agree. Together, these rules allow one to easily use ASCII characters, arbitrary byte values, and UTF-8 sequences to produce arrays of bytes. Here is an example using all three:

```
julia> b"DATA\xff\u2200"  
8-element Array{UInt8,1}:  
0x44  
0x41  
0x54  
0x41  
0xff  
0xe2
```

```
| 104
```

```
| 0x88
```

```
| 0x80
```

## CHAPTER 10. STRINGS

The ASCII string "DATA" corresponds to the bytes 68, 65, 84, 65. `\xff` produces the single byte 255. The Unicode escape `\u2200` is encoded in UTF-8 as the three bytes 226, 136, 128. Note that the resulting byte array does not correspond to a valid UTF-8 string – if you try to use this as a regular string literal, you will get a syntax error:

```
julia> "DATA\xff\u2200"  
ERROR: syntax: invalid UTF-8 sequence
```

Also observe the significant distinction between `\xff` and `\uff`: the former escape sequence encodes the byte 255, whereas the latter escape sequence represents the code point 255, which is encoded as two bytes in UTF-8:

```
julia> b"\xff"  
1-element Array{UInt8,1}:  
 0xff
```

```
julia> b"\uff"  
2-element Array{UInt8,1}:  
 0xc3  
 0xbf
```

Character literals use the same behavior.

For code points less than `\u80`, it happens that the UTF-8 encoding of each code point is just the single byte produced by the corresponding `\x` escape, so the distinction can safely be ignored. For the escapes `\x80` through `\xff` as compared to `\u80` through `\uff`, however, there is a major difference: the former escapes all encode single bytes, which – unless followed by very

~~Specific VERSION numbers LITERALS~~ form valid UTF-8 data, whereas the latter escapes all represent Unicode code points with two-byte encodings.

If this is all extremely confusing, try reading "[The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets](#)". It's an excellent introduction to Unicode and UTF-8, and may help alleviate some confusion regarding the matter.

## 10.11 Version Number Literals

Version numbers can easily be expressed with non-standard string literals of the form `v"..."`. Version number literals create `VersionNumber` objects which follow the specifications of [semantic versioning](#), and therefore are composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations. For example, `v"0.2.1-rc1+win64"` is broken into major version 0, minor version 2, patch version 1, pre-release `rc1` and build `win64`. When entering a version literal, everything except the major version number is optional, therefore e.g. `v"0.2"` is equivalent to `v"0.2.0"` (with empty pre-release/build annotations), `v"2"` is equivalent to `v"2.0.0"`, and so on.

`VersionNumber` objects are mostly useful to easily and correctly compare two (or more) versions. For example, the constant `VERSION` holds Julia version number as a `VersionNumber` object, and therefore one can define some version-specific behavior using simple statements as:

```
if v"0.2" <= VERSION < v"0.3-"  
    # do something specific to 0.2 release series  
end
```

Note that in the above example the non-standard version number `v"0.3-"` is used, with a trailing `-`: this notation is a Julia extension of the standard, and

it's used to indicate a version which is lower than any [CHAPTER 10, STRINGS](#) all of its pre-releases. So in the above example the code would only run with stable 0.2 versions, and exclude such versions as v"0.3.0-rc1". In order to also allow for unstable (i.e. pre-release) 0.2 versions, the lower bound check should be modified like this: v"0.2- " <= VERSION.

Another non-standard version specification extension allows one to use a trailing + to express an upper limit on build versions, e.g. VERSION > v"0.2-rc1+" can be used to mean any version above 0.2-rc1 and any of its builds: it will return `false` for version v"0.2-rc1+win64" and `true` for v"0.2-rc2".

It is good practice to use such special versions in comparisons (particularly, the trailing - should always be used on upper bounds unless there's a good reason not to), but they must not be used as the actual version number of anything, as they are invalid in the semantic versioning scheme.

Besides being used for the `VERSION` constant, `VersionNumber` objects are widely used in the `Pkg` module, to specify packages versions and their dependencies.

## 10.12 Raw String Literals

Raw strings without interpolation or unescaping can be expressed with non-standard string literals of the form `raw"..."`. Raw string literals create ordinary `String` objects which contain the enclosed contents exactly as entered with no interpolation or unescaping. This is useful for strings which contain code or markup in other languages which use \$ or \ as special characters.

The exception is that quotation marks still must be escaped, e.g. `raw"\\"` is equivalent to `"\\\"`. To make it possible to express all strings, backslashes then also must be escaped, but only when appearing right before a quote character:

```
|1.0.12> julia> println(Raw"\\"\\\"")  
|\\ \\"
```

107

Notice that the first two backslashes appear verbatim in the output, since they do not precede a quote character. However, the next backslash character escapes the backslash that follows it, and the last backslash escapes a quote, since these backslashes appear before a quote.



# Chapter 11

## 함수

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, in the sense that functions can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "assignment form":

```
julia> f(x,y) = x + y
f (generic function with 1 method)
```

In the assignment form, the body of the function must be a single expression, although it can be a compound expression (see [Compound Expressions](#)).

Short, simple function definitions are common in Julia. The syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

```
julia> f(2,3)  
5
```

Without parentheses, the expression `f` refers to the function object, and can be passed around like any value:

```
julia> g = f;  
  
julia> g(2,3)  
5
```

As with variables, Unicode can also be used for function names:

```
julia> Σ(x,y) = x + y  
Σ (generic function with 1 method)  
  
julia> Σ(2, 3)  
5
```

## 11.1 Argument Passing Behavior

Julia function arguments follow a convention sometimes called “pass-by-sharing”, which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable bindings (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as `Arrays`) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, `f`, from the previous section this is the value of the expression `x + y`. As in C and most other imperative or functional languages, the `return` keyword causes a function to return immediately, providing an expression whose value is returned:

```
function g(x,y)
    return x * y
    x + y
end
```

Since function definitions can be entered into interactive sessions, it is easy to compare these definitions:

```
julia> f(x,y) = x + y
f (generic function with 1 method)

julia> function g(x,y)

    return x * y

    x + y

end

g (generic function with 1 method)
```

```
julia> f(2,3)
```

```
|1|2|julia> g(2,3)
```

```
|6
```

## CHAPTER 11. 함수

Of course, in a purely linear function body like `g`, the usage of `return` is pointless since the expression `x + y` is never evaluated and we could simply make `x * y` the last expression in the function and omit the `return`. In conjunction with other control flow, however, `return` is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length `x` and `y`, avoiding overflow:

```
julia> function hypot(x,y)

    x = abs(x)

    y = abs(y)

    if x > y

        r = y/x

        return x*sqrt(1+r*r)

    end

    if y == 0

        return zero(x)

    end

    r = x/y
```

```
    end  
hypot (generic function with 1 method)  
  
julia> hypot(3, 4)  
5.0
```

There are three possible points of return from this function, returning the values of three different expressions, depending on the values of `x` and `y`. The `return` on the last line could be omitted since it is the last expression.

### 11.3 Operators Are Functions

In Julia, most operators are just functions with support for special syntax. (The exceptions are operators with special evaluation semantics like `&&` and `||`. These operators cannot be functions since [Short-Circuit Evaluation](#) requires that their operands are not evaluated before evaluation of the operator.) Accordingly, you can also apply them using parenthesized argument lists, just as you would any other function:

```
julia> 1 + 2 + 3  
6  
  
julia> +(1,2,3)  
6
```

The infix form is exactly equivalent to the function application form – in fact the former is parsed to produce the function call internally. This also means that you can assign and pass around operators such as `+` and `*` just like you would with other function values:

```
|14 julia> f = +;
```

## CHAPTER 11. 함수

```
julia> f(1,2,3)
```

```
6
```

Under the name `f`, the function does not support infix notation, however.

## 11.4 Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names. These are:

Expression	Calls
<code>[A B C ...]</code>	<code>hcat</code>
<code>[A; B; C; ...]</code>	<code>vcat</code>
<code>[A B; C D; ...]</code>	<code>hvcat</code>
<code>A'</code>	<code>adjoint</code>
<code>A.'</code>	<code>transpose</code>
<code>1:n</code>	<code>colon</code>
<code>A[i]</code>	<code>getindex</code>
<code>A[i]=x</code>	<code>setindex!</code>

## 11.5 Anonymous Functions

Functions in Julia are [first-class objects](#): they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

```
julia> x -> x^2 + 2x - 1  
#1 (generic function with 1 method)
```

```
julia> function (x)
```

```
end  
#3 (generic function with 1 method)
```

This creates a function taking one argument  $x$  and returning the value of the polynomial  $x^2 + 2x - 1$  at that value. Notice that the result is a generic function, but with a compiler-generated name based on consecutive numbering.

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is `map`, which applies a function to each value of an array and returns a new array containing the resulting values:

```
julia> map(round, [1.2,3.5,1.7])  
3-element Array{Float64,1}:  
 1.0  
 4.0  
 2.0
```

This is fine if a named function effecting the transform already exists to pass as the first argument to `map`. Often, however, a ready-to-use, named function does not exist. In these situations, the anonymous function construct allows easy creation of a single-use function object without needing a name:

```
julia> map(x -> x^2 + 2x - 1, [1,3,-1])  
3-element Array{Int64,1}:  
 2  
 14  
 -2
```

An anonymous function accepting multiple arguments can be written using the syntax `(x,y,z)->2x+y-z`. A zero-argument anonymous function is written as `()->3`. The idea of a function with no arguments may seem strange, but is useful for "delaying" a computation. In this usage, a block of code is wrapped in a zero-argument function, which is later invoked by calling it as `f`.

## 11.6 Tuples

Julia has a built-in data structure called a tuple that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is immutable). Tuples are constructed with commas and parentheses, and can be accessed via indexing:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"
```

Notice that a length-1 tuple must be written with a comma, `(1,)`, since `(1)` would just be a parenthesized value. `()` represents the empty (length-0) tuple.

The components of tuples can optionally be named, in which case a named tuple is constructed:

```
julia> x = (a=1, b=1+1)  
(a = 1, b = 2)
```

```
julia> x.a  
1
```

Named tuples are very similar to tuples, except that fields can additionally be accessed by name using dot syntax (`x.a`).

## 11.8 Multiple Return Values

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

```
julia> function foo(a,b)  
    a+b, a*b  
end  
foo (generic function with 1 method)
```

If you call it in an interactive session without assigning the return value anywhere, you will see the tuple returned:

```
julia> foo(2,3)  
(5, 6)
```

A typical usage of such a pair of return values, however, exchanges each value into a variable. Julia supports simple tuple "destructuring" that facilitates this:

```
julia> x, y = foo(2,3)
```

```
(5, 6)
```

```
julia> x
```

```
5
```

```
julia> y
```

```
6
```

You can also return multiple values via an explicit usage of the `return` keyword:

```
function foo(a,b)
    return a+b, a*b
end
```

This has the exact same effect as the previous definition of `foo`.

## 11.9 Argument destructuring

The destructuring feature can also be used within a function argument. If a function argument name is written as a tuple (e.g. `(x, y)`) instead of just a symbol, then an assignment `(x, y) = argument` will be inserted for you:

```
julia> minmax(x, y) = (y < x) ? (y, x) : (x, y)
```

```
julia> range((min, max)) = max - min
```

```
julia> range(minmax(10, 2))
```

```
8
```

~~Note that VARARGS FUNCTIONS~~ Notice that VARARGS FUNCTIONS uses parentheses in the definition of `range`. Without those, range would be a two-argument function, and this example would not work.<sup>10</sup>

## 11.10 Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as “varargs” functions, which is short for “variable number of arguments”. You can define a varargs function by following the last argument with an ellipsis:

```
julia> bar(a,b,x...) = (a,b,x)  
bar (generic function with 1 method)
```

The variables `a` and `b` are bound to the first two argument values as usual, and the variable `x` is bound to an iterable collection of the zero or more values passed to `bar` after its first two arguments:

```
julia> bar(1,2)  
(1, 2, ())
```

```
julia> bar(1,2,3)  
(1, 2, (3,))
```

```
julia> bar(1, 2, 3, 4)  
(1, 2, (3, 4))
```

```
julia> bar(1,2,3,4,5,6)  
(1, 2, (3, 4, 5, 6))
```

In all these cases, `x` is bound to a tuple of the trailing values passed to `bar`.

It is possible to constrain the number of values passed as a variable argument; this will be discussed later in [Parametrically-constrained Varargs methods](#).

On the flip side, it is often handy to "splat" the values contained in a collection into a function call as individual arguments. To do this, one also uses ... but in the function call instead:

```
julia> x = (3, 4)  
(3, 4)
```

```
julia> bar(1,2,x...)  
(1, 2, (3, 4))
```

In this case a tuple of values is spliced into a varargs call precisely where the variable number of arguments go. This need not be the case, however:

```
julia> x = (2, 3, 4)  
(2, 3, 4)
```

```
julia> bar(1,x...)  
(1, 2, (3, 4))
```

```
julia> x = (1, 2, 3, 4)  
(1, 2, 3, 4)
```

```
julia> bar(x...)  
(1, 2, (3, 4))
```

Furthermore, the iterable object splatted into a function call need not be a tuple:

```
julia> x = [3,4]  
2-element Array{Int64,1}:  
3  
4
```

```
julia> bar(1,2,x...)
```

```
(1, 2, (3, 4))
```

```
julia> x = [1,2,3,4]
```

```
4-element Array{Int64,1}:
```

```
1
```

```
2
```

```
3
```

```
4
```

```
julia> bar(x...)
```

```
(1, 2, (3, 4))
```

Also, the function that arguments are splatted into need not be a varargs function (although it often is):

```
julia> baz(a,b) = a + b;
```

```
julia> args = [1,2]
```

```
2-element Array{Int64,1}:
```

```
1
```

```
2
```

```
julia> baz(args...)
```

```
3
```

```
julia> args = [1,2,3]
```

```
3-element Array{Int64,1}:
```

```
1
```

```
2
```

```
3
```

```
julia> baz(args...)  
ERROR: MethodError: no method matching baz(::Int64, ::Int64,  
→ ::Int64)  
Closest candidates are:  
baz(::Any, ::Any) at none:1
```

As you can see, if the wrong number of elements are in the splatted container, then the function call will fail, just as it would if too many arguments were given explicitly.

## 11.11 Optional Arguments

In many cases, function arguments have sensible default values and therefore might not need to be passed explicitly in every call. For example, the library function `parse(T, num, base)` interprets a string as a number in some base. The `base` argument defaults to `10`. This behavior can be expressed concisely as:

```
function parse(T, num, base=10)  
    ###  
end
```

With this definition, the function can be called with either two or three arguments, and `10` is automatically passed when a third argument is not specified:

```
julia> parse(Int, "12", 10)  
12  
  
julia> parse(Int, "12", 3)  
5
```

```
|11.12> julia> parse(Int, "12")
```

```
|12
```

123

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see [Note on Optional and keyword Arguments](#)).

## 11.12 Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function `plot` that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like `plot(x, y, width=2)`, where we have chosen to specify only line width. Notice that this serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon in the signature:

```
function plot(x, y; style="solid", width=1, color="black")
    ##
end
```

When the function is called, the semicolon is optional: one can either call `plot(x, y, width=2)` or `plot(x, y; width=2)`, but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.

**Key**word argument default values are evaluated only when **CHAPTER 1** (when a corresponding keyword argument is not passed), and in left-to-right order. Therefore default expressions may refer to prior keyword arguments.

The types of keyword arguments can be made explicit as follows:

```
function f(;x::Int=1)
    ###
end
```

Extra keyword arguments can be collected using ..., as in varargs functions:

```
function f(x; y=0, kwargs...)
    ###
end
```

Inside f, kwargs will be a collection of (key, value) tuples, where each key is a symbol. Such collections can be passed as keyword arguments using a semicolon in a call, e.g. f(x, z=1; kwargs...). Dictionaries can also be used for this purpose.

One can also pass (key, value) tuples, or any iterable expression (such as a => pair) that can be assigned to such a tuple, explicitly after a semicolon. For example, plot(x, y; (:width,2)) and plot(x, y; :width => 2) are equivalent to plot(x, y, width=2). This is useful in situations where the keyword name is computed at runtime.

The nature of keyword arguments makes it possible to specify the same argument more than once. For example, in the call plot(x, y; options..., width=2) it is possible that the options structure also contains a value for width. In such a case the rightmost occurrence takes precedence; in this example, width is certain to have the value 2.

When optional and keyword argument default expressions are evaluated, only previous arguments are in scope. For example, given this definition:

```
function f(x, a=b, b=1)
    ###
end
```

the `b` in `a=b` refers to a `b` in an outer scope, not the subsequent argument `b`.

## 11.14 Do-Block Syntax for Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling `map` on a function with several cases:

```
map(x->begin
        if x < 0 && iseven(x)
            return 0
        elseif x == 0
            return 1
        else
            return x
        end
    end,
    [A, B, C])
```

Julia provides a reserved word `do` for rewriting this code more clearly:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
```

```
126     return 0
  elseif x == 0
    return 1
  else
    return x
  end
end
```

## CHAPTER 11. 함수

The `do x` syntax creates an anonymous function with argument `x` and passes it as the first argument to `map`. Similarly, `do a, b` would create a two-argument anonymous function, and a plain `do` would declare that what follows is an anonymous function of the form `() -> ....`

How these arguments are initialized depends on the "outer" function; here, `map` will sequentially set `x` to `A, B, C`, calling the anonymous function on each, just as would happen in the syntax `map(func, [A, B, C])`.

This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from `map`, such as managing system state. For example, there is a version of `open` that runs code ensuring that the opened file is eventually closed:

```
open("outfile", "w") do io
  write(io, data)
end
```

This is accomplished by the following definition:

```
function open(f::Function, args...)
  io = open(args...)
  try
    f(io)
```

```
    close(io)
end
end
```

Here, `open` first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the `do ... end` block. After your function exits, `open` will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception. (The `try/finally` construct will be described in [Control Flow](#).)

With the `do` block syntax, it helps to check the documentation or implementation to know how the arguments of the user function are initialized.

## 11.15 Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have “vectorized” versions of functions, which simply apply a given function  $f(x)$  to each element of an array  $A$  to yield a new array via  $f(A)$ . This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: if loops are slow, the “vectorized” version of a function can call fast library code written in a low-level language. In Julia, vectorized functions are not required for performance, and indeed it is often beneficial to write your own loops (see [Performance Tips](#)), but they can still be convenient. Therefore, any Julia function  $f$  can be applied elementwise to any array (or other collection) with the syntax  $f.(A)$ . For example `sin` can be applied to all elements in the vector  $A$ , like so:

```
julia> A = [1.0, 2.0, 3.0]
3-element Array{Float64,1}:
 1.0
 2.0
```

```
julia> sin.(A)
3-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Of course, you can omit the dot if you write a specialized "vector" method of `f`, e.g. via `f(A::AbstractArray) = map(f, A)`, and this is just as efficient as `f.(A)`. But that approach requires you to decide in advance which functions you want to vectorize.

More generally, `f.(args...)` is actually equivalent to `broadcast(f, args...)`, which allows you to operate on multiple arrays (even of different shapes), or a mix of arrays and scalars (see [Broadcasting](#)). For example, if you have `f(x, y) = 3x + 4y`, then `f.(pi, A)` will return a new array consisting of `f(pi, a)` for each `a` in `A`, and `f.(vector1, vector2)` will return a new vector consisting of `f(vector1[i], vector2[i])` for each index `i` (throwing an exception if the vectors have different length).

```
julia> f(x,y) = 3x + 4y;
julia> A = [1.0, 2.0, 3.0];
julia> B = [4.0, 5.0, 6.0];
julia> f.(pi, A)
3-element Array{Float64,1}:
 13.42477796076938
 17.42477796076938
 21.42477796076938
```

```
julia> f.(A, B)
3-element Array{Float64,1}:
 19.0
 26.0
 33.0
```

Moreover, nested `f.(args...)` calls are fused into a single `broadcast` loop. For example, `sin.(cos.(X))` is equivalent to `broadcast(x -> sin(cos(x)), X)`, similar to `[sin(cos(x)) for x in X]`: there is only a single loop over `X`, and a single array is allocated for the result. [In contrast, `sin(cos(X))` in a typical “vectorized” language would first allocate one temporary array for `tmp=cos(X)`, and then compute `sin(tmp)` in a separate loop, allocating a second array.] This loop fusion is not a compiler optimization that may or may not occur, it is a syntactic guarantee whenever nested `f.(args...)` calls are encountered. Technically, the fusion stops as soon as a “non-dot” function call is encountered; for example, in `sin.(sort(cos.(X)))` the `sin` and `cos` loops cannot be merged because of the intervening `sort` function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is pre-allocated, so that repeated calls do not allocate new arrays over and over again for the results (see [Pre-allocating outputs](#)). A convenient syntax for this is `X .= ...`, which is equivalent to `broadcast!(identity, X, ...)` except that, as above, the `broadcast!` loop is fused with any nested “dot” calls. For example, `X .= sin.(Y)` is equivalent to `broadcast!(sin, X, Y)`, overwriting `X` with `sin.(Y)` in-place. If the left-hand side is an array-indexing expression, e.g. `X[2:end] .= sin.(Y)`, then it translates to `broadcast!` on a `view`, e.g. `broadcast!(sin, view(X, 2:endof(X)), Y)`, so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can

CHAPTER 10.5

bogus and lead to code that is difficult to read, the macro `CHAPTER 10.5` provides to convert every function call, operation, and assignment in an expression into the "dotted" version.

```
julia> Y = [1.0, 2.0, 3.0, 4.0];  
  
julia> X = similar(Y); # pre-allocate output array  
  
julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))  
4-element Array{Float64,1}:  
 0.5143952585235492  
 -0.4042391538522658  
 -0.8360218615377305  
 -0.6080830096407656
```

Binary (or unary) operators like `.+` are handled with the same mechanism: they are equivalent to `broadcast` calls and are fused with other nested "dot" calls. `X .+= Y` etcetera is equivalent to `X .= X .+ Y` and results in a fused in-place assignment; see also [dot operators](#).

## 11.16 Further Reading

We should mention here that this is far from a complete picture of defining functions. Julia has a sophisticated type system and allows multiple dispatch on argument types. None of the examples given here provide any type annotations on their arguments, meaning that they are applicable to all types of arguments. The type system is described in [Types](#) and defining a function in terms of methods chosen by multiple dispatch on run-time argument types is described in [Methods](#).

# Chapter 12

## Control Flow

Julia는 다양한 제어 흐름 구조를 제공합니다.

복합 표현: `begin` 및 `(;)`.

조건부 평가: `if-elseif-else` 및 `? :` (삼항 연산자).

단락 평가: `&&`, `||` 및 연속 비교문.

반복 평가: 루프: `while` 및 `for`.

예외 처리: `try-catch`, `error` 및 `throw`.

태스크(일명 코루틴): `yieldto`.

처음 5개의 제어 흐름 메커니즘은 고급 프로그래밍 언어의 표준입니다. 하지만 는 그렇지 않습니다. 태스크는 비지역적 제어 흐름을 제공하여, 일시적으로 중단된 계산을 바꾸는 것을 가능하게 만듭니다. 태스크는 강력한 구조입니다: Julia는 예외 처리 및 협력적 멀티태스킹 모두를 태스크를 사용하여 구현합니다. 일상적인 프로그래밍에서는 태스크를 사용할 필요가 없지만, 몇몇 문제는 태스크를 사용함으로써 더 쉽게 해결될 수 있습니다.

### 12.1 복합 표현

때로는 여러 하위식을 순서대로 평가하는 단 하나의 식이 더 편리하며, 이 경우 마지막 하위식의 값을 그 값으로 반환하게 됩니다. 이를 수행하는 두 개의 Julia 구조가 있습니다:

**begin** 구문과 ( ; ) 체인 구문입니다. 두 복잡한 구문을 대체하는 데 유용합니다. 다음은 **begin** 구문의 예제입니다.

```
julia> z = begin  
           x = 1  
  
           y = 2  
  
           x + y  
  
       end  
3
```

위와 같이 식의 길이가 매우 짧고 단순하다면, 유용한 ( ; ) 체인 구문을 사용해 한 줄로 쉽게 표현할 수 있습니다.

```
julia> z = (x = 1; y = 2; x + y)  
3
```

이 구문은 함수 문서에 소개된 간결한 단일 행 함수를 정의할 때 특히 유용합니다. 전형적인 구문처럼 보이겠지만, **begin** 블록 내부가 여러 줄일 필요도 없고, ( ; ) 체인이 전부 한 줄에서 이루어질 필요도 없습니다.

```
julia> begin x = 1; y = 2; x + y end  
3  
  
julia> (x = 1;  
           y = 2;  
           x + y)
```

3

조건부 평가는 논리식의 값에 따라 일부 코드의 실행 여부를 결정합니다. 다음은 `if-elseif-else` 조건 구문의 구조입니다.

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

조건식 `x < y`가 `true`이면 해당 블록이 실행됩니다. 참이 아니라면, 조건식 `x > y`를 평가하고 `true`이면 해당 블록이 실행됩니다. 만약 두 표현 둘 다 참이 아니라면, `else` 블록이 실행됩니다. 다음은 실행 예제입니다.

```
julia> function test(x, y)

    if x < y

        println("x is less than y")

    elseif x > y

        println("x is greater than y")

    else

        println("x is equal to y")

    end
```

```
    end  
test (generic function with 1 method)  
  
julia> test(1, 2)  
x is less than y  
  
julia> test(2, 1)  
x is greater than y  
  
julia> test(1, 1)  
x is equal to y
```

`elseif`와 `else` 블록은 선택 사항이며, 원하는 만큼 많은 `elseif` 블록을 사용할 수 있습니다. `if-elseif-else` 구문 안의 조건식은 어느 한 식이 처음으로 `true`로 평가될 때까지 평가되고, 그 후에 관련 블록이 실행되며, 이후로는 어떤 식이나 블록도 실행되지 않습니다.

`if` 블록은 지역 범위를 만들지 않기 때문에 한 마디로 "구멍이 났다"고 할 수 있습니다. 이는 `if` 절 안에서 정의된 새로운 변수가 `if` 블록 다음에도 사용될 수 있음을 의미합니다. 따라서, 위에서 정의한 `test` 함수를 다음과 같이 정의할 수도 있습니다.

```
julia> function test(x,y)  
  
    if x < y  
  
        relation = "less than"  
  
    elseif x == y  
  
        relation = "equal to"
```

```
    relation = "greater than"

end

println("x is ", relation, " y.")

end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.
```

`relation` 변수는 `if` 블록 안에서 선언되었지만, 블록 밖에서 사용되고 있습니다. 그러나, 모든 코드 경로가 이 구문을 통해 변수 값을 정의할 수 있는지 확인해야 합니다. 위 함수를 다음과 같이 변경하면 런타임 오류가 발생합니다.

```
julia> function test(x,y)

    if x < y

        relation = "less than"

    elseif x == y

        relation = "equal to"

    end

    println("x is ", relation, " y.")
```

```
test (generic function with 1 method)
```

```
julia> test(1,2)
```

```
x is less than y.
```

```
julia> test(2,1)
```

```
ERROR: UndefVarError: relation not defined
```

```
Stacktrace:
```

```
[1] test(::Int64, ::Int64) at ./none:7
```

if 블록도 값을 반환하기 때문에 다른 많은 언어에서 오는 사용자에게는 어색해 보일 수 있습니다. 이 같은 단순히 선택한 분기에서 마지막으로 실행한 명령문의 반환값이며, 따라서

```
julia> x = 3
```

```
3
```

```
julia> if x > 0
```

```
    "positive!"
```

```
else
```

```
    "negative..."
```

```
end
```

```
"positive!"
```

아주 짧은(한 줄로 된) 조건문은 다음 절에 설명되었듯이 Julia의 단락 회로 평가를 통해 자주 표현된다는 것을 유념하시기 바랍니다.

C, MATLAB, Perl, Python, Ruby와는 다르게 조건식의 값이 `true`나 `false`가 아니면 오류가 발생하며, 이는 Java와 같이 자료형을 엄격하게 다루는 언어와 비슷하다고 할 수

```
julia> if 1
           println("true")
       end
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

이 오류는 조건부에 잘못된 자료형을 넣었음을 나타냅니다. Int64형 대신 Bool형이 들어가야 하죠.

소위 "삼항 연산자"라고 불리는 ?:는 if-elseif-else 구문과 밀접한 관련이 있습니다. 후자가 긴 코드 블록의 조건 실행에 사용되는 것과 달리, 전자는 단일식에서의 조건부 선택이 필요한 곳에서 사용됩니다. 이 연산자는 대부분의 다른 언어에서도 피연산자 셋을 취하는 유일한 연산자라는 칭호를 얻었습니다.

```
| a ? b : c
```

? 앞의 a는 조건식이고, 삼항 연산자는 a가 true이면 : 앞의 b를, false이면 : 뒤의 c를 실행합니다. 여기서 ?와 : 주위에는 공백이 있어야 함을 명심하십시오. a?b:c와 같은 식은 유효하지 않은 식입니다.(다만 ?와 : 각각의 뒤에 개행 문자는 사용 가능)

이 동작을 이해하는 가장 쉬운 방법은 예제를 보는 것입니다. 이전 예제에서 println 호출은 세 브랜치 모두에서 공유되었습니다. 실제로 고른 것은 오직 출력할 리터럴 문자열이었습니다. 이제 삼항 연산자를 사용하여 보다 간결하게 예제를 작성할 수 있습니다. 명확히 하기 위해, 먼저 둘 중 하나를 고르는 버전을 사용해 볼시다.

```
julia> x = 1; y = 2;
julia> println(x < y ? "less than" : "not less than")
less than
```

138  
julia> x = 1; y = 0;

## CHAPTER 12. CONTROL FLOW

julia> println(x < y ? "less than" : "not less than")  
not less than

`x < y` 식이 참이면, 전체 삼항 연산자 식은 "" 문자열을 평가하고, 거짓이면 "" 문자열을 평가할 것입니다. 기준의 셋 중 하나를 고르는 예제를 구현하려면 삼항 연산자를 여러 번 사용하여 중첩할 필요가 있습니다.

```
julia> test(x, y) = println(x < y ? "x is less than y" :  
                           x > y ? "x is greater than y" : "x is  
                           → equal to y")  
test (generic function with 1 method)  
  
julia> test(1, 2)  
x is less than y  
  
julia> test(2, 1)  
x is greater than y  
  
julia> test(1, 1)  
x is equal to y
```

연결을 쉽게 하기 위해 연산자는 오른쪽에서 왼쪽으로 연결됩니다.

`if-elseif-else`와 같이 조건식이 각각 `true`나 `false`로 평가될 때만 : 앞뒤로 있는 식이 평가된다는 점 역시 중요합니다.

```
julia> v(x) = (println(x); x)  
v (generic function with 1 method)  
  
julia> 1 < 2 ? v("yes") : v("no")
```

```
"yes"
```

```
julia> 1 > 2 ? v("yes") : v("no")
```

```
no
```

```
"no"
```

### 12.3 단락 평가

단락 평가는 조건부 평가와 상당히 유사합니다. 이 동작은 `&&` 및 `||` 연산자가 있는 대부분의 명령형 프로그래밍 언어에서 찾을 수 있습니다. 이런 연산자로 연결된 일련의 표현식에서, 최종 논리값을 결정하는 데 필요한 최소 식만 평가됩니다. 명쾌하게 말하자면, 이는 다음을 의미합니다.

표현식 `a && b`에서, 하위 표현식 `b`는 오직 `a`가 `true`로 평가될 때만 평가를 받는다.

표현식 `a || b`에서, 하위 표현식 `b`는 오직 `a`가 `false`로 평가될 때만 평가를 받는다.

왜냐하면, `a`가 `false`이면, `b`의 값에 관계없이 `a && b`는 무조건 `false`가 되고, `a`가 `true`이면, `b`의 값에 관계없이 `a && b`는 무조건 `true`가 되기 때문입니다. `&&`와 `||` 모두 오른쪽에 연관되지만, `&&`가 `||`보다 우선 순위가 더 높습니다. 실험해 보면 동작을 이해하기 쉽습니다.

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)
```

```
julia> f(x) = (println(x); false)
f (generic function with 1 method)
```

```
julia> t(1) && t(2)
```

```
1
```

```
2
```

```
julia> t(1) && f(2)
```

```
1
```

```
2
```

```
false
```

```
julia> f(1) && t(2)
```

```
1
```

```
false
```

```
julia> f(1) && f(2)
```

```
1
```

```
false
```

```
julia> t(1) || t(2)
```

```
1
```

```
true
```

```
julia> t(1) || f(2)
```

```
1
```

```
true
```

```
julia> f(1) || t(2)
```

```
1
```

```
2
```

```
true
```

```
julia> f(1) || f(2)
```

```
1
```

```
2
```

&& 및 || 연산자의 다양한 조합의 연관성과 우선 순위를 통해 같은 방식으로 쉽게 실험할 수 있습니다.

이 동작은 Julia에서 매우 짧은 if 문의 대용으로 자주 사용됩니다. if <> <> end 대신에, <> 그리고 나서 <>이라고 읽을 수 있는 <> && <>을 쓸 수 있습니다. 비슷하게, if ! <> <> end 대신에, <> 아니면 <>이라고 읽을 수 있는 <> || <>을 쓸 수 있습니다.

예제로 재귀적 팩토리얼 함수를 다음과 같이 선언할 수 있습니다.

```
julia> function fact(n::Int)
           n >= 0 || error("n must be non-negative")
           n == 0 && return 1
           n * fact(n-1)
       end
fact (generic function with 1 method)

julia> fact(5)
120

julia> fact(0)
1

julia> fact(-1)
ERROR: n must be non-negative
Stacktrace:
 [1] fact(::Int64) at ./none:2
```

```
julia> f(1) & t(2)
```

```
1
```

```
2
```

```
false
```

```
julia> t(1) | t(2)
```

```
1
```

```
2
```

```
true
```

`if`, `elseif` 또는 삼항 연산자에서 사용되는 조건식과 마찬가지로, `&&`나 `||` 역시 피연산자가 논리값(`true` 또는 `false`)을 가져야 합니다. 조건부 체인의 가장 마지막 항목을 제외하고는 어디에도 비논리값을 사용하면 오류가 발생합니다.

```
julia> 1 && true
```

```
ERROR: TypeError: non-boolean (Int64) used in boolean context
```

반면에 조건부 체인 끝에는 어떤 표현식이든 사용할 수 있습니다. 이는 선행 조건에 따라 평가되고 반환될 것이기 때문입니다.

```
julia> true && (x = (1, 2, 3))
```

```
(1, 2, 3)
```

```
julia> false && (x = (1, 2, 3))
```

```
false
```

반복 평가식에는 두 구문이 있습니다. 바로 `while` 루프와 `for` 루프입니다. 다음은 `while` 루프의 예제입니다.

```
julia> i = 1;

julia> while i <= 5

        println(i)

        i += 1

end

1
2
3
4
5
```

`while` 루프는 조건식(여기서는 `i <= 5`)을 평가하여, `true`가 아닐 때까지 내내 `while` 루프를 반복하여 실행합니다. `while` 루프에 도착했을 때, 조건식이 `false`이면 루프를 실행하지 않습니다.

`for` 루프는 평범한 반복 평가문을 작성하기 쉽게 만들어 줍니다. 위의 `while` 루프와 같이 위아래로 세는 것이 일반적이므로 `for` 루프를 통해 보다 간결하게 표현할 수 있습니다.

```
julia> for i = 1:5

        println(i)

end

1
```

```
3  
4  
5
```

여기에서 `1:5`는 범위 객체이며 숫자 1, 2, 3, 4, 5의 순서를 나타냅니다. `for` 루프는 이 값을 반복하며, 각 수들을 차례로 변수 `i`에 할당합니다. 앞의 `while` 루프 형식과 `for` 루프 형식의 중요한 차이점 중 하나는 바로 변수가 표시되는 범위입니다. 만약 변수 `i`가 다른 영역에서 선언되지 않았다면, `for` 루프 형식에서는 `for` 루프 내부에서만 볼 수 있고, 루프 외부나 루프 종료 이후로는 볼 수 없습니다. 이를 테스트하려면 새로운 대화형 세션 인스턴스나 다른 변수 이름이 필요할 겁니다.

```
julia> for j = 1:5  
    println(j)  
end  
1  
2  
3  
4  
5  
  
julia> j  
ERROR: UndefVarError: j not defined
```

변수 범위에 관한 자세한 설명과 그것이 Julia에서 어떻게 작동하는지는 `Scope of Variables` 문서를 통해 확인하십시오.

일반적으로, `for` 루프 구조는 어떤 컨테이너든 반복할 수 있습니다. 이 경우, 코드의 더 명확한 가독성을 위해 = 대신 일반적으로 `in`이나 `ga` 대용(그러나 완전히 동등한) 키워드로 사용됩니다.

12.4. 반복 평가: 루프 [1, 4, 0]

145

```
    println(i)

end

1
4
0

julia> for s  ["foo", "bar", "baz"]

    println(s)

end

foo
bar
baz
```

다양한 유형의 반복 가능한 컨테이너가 매뉴얼 뒷부분(예: Multi-dimensional Arrays 문서 참조)에서 소개되고 논의될 것입니다.

테스트 조건이 위치되기 전에 **while** 반복을 종료하거나, 반복용 변수가 끝에 도달하기 전에 **for** 루프를 멈추는 것이 편리할 때가 있습니다. 이는 **break** 키워드로 수행할 수 있습니다.

```
julia> i = 1;

julia> while true

    println(i)

    if i >= 5
```

```
    break  
  
    end  
  
    i += 1  
  
end  
1  
2  
3  
4  
5  
  
julia> for j = 1:1000  
  
    println(j)  
  
    if j >= 5  
  
        break  
  
    end  
  
end  
1  
2  
3  
4  
5
```

`break` 키워드 없이는 `while` 루프는 절대 스스로 종료되지 않을 것이며, `for` 루프는 1000까지 세고 말 것입니다. 이 루프 모두 `break`를 사용하여 빠져나갈 수 있습니다.

다면 상황에 따라 반복을 중지하고 즉시 다음 단계로 넘어가는 것이 더 유용할 수 있습니다.  
`continue` 키워드는 다음을 수행합니다.

```
julia> for i = 1:10  
         if i % 3 != 0  
             continue  
         end  
         println(i)  
     end  
3  
6  
9
```

이는 조건을 무효화하고 조건을 무효화하고 `println` 호출을 `if` 블록 안에 두어 더 똑같은 동작을 명확하게 나타낼 수 있기 때문에 다소 고안된 예제입니다. 실제 코드에서는 `continue` 뒤에 평가할 코드가 더 많을 것이며, 종종 `continue`가 여러 번 사용될 수도 있습니다.

다중 중첩 `for` 루프는 하나의 외부 루프로 결합되어, 반복용 변수의 데카르트 곱을 형성합니다.

```
julia> for i = 1:2, j = 3:4  
         println((i, j))  
     end  
(1, 3)  
(1, 4)
```

```
|148 (2, 3)  
|(2, 4)
```

## CHAPTER 12. CONTROL FLOW

이러한 루프 내부의 `break`문은 내부의 루프 하나만을 종료하는 것이 아닌, 루프 전체를 종료합니다.

### 12.5 예외 처리

예기치 않은 조건이 발생하면 함수가 호출자에게 적절한 값을 반환하지 못할 수 있습니다. 이런 경우에는 예외적인 조건에서 진단 오류 메시지를 출력하는 동안 프로그램을 종료하는 것이 좋을 수도 있지만, 프로그래머가 예외적인 상황을 처리하는 코드를 제공한 경우 해당 코드가 적절한 조치를 취하도록 하는 것이 최선의 방법입니다.

#### 기본 제공 ‘예외’

예기치 않은 조건이 일어나면 `Exception`이 발생합니다. 아래에 나열된 기본 제공 `Exception`은 모두 정상적인 제어 흐름을 방해합니다.

예를 들어, 음의 실수 값에 적용된 `sqrt` 함수는 `DomainError`를 `throw`합니다.

```
|julia> sqrt(-1)  
ERROR: DomainError with -1.0:  
  sqrt will only return a complex result if called with a complex  
  ↵ argument. Try sqrt(Complex(x)).  
Stacktrace:  
[...]
```

다음과 같은 방법으로 사용자 정의 예외를 직접 만들 수 있습니다.

```
|julia> struct MyCustomException <: Exception end
```

#### `throw` 함수

예외는 `throw`를 사용하여 명시적으로 만들 수 있습니다. 예를 들어, 인수가 음수이면 인수가 음수가 아닌 숫자로만 정의된 함수를 작성하여 `DomainError`를 `throw`할 수 있습니다.

12 Exception 처리
ArgumentError
BoundsError
CompositeException
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException
KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
ParseError
SystemError
TypeError
UndefRefError
UndefVarError
UnicodeError

```
julia> f(x) = x>=0 ? exp(-x) : throw(DomainError(x, "argument must
→ be nonnegative"))
f (generic function with 1 method)

julia> f(1)
0.36787944117144233

julia> f(-1)
ERROR: DomainError with -1:
argument must be nonnegative
Stacktrace:
 [1] f(::Int64) at ./none:1
```

괄호가 없는 DomainError는 예외가 아니라 예외 유형임을 기억하십시오. Exception

```
julia> typeof(DomainError(nothing)) <: Exception
true
```

```
julia> typeof(DomainError) <: Exception
false
```

또한 일부 예외 유형은 오류 보고에 사용되는 하나 이상의 인수를 필요로 합니다.

```
julia> throw(UndefVarError(:x))
ERROR: UndefVarError: x not defined
```

이 메커니즘은 `UndefVarError`가 쓰여지는 방식에 따라 사용자 정의 예외 유형에 의해 쉽게 구현될 수 있습니다.

```
julia> struct MyUndefVarError <: Exception
           var::Symbol
       end
```

```
julia> Base.showerror(io::IO, e::MyUndefVarError) = print(io,
   → e.var, " not defined")
```

!!! 주의 오류 메시지를 작성할 때 첫 번째 단어를 소문자로 만드는 것이 좋습니다. 예를 들어, `size(A) == size(B) || throw(DimensionMismatch("size of A not equal to size of B"))`

```
`size(A) == size(B) || throw(DimensionMismatch("Size of A not
equal to size of B"))`.
```

```
,      . `size(A,1) == size(B,2) || throw(  
DimensionMismatch("A has first dimension..."))`.
```

## 오류

`error` 함수는 정상적인 제어 흐름을 방해하는 `ErrorException`을 생성하는 데 사용됩니다.

음수의 제곱근을 취하면 즉시 실행을 멈추고 싶다고 합시다. 이것을 하기 위해 인수가 음수이면 오류가 발생하는 `sqrt` 함수의 까다로운 버전을 정의할 수 있습니다.

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not  
→ allowed")  
fussy_sqrt (generic function with 1 method)  
  
julia> fussy_sqrt(2)  
1.4142135623730951  
  
julia> fussy_sqrt(-1)  
ERROR: negative x not allowed  
Stacktrace:  
[1] fussy_sqrt(::Int64) at ./none:1
```

`fussy_sqrt`가 호출 함수의 실행을 계속하려 하는 것이 아니라 다른 함수에서 음수 값으로 호출되면, 즉시 반환되어 대화식 세션에 오류 메시지를 표시합니다.

```
julia> function verbose_fussy_sqrt(x)  
  
    println("before fussy_sqrt")  
  
    r = fussy_sqrt(x)  
  
    println("after fussy_sqrt")
```

```

        return r

    end
verbose_fussy_sqrt (generic function with 1 method)

julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
Stacktrace:
 [1] fussy_sqrt at ./none:1 [inlined]
 [2] verbose_fussy_sqrt(::Int64) at ./none:3

```

## 경고 및 정보 메시지

또한 Julia는 표준 오류 입출력에 메시지를 쓰지만 `Exception`을 `throw`하지도 않고, 때문에 실행을 중단하지도 않는 다른 함수를 제공합니다.

```

julia> info("Hi"); 1+1
INFO: Hi
2

julia> warn("Hi"); 1+1
WARNING: Hi
2

julia> error("Hi"); 1+1

```

```
ERROR: Hi
```

Stacktrace:

```
[1] error(::String) at ./error.jl:33
```

## try/catch문

try/catch문은 Exception을 테스트할 수 있습니다. 예를 들어, 사용자 정의 제곱근 함수를 Exception을 사용하여 필요에 따라 실수 또는 복소수 제곱근 방법을 자동으로 호출하도록 작성할 수 있습니다.

```
julia> f(x) = try
           sqrt(x)
       catch
           sqrt(complex(x, 0))
       end
f (generic function with 1 method)

julia> f(1)
1.0

julia> f(-1)
0.0 + 1.0im
```

이 함수를 계산하는 실제 코드에서는 예외를 잡는 대신 x와 0을 비교한다는 점에 유의해야 합니다. 단순히 비교하고 분기하는 것보다 예외는 훨씬 느립니다.

또한 try/catch문은 Exception이 변수에 저장되도록 합니다. 이 고안된 예제에서, 다음 예제는 x가 색인 가능한 경우 x의 두 번째 요소의 제곱근을 계산하고, 그렇지 않으면 x가 실수임을 가정하고 제곱근을 반환합니다.

|154 julia> sqrt\_second(x) = try

## CHAPTER 12. CONTROL FLOW

```
    sqrt(x[2])  
  
    catch y  
  
        if isa(y, DomainError)  
  
            sqrt(complex(x[2], 0))  
  
        elseif isa(y, BoundsError)  
  
            sqrt(x)  
  
        end
```

```
    end  
sqrt_second (generic function with 1 method)
```

julia> sqrt\_second([1 4])

2.0

julia> sqrt\_second([1 -4])

0.0 + 2.0im

julia> sqrt\_second(9)

3.0

julia> sqrt\_second(-9)

ERROR: DomainError with -9.0:

| 12.5 예외 처리  
| `sqrt` will only return a complex result if called with a complex<sup>155</sup>  
| ↵ argument. Try `sqrt(Complex(x))`.  
| Stacktrace:  
| [...]

`catch` 다음의 기호는 항상 예외 이름으로 해석될 것이고, 때문에 한 줄로 `try/catch`문을 작성할 때 주의해야 합니다. 다음 코드는 오류가 발생하더라도 `x`의 값을 반환하지 않습니다.

| **try** bad() **catch** x **end**

대신 세미콜론을 사용하거나 `catch` 다음에 개행 문자를 삽입하십시오.

| **try** bad() **catch**; x **end**  
  
| **try** bad()  
| **catch**  
|     x  
| **end**

`catch`절은 꼭 필요한 것은 아닙니다. 생략한다면 기본 반환값은 `nothing`입니다.

| **julia>** **try** error() **end** # Returns nothing

`try/catch`문의 장점은 호출 함수의 스택에서 훨씬 더 높은 레벨로 깊게 중첩된 계산을 즉시 풀 수 있는 능력에 있습니다. 오류가 발생하지 않은 상황이 있지만 스택을 풀어 더 높은 레벨로 값을 전달하는 것이 바람직합니다. Julia는 고급 오류 처리를 위해 `rethrow`, `backtrace`, `catch_backtrace`와 같은 함수들을 제공합니다.

## **finally**문

상태 변경을 수행하거나 파일과 같은 리소스를 사용하는 코드에서는 일반적으로 코드가 끝났을 때 수행해야 하는 정리 작업(예: 파일 닫기)이 있습니다. 예외는 이 작업을 어쩌면 복잡하게 만들 수 있습니다. 왜냐하면 코드 블록이 정상적으로 끝나기 전에 종료될 수 있기

때문입니다. `finally` 키워드는 주어진 코드 블록의 종료가 정상으로 무릎이 떠도 않고 특정 코드를 실행하게 해줍니다.

예를 들면, 열린 파일을 확실히 닫을 수 있는 방법은 다음과 같습니다.

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

제어가 `try` 블록을 떠날 때(`return` 때문에 끝나든, 정상적으로 끝나든) `close(f)`가 실행됩니다. 만약 여기서 `try` 블록이 예외로 인해 종료되면 예외는 계속 증식할 것입니다. `catch` 블록은 `try` 및 `finally`와 결합할 수 있으므로, 이 상황에서는 `catch`가 오류를 처리한 후에 `finally`문이 실행되면 좋을 것입니다.

## 12.6 태스크(일명 코루틴)

태스크는 유연한 방식으로 계산을 일시 중단하고 다시 시작할 수 있게 해주는 제어 흐름 기능입니다. 이 기능은 다른 프로그래밍 언어에서는 대칭 코루틴, 경량 스레드, 협업 멀티태스킹 또는 원샷 컨티뉴에이션과 같은 다른 이름으로 불립니다.

컴퓨팅 작업(실제로는 특정 기능 실행)이 `Task`로 지정되면, 다른 `Task`로 전환하여 그 태스크를 중단할 수 있습니다. 원래의 `Task`는 나중에 다시 시작될 수 있으며, 중단된 그 시점에서 바로 시작됩니다. 처음에는 함수 호출과 비슷하게 보일 수 있지만, 두 가지 중요한 차이점이 있습니다. 첫째, 태스크 전환은 공간을 사용하지 않아 호출 스택을 사용하지 않고도 얼마든지 태스크 전환이 발생할 수 있습니다. 둘째, 함수 호출과는 달리 태스크간 전환은 임의의 순서로 발생할 수 있습니다. 함수 호출은 호출된 함수가 제어가 호출 함수로 돌아가기 전에 실행을 완료해야 하는 구조입니다.

이러한 종류의 제어 흐름은 특정 문제를 훨씬 쉽게 해결할 수 있습니다. 일부 문제에서 필요한 작업의 다양한 부분은 함수 호출에 의해 자연스럽게 관련되지 않습니다. 수행해야 할 작업

증거6. 명확한 코드를 만들 수 있다면 “수신자”가 없기 때문입니다. 한 예로 복잡한 프로시저가 값을 생성하고, 다른 복잡한 프로시저가 값을 소비하는 생산자-소비자 문제가 있습니다. 소비자는 단순히 값을 얻기 위해 생산자 함수를 호출할 수 없습니다. 왜냐하면 생산자가 생성할 값이 더 많아 반환할 준비가 되지 않았기 때문입니다. 태스크를 통해 생산자와 소비자는 필요한 만큼 오래 실행하고 필요한 만큼 값을 주고 받을 수 있습니다.

Julia는 이 문제를 해결하기 위한 Channel 메커니즘을 제공합니다. Channel은 여러 태스크를 읽고 쓸 수 있는 대기 가능한 선입선출(FIFO) 대기열(queue)입니다.

`put!` 호출을 통해 값을 생성하는 생산자 태스크를 정의해 봅시다. 값을 소비하려면 생산자가 새 태스크를 실행하도록 예약해야 합니다. 인수가 하나인 함수를 인수로 받아들이는 특별한 Channel 생산자는 채널에 둑여진 작업을 실행하는 데 사용할 수 있습니다. 그런 다음 채널 객체에서 반복적으로 값을 `take!`를 통해 가져올 수 있습니다.

```
julia> function producer(c::Channel)

    put!(c, "start")

    for n=1:4

        put!(c, 2n)

    end

    put!(c, "stop")

end;

julia> chnl = Channel(producer);

julia> take!(chnl)
"start"
```

```
julia> take!(chnl)
```

```
2
```

```
julia> take!(chnl)
```

```
4
```

```
julia> take!(chnl)
```

```
6
```

```
julia> take!(chnl)
```

```
8
```

```
julia> take!(chnl)
```

```
"stop"
```

이 동작을 생각하는 한 가지 방법은 `producer`가 여러 번 반환이 가능하다는 것입니다. `put!` 호출 사이에 생성자의 실행이 일시 중단되고 소비자가 제어권을 가집니다.

반환된 `Channel`은 `for` 루프에서 반복용 객체로 사용될 수 있습니다. 이때 루프 변수는 생성된 모든 값을 취합니다. 채널이 닫히면 루프도 종료됩니다.

```
julia> for x in Channel(producer)
```

```
    println(x)
```

```
end
```

```
start
```

```
2
```

```
4
```

```
6
```

```
8
```

```
stop
```

생산자 측에서 채널을 고정하여 유포자 측으로 닫을 필요는 없었음을 알아두십시오. 이는 채널을 태스크에 묶는 동작이 채널의 수명과 묶인 태스크의 수명을 연결짓기 때문입니다. 채널 객체는 태스크가 종료되면 자동으로 닫힙니다. 여러 채널을 하나의 태스크에 묶을 수 있고, 그 반대로도 가능합니다.

태스크 생성자가 인수가 없는 함수를 예상하는 동안, 태스크에 묶인 채널을 만드는 채널 메소드는 채널 유형의 단일 인수를 협용하는 함수를 필요로 합니다. 공통 패턴은 생산자가 매개 변수화된 경우이며, 이 경우 부분 함수 응용 프로그램은 인수가 없거나 1개의 인수를 갖는 익명 함수를 작성하는 데 필요합니다.

태스크 객체의 경우 직접 또는 편리한 매크로를 사용하여 수행할 수 있습니다.

```
function mytask(myarg)
    ...
end

taskHdl = Task(() -> mytask(7))
#
taskHdl = @task mytask(7)
```

고급 작업 배분 패턴을 조율하기 위해, 태스크 및 채널 생성자와 바인드 및 스케줄을 사용하여 일련의 채널을 생산자/소비자 태스크 집합과 명시적으로 연결할 수 있습니다.

현재 Julia의 태스크 기능은 별도의 CPU 코어를 사용하도록 스케줄되어 있지 않습니다. 진짜 귀臬 스레드는 Parallel Computing 주제에서 논의하겠습니다.

## 코어 태스크 연산

작업 중 태스크가 어떻게 전환되는지 이해하기 위해 `yieldto` 저수준 구조를 탐험해 봅시다. `yieldto(task, value)`는 현재 태스크를 잠시 중단하고, 지정된 태스크로 전환하며, 태스크가 특정한 값을 반환하도록 하는 `yieldto` 호출을 하도록 합니다. 태스크 방식 제어 허름을 사용하려면 `yieldto`만이 유일한 해법임을 명심하십시오. 호출하고 반환하는 동작 대신 항상 다른 태스크로 전환할 뿐입니다. 이것이 이 기능이 "대칭 코루틴"이라고 불리는 이유입니다. 각 태스크는 동일한 메커니즘을 통해 다른 태스크로 전환하거나 전환됩니다.

`yieldto`는 강력하지만, 하지만 대부분의 태스크가 그 이후에 작업을 더 하거나 다른 태스크로 전환하는 경우 그런지 한번 볼까요. 현재 태스크를 전환한다면 아마 특정 시점에서 다시 전환하고 싶을 겁니다. 하지만 언제 전환을 해야 하는지, 무슨 태스크를 전환해야 할지를 파악하는 데에 상당한 조율이 필요할 것입니다. 예를 들어, `put!`과 `take!`는 채널 컨텍스트에서 사용될 때 소비자가 누구인지를 기억하기 위해 상태를 유지하는 차단 작업입니다. 소비 작업을 수동으로 추적할 필요가 없으므로 `put!`을 저수준인 `yieldto`보다 쉽게 사용할 수 있습니다.

`yieldto` 외에, 태스크를 효과적으로 사용하기 위해서는 몇 가지 기본적인 함수가 더 필요합니다.

`current_task`는 현재 실행중인 태스크를 참조합니다.

`istaskdone`는 태스크가 종료했는지 여부를 묻습니다.

`istaskstarted`는 태스크가 실행 중인지 여부를 묻습니다.

`task_local_storage` 현재 태스크와 관련된 키값 저장소를 조작합니다.

## 태스크와 이벤트

대부분의 태스크 전환은 입출력 요청과 같은 이벤트를 기다린 결과로 발생하며, 이는 표준 라이브러리에 포함된 스케줄러에 의해 수행됩니다. 스케줄러는 실행 가능한 작업 대기열을 관리하고 메시지 도착과 같은 외부 이벤트를 기반으로 작업을 다시 시작하는 이벤트 루프를 실행합니다.

이벤트를 기다리는 기본적인 함수로는 `wait`가 있습니다. 여러 객체는 `wait`를 구현할 수 있는데, 그 예로 `Process` 객체가 주어진다면, `wait`는 그 객체가 종료될 때까지 기다릴 것입니다. `wait`는 종종 명시적이지 않습니다. 예를 들자면, `wait`는 데이터를 사용할 수 있을 때까지 기다리기 위해 `read` 호출 내부에서 발생할 수 있습니다.

이 모든 경우에, `wait`는 태스크를 대기열에 넣고 재시작하는 `Condition` 객체에서 궁극적으로 작동합니다. 태스크가 `Condition`에서 `wait`를 호출하면, 태스크는 실행 불가능한 것으로 표시되고, 조건 대기열에 추가되며 스케줄러로 전환됩니다. 스케줄러는 실행할 다른 태스크를 선택하거나 외부 이벤트 대기를 차단합니다. 모든 것이 잘되면 결국 이벤트 처리기가 조건에서 `notify`를 호출하여 해당 조건을 기다리는 작업을 다시 실행 가능하게 만듭니다.

~~파이썬을 호출할 때 명시적으로~~ 생성된 태스크는 처음에는 스케줄러에게 알려지지 않습니다. 원한다면 `yieldto`를 사용하여 수동으로 작업을 관리할 수도 있습니다. 하지만 그런 태스크가 이벤트를 기다리면 예상대로 이벤트가 발생할 때 자동적으로 재시작됩니다. 이벤트를 기다리지 않고 언제든지 스케줄러가 작업을 실행할 수 있게 하는 것도 가능합니다. 이 방법은 `schedule`을 호출하거나, `schedule` 또는 `@async` 매크로(자세한 사항은 Parallel Computing 참조)를 사용하여 수행할 수 있습니다.

## 태스크 상태

태스크에는 실행 상태를 설명하는 `state` 필드가 있습니다. 태스크의 모든 `state`는 다음과 같습니다.

상태	의미
<code>:runnable</code>	현재 실행 중이거나 전환할 수 있는 상태
<code>:waiting</code>	특정 이벤트를 기다리고 있어 블록된 상태
<code>:queued</code>	스케줄러의 실행 대기열에 있어 곧 다시 시작될 상태
<code>:done</code>	실행을 성공적으로 완료한 상태
<code>:failed</code>	알 수 없는 예외로 종료된 상태



# Chapter 13

## Scope of Variables

The scope of a variable is the region of code within which a variable is visible. Variable scoping helps avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called `x` without the two `x`'s referring to the same thing. Similarly there are many other cases where different blocks of code can use the same name without referring to the same thing. The rules for when the same variable name does or doesn't refer to the same thing are called scope rules; this section spells them out in detail.

Certain constructs in the language introduce scope blocks, which are regions of code that are eligible to be the scope of some set of variables. The scope of a variable cannot be an arbitrary set of source lines; instead, it will always line up with one of these blocks. There are two main types of scopes in Julia, global scope and local scope, the latter can be nested. The constructs introducing scope blocks are:



# Chapter 14

Scope blocks that may nest only in other global scope blocks:

- global scope
  - \* module, baremodule
  - \* at interactive prompt (REPL)
- local scope (don't allow nesting)
  - \* type, immutable, macro

Scope blocks which may nest anywhere (in global or local scope):

- local scope
  - \* for, while, try-catch-finally, let
  - \* functions (either syntax, anonymous & do-blocks)
  - \* comprehensions, broadcast-fusing

Notably missing from this table are [begin blocks](#) and [if blocks](#) which do not introduce new scope blocks. Both types of scopes follow somewhat different rules which will be explained below.

Julia uses [lexical scoping](#), meaning that a function's scope does not inherit from its caller's scope, but from the scope in which the function was defined. For example, in the following code the `x` inside `foo` refers to the `x` in the global scope of its module `Bar`:

```
julia> module Bar  
  
      x = 1  
  
      foo() = x  
  
end;
```

and not a `x` in the scope where `foo` is used:

```
julia> import .Bar  
  
julia> x = -1;  
  
julia> Bar.foo()  
1
```

Thus lexical scope means that the scope of variables can be inferred from the source code alone.

## 14.1 Global Scope

Each module introduces a new global scope, separate from the global scope of all other modules; there is no all-encompassing global scope. Modules can introduce variables of other modules into their scope through the [using](#) or [import](#) statements or through qualified access using the dot-notation, i.e. each module is a so-called namespace. Note that variable bindings can only be changed within their global scope and not from an outside module.

```
julia> module A  
  
      a = 1 # a global in A's scope
```

```
    end;

julia> module B

    module C

        c = 2

    end

    b = C.c      # can access the namespace of a nested
→   global scope

            # through a qualified access

    import ..A # makes module A available

    d = A.a

end;

julia> module D

    b = a # errors as D's global scope is separate from A's
end;

ERROR: UndefVarError: a not defined

julia> module E
```

```
import ..A # make module A available
```

```
A.a = 2      # throws below error
```

```
end;
```

```
ERROR: cannot assign variables in other modules
```

Note that the interactive prompt (aka REPL) is in the global scope of the module **Main**.

## 14.2 Local Scope

A new local scope is introduced by most code blocks (see above [table](#) for a complete list). A local scope inherits all the variables from a parent local scope, both for reading and writing. Additionally, the local scope inherits all globals that are assigned to in its parent global scope block (if it is surrounded by a global `if` or `begin` scope). Unlike global scopes, local scopes are not namespaces, thus variables in an inner scope cannot be retrieved from the parent scope through some sort of qualified access.

The following rules and examples pertain to local scopes. A newly introduced variable in a local scope does not back-propagate to its parent scope. For example, here the `z` is not introduced into the top-level scope:

```
julia> for i = 1:10
```

```
    z = i
```

```
end
```

```
julia> z
```

```
ERROR: UndefVarError: z not defined
```

(Note, in LOCAL SCOPES following examples it is assumed that their top-level is in global scope with a clean workspace, for instance a newly started REPL.)

Inside a local scope a variable can be forced to be a new local variable using the `local` keyword:

```
julia> x = 0;

julia> for i = 1:10

    local x # this is also the default

    x = i + 1

end

julia> x
0
```

Inside a local scope a global variable can be assigned to by using the keyword `global`:

```
julia> for i = 1:10

    global z

    z = i

end

julia> z
10
```

The location of both the `local` and `global` keywords within the `CHAPTER14` block is irrelevant. The following is equivalent to the last example (although stylistically worse):

```
julia> for i = 1:10  
      z = i  
      global z  
    end  
  
julia> z  
10
```

The `local` and `global` keywords can also be applied to destructuring assignments, e.g. `local x, y = 1, 2`. In this case the keyword affects all listed variables.

Local scopes are introduced by most block keywords, with notable exceptions of `begin` and `if`.

In a local scope, all variables are inherited from its parent global scope block unless:

an assignment would result in a modified global variable, or

a variable is specifically marked with the keyword `local`.

Thus global variables are only inherited for reading but not for writing:

```
julia> x, y = 1, 2;  
  
julia> function foo()
```

```
x = 2          # assignment introduces a new local  
  
        return x + y # y refers to the global  
  
end;  
  
julia> foo()  
4  
  
julia> x  
1
```

An explicit `global` is needed to assign to a global variable:

### Avoiding globals

Avoiding changing the value of global variables is considered by many to be a programming best-practice. One reason for this is that remotely changing the state of global variables in other modules should be done with care as it makes the local behavior of the program hard to reason about. This is why the scope blocks that introduce local scope require the `global` keyword to declare the intent to modify a global variable.

```
julia> x = 1;  
  
julia> function foobar()  
  
        global x = 2  
  
    end;
```

```
julia> foobar();
```

```
julia> x
```

```
2
```

Note that nested functions can modify their parent scope's local variables:

```
julia> x, y = 1, 2;
```

```
julia> function baz()
```

```
    x = 2 # introduces a new local
```

```
    function bar()
```

```
        x = 10          # modifies the parent's x
```

```
        return x + y # y is global
```

```
    end
```

```
    return bar() + x # 12 + 10 (x is modified in call of
```

```
→ bar())
```

```
end;
```

```
julia> baz()
```

```
22
```

```
julia> x, y # verify that global x and y are unchanged
```

```
(1, 2)
```

The reason to allow modifying local variables of parent scopes in nested functions is to allow constructing **closures** which have a private state, for instance the *state* variable in the following example:

```
julia> let state = 0
           global counter() = (state += 1)
       end;

julia> counter()
1

julia> counter()
2
```

See also the closures in the examples in the next two sections.

The distinction between inheriting global scope and nesting local scope can lead to some slight differences between functions defined in local vs. global scopes for variable assignments. Consider the modification of the last example by moving **bar** to the global scope:

```
julia> x, y = 1, 2;

julia> function bar()
           x = 10 # local, no longer a closure variable
           return x + y
       end;

julia> function quz()
```

```
x = 2 # local
```

```
    return bar() + x # 12 + 2 (x is not modified)
```

```
end;
```

```
julia> quz()
```

```
14
```

```
julia> x, y # verify that global x and y are unchanged
```

```
(1, 2)
```

Note that the above nesting rules do not pertain to type and macro definitions as they can only appear at the global scope. There are special scoping rules concerning the evaluation of default and keyword function arguments which are described in the [Function section](#).

An assignment introducing a variable used inside a function, type or macro definition need not come before its inner usage:

```
julia> f = y -> y + a;
```

```
julia> f(3)
```

```
ERROR: UndefVarError: a not defined
```

```
Stacktrace:
```

```
[...]
```

```
julia> a = 1
```

```
1
```

```
julia> f(3)
```

```
4
```

This behavior seems slightly odd for a normal variable, but allows for named functions – which are just normal variables holding function objects – to be used before they are defined. This allows functions to be defined in whatever order is intuitive and convenient, rather than forcing bottom up ordering or requiring forward declarations, as long as they are defined by the time they are actually called. As an example, here is an inefficient, mutually recursive way to test if positive integers are even or odd:

```
julia> even(n) = (n == 0) ? true : odd(n - 1);  
  
julia> odd(n) = (n == 0) ? false : even(n - 1);  
  
julia> even(3)  
false  
  
julia> odd(3)  
true
```

Julia provides built-in, efficient functions to test for oddness and evenness called `iseven` and `isodd` so the above definitions should only be considered to be examples of scope, not efficient design.

## Let Blocks

Unlike assignments to local variables, `let` statements allocate new variable bindings each time they run. An assignment modifies an existing value location, and `let` creates new locations. This difference is usually not important, and is only detectable in the case of variables that outlive their scope via closures. The `let` syntax accepts a comma-separated series of assignments and variable names:

```
|176 julia> x, y, z = -1, -1, -1;
```

CHAPTER 14.

```
julia> let x = 1, z
           println("x: $x, y: $y") # x is local variable, y the
→   global

           println("z: $z") # errors as z has not been assigned
→   yet but is local

       end
x: 1, y: -1
ERROR: UndefVarError: z not defined
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x` since the two `x` variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
julia> Fs = Array{Any}(2); i = 1;

julia> while i <= 2
           Fs[i] = ()->i
           i += 1

       end

julia> Fs[1]()
```

```
julia> Fs[2]()
3
```

Here we create and store two closures that return variable `i`. However, it is always the same variable `i`, so the two closures behave identically. We can use `let` to create a new binding for `i`:

```
julia> Fs = Array{Any}(2); i = 1;

julia> while i <= 2

        let i = i

            Fs[i] = ()->i

        end

        i += 1

    end

julia> Fs[1]()
1

julia> Fs[2]()
2
```

Since the `begin` construct does not introduce a new scope, it can be useful to use a zero-argument `let` to just introduce a new scope block without creating any new bindings:

|178  
julia> let

CHAPTER 14.

```
    local x = 1

    let

        local x = 2

    end

    x

end

1
```

Since `let` introduces a new scope block, the inner local `x` is a different variable than the outer local `x`.

## For Loops and Comprehensions

`for` loops, `while` loops, and [Comprehensions](#) have the following behavior: any new variables introduced in their body scopes are freshly allocated for each loop iteration, as if the loop body were surrounded by a `let` block:

```
julia> Fs = Array{Any}(2);

julia> for j = 1:2

    Fs[j] = ()->j

end
```

```
julia> Fs[1]()
```

```
julia> Fs[2]()
```

```
2
```

A `for` loop or comprehension iteration variable is always a new variable:

```
julia> function f()
    i = 0
    for i = 1:3
    end
    return i
end;
```

```
julia> f()
```

```
0
```

However, it is occasionally useful to reuse an existing variable as the iteration variable. This can be done conveniently by adding the keyword `outer`:

```
julia> function f()
    i = 0
    for outer i = 1:3
    end
    return i
end;
```

```
|180 Julia> f()
```

```
| 3
```

CHAPTER 14.

## 14.3 Constants

A common use of variables is giving names to specific, unchanging values. Such variables are only assigned once. This intent can be conveyed to the compiler using the `const` keyword:

```
Julia> const e = 2.71828182845904523536;
```

```
Julia> const pi = 3.14159265358979323846;
```

Multiple variables can be declared in a single `const` statement:

```
Julia> const a, b = 1, 2
```

```
(1, 2)
```

The `const` declaration should only be used in global scope on globals. It is difficult for the compiler to optimize code involving global variables, since their values (or even their types) might change at almost any time. If a global variable will not change, adding a `const` declaration solves this performance problem.

Local constants are quite different. The compiler is able to determine automatically when a local variable is constant, so local constant declarations are not necessary, and in fact are currently not supported.

Special top-level assignments, such as those performed by the `function` and `struct` keywords, are constant by default.

Note that `const` only affects the variable binding; the variable may be bound to a mutable object (such as an array), and that object may still be modified.

# Chapter 15

## Types

Type systems have traditionally fallen into two quite different camps: static type systems, where every program expression must have a type computable before the execution of the program, and dynamic type systems, where nothing is known about types until run time, when the actual values manipulated by the program are available. Object orientation allows some flexibility in statically typed languages by letting code be written without the precise types of values being known at compile time. The ability to write code that can operate on different types is called polymorphism. All code in classic dynamically typed languages is polymorphic: only by explicitly checking types, or when objects fail to support operations at run-time, are the types of any values ever restricted.

Julia's type system is dynamic, but gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types. This can be of great assistance in generating efficient code, but even more significantly, it allows method dispatch on the types of function arguments to be deeply integrated with the language. Method dispatch is explored in detail in [Methods](#), but is rooted in the type system presented here.

The default behavior in Julia when types are omitted is to allow values to be of any type. Thus, one can write many useful Julia programs without ever explicitly

CHAPTER 15. TYPES

Using types. When additional expressiveness is needed, however, we gradually introduce explicit type annotations into previously "untyped" code. Doing so will typically increase both the performance and robustness of these systems, and perhaps somewhat counterintuitively, often significantly simplify them.

Describing Julia in the lingo of [type systems](#), it is: dynamic, nominative and parametric. Generic types can be parameterized, and the hierarchical relationships between types are [explicitly declared](#), rather than [implied by compatible structure](#). One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes. While this might at first seem unduly restrictive, it has many beneficial consequences with surprisingly few drawbacks. It turns out that being able to inherit behavior is much more important than being able to inherit structure, and inheriting both causes significant difficulties in traditional object-oriented languages. Other high-level aspects of Julia's type system that should be mentioned up front are:

There is no division between object and non-object values: all values in Julia are true objects having a type that belongs to a single, fully connected type graph, all nodes of which are equally first-class as types.

There is no meaningful concept of a "compile-time type": the only type a value has is its actual type when the program is running. This is called a "run-time type" in object-oriented languages where the combination of static compilation with polymorphism makes this distinction significant.

Only values, not variables, have types – variables are simply names bound to values.

Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols, by values of any type for

are stored like C types or structs with no pointers to other objects), and also by tuples thereof. Type parameters may be omitted when they do not need to be referenced or restricted.

Julia's type system is designed to be powerful and expressive, yet clear, intuitive and unobtrusive. Many Julia programmers may never feel the need to write code that explicitly uses types. Some kinds of programming, however, become clearer, simpler, faster and more robust with declared types.

## 15.1 Type Declarations

The `::` operator can be used to attach type annotations to expressions and variables in programs. There are two primary reasons to do this:

1. As an assertion to help confirm that your program works the way you expect,
2. To provide extra type information to the compiler, which can then improve performance in some cases

When appended to an expression computing a value, the `::` operator is read as "is an instance of". It can be used anywhere to assert that the value of the expression on the left is an instance of the type on the right. When the type on the right is concrete, the value on the left must have that type as its implementation – recall that all concrete types are final, so no implementation is a subtype of any other. When the type is abstract, it suffices for the value to be implemented by a concrete type that is a subtype of the abstract type. If the type assertion is not true, an exception is thrown, otherwise, the left-hand value is returned:

```
|184 julia> (1+2)::AbstractFloat
```

## CHAPTER 15. TYPES

```
ERROR: TypeError: in typeassert, expected AbstractFloat, got Int64
```

```
julia> (1+2)::Int
```

```
3
```

This allows a type assertion to be attached to any expression in-place.

When appended to a variable on the left-hand side of an assignment, or as part of a `local` declaration, the `::` operator means something a bit different: it declares the variable to always have the specified type, like a type declaration in a statically-typed language such as C. Every value assigned to the variable will be converted to the declared type using `convert`:

```
julia> function foo()

    x::Int8 = 100

    x

end

foo (generic function with 1 method)
```

```
julia> foo()
```

```
100
```

```
julia> typeof(ans)
```

```
Int8
```

This feature is useful for avoiding performance "gotchas" that could occur if one of the assignments to a variable changed its type unexpectedly.

This "declaration" behavior only occurs in specific contexts:

```
| 15.2 ABSTRACT TYPES
```

```
| local x::Int8 # in a local declaration
```

185

```
| x::Int8 = 10 # as the left-hand side of an assignment
```

and applies to the whole current scope, even before the declaration. Currently, type declarations cannot be used in global scope, e.g. in the REPL, since Julia does not yet have constant-type globals.

Declarations can also be attached to function definitions:

```
function sinc(x)::Float64
    if x == 0
        return 1
    end
    return sin(pi*x)/(pi*x)
end
```

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to `Float64`.

## 15.2 Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

Recall that in [Integers and Floating-Point Numbers](#), we introduced a variety of concrete types of numeric values: `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `Float16`, `Float32`, and `Float64`. Although they have different representation sizes, `Int8`, `Int16`, `Int32`, `Int64` and `Int128` all have in common that they are signed integer

186 types. Likewise `UInt8`, `UInt16`, `UInt32`, `UInt64` and `CHAPTER15. TYPES` signed integer types, while `Float16`, `Float32` and `Float64` are distinct in being floating-point types rather than integers. It is common for a piece of code to make sense, for example, only if its arguments are some kind of integer, but not really depend on what particular kind of integer. For example, the greatest common denominator algorithm works for all kinds of integers, but will not work for floating-point numbers. Abstract types allow the construction of a hierarchy of types, providing a context into which concrete types can fit. This allows you, for example, to easily program to any type that is an integer, without restricting an algorithm to a specific type of integer.

Abstract types are declared using the `abstract` type keyword. The general syntaxes for declaring an abstract type are:

```
abstract type «name» end  
abstract type «name» <: «supertype» end
```

The `abstract` type keyword introduces a new abstract type, whose name is given by «name». This name can be optionally followed by `<:` and an already-existing type, indicating that the newly declared abstract type is a subtype of this "parent" type.

When no supertype is given, the default supertype is `Any` – a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, `Any` is commonly called "top" because it is at the apex of the type graph. Julia also has a predefined abstract "bottom" type, at the nadir of the type graph, which is written as `Union{}`. It is the exact opposite of `Any`: no object is an instance of `Union{}` and all types are supertypes of `Union{}`.

Let's consider some of the abstract types that make up Julia's numerical hierarchy:

```
abstract type Number end
```

```
abstract type Real    <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer <: Real end
abstract type Signed <: Integer end
abstract type Unsigned <: Integer end
```

The `Number` type is a direct child type of `Any`, and `Real` is its child. In turn, `Real` has two children (it has more, but only two are shown here; we'll get to the others later): `Integer` and `AbstractFloat`, separating the world into representations of integers and representations of real numbers. Representations of real numbers include, of course, floating-point types, but also include other types, such as rationals. Hence, `AbstractFloat` is a proper subtype of `Real`, including only floating-point representations of real numbers. Integers are further subdivided into `Signed` and `Unsigned` varieties.

The `<:` operator in general means "is a subtype of", and, used in declarations like this, declares the right-hand type to be an immediate supertype of the newly declared type. It can also be used in expressions as a subtype operator which returns `true` when its left operand is a subtype of its right operand:

```
julia> Integer <: Number
true

julia> Integer <: AbstractFloat
false
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
function myplus(x,y)
    x+y
end
```

The first thing to note is that the above argument declarations are equivalent to `x::Any` and `y::Any`. When this function is invoked, say as `myplus(2, 5)`, the dispatcher chooses the most specific method named `myplus` that matches the given arguments. (See [Methods](#) for more information on multiple dispatch.)

Assuming no method more specific than the above is found, Julia next internally defines and compiles a method called `myplus` specifically for two `Int` arguments based on the generic function given above, i.e., it implicitly defines and compiles:

```
function myplus(x::Int, y::Int)
    x+y
end
```

and finally, it invokes this specific method.

Thus, abstract types allow programmers to write generic functions that can later be used as the default method by many combinations of concrete types. Thanks to multiple dispatch, the programmer has full control over whether the default or more specific method is used.

An important point to note is that there is no loss in performance if the programmer relies on a function whose arguments are abstract types, because it is recompiled for each tuple of argument concrete types with which it is invoked. (There may be a performance issue, however, in the case of function arguments that are containers of abstract types; see [Performance Tips](#).)

## 15.3 Primitive Types

A primitive type is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Unlike most languages, Julia lets you declare your own primitive types, rather than

~~Providing primitive types~~ set of built-in ones. In fact, the standard primitive types<sup>189</sup> are all defined in the language itself:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8      <: Signed     8 end
primitive type UInt8    <: Unsigned   8 end
primitive type Int16     <: Signed     16 end
primitive type UInt16   <: Unsigned   16 end
primitive type Int32     <: Signed     32 end
primitive type UInt32   <: Unsigned   32 end
primitive type Int64     <: Signed     64 end
primitive type UInt64   <: Unsigned   64 end
primitive type Int128    <: Signed    128 end
primitive type UInt128   <: Unsigned 128 end
```

The general syntaxes for declaring a primitive type are:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

The number of bits indicates how much storage the type requires and the name gives the new type a name. A primitive type can optionally be declared to be a subtype of some supertype. If a supertype is omitted, then the type defaults to having `Any` as its immediate supertype. The declaration of `Bool` above therefore means that a boolean value takes eight bits to store, and has `Integer` as its immediate supertype. Currently, only sizes that are multiples

CHAPTER 15 TYPES  
698 bits are supported. Therefore, boolean values, although they really just a single bit, cannot be declared to be any smaller than eight bits.

The types `Bool`, `Int8` and `UInt8` all have identical representations: they are eight-bit chunks of memory. Since Julia's type system is nominative, however, they are not interchangeable despite having identical structure. A fundamental difference between them is that they have different supertypes: `Bool`'s direct supertype is `Integer`, `Int8`'s is `Signed`, and `UInt8`'s is `Unsigned`. All other differences between `Bool`, `Int8`, and `UInt8` are matters of behavior – the way functions are defined to act when given objects of these types as arguments. This is why a nominative type system is necessary: if structure determined type, which in turn dictates behavior, then it would be impossible to make `Bool` behave any differently than `Int8` or `UInt8`.

## 15.4 Composite Types

**Composite types** are called records, structs, or objects in various languages. A composite type is a collection of named fields, an instance of which can be treated as a single value. In many languages, composite types are the only kind of user-definable type, and they are by far the most commonly used user-defined type in Julia as well.

In mainstream object oriented languages, such as C++, Java, Python and Ruby, composite types also have named functions associated with them, and the combination is called an "object". In purer object-oriented languages, such as Ruby or Smalltalk, all values are objects whether they are composites or not. In less pure object oriented languages, including C++ and Java, some values, such as integers and floating-point values, are not objects, while instances of user-defined composite types are true objects with associated methods. In Julia, all values are objects, but functions are not bundled with the objects they operate on. This is necessary since Julia chooses which method of a

## FUNCTION COMPOSITE TYPES

These dispatch, meaning that the types of all of a function's arguments are considered when selecting a method, rather than just the first one (see [Methods](#) for more information on methods and dispatch). Thus, it would be inappropriate for functions to "belong" to only their first argument. Organizing methods into function objects rather than having named bags of methods "inside" each object ends up being a highly beneficial aspect of the language design.

Composite types are introduced with the `struct` keyword followed by a block of field names, optionally annotated with types using the `::` operator:

```
julia> struct Foo  
    bar  
    baz::Int  
    qux::Float64  
end
```

Fields with no type annotation default to `Any`, and can accordingly hold any type of value.

New objects of type `Foo` are created by applying the `Foo` type object like a function to values for its fields:

```
julia> foo = Foo("Hello, world.", 23, 1.5)  
Foo("Hello, world.", 23, 1.5)  
  
julia> typeof(foo)  
Foo
```

When a type is applied like a function it is called a constructor. Constructors are generated automatically (these are called default constructors). One accepts any arguments and calls `convert` to convert them to the types of the fields, and the other accepts arguments that match the field types exactly. The reason both of these are generated is that this makes it easier to add new definitions without inadvertently replacing a default constructor.

Since the `bar` field is unconstrained in type, any value will do. However, the value for `baz` must be convertible to `Int`:

```
julia> Foo((), 23.5, 1)
ERROR: InexactError: convert(Int64, 23.5)
Stacktrace:
 [1] convert at ./float.jl:703 [inlined]
 [2] Foo(::Tuple{}, ::Float64, ::Int64) at ./none:2
```

You may find a list of field names using the `fieldnames` function.

```
julia> fieldnames(Foo)
3-element Array{Symbol,1}:
 :bar
 :baz
 :qux
```

You can access the field values of a composite object using the traditional `foo.bar` notation:

```
julia> foo.bar
"Hello, world."
```

```
julia> foo.baz
```

1.5

Composite objects declared with `struct` are immutable; they cannot be modified after construction. This may seem odd at first, but it has several advantages:

It can be more efficient. Some structs can be packed efficiently into arrays, and in some cases the compiler is able to avoid allocating immutable objects entirely.

It is not possible to violate the invariants provided by the type's constructors.

Code using immutable objects can be easier to reason about.

An immutable object might contain mutable objects, such as arrays, as fields. Those contained objects will remain mutable; only the fields of the immutable object itself cannot be changed to point to different objects.

Where required, mutable composite objects can be declared with the keyword `mutable struct`, to be discussed in the next section.

Composite types with no fields are singletons; there can be only one instance of such types:

```
julia> struct NoFields  
  
        end  
  
julia> NoFields() === NoFields()  
true
```

`TYPE` function confirms that the "two" constructed instances are actually one and the same. Singleton types are described in further detail below.

There is much more to say about how instances of composite types are created, but that discussion depends on both [Parametric Types](#) and on [Methods](#), and is sufficiently important to be addressed in its own section: [Constructors](#).

## 15.5 Mutable Composite Types

If a composite type is declared with `mutable struct` instead of `struct`, then instances of it can be modified:

```
julia> mutable struct Bar  
        baz  
        qux::Float64  
  
    end  
  
julia> bar = Bar("Hello", 1.5);  
  
julia> bar.qux = 2.0  
2.0  
  
julia> bar.baz = 1//2  
1//2
```

In order to support mutation, such objects are generally allocated on the heap, and have stable memory addresses. A mutable object is like a little container that might hold different values over time, and so can only be reliably identified with its address. In contrast, an instance of an immutable type is associated

~~15.6. DECLARED TYPES~~ -- the field values alone tell you everything about the object. In deciding whether to make a type mutable, ask whether two instances with the same field values would be considered identical, or if they might need to change independently over time. If they would be considered identical, the type should probably be immutable.

To recap, two essential properties define immutability in Julia:

An object with an immutable type is passed around (both in assignment statements and in function calls) by copying, whereas a mutable type is passed around by reference.

It is not permitted to modify the fields of a composite immutable type.

It is instructive, particularly for readers whose background is C/C++, to consider why these two properties go hand in hand. If they were separated, i.e., if the fields of objects passed around by copying could be modified, then it would become more difficult to reason about certain instances of generic code. For example, suppose `x` is a function argument of an abstract type, and suppose that the function changes a field: `x.isprocessed = true`. Depending on whether `x` is passed by copying or by reference, this statement may or may not alter the actual argument in the calling routine. Julia sidesteps the possibility of creating functions with unknown effects in this scenario by forbidding modification of fields of objects passed around by copying.

## 15.6 Declared Types

The three kinds of types discussed in the previous three sections are actually all closely related. They share the same key properties:

They are explicitly declared.

They have names.

They may have parameters.

Because of these shared properties, these types are internally represented as instances of the same concept, `DataType`, which is the type of any of these types:

```
julia> typeof(Real)
```

```
DataType
```

```
julia> typeof(Int)
```

```
DataType
```

A `DataType` may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a primitive type is a `DataType` with nonzero size, but no field names. A composite type is a `DataType` that has field names or is empty (zero size).

Every concrete value in the system is an instance of some `DataType`.

## 15.7 Type Unions

A type union is a special abstract type which includes as objects all instances of any of its argument types, constructed using the special `Union` function:

```
julia> IntOrString = Union{Int,AbstractString}
```

```
Union{Int64, AbstractString}
```

```
julia> 1 :: IntOrString
```

```
1
```

```
julia> "Hello!" :: IntOrString
```

```
"Hello!"
```

```
julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64,
    ↵ AbstractString}, got Float64
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer.

## 15.8 Parametric Types

An important and powerful feature of Julia's type system is that it is parametric: types can take parameters, so that type declarations actually introduce a whole family of new types – one for each possible combination of parameter values. There are many languages that support some version of [generic programming](#), wherein data structures and algorithms to manipulate them may be specified without specifying the exact types involved. For example, some form of generic programming exists in ML, Haskell, Ada, Eiffel, C++, Java, C#, F#, and Scala, just to name a few. Some of these languages support true parametric polymorphism (e.g. ML, Haskell, Scala), while others support ad-hoc, template-based styles of generic programming (e.g. C++, Java). With so many different varieties of generic programming and parametric types in various languages, we won't even attempt to compare Julia's parametric types to other languages, but will instead focus on explaining Julia's system in its own right. We will note, however, that because Julia is a dynamically typed language and doesn't need to make all type decisions at compile time, many traditional difficulties encountered in static parametric type systems can be relatively easily handled.

All declared types (the `DataType` variety) can be parameterized, with the same syntax in each case. We will discuss them in the following order: first, parametric composite types, then parametric abstract types, and finally para-

## Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

```
julia> struct Point{T}

    x::T

    y::T

end
```

This declaration defines a new parametric type, `Point{T}`, holding two "coordinates" of type `T`. What, one may ask, is `T`? Well, that's precisely the point of parametric types: it can be any type at all (or a value of any bits type, actually, although here it's clearly used as a type). `Point{Float64}` is a concrete type equivalent to the type defined by replacing `T` in the definition of `Point` with `Float64`. Thus, this single declaration actually declares an unlimited number of types: `Point{Float64}`, `Point{AbstractString}`, `Point{Int64}`, etc. Each of these is now a usable concrete type:

```
julia> Point{Float64}
Point{Float64}

julia> Point{AbstractString}
Point{AbstractString}
```

The type `Point{Float64}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{AbstractString}` is a "point" whose "coordinates" are string objects (see [Strings](#)).

**P6.3. PARAMETRIC TYPES** type object, containing all instances `Point{Float64}`, `Point{AbstractString}`, etc. as subtypes:

```
julia> Point{Float64} <: Point
true

julia> Point{AbstractString} <: Point
true
```

Other types, of course, are not subtypes of it:

```
julia> Float64 <: Point
false

julia> AbstractString <: Point
false
```

Concrete `Point` types with different values of `T` are never subtypes of each other:

```
julia> Point{Float64} <: Point{Int64}
false

julia> Point{Float64} <: Point{Real}
false
```

### Warning

This last point is very important: even though `Float64 <: Real` we DO NOT have `Point{Float64} <: Point{Real}`.

In other words, in the parlance of type theory, Julia's type parameters are invariant, rather than being covariant (or even contravariant). This is for practical reasons: while any instance of `Point{Float64}` may conceptually be

an instance of `Point{Real}` as well, the two types have different representations in memory:

An instance of `Point{Float64}` can be represented compactly and efficiently as an immediate pair of 64-bit values;

An instance of `Point{Real}` must be able to hold any pair of instances of `Real`. Since objects that are instances of `Real` can be of arbitrary size and structure, in practice an instance of `Point{Real}` must be represented as a pair of pointers to individually allocated `Real` objects.

The efficiency gained by being able to store `Point{Float64}` objects with immediate values is magnified enormously in the case of arrays: an `Array{Float64}` can be stored as a contiguous memory block of 64-bit floating-point values, whereas an `Array{Real}` must be an array of pointers to individually allocated `Real` objects – which may well be `boxed` 64-bit floating-point values, but also might be arbitrarily large, complex objects, which are declared to be implementations of the `Real` abstract type.

Since `Point{Float64}` is not a subtype of `Point{Real}`, the following method can't be applied to arguments of type `Point{Float64}`:

```
function norm(p::Point{Real})
    sqrt(p.x^2 + p.y^2)
end
```

A correct way to define a method that accepts all arguments of type `Point{T}` where `T` is a subtype of `Real` is:

```
function norm(p::Point{<:Real})
    sqrt(p.x^2 + p.y^2)
end
```

(ESPRIVIDPARAMETRICTYPES) Define function `norm{T<:Real}(p::Point{T})` or function `norm(p::Point{T} where T<:Real)`; see [UnionAll Types](#).)

More examples will be discussed later in [Methods](#).

How does one construct a `Point` object? It is possible to define custom constructors for composite types, which will be discussed in detail in [Constructors](#), but in the absence of any special constructor declarations, there are two default ways of creating new composite objects, one in which the type parameters are explicitly given and the other in which they are implied by the arguments to the object constructor.

Since the type `Point{Float64}` is a concrete type equivalent to `Point` declared with `Float64` in place of `T`, it can be applied as a constructor accordingly:

```
julia> Point{Float64}(1.0, 2.0)
Point{Float64}(1.0, 2.0)

julia> typeof(ans)
Point{Float64}
```

For the default constructor, exactly one argument must be supplied for each field:

```
julia> Point{Float64}(1.0)
ERROR: MethodError: Cannot `convert` an object of type Float64 to
→ an object of type Point{Float64}
This may have arisen from a call to the constructor
→ Point{Float64}(...),
since type constructors fall back to convert methods.
Stacktrace:
 [1] Point{Float64}(::Float64) at ./sysimg.jl:114
```

```
julia> Point{Float64}(1.0,2.0,3.0)
ERROR: MethodError: no method matching Point{Float64}(:::Float64,
→ ::Float64, ::Float64)
```

Only one default constructor is generated for parametric types, since overriding it is not possible. This constructor accepts any arguments and converts them to the field types.

In many cases, it is redundant to provide the type of `Point` object one wants to construct, since the types of arguments to the constructor call already implicitly provide type information. For that reason, you can also apply `Point` itself as a constructor, provided that the implied value of the parameter type `T` is unambiguous:

```
julia> Point(1.0,2.0)
Point{Float64}(1.0, 2.0)
```

```
julia> typeof(ans)
Point{Float64}
```

```
julia> Point(1,2)
Point{Int64}(1, 2)
```

```
julia> typeof(ans)
Point{Int64}
```

In the case of `Point`, the type of `T` is unambiguously implied if and only if the two arguments to `Point` have the same type. When this isn't the case, the constructor will fail with a `MethodError`:

```
julia> Point(1,2.5)
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
```

```
Point(::T, !Matched::T) where T at none:2
```

Constructor methods to appropriately handle such mixed cases can be defined, but that will not be discussed until later on in [Constructors](#).

## Parametric Abstract Types

Parametric abstract type declarations declare a collection of abstract types, in much the same way:

```
julia> abstract type Pointy{T} end
```

With this declaration, `Pointy{T}` is a distinct abstract type for each type or integer value of `T`. As with parametric composite types, each such instance is a subtype of `Pointy`:

```
julia> Pointy{Int64} <: Pointy
true
```

```
julia> Pointy{1} <: Pointy
true
```

Parametric abstract types are invariant, much as parametric composite types are:

```
julia> Pointy{Float64} <: Pointy{Real}
false
```

```
julia> Pointy{Real} <: Pointy{Float64}
false
```

The notation `Pointy{<:Real}` can be used to express the analogue of a covariant type, while `Pointy{>:Int}` the analogue of a contravariant type, but technically these represent sets of types (see [UnionAll Types](#)).

```
julia> Pointy{Float64} <: Pointy{<:Real}
true
```

```
julia> Pointy{Real} <: Pointy{>:Int}
true
```

Much as plain old abstract types serve to create a useful hierarchy of types over concrete types, parametric abstract types serve the same purpose with respect to parametric composite types. We could, for example, have declared `Point{T}` to be a subtype of `Pointy{T}` as follows:

```
julia> struct Point{T} <: Pointy{T}

    x::T

    y::T

end
```

Given such a declaration, for each choice of `T`, we have `Point{T}` as a subtype of `Pointy{T}`:

```
julia> Point{Float64} <: Pointy{Float64}
true

julia> Point{Real} <: Pointy{Real}
true
```

```
|julia> Point{AbstractString} <: Pointy{AbstractString}
```

```
true
```

This relationship is also invariant:

```
|julia> Point{Float64} <: Pointy{Real}
```

```
false
```

```
|julia> Point{Float64} <: Pointy{<:Real}
```

```
true
```

What purpose do parametric abstract types like `Pointy` serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line  $x = y$ :

```
|julia> struct DiagPoint{T} <: Pointy{T}
```

```
    x::T
```

```
end
```

Now both `Point{Float64}` and `DiagPoint{Float64}` are implementations of the `Pointy{Float64}` abstraction, and similarly for every other possible choice of type `T`. This allows programming to a common interface shared by all `Pointy` objects, implemented for both `Point` and `DiagPoint`. This cannot be fully demonstrated, however, until we have introduced methods and dispatch in the next section, [Methods](#).

There are situations where it may not make sense for type parameters to range freely over all possible types. In such situations, one can constrain the range of `T` like so:

```
|julia> abstract type Pointy{T<:Real} end
```

such a declaration, it is acceptable to use any type **Real** in place of T, but not types that are not subtypes of Real:

```
julia> Pointy{Float64}
Pointy{Float64}

julia> Pointy{Real}
Pointy{Real}

julia> Pointy{AbstractString}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got
→ Type{AbstractString}

julia> Pointy{1}
ERROR: TypeError: in Pointy, in T, expected T<:Real, got Int64
```

Type parameters for parametric composite types can be restricted in the same manner:

```
struct Point{T<:Real} <: Pointy{T}
    x::T
    y::T
end
```

To give a real-world example of how all this parametric type machinery can be useful, here is the actual definition of Julia's **Rational** immutable type (except that we omit the constructor here for simplicity), representing an exact ratio of integers:

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
end
```

15.8 PARAMETRIC TYPES ratios of integer values, so the parameter type `2017` restricted to being a subtype of `Integer`, and a ratio of integers represents a value on the real number line, so any `Rational` is an instance of the `Real` abstraction.

## Tuple Types

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. For example, a 2-element tuple type resembles the following immutable type:

```
struct Tuple2{A,B}  
    a::A  
    b::B  
end
```

However, there are three key differences:

Tuple types may have any number of parameters.

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are.

Tuples do not have field names; fields are only accessed by index.

Tuple values are written with parentheses and commas. When a tuple is constructed, an appropriate tuple type is generated on demand:

```
julia> typeof((1, "foo", 2.5))  
Tuple{Int64, String, Float64}
```

See the implications of covariance:

CHAPTER 15. TYPES

```
julia> Tuple{Int,AbstractString} <: Tuple{Real,Any}
true

julia> Tuple{Int,AbstractString} <: Tuple{Real,Real}
false

julia> Tuple{Int,AbstractString} <: Tuple{Real,}
false
```

Intuitively, this corresponds to the type of a function's arguments being a subtype of the function's signature (when the signature matches).

## Vararg Tuple Types

The last parameter of a tuple type can be the special type `Vararg`, which denotes any number of trailing elements:

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}

julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

**Named Parameter Types** corresponds to zero or more elements of type `Vararg{T,N}`. Vararg tuple types are used to represent the arguments accepted by varargs methods (see [Varargs Functions](#)).

The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `NTuple{N,T}` is a convenient alias for `Tuple{Vararg{T,N}}`, i.e. a tuple type containing exactly `N` elements of type `T`.

## Named Tuple Types

Named tuples are instances of the `NamedTuple` type, which has two parameters: a tuple of symbols giving the field names, and a tuple type giving the field types.

```
julia> typeof((a=1, b="hello"))
NamedTuple{(:a, :b), Tuple{Int64, String}}
```

A `NamedTuple` type can be used as a constructor, accepting a single tuple argument. The constructed `NamedTuple` type can be either a concrete type, with both parameters specified, or a type that specifies only field names:

```
julia> NamedTuple{(:a, :b), Tuple{Float32, String}}((1, ""))
(a = 1.0f0, b = "")

julia> NamedTuple{(:a, :b)}((1, ""))
(a = 1, b = "")
```

If field types are specified, the arguments are converted. Otherwise the types of the arguments are used directly.

## Singleton Types

There is a special kind of abstract parametric type that must be mentioned here: singleton types. For each type, `T`, the "singleton type" `Type{T}` is an

Abstract type whose only instance is the object T. Since this is difficult to parse, let's look at some examples:

```
julia> isa(Float64, Type{Float64})
```

```
true
```

```
julia> isa(Real, Type{Float64})
```

```
false
```

```
julia> isa(Real, Type{Real})
```

```
true
```

```
julia> isa(Float64, Type{Real})
```

```
false
```

In other words, `isa(A, Type{B})` is true if and only if A and B are the same object and that object is a type. Without the parameter, `Type` is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

```
julia> isa(Type{Float64}, Type)
```

```
true
```

```
julia> isa(Float64, Type)
```

```
true
```

```
julia> isa(Real, Type)
```

```
true
```

Any object that is not a type is not an instance of `Type`:

```
julia> isa(1, Type)
```

```
false
```

```
julia> isa("foo", Type)  
false
```

Until we discuss [Parametric Methods](#) and [conversions](#), it is difficult to explain the utility of the singleton type construct, but in short, it allows one to specialize function behavior on specific type values. This is useful for writing methods (especially parametric ones) whose behavior depends on a type that is given as an explicit argument rather than implied by the type of one of its arguments.

A few popular languages have singleton types, including Haskell, Scala and Ruby. In general usage, the term "singleton type" refers to a type whose only instance is a single value. This meaning applies to Julia's singleton types, but with that caveat that only type objects have singleton types.

## Parametric Primitive Types

Primitive types can also be declared parametrically. For example, pointers are represented as primitive types which would be declared in Julia like this:

```
# 32-bit system:  
primitive type Ptr{T} 32 end  
  
# 64-bit system:  
primitive type Ptr{T} 64 end
```

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter  $T$  is not used in the definition of the type itself – it is just an abstract tag, essentially defining an entire family of types with identical structure, differentiated only by their type parameter. Thus, `Ptr{Float64}` and `Ptr{Int64}` are distinct types, even though they have identical representations. And of course, all specific pointer types are subtypes of the umbrella `Ptr` type:

```
212 julia> Ptr{Float64} <: Ptr  
true
```

## CHAPTER 15. TYPES

```
julia> Ptr{Int64} <: Ptr  
true
```

### 15.9 UnionAll Types

We have said that a parametric type like `Ptr` acts as a supertype of all its instances (`Ptr{Int64}` etc.). How does this work? `Ptr` itself cannot be a normal data type, since without knowing the type of the referenced data the type clearly cannot be used for memory operations. The answer is that `Ptr` (or other parametric types like `Array`) is a different kind of type called a `UnionAll` type. Such a type expresses the iterated union of types for all values of some parameter.

`UnionAll` types are usually written using the keyword `where`. For example `Ptr` could be more accurately written as `Ptr{T} where T`, meaning all values whose type is `Ptr{T}` for some value of `T`. In this context, the parameter `T` is also often called a "type variable" since it is like a variable that ranges over types. Each `where` introduces a single type variable, so these expressions are nested for types with multiple parameters, for example `Array{T,N} where N where T`.

The type application syntax `A{B,C}` requires `A` to be a `UnionAll` type, and first substitutes `B` for the outermost type variable in `A`. The result is expected to be another `UnionAll` type, into which `C` is then substituted. So `A{B,C}` is equivalent to `A{B}{C}`. This explains why it is possible to partially instantiate a type, as in `Array{Float64}`: the first parameter value has been fixed, but the second still ranges over all possible values. Using explicit `where` syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional

Type variables can be restricted with subtype relations. `Array{T} where T<:Integer` refers to all arrays whose element type is some kind of `Integer`. The syntax `Array{<:Integer}` is a convenient shorthand for `Array{T} where T<:Integer`. Type variables can have both lower and upper bounds. `Array{T} where Int<:T<:Number` refers to all arrays of `Numbers` that are able to contain `Ints` (since `T` must be at least as big as `Int`). The syntax `where T>:Int` also works to specify only the lower bound of a type variable, and `Array{>:Int}` is equivalent to `Array{T} where T>:Int`.

Since `where` expressions nest, type variable bounds can refer to outer type variables. For example `Tuple{T, Array{S}}` `where S<:AbstractArray{T}` `where T<:Real` refers to 2-tuples whose first element is some `Real`, and whose second element is an `Array` of any kind of array whose element type contains the type of the first tuple element.

The `where` keyword itself can be nested inside a more complex declaration. For example, consider the two types created by the following declarations:

```
julia> const T1 = Array{Array{T, 1}} where T, 1
Array{Array{T, 1}} where T, 1

julia> const T2 = Array{Array{T, 1}, 1} where T
Array{Array{T, 1}, 1} where T
```

Type `T1` defines a 1-dimensional array of 1-dimensional arrays; each of the inner arrays consists of objects of the same type, but this type may vary from one inner array to the next. On the other hand, type `T2` defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Note that `T2` is an abstract type, e.g., `Array{Array{Int, 1}, 1} <: T2`, whereas `T1` is a concrete type. As a consequence,

~~CHAPTER 14 TYPES~~ can be constructed with a zero-argument constructor ~~CHAPTER 15 TYPES~~ not.

There is a convenient syntax for naming such types, similar to the short form of function definition syntax:

```
| Vector{T} = Array{T, 1}
```

This is equivalent to `const Vector = Array{T, 1}` where `T`. Writing `Vector{Float64}` is equivalent to writing `Array{Float64, 1}`, and the umbrella type `Vector` has as instances all `Array` objects where the second parameter – the number of array dimensions – is 1, regardless of what the element type is. In languages where parametric types must always be specified in full, this is not especially helpful, but in Julia, this allows one to write just `Vector` for the abstract type including all one-dimensional dense arrays of any element type.

## 15.10 Type Aliases

Sometimes it is convenient to introduce a new name for an already expressible type. This can be done with a simple assignment statement. For example, `UInt` is aliased to either `UInt32` or `UInt64` as is appropriate for the size of pointers on the system:

```
# 32-bit system:  
julia> UInt  
UInt32  
  
# 64-bit system:  
julia> UInt  
UInt64
```

This is accomplished via the following code in `base/boot.jl`:

```
if Int == Int64
    const UInt = UInt64
else
    const UInt = UInt32
end
```

Of course, this depends on what `Int` is aliased to – but that is predefined to be the correct type – either `Int32` or `Int64`.

(Note that unlike `Int`, `Float` does not exist as a type alias for a specific sized `AbstractFloat`. Unlike with integer registers, the floating point register sizes are specified by the IEEE-754 standard. Whereas the size of `Int` reflects the size of a native pointer on that machine.)

## 15.11 Operations on Types

Since types in Julia are themselves objects, ordinary functions can operate on them. Some functions that are particularly useful for working with or exploring types have already been introduced, such as the `<:` operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The `isa` function tests if an object is of a given type and returns true or false:

```
julia> isa(1, Int)
true

julia> isa(1, AbstractFloat)
false
```

The `typeof` function, already used throughout the manual in examples, returns the type of its argument. Since, as noted above, types are objects, they also have types, and we can ask what their types are:

```
216 julia> typeof(Rational{Int})
```

## CHAPTER 15. TYPES

```
DataType
```

```
julia> typeof(Union{Real, Float64, Rational})
```

```
DataType
```

```
julia> typeof(Union{Real, String})
```

```
Union
```

What if we repeat the process? What is the type of a type of a type? As it happens, types are all composite values and thus all have a type of `DataType`:

```
julia> typeof(DataType)
```

```
DataType
```

```
julia> typeof(Union)
```

```
DataType
```

`DataType` is its own type.

Another operation that applies to some types is `supertype`, which reveals a type's supertype. Only declared types (`DataType`) have unambiguous supertypes:

```
julia> supertype(Float64)
```

```
AbstractFloat
```

```
julia> supertype(Number)
```

```
Any
```

```
julia> supertype(AbstractString)
```

```
Any
```

```
Any
```

If you apply `supertype` to other type objects (or non-type objects), a `MethodError` is raised:

```
julia> supertype(Union{Float64, Int64})  
ERROR: MethodError: no method matching  
    → supertype(::Type{Union{Float64, Int64}})  
Closest candidates are:  
    supertype(!Matched::DataType) at operators.jl:42  
    supertype(!Matched::UnionAll) at operators.jl:47
```

## 15.12 Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the `show` function. For example, suppose we define a type to represent complex numbers in polar form:

```
julia> struct Polar{T<:Real} <: Number  
  
    r::T  
  
    θ::T  
  
end  
  
julia> Polar(r::Real,θ::Real) = Polar(promote(r,θ)...)
```

```
Polar
```

Here, we've added a custom constructor function so that it can take arguments of different `Real` types and promote them to a common type (see [Constructors](#)

[2.18 Conversion and Promotion](#)). (Of course, we would have to define lots of other methods, too, to make it act like a [Number](#), e.g. `+`, `*`, `one`, `zero`, promotion rules and so on.) By default, instances of this type display rather simply, with information about the type name and the field values, as e.g. `Polar{Float64}(3.0, 4.0)`.

If we want it to display instead as `3.0 * exp(4.0im)`, we would define the following method to print the object to a given output object `io` (representing a file, terminal, buffer, etcetera; see [Networking and Streams](#)):

```
julia> Base.show(io::IO, z::Polar) = print(io, z.r, " * exp(",  
    ↪ z.θ, "im)")
```

More fine-grained control over display of `Polar` objects is possible. In particular, sometimes one wants both a verbose multi-line printing format, used for displaying a single object in the REPL and other interactive environments, and also a more compact single-line format used for `print` or for displaying the object as part of another object (e.g. in an array). Although by default the `show(io, z)` function is called in both cases, you can define a different multi-line format for displaying an object by overloading a three-argument form of `show` that takes the `text/plain` MIME type as its second argument (see [Multimedia I/O](#)), for example:

```
julia> Base.show(io::IO, ::MIME"text/plain", z::Polar{T}) where{T}  
    ↪ =  
  
        print(io, "Polar{$T} complex number:\n    ", z)
```

(Note that `print(..., z)` here will call the 2-argument `show(io, z)` method.) This results in:

```
julia> Polar(3, 4.0)  
Polar{Float64} complex number:
```

```
julia> [Polar(3, 4.0), Polar(4.0, 5.3)]  
2-element Array{Polar{Float64},1}:  
 3.0 * exp(4.0im)  
 4.0 * exp(5.3im)
```

where the single-line `show(io, z)` form is still used for an array of `Polar` values. Technically, the REPL calls `display(z)` to display the result of executing a line, which defaults to `show(STDOUT, MIME("text/plain"), z)`, which in turn defaults to `show(STDOUT, z)`, but you should not define new `display` methods unless you are defining a new multimedia display handler (see [Multimedia I/O](#)).

Moreover, you can also define `show` methods for other MIME types in order to enable richer display (HTML, images, etcetera) of objects in environments that support this (e.g. IJulia). For example, we can define formatted HTML display of `Polar` objects, with superscripts and italics, via:

```
julia> Base.show(io::IO, ::MIME"text/html", z::Polar{T}) where {T}  
→ =  
  
    println(io, "<code>Polar{$T}</code> complex number: ",  
  
            z.r, " <i>e</i><sup>", z.θ, " <i>i</i></sup>")
```

A `Polar` object will then display automatically using HTML in an environment that supports HTML display, but you can call `show` manually to get HTML output if you want:

```
julia> show(STDOUT, "text/html", Polar(3.0, 4.0))  
<code>Polar{Float64}</code> complex number: 3.0 <i>e</i><sup>4.0  
→ <i>i</i></sup>
```

As a rule of thumb, the single-line `show` method should return a valid expression for creating the shown object. When this `show` method contains infix operators, such as the multiplication operator (`*`) in our single-line `show` method for `Polar` above, it may not parse correctly when printed as part of another object. To see this, consider the expression object (see [Program representation](#)) which takes the square of a specific instance of our `Polar` type:

```
julia> a = Polar(3, 4.0)
Polar{Float64} complex number:
3.0 * exp(4.0im)

julia> print(:($a^2))
3.0 * exp(4.0im) ^ 2
```

Because the operator `^` has higher precedence than `*` (see [Operator Precedence and Associativity](#)), this output does not faithfully represent the expression `a ^ 2` which should be equal to `(3.0 * exp(4.0im)) ^ 2`. To solve this issue, we must make a custom method for `Base.show_unquoted(io::IO, z::Polar, indent::Int, precedence::Int)`, which is called internally by the expression object when printing:

```
julia> function Base.show_unquoted(io::IO, z::Polar, ::Int,
→ precedence::Int)

    if Base.operator_precedence(:*) <= precedence

        print(io, "(")

        show(io, z)
```

```
    else  
  
        show(io, z)  
  
    end  
  
end  
  
julia> :($a^2)  
:( (3.0 * exp(4.0im)) ^ 2)
```

The method defined above adds parentheses around the call to `show` when the precedence of the calling operator is higher than or equal to the precedence of multiplication. This check allows expressions which parse correctly without the parentheses (such as `:($a + 2)` and `:($a == 2)`) to omit them when printing:

```
julia> :($a + 2)  
:(3.0 * exp(4.0im) + 2)  
  
julia> :($a == 2)  
:(3.0 * exp(4.0im) == 2)
```

## 15.13 "Value types"

In Julia, you can't dispatch on a value such as `true` or `false`. However, you can dispatch on parametric types, and Julia allows you to include "plain bits" values (Types, Symbols, Integers, floating-point numbers, tuples, etc.) as type parameters. A common example is the dimensionality parameter in `Array{T,N}`, where `T` is a type (e.g., `Float64`) but `N` is just an `Int`.

You can create your own custom types that take values as parameters, and use them to control dispatch of custom types. By way of illustration of this idea, let's introduce a parametric type, `Val{x}`, and a constructor `Val(x) = Val{x}()`, which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

`Val` is defined as:

```
julia> struct Val{x}

        end

julia> Base.@pure Val(x) = Val{x}()
```

There is no more to the implementation of `Val` than this. Some functions in Julia's standard library accept `Val` instances as arguments, and you can also use it to write your own functions. For example:

```
julia> firstlast(::Val{true}) = "First"
firstlast (generic function with 1 method)

julia> firstlast(::Val{false}) = "Last"
firstlast (generic function with 2 methods)

julia> firstlast(Val(true))
"First"

julia> firstlast(Val(false))
"Last"
```

For consistency across Julia, the call site should always pass a `Val` instance rather than using a type, i.e., use `foo(Val(:bar))` rather than `foo(Val{:bar})`.

15.14 NULLABLE TYPES: REPRESENTING MISSING VALUES 223

including `Val`; in unfavorable cases, you can easily end up making the performance of your code much worse. In particular, you would never want to write actual code as illustrated above. For more information about the proper (and improper) uses of `Val`, please read the more extensive discussion in [the performance tips](#).

## 15.14 Nullable Types: Representing Missing Values

In many settings, you need to interact with a value of type `T` that may or may not exist. To handle these settings, Julia provides a parametric type called `Nullable{T}`, which can be thought of as a specialized container type that can contain either zero or one values. `Nullable{T}` provides a minimal interface designed to ensure that interactions with missing values are safe. At present, the interface consists of several possible interactions:

Construct a `Nullable` object.

Check if a `Nullable` object has a missing value.

Access the value of a `Nullable` object with a guarantee that a `NullException` will be thrown if the object's value is missing.

Access the value of a `Nullable` object with a guarantee that a default value of type `T` will be returned if the object's value is missing.

Perform an operation on the value (if it exists) of a `Nullable` object, getting a `Nullable` result. The result will be missing if the original value was missing.

Performing a test on the value (if it exists) of a `Nullable` object, getting a result that is missing if either the `Nullable` itself was missing, or the test failed.

224 Perform general operations on single **Nullable** objects  
CHAPTER 15 TYPES

missing data.

## Constructing **Nullable** objects

To construct an object representing a missing value of type T, use the **Nullable{T}()** function:

```
julia> x1 = Nullable{Int64}()  
Nullable{Int64}()  
  
julia> x2 = Nullable{Float64}()  
Nullable{Float64}()  
  
julia> x3 = Nullable{Vector{Int64}}()  
Nullable{Array{Int64,1}}()
```

To construct an object representing a non-missing value of type T, use the **Nullable(x::T)** function:

```
julia> x1 = Nullable(1)  
Nullable{Int64}(1)  
  
julia> x2 = Nullable(1.0)  
Nullable{Float64}(1.0)  
  
julia> x3 = Nullable([1, 2, 3])  
Nullable{Array{Int64,1}}([1, 2, 3])
```

Note the core distinction between these two ways of constructing a **Nullable** object: in one style, you provide a type, T, as a function parameter; in the other style, you provide a single value of type T as an argument.

You can check if a **Nullable** object has any value using `isnull`:

```
julia> isnull(Nullable{Float64}())
true

julia> isnull(Nullable(0.0))
false
```

Safely accessing the value of a **Nullable** object

You can safely access the value of a **Nullable** object using `get`:

```
julia> get(Nullable{Float64}())
ERROR: NullException()
Stacktrace:
[...]

julia> get(Nullable(1.0))
1.0
```

If the value is not present, as it would be for `Nullable{Float64}`, a `NullException` error will be thrown. The error-throwing nature of the `get` function ensures that any attempt to access a missing value immediately fails.

In cases for which a reasonable default value exists that could be used when a **Nullable** object's value turns out to be missing, you can provide this default value as a second argument to `get`:

```
julia> get(Nullable{Float64}(), 0.0)
0.0

julia> get(Nullable(1.0), 0.0)
1.0
```

Make sure the type of the default value passed to `get` and that of the `Nullable` object match to avoid type instability, which could hurt performance. Use `convert` manually if needed.

### Performing operations on `Nullable` objects

`Nullable` objects represent values that are possibly missing, and it is possible to write all code using these objects by first testing to see if the value is missing with `isnull`, and then doing an appropriate action. However, there are some common use cases where the code could be more concise or clear by using a higher-order function.

The `map` function takes as arguments a function `f` and a `Nullable` value `x`. It produces a `Nullable`:

If `x` is a missing value, then it produces a missing value;

If `x` has a value, then it produces a `Nullable` containing `f(get(x))` as value.

This is useful for performing simple operations on values that might be missing if the desired behaviour is to simply propagate the missing values forward.

The `filter` function takes as arguments a predicate function `p` (that is, a function returning a boolean) and a `Nullable` value `x`. It produces a `Nullable` value:

If `x` is a missing value, then it produces a missing value;

If `p(get(x))` is true, then it produces the original value `x`;

If `p(get(x))` is false, then it produces a missing value.

In this way, `Nullable`s represent missing values as allowable values, and converting non-allowable values to missing values.

While `map` and `filter` are useful in specific cases, by far the most useful higher-order function is `broadcast`, which can handle a wide variety of cases, including making existing operations work and propagate `Nullables`. An example will motivate the need for `broadcast`. Suppose we have a function that computes the greater of two real roots of a quadratic equation, using the quadratic formula:

```
julia> root(a::Real, b::Real, c::Real) = (-b + √(b^2 - 4a*c)) / 2a
root (generic function with 1 method)
```

We may verify that the result of `root(1, -9, 20)` is `5.0`, as we expect, since `5.0` is the greater of two real roots of the quadratic equation.

Suppose now that we want to find the greatest real root of a quadratic equations where the coefficients might be missing values. Having missing values in datasets is a common occurrence in real-world data, and so it is important to be able to deal with them. But we cannot find the roots of an equation if we do not know all the coefficients. The best solution to this will depend on the particular use case; perhaps we should throw an error. However, for this example, we will assume that the best solution is to propagate the missing values forward; that is, if any input is missing, we simply produce a missing output.

The `broadcast` function makes this task easy; we can simply pass the `root` function we wrote to `broadcast`:

```
julia> broadcast(root, Nullable(1), Nullable(-9), Nullable(20))
Nullable{Float64}(5.0)
```

228

CHAPTER 15. TYPES

```
julia> broadcast(root, Nullable(1), Nullable{Int}()),
    ↳ Nullable{Int}()
Nullable{Float64}()
```

```
julia> broadcast(root, Nullable{Int}(), Nullable(-9),
    ↳ Nullable(20))
Nullable{Float64}()
```

If one or more of the inputs is missing, then the output of `broadcast` will be missing.

There exists special syntactic sugar for the `broadcast` function using a dot notation:

```
julia> root.(Nullable(1), Nullable(-9), Nullable(20))
Nullable{Float64}(5.0)
```

In particular, the regular arithmetic operators can be `broadcast` conveniently using `.-`-prefixed operators:

```
julia> Nullable(2) ./ Nullable(3) .+ Nullable(1.0)
Nullable{Float64}(1.66667)
```

# Chapter 16

## 메소드

함수에서 함수는 인수의 튜플을 반환 값으로 매핑하는 객체이거나 적절한 값을 반환 할 수 없는 경우 예외를 throw합니다. 두 가지 정수를 더하는 것은 부동 소수점 수에 정수를 더하는 것과는 다른 두 개의 부동 소수점 수를 더하는 것과는 매우 다릅니다. . 이러한 구현의 차이점에도 불구하고 이러한 작업은 모두 "추가"라는 일반적인 개념에 속합니다. 따라서 Julia에서 이러한 동작은 모두 하나의 객체 인 + 함수에 속합니다.

동일한 개념의 여러 구현을 원활하게 사용하기 위해 함수를 한꺼번에 정의 할 필요는 없지만 인수 유형 및 개수의 특정 조합에 특정 동작을 제공함으로써 구분할 수 있습니다. 함수에 대해 가능한 한 행동의 정의를 메소드 라고합니다. 지금까지 우리는 하나의 메소드로 정의 된 함수의 예제만을 제시했으며, 모든 유형의 인수에 적용 할 수 있습니다. 그러나 메소드 정의의 서명에는 그 수 이외에 인수의 유형을 표시하기 위해 주석을 달 수 있으며 단일 메소드 정의 이상이 제공 될 수 있습니다. 함수가 특정 튜플의 인수에 적용되면 해당 인수에 적용 할 수 있는 가장 구체적인 메서드가 적용됩니다. 따라서 함수의 전반적인 동작은 다양한 메서드 정의의 동작을 패치하는 것입니다. 패치 워크가 잘 설계된 경우, 메소드의 구현이 상당히다를 수도 있지만, 함수의 외부 동작은 매끄럽고 일관성있게 나타납니다.

함수가 적용될 때 실행할 메소드의 선택은 dispatch 라고합니다. Julia는 파견 프로세스가 주어진 인수의 수와 함수의 모든 인수 유형에 따라 함수의 메소드 중 어느 것을 호출 할 것인지 선택할 수 있습니다. 이것은 전통적인 객체 지향 언어와는 다르다. 디스패치는 특별한 인수 구문을 사용하는 첫 번째 인수에만 기반을 두며 종종 인수로 명시 적으로 작성되지 않고 암시된다. [^ 1] 함수의 인수를 모두 사용하여 첫 번째 메서드가 아닌 호출 할 메서드를

선택하는 것이 [다중 디스패치](#)로 알려져 있습니다. 다중 디스패치는 C++과 Java에 사용되는 유용합니다. 다른 연산보다 하나의 인수에 "속하는" 연산을 인위적으로 판단하는 것은 의미가 없습니다.  $x + y$  의 더하기 연산이  $y$  보다  $x$ 에 속해 있습니까? 수학 연산자의 구현은 일반적으로 모든 인수의 유형에 따라 다릅니다. 그러나 수학적 작업을 넘어서더라도, 다중 파견은 프로그램을 구조화하고 조직화하는 데 있어 유쾌하고 편리한 패러다임입니다.

## 16.1 메소드 정의

지금까지 우리는 예제에서 제약되지 않은 인자 타입을 가진 하나의 메소드를 가진 함수만을 정의했다. 이러한 함수는 전통적인 동적 유형 지정 언어에서처럼 작동합니다. 그럼에도 불구하고 우리는 다중 디스패치와 메소드를 의식하지 않고 거의 연속적으로 사용했습니다. 앞서 말한 `+` 함수와 같은 모든 Julia의 표준 함수와 연산자에는 인수 유형과 개수의 다양한 조합을 통해 동작을 정의하는 많은 메소드가 있습니다.

함수를 정의 할 때 [Composite Types](#) 섹션에서 소개 한 `:: type-assertion` 연산자를 사용하여 적용 할 수 있는 매개 변수의 유형을 선택적으로 제한 할 수 있습니다. :

```
julia> f(x::Float64, y::Float64) = 2x + y  
f (generic function with 1 method)
```

이 함수 정의는  $x$  와  $y$  가 모두 타입의 값인 호출에만 적용됩니다 [Float64](#):

```
julia> f(2.0, 3.0)  
7.0
```

다른 유형의 인수에 적용하면 [MethodError](#) 가 됩니다.

```
julia> f(2.0, 3)  
ERROR: MethodError: no method matching f(::Float64, ::Int64)  
Closest candidates are:
```

<sup>1</sup> 예를 들어, `obj.meth(arg1, arg2)` 와 같은 메소드 호출에서 C ++이나 Java에서 객체 `obj`는 메소드 호출을 "받는다"는 의미가 아니라 `this` 키워드를 통해 메소드에 암묵적으로 전달됩니다. 명시적인 메소드 인수. 현재 `this` 객체가 메소드 호출의 수신자 일 때 `meth (arg1, arg2)` 만 쓰고 `this` 를 수신 객체로 함축하여 생략 할 수 있습니다.

```
16.1 (:!Matched::Float64, !Matched::Float64) at none:1
```

231

```
julia> f(Float32(2.0), 3.0)
```

```
ERROR: MethodError: no method matching f(::Float32, ::Float64)
```

```
Closest candidates are:
```

```
f(!Matched::Float64, ::Float64) at none:1
```

```
julia> f(2.0, "3.0")
```

```
ERROR: MethodError: no method matching f(::Float64, ::String)
```

```
Closest candidates are:
```

```
f(::Float64, !Matched::Float64) at none:1
```

```
julia> f("2.0", "3.0")
```

```
ERROR: MethodError: no method matching f(::String, ::String)
```

보시다시피, 인수는 정확히 **Float64** 유형이어야합니다. 정수 또는 32 비트 부동 소수점 값과 같은 다른 숫자 유형은 자동으로 '64 비트 부동 소수점으로 변환되지 않으며 숫자로 해석되는 문자열도 아닙니다. **Float64**는 구체적인 타입이고 Julia에서 구체적인 타입을 서브 클래싱 할 수 없기 때문에, 그러한 정의는 정확히 **Float64** 타입의 인자에만 적용될 수 있습니다. 그러나 선언 된 매개 변수 유형이 추상 인 경우보다 일반적인 메소드를 작성하는 것이 종종 유용 할 수 있습니다.

```
julia> f(x::Number, y::Number) = 2x - y
```

```
f (generic function with 2 methods)
```

```
julia> f(2.0, 3)
```

```
1.0
```

이 메소드 정의는 **Number**의 인스턴스인 모든 인수 쌍에 적용됩니다. 그들은 각각 숫자 값인 한 동일한 유형일 필요는 없습니다. 서로 다른 숫자 타입을 처리하는 문제는  $2x - y$  표현식의 산술 연산에 위임됩니다.

~~여기~~ 메소드로 함수를 정의하려면 함수의 수와 유형을 여러 번 정의해야 합니다. 첫 번째 메서드 정의는 함수 자체를 만들고 후속 메서드 정의는 기존 함수 자체에 새 메서드를 추가합니다. 인수의 수와 유형과 일치하는 가장 구체적인 메소드 정의는 함수가 적용될 때 실행됩니다. 따라서, 위의 두 메소드 정의는 함께 추상 타입 `Number` 의 모든 인스턴스 쌍에 대해 `f` 에 대한 동작을 정의하지만 `Float64` 쌍에 고유 한 다른 동작을 사용하여 정의됩니다. 값. 인수 중 하나가 64 비트 부동 소수점이지만 다른 하나는 부동 소수점인 경우 `f(Float64, Float64)` 메서드를 호출 할 수 없으며보다 일반적인 `f(Number, Number)` 메서드를 사용해야합니다.

```
julia> f(2.0, 3.0)
```

```
7.0
```

```
julia> f(2, 3.0)
```

```
1.0
```

```
julia> f(2.0, 3)
```

```
1.0
```

```
julia> f(2, 3)
```

```
1
```

$2x + y$  정의는 첫 번째 경우에만 사용되는 반면,  $2x-y$  정의는 다른 것에 사용됩니다. 자동 주조 또는 함수 인수 변환이 수행되지 않습니다. Julia의 모든 변환은 비 마법적이고 완전히 명시 적입니다. 그러나 [전환 및 판촉](#) 은 충분히 발전된 기술의 영리한 적용이 마법과 구별 될 수 없음을 보여줍니다.<sup>1</sup>

숫자가 아닌 값과 2 개보다 적거나 많은 인수의 경우, `f` 함수는 정의되지 않은 채로 남아 있으며, 여전히 [MethodError](#) 가됩니다 :

```
julia> f("foo", 3)
```

```
ERROR: MethodError: no method matching f(::String, ::Int64)
```

```
Closest candidates are:
```

```
julia> f()  
ERROR: MethodError: no method matching f()  
Closest candidates are:  
  f(!Matched::Float64, !Matched::Float64) at none:1  
  f(!Matched::Number, !Matched::Number) at none:1
```

대화 형 세션에서 함수 객체 자체를 입력하면 함수에 어떤 메소드가 있는지 쉽게 알 수 있습니다.

```
julia> f  
f (generic function with 2 methods)
```

이 출력은 `f` 가 두 가지 메소드를 가진 함수 객체라는 것을 알려줍니다. 이러한 메소드의 서명이 무엇인지 알아 보려면 `methods` 함수를 사용하십시오.

```
julia> methods(f)  
# 2 methods for generic function "f":  
[1] f(x::Float64, y::Float64) in Main at none:1  
[2] f(x::Number, y::Number) in Main at none:1
```

`f` 에는 두 개의 `Float64` 인수를 취하는 메소드와 `Number` 타입의 인수를 취하는 메소드가 있습니다. 또한 메소드가 정의 된 파일과 행 번호를 표시합니다. 이 메소드가 REPL에 정의되었기 때문에 명백한 행 번호는 `none:1`입니다.

`::` 를 사용한 타입 선언이 없으면 메소드 매개 변수의 타입은 기본적으로 `Any`이며, 이는 Julia의 모든 값이 추상 타입 `Any`의 인스턴스이기 때문에 제약이 없다는 것을 의미합니다. 따라서 `f` 에 대한 catch-all 메소드를 다음과 같이 정의 할 수 있습니다 :

```
julia> f(x,y) = println("Whoa there, Nelly.")  
f (generic function with 3 methods)
```

234  
Julia> f("foo", 1)

Whoa there, Nelly.

## CHAPTER 16. 메소드

이 catch-all은 한 쌍의 매개 변수 값에 대해 가능한 다른 메서드 정의보다 덜 구체적이므로 다른 메서드 정의가 적용되지 않는 인수 쌍에서만 호출됩니다.

단순한 개념으로 보일지라도, 가치 유형에 대한 다중 파견은 아마도 줄리아 언어의 가장 강력하고 핵심적인 단일 기능 일 것입니다. 핵심 운영에는 일반적으로 수십 가지 방법이 있습니다 :

```
julia> methods(+)
# 180 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:227
[2] +(x::Bool, y::Bool) in Base at bool.jl:89
[3] +(x::Bool) in Base at bool.jl:86
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:96
[5] +(x::Bool, z::Complex) in Base at complex.jl:234
[6] +(a::Float16, b::Float16) in Base at float.jl:373
[7] +(x::Float32, y::Float32) in Base at float.jl:375
[8] +(x::Float64, y::Float64) in Base at float.jl:376
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:228
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:242
[11] +(x::Char, y::Integer) in Base at char.jl:40
[12] +(c::BigInt, x::BigFloat) in Base.MPFR at mpfr.jl:307
[13] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt) in
    → Base.GMP at gmp.jl:392
[14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt) in Base.GMP at
    → gmp.jl:391
[15] +(a::BigInt, b::BigInt, c::BigInt) in Base.GMP at gmp.jl:390
[16] +(x::BigInt, y::BigInt) in Base.GMP at gmp.jl:361
[17] +(x::BigInt, c::Union{UInt16, UInt32, UInt64, UInt8}) in
    → Base.GMP at gmp.jl:398
```

```
[180] +(a, b, c, xs...) in Base at operators.jl:424
```

유연한 파라메트릭 유형 시스템과의 다중 디스패치 기능을 통해 Julia는 구현 세부 정보에서 분리 된 상위 수준의 알고리즘을 추상적으로 표현할 수 있지만 런타임에 각 사례를 처리 할 수 있는 효율적인 특수 코드를 생성 할 수 있습니다.

## 16.2 방법 모호성

일부 인수 조합에 적용 할 수 있는 고유 한 가장 구체적인 메소드가 없도록 함수 메소드 세트를 정의 할 수 있습니다. :

```
julia> g(x::Float64, y) = 2x + y
g (generic function with 1 method)
```

```
julia> g(x, y::Float64) = x + 2y
g (generic function with 2 methods)
```

```
julia> g(2.0, 3)
7.0
```

```
julia> g(2, 3.0)
8.0
```

```
julia> g(2.0, 3.0)
ERROR: MethodError: g(::Float64, ::Float64) is ambiguous.
→ Candidates:
  g(x, y::Float64) in Main at none:1
  g(x::Float64, y) in Main at none:1
Possible fix, define
  g(::Float64, ::Float64)
```

036서 g (2.0, 3.0) 호출은 g (Float64, Any) 또는 CHAPTER Float64 메소드에 의해 처리 될 수 있으며, 어느 쪽도 더 구체적이지 않습니다. 이런 경우 Julia는 임의로 메서드를 선택하는 것이 아니라 **MethodError** 를 발생시킵니다. 교차 사례의 적절한 방법을 지정하여 메소드 모호성을 피할 수 있습니다. :

```
julia> g(x::Float64, y::Float64) = 2x + 2y
g (generic function with 3 methods)

julia> g(2.0, 3)
7.0

julia> g(2, 3.0)
8.0

julia> g(2.0, 3.0)
10.0
```

일시적인 경우보다 구체적인 방법이 정의 될 때까지 모호성이 존재하기 때문에 모호성을 제거하는 방법을 먼저 정의하는 것이 좋습니다.

보다 복잡한 경우, 방법 모호성 해결은 디자인의 특정 요소를 포함합니다. 이 주제는 아래에서 더 자세히 다루어집니다.

### 16.3 파라 메 트릭 메소드

메소드 정의는 선택적으로 서명을 한정하는 매개 변수를 가질 수 있습니다.

```
julia> same_type(x::T, y::T) where {T} = true
same_type (generic function with 1 method)

julia> same_type(x,y) = false
same_type (generic function with 2 methods)
```

첫 번째 방법은 두 원소 모두 동일한 구체 유형일 때마다 적용됩니다. 두 번째 방법은 모든 경우를 포괄하는 포괄적인 방식으로 작동합니다. 따라서 전반적으로 두 인수가 같은 유형인지 여부를 확인하는 부울 함수를 정의합니다.

```
julia> same_type(1, 2)
```

```
true
```

```
julia> same_type(1, 2.0)
```

```
false
```

```
julia> same_type(1.0, 2.0)
```

```
true
```

```
julia> same_type("foo", 2.0)
```

```
false
```

```
julia> same_type("foo", "bar")
```

```
true
```

```
julia> same_type(Int32(1), Int64(2))
```

```
false
```

이러한 정의는 형식 서명이 UnionAll ([UnionAll Types](#) 참조) 유형인 메소드에 해당합니다.

이러한 종류의 파견에 의한 기능 행동의 정의는 매우 흔하며, 심지어 줄리아에서도 관용적입니다. 메소드 유형 매개 변수는 인수의 유형으로 사용되는 것으로 제한되지 않으며 함수의 본문 또는 본문에 값이 있는 모든 위치에서 사용할 수 있습니다. 다음은 메소드 시그니처의 매개 변수 유형 Vector {T}에 대한 유형 매개 변수로 메소드 유형 매개 변수 T가 사용되는 예제입니다.

```
julia> myappend(v::Vector{T}, x::T) where {T} = [v..., x]
```

```
myappend ( )
```

```
julia> myappend([1,2,3],4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> myappend([1,2,3],2.5)
ERROR: MethodError : myappend    .(:Array{Int64,1}, ::Float64)
:
myappend(:Array{T,1}, !Matched::T) where T at none:1

julia> myappend([1.0,2.0,3.0],4.0)
4-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0

julia> myappend([1.0,2.0,3.0],4)
ERROR : MethodError : myappend    .(:Array{Float64,1}, ::Int64)
:
myappend(:Array{T,1}, !Matched::T) where T at none:1
```

보시다시피, 첨부 된 요소의 유형은 추가되는 벡터의 요소 유형과 일치해야합니다. 그렇지 않으면 **MethodError** 가 발생합니다. 다음 예제에서는 메서드 유형 매개 변수 T 가 반환 값으로 사용됩니다.

```
julia> mytypeof(x::T) where {T} = T
mytypeof( )
```

```
julia> mytypeof(1)
```

```
Int64
```

```
julia> mytypeof(1.0)
```

```
Float64
```

타입 선언에 타입 파라미터에 하위 타입 제약 조건을 넣을 수 있는 것처럼 (파라 메트릭 타입 참조), 메소드의 타입 파라미터를 제한 할 수도 있습니다 :

```
julia> same_type_numeric(x::T, y::T) where {T<:Number} = true
same_type_numeric (   )
```

```
julia> same_type_numeric(x::Number, y::Number) = false
same_type_numeric (   s)
```

```
julia> same_type_numeric(1, 2)
true
```

```
julia> same_type_numeric(1, 2.0)
false
```

```
julia> same_type_numeric(1.0, 2.0)
true
```

```
julia> same_type_numeric("foo", 2.0)
ERROR: MethodError : same_type_numeric .(::String, ::Float64)
:
same_type_numeric(!Matched::T<:Number, ::T<:Number) where
→ T<:Number at none:1
same_type_numeric(!Matched::Number, ::Number) at none:1
```

```
julia> same_type_numeric("foo", "bar")
```

```
julia> same_type_numeric(Int32(1), Int64(2))  
false
```

same\_type\_numeric 함수는 위에 정의된 same\_type 함수와 매우 비슷하게 동작하지만 숫자 쌍에 대해서만 정의됩니다.

파라 메트릭 메소드는 타입 작성에 사용되는 `where` 표현식과 같은 구문을 허용합니다 ([UnionAll Types](#) 참조). 하나의 매개 변수 만있는 경우에는 `{T}` 에있는 중괄호를 생략 할 수 있지만 명확성을 위해 선호하는 경우가 많습니다. 여러 매개 변수는 쉼표로 구분할 수 있습니다. e.g. `where {T, S<:Real}`, 또는 `where` 중첩을 사용하여 작성 , e.g. `where S<:Real where T`.

## 16.4 ##재정의 메소드

메소드를 재정의하거나 새 메소드를 추가 할 때 이러한 변경 사항이 즉시 적용되지 않는다는 것을 인식하는 것이 중요합니다. Julia가 일반적인 JIT 트릭이나 오버 헤드없이 정적으로 코드를 추론하고 컴파일하여 실행하는 것이 중요합니다. 실제로 새로운 메소드 정의는 Tasks와 쓰레드 (그리고 이전에 정의 된 `@generated` 함수들) 를 포함하여 현재 런타임 환경에서 볼 수 없을 것입니다. 예를 들어 이것이 무엇을 의미하는지 알아 봅시다.

```
julia> function tryeval()  
  
    @eval newfun() = 1  
  
    newfun()  
  
end  
tryeval (generic function with 1 method)  
  
julia> tryeval()
```

```
ERROR: MethodError: newfun ()  
       : current world is xxxx2 world age xxxx1.  
       :  
none:1  newfun()  (this world context      .)  
in tryeval() at none:1  
...  
...
```

```
julia> newfun()
```

```
1
```

이 예제에서는 `newfun`에 대한 새로운 정의가 생성되었지만 즉시 호출 할 수 없음을 관찰합니다. 새로운 전역 변수는 `tryeval` 함수에서 즉시 볼 수 있으므로 `return newfun` (괄호없이)을 쓸 수 있습니다. 그러나 당신이나 당신의 호출자, 또는 그들이 부르는 기능도 아닙니다. 이 새로운 메소드 정의를 호출 할 수 있습니다!

그러나 예외가 있습니다 : 미래의 `newfun` 호출은 예상대로 REPL에서 작동합니다. `newfun`의 새로운 정의를 보고 호출 할 수 있습니다.

그러나 'tryeval'에 대한 향후 호출은 이전 선언문 REPL에서처럼 `newfun`의 정의를 계속 볼 것이며, 따라서 `tryeval`에 대한 호출 전에 나타납니다.

어떻게 작동하는지 직접 확인해보십시오.

이 동작의 구현은 "세계 시대 카운터"입니다. 단조 증가 값은 각 메소드 정의 연산을 추적합니다. 이를 통해 "주어진 런타임 환경에서 볼 수 있는 메소드 정의 세트"를 단일 숫자 또는 "world age"로 설명 할 수 있습니다. 또한 서수 값을 비교하여 두 `worlds`에서 사용할 수 있는 방법을 비교할 수 있습니다. 위 예제에서 "current world"("newfun"메서드가 있는)는 `tryeval` 실행이 시작될 때 수정 된 작업 로컬 "런타임 환경" 보다 큰 것입니다.

때로는 이것을 피하는 것이 필요합니다 (예를 들어 위의 REPL을 구현하는 경우). 다행히도 쉬운 해결책이 있습니다 : `Base.invokelatest`를 사용하여 함수를 호출하십시오 :

```
julia> function tryeval2()
```

```

@eval newfun2() = 2

Base.invokelatest(newfun2)

end

tryeval2(  )

```

**julia>** tryeval2()

2

마지막으로, 이 규칙이 적용되는 좀 더 복잡한 예제를 살펴 보겠습니다. 처음에는 하나의 메소드를 가지고 있는 함수  $f(x)$  를 정의합니다 :

**julia>** f(x) = " "

f( )

$f(x)$  를 사용하는 다른 연산을 시작합니다.:

**julia>** g(x) = f(x)

g( )

**julia>** t = @async f(wait()); yield();

이제 우리는  $f(x)$  에 몇 가지 새로운 메소드를 추가합니다 :

**julia>** f(x::Int) = "Int "

f( )

**julia>** f(x::Type{Int}) = "Type{Int} "

f( )

결과가 어떻게 다른지 비교해봅시다.:

```
"Int "
julia> g(1)
"Int "
julia> wait(schedule(t, 1))
" "
julia> t = @async f(wait()); yield();
julia> wait(schedule(t, 1))
"Int "
```

## 16.5 파라 메트릭 방법을 사용한 디자인 패턴

성능이나 유용성에 복잡한 디스패치 로직이 필요하지는 않지만 때로는 일부 알고리즘을 표현하는 가장 좋은 방법 일 수 있습니다. 이런 방식으로 디스패치를 사용할 때 때로는 나타나는 몇 가지 일반적인 디자인 패턴이 있습니다.

### super-type에서 type 매개 변수 추출

여기 `AbstractArray`의 임의의 하위 유형의 요소 유형 `T`를 반환하기 위한 올바른 코드 템플릿이 있습니다.:

```
abstract type AbstractArray{T, N} end
eltype(::Type{<:AbstractArray{T}}) where {T} = T
```

삼각 괄호를 사용합니다. 예를 들어 `T` 가 `UnionAll` 타입인 경우에 주목합니다. `eltype(Array{T}) where T <: Integer` 이면 `Any` 가 반환됩니다. (`Base`에 있는 `eltype`의 버전도 마찬가지입니다). Note that if `T` is a `UnionAll`

244a v0.6에서 삼각형 패턴의 출현 이전에 유일한 올바른 방법이었던 APPROXIMATE 방법은 매우 표 같습니다. :

```
abstract type AbstractArray{T, N} end
eltype(::Type{AbstractArray}) = Any
eltype(::Type{AbstractArray{T}}) where {T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A<:AbstractArray} = eltype(supertype(A))
```

또 다른 가능성은 다음과 같습니다. 매개 변수 T 가 더 좁게 일치해야하는 경우에 적용하는 것이 유용 할 수 있습니다.

```
eltype(::Type{AbstractArray{T, N} where {T<:S, N<:M}}) where {M,
→ S} = Any
eltype(::Type{AbstractArray{T, N} where {T<:S}}) where {N, S} =
→ Any
eltype(::Type{AbstractArray{T, N} where {N<:M}}) where {M, T} = T
eltype(::Type{AbstractArray{T, N}}) where {T, N} = T
eltype(::Type{A}) where {A <: AbstractArray} =
→ eltype(supertype(A))
```

일반적인 실수 중 하나는 내부 검사를 사용하여 요소 유형을 얻는 것입니다. :

```
eltype_wrong(::Type{A}) where {A<:AbstractArray} = A.parameters[1]
```

그러나 이것이 실패 할 경우를 만드는 것은 어렵지 않습니다. :

```
struct BitVector <: AbstractArray{Bool, 1}; end
```

여기서 우리는 매개 변수를 가지지 않는 BitVector 타입을 만들었지만, element-type이 여전히 완전하게 지정되고 T 는 Bool 과 같습니다!

일반적인 코드를 만들 때 유형의 레이아웃을 변경하고 유사한 유형의 객체를 생성해야하는 경우가 종종 있습니다. 유형 매개 변수의 변경이 필요합니다. 예를 들어 임의의 요소 유형을 가진 일종의 추상 배열을 가질 수 있으며 특정 요소 유형으로 계산을 작성하려고합니다. 이 유형 변환을 계산하는 방법을 설명하는 각 `AbstractArray{T}` 하위 유형에 대한 메소드를 구현해야합니다. 하나의 부속 유형에 대해 다른 매개 변수를 갖는 다른 부속 유형으로의 변환이 없습니다. (빠른 검토 : 이것이 왜 있는지 보시겠습니까?)

`AbstractArray` 의 부속유형은 일반적으로 이를 달성하는 두가지 메소드를 구현합니다. 입력배열을 특정 `AbstractArray {T, N}` 추상유형의 부속유형으로 변환하는 메소드 및 특정요소 유형을 가진 초기화되지 않은 새로운 배열을 만드는 방법. 이들의 샘플구현은 표준 라이브러리에서 찾을 수 있습니다. 다음은 `input` 과 `output` 이 같은 타입으로되어 있다는 것을 보장하는 기본적인 예제사용법입니다 :

```
| input = convert(AbstractArray{Eltype}, input)
| output = similar(input, Eltype)
```

이를 확장하기 위해 알고리즘에 입력배열의 복사본이 필요한경우 반환값이 원래입력의 별칭일 수 있으므로 `convert` 는 충분하지 않습니다. 출력배열을 만들기 위해 `similar`와 입력데이터로 채우기 위해 `copy!` 를 결합하는 것은 변경가능한 입력인수 복사본에 대한 요구사항을 표현하는 일반적인 방법입니다.:

```
| copy_with_eltype(input, Eltype) = copy!(similar(input, Eltype),
|   → input)
```

## 반복 디스패치

다단계 매개변수 목록을 발송하려면 종종 각 발송 단계를 별개의 기능으로 분리하는 것이 가장 좋습니다. 단일발송에 대한 접근방식과 비슷할 수도 있지만, 아래에서 보게 되겠지만 더 유연합니다.

예를들어 배열의 요소유형을 디스패치하려고하면 종종 모호한 상황이 발생합니다. 대신 일반적으로 코드는 컨테이너유형에 먼저 전달되고 `eltype`를 기반으로 한 것 보다 구체적인

메서드로 순환됩니다. 대부분의 경우 알고리즘은이 계층적 접근방식을 편리하게 사용하지만 다른 경우에는 수동으로 이 엄격성을 해결해야합니다. 이 디스패치 브랜칭은 두 개의 행렬을 합산하는 로직에서 볼 수 있습니다. :

```
#      map .
+(a::Matrix, b::Matrix) = map(+, a, b)
#
#
#
+(a, b) = +(promote(a, b)...)
#
#
#
+(a::Float64, b::Float64) = Core.add(a, b)
```

## Trait-based 디스패치

위의 iterated 디스패치를 자연스럽게 확장하면 형식계층에 정의된 집합과 독립적인 형식집합을 디스패치 할 수 있는 계층을 메서드선택에 추가 할 수 있습니다. 우리는 문제의 유형의 을 작성하여 그러한 집합을 구성 할 수 있지만, 생성 후에 유형을 변경할 수 없으므로 이 집합은 확장 할 수 없습니다. 그러나 이러한 확장가능 세트는 종종 "Holy-trait" 이라고하는 디자인 패턴으로 프로그래밍 될 수 있습니다.

이 패턴은 함수인수가 속할 수 있는 각 특성집합에 대해 다른 단일값 (또는 유형)을 계산하는 일반함수를 정의하여 구현됩니다. 이 기능이 순수하면 정상적인 발송과 비교하여 성능에 영향을 주지 않습니다.

이전 섹션의 예제는 `map` 및 `promote`의 구현세부사항을 설명했습니다. 이 두 특성은 이러한 특성에 따라 작동합니다. `map`의 구현과 같이 행렬을 반복 할 때 중요한 질문 중 하나는 데이터를 순회하기 위해 사용하는 순서입니다. `AbstractArray` 보조형이 `Base.IndexStyle` 형질을 구현할 때 `map`과 같은 다른함수는 이러한 정보를 보내서 최상의 알고리즘을 선택합니다 (`Abstract Array Interface`). 즉, 일반정의 + 특성클래스를 사용하면 시스템에서 가장 빠른 버전을 선택할 수 있기 때문에 각 하위유형에서 `map`의 사용자정의버전을 구현할 필요가 없습니다. trait-based의 디스패치를 보여주는 `map`의 장난감 구현 :

```

16.5 (f, a::AbstractArray, b::AbstractArray) = 247
    ↳ map(Base.IndexStyle(a, b), f, a, b)
# :
map(::Base.IndexCartesian, f, a::AbstractArray, b::AbstractArray)
    ↳ ... =
#   ( )
map(::Base.IndexLinear, f, a::AbstractArray, b::AbstractArray) =
    ↳ ...

```

이 trait-based 접근법은 스칼라 +에 의해 채택 된 `promote` 메커니즘에도 존재합니다. 이것은 `promote_type` 을 사용하는데, 이것은 최적의 공통 유형을 반환합니다. 두 가지 유형의 피연산자가 주어진 경우 연산을 계산합시다. 이를 통해 모든 유형의 인수에 대해 모든 함수를 구현하는 문제를 줄이고, 각 유형에서 일반 유형으로 변환 작업을 구현하는 훨씬 작은 문제와 선호되는 preferred pair-wise promotion rules 표를 줄일 수 있습니다.

## 출력 유형 계산

trait-based 프로모션에 대한 논의는 다음 디자인패턴으로의 전환을 제공합니다. 매트릭스 작동을 위한 출력 요소 유형 계산.

추가 같은 기본 연산을 구현하기 위해 `promote_type` 함수를 사용하여 원하는 출력유형을 계산합니다. (이전과 마찬가지로 + 호출로 `promote` 호출에서 이것을 보았습니다).

행렬에 대한 함수의 경우보다 더 복잡한 연산 순서에 대해 예상되는 반환 형식을 계산해야 할 수도 있습니다. 이 작업은 종종 다음 단계로 수행됩니다.

1. 알고리즘의 구조에 의해 수행되는 일련의 연산을 나타내는 작은 함수 `op` 를 작성합니다.
2. 결과 행렬의 요소 타입 `R` 을 `promote_op (op, argument_types ...)` 로 계산합니다. 여기서 `argument_types` 는 각 입력 배열에 적용된 `eltype` 에서 계산됩니다.
3. 출력 행렬을 `similar(R, dims)` 로 만듭니다. 여기서 `dims` 는 출력 배열의 원하는 차원입니다.

248 a more specific example, a generic square-matrix multiplication function might look like:

좀 더 구체적인 예를 들어, 일반제곱 - 행렬곱셈 pseudo코드는 다음과 같습니다.

```
function matmul(a::AbstractMatrix, b::AbstractMatrix)
    op = (ai, bi) -> ai * bi + ai * bi

    ## `one(eltype (a))`      .:
    # R = typeof(op(one(eltype(a)), one(eltype(b)))) 

    ## `a [1]`      ,
    # R = typeof(op(a[1], b[1])) 

    ## `+` `promote_type`      .
    ## Bool      .:
    # R = promote_type(ai, bi)

    #      .(   ):
    # R = return_types(op, (eltype(a), eltype(b))) 

    ##  .:
    R = promote_op(op, eltype(a), eltype(b))
    ##  .

    output = similar(b, R, (size(a, 1), size(b, 2)))
    if size(a, 2) > 0
        for j in 1:size(b, 2)
            for i in 1:size(b, 1)
                ## `R` `Any`, `zero(Any)`      `ab = zero(R)`      .
                ##     typeof (a * b)! = typeof (a * b + a * b) == R
                →     `ab::R`      `ab`      .
```

## 16.6. 매개 변수 적으로 제한된 VARARGS 메소드

249

```
for k in 2:size(a, 2)
    ab += a[i, k] * b[k, j]
end
output[i, j] = ab
end
end
return output
end
```

## 별도의 변환과 귀널로직

컴파일 시간을 줄이고 복잡성을 테스트하는 한 가지 방법은 원하는 유형으로 변환하기 위한 로직과 계산을 분리하는 것입니다. 이를 통해 컴파일러는 변환 논리를 대형 귀널의 나머지 본문과 독립적으로 특수화하고 인라인 할 수 있습니다.

이것은 더 큰 클래스 유형에서 변환 할 때 나타나는 공통 패턴입니다 알고리즘이 실제로 지원하는 특정 인수 유형으로 변환합니다.

```
complexfunction(arg)::Int) = ...
complexfunction(arg)::Any) = complexfunction(convert(Int, arg))

matmul(a::T, b::T) = ...
matmul(a, b) = matmul(promote(a, b)...)
```

## 16.6 매개 변수 적으로 제한된 Varargs 메소드

함수매개변수는 "varargs"함수 ([Varargs Functions](#))에 제공 될 수 있는 인수의 수를 제한하는데 사용될 수도 있습니다. Vararg {T, N} 표기법은 그러한 제약을 나타내기 위해 사용됩니다. 예 :

```
julia> bar(a,b,x)::Vararg{Any,2}) = (a,b,x)
bar ( )
```

```
julia> bar(1,2,3)
ERROR: MethodError: bar(::Int64, ::Int64, ::Int64)
. :
bar(::Any, ::Any, ::Any, !Matched::Any) at none:1

julia> bar(1,2,3,4)
(1, 2, (3, 4))

julia> bar(1,2,3,4,5)
ERROR: MethodError: bar(::Int64, ::Int64, ::Int64, ::Int64,
→ ::Int64)
. :
bar(::Any, ::Any, ::Any, ::Any) at none:1
```

보다 유용하게 파라미터에 의해 varargs 메소드를 제한하는 것이 가능합니다. 예 :

```
function getindex(A::AbstractArray{T,N},
→ indexes::Vararg{Number,N}) where {T,N}
```

의 수가 배열의 차원과 일치 할 때만 호출됩니다.

When only the type of supplied arguments needs to be constrained Vararg{T} can be equivalently written as T.... For instance f(x::Int...) = x is a shorthand for f(x::Vararg{Int}) = x.

오직 제공된 인자의 타입만이 제약을 받아야 할 때 Vararg{T} 는 T... 와 같이 쓰일 수 있습니다. 예를 들어, f(x::Int ...) = x 는 f(x::Vararg{Int}) = x 의 속기입니다.

## 16.7 키워드 인수 선택 사항에 대한 참고 사항

Functions 에서 간략하게 언급했듯이 선택적 인수는 여러 메소드정의의 구문으로 구현됩니다. 예를 들어, 이 정의는 다음과 같습니다.

다음 세 가지 방법으로 변환됩니다.

```
f(a,b) = a+2b
f(a) = f(a,2)
f() = f(1,2)
```

This means that calling `f()` is equivalent to calling `f(1,2)`. In this case the result is 5, because `f(1,2)` invokes the first method of `f` above. However, this need not always be the case. If you define a fourth method that is more specialized for integers: 이것은 `f()` 를 호출하는 것이 `f(1,2)` 를 호출하는 것과 동일하다는 것을 의미합니다. 이 경우 결과는 5입니다. 왜냐하면 `f(1,2)` 가 위의 `f` 의 첫 번째 메소드를 호출하기 때문입니다. 그러나, 항상 그런 것은 아닙니다. 정수에보다 특수화 된 네 번째 메서드를 정의하면 다음과 같습니다.

```
f(a::Int,b::Int) = a-2b
```

`f()` 와 `f(1,2)` 의 결과는 -3 입니다. 즉, 선택적 인수는 해당함수의 특정메서드가 아닌 함수에 연결됩니다. 그것은 메소드가 불려지는 옵션 인수의 형태에 의존합니다. 선택적 인수가 전역 변수로 정의되면 선택적 인수의 유형이 런타임에 변경 될 수도 있습니다.

키워드 인수는 일반적인 위치 인수와는 상당히 다르게 작동합니다. 특히, 메소드 디스패치에 참여하지 않습니다. 메소드는 일치하는 메소드가 식별 된 후에 처리되는 키워드 인수를 가진 위치인수에 기반하여 발송됩니다.

## 16.8 함수같은 객체

메서드는 형식과 관련이 있으므로 형식에 메서드를 추가하여 임의의 Julia 객체를 "호출 가능"하게 만들 수 있습니다. (이러한 "호출 가능"한 객체를 "functors"라고 합니다.)

예를 들어 다항식의 계수를 저장하는 유형을 정의 할 수 있습니다. 다항식을 계산하는 함수 :

```
252 julia> struct Polynomial{R}
           coeffs::Vector{R}

           end

julia> function (p::Polynomial)(x)

           v = p.coeffs[end]

           for i = (length(p.coeffs)-1):-1:1

               v = v*x + p.coeffs[i]

           end

           return v

       end
```

## CHAPTER 16. 메소드

함수는 이를 대신 형식으로 지정됩니다. 함수 본문에서 p 는 호출 된 객체를 나타냅니다. 은 다음과 같이 사용할 수 있습니다.

```
julia> p = Polynomial([1,10,100])
Polynomial{Int64}([1, 10, 100])
```

```
julia> p(3)
```

931

이 메카니즘은 타입 생성자와 클로저 (주변 환경을 참조하는 내부 함수)가 Julia에서 어떻게 작동 하는지를 결정하는 열쇠이기도합니다. 나중에이 매뉴얼에서 설명합니다.

때때로 메소드를 추가하지 않고 일반함수를 도입하는 것이 유용하기도 합니다. 이것은 인터페이스 정의와 구현을 분리하는데 사용할 수 있습니다. 문서화 또는 코드 가독성을 위해 수행 될 수도 있습니다. 이를 위한 문법은 인자의 튜플이 없는 빈function 블록입니다 :

```
function emptyfunc
end
```

## 16.10 방법 설계 및 모호한 방지

Julia의 방법 다형성은 가장 강력한 기능 중 하나이지만, 이 힘을 이용하는 것은 설계상의 어려움을 야기 할 수 있습니다. 특히, 보다 복잡한 메소드 계층 구조에서는 모호성 (ambiguities)이 발생하는 경우는 드뭅니다. 위에서, 모호성을 해결할 수 있다고 지적했습니다.

```
f(x, y::Int) = 1
f(x::Int, y) = 2
```

메소드를 정의함으로서

```
f(x::Int, y::Int) = 3
```

이것은 종종 올바른 전략일 수 있습니다. 그러나 이 조언을 맹목적으로 따르는 것이 비생산적일 수 있는 상황이 있습니다. 특히, 일반 함수의 메서드가 많을 수록 모호성 가능성이 커집니다. 메소드 계층구조가 간단한 예제보다 복잡해지면 대체 전략에 대해 신중하게 생각하는 것이 가치가 있습니다. 아래에서는 특정 문제와 이러한 문제를 해결할 수 있는 몇 가지 대안을 논의합니다.

### 튜플 및 NTuple 인수

`Tuple` (그리고 `NTuple`) 인자는 특별한 도전을 제시합니다. 예를 들어,

```
f(x::NTuple{N, Int}) where {N} = 1
f(x::NTuple{N, Float64}) where {N} = 2
```

¶ 540일 가능성 때문에 모호하다면 :Int 나 Float64 변형이 호출되어 API가 다른지를 결정하는 요소가 없습니다. 모호성을 해결하기 위해 한 가지 방법은 빈 튜플에 대한 메서드를 정의하는 것입니다.

```
f(x::Tuple{}) = 3
```

또는 하나의 메서드를 제외한 모든 메서드에 대해 적어도 하나의 요소가 튜플에 있다고 주장 할 수 있습니다.

```
f(x::NTuple{N, Int}) where {N} = 1           # .
f(x::Tuple{Float64, Vararg{Float64}}) = 2    #   Float64 .
```

## 디자인 직교화

이상의 인수를 디스패치하려는 상황일 때 "wrapper"기능이 보다 단순한 설계를 할 수 있는지 고려해야합니다. 예를 들어 여러 변형을 작성하는 대신에 :

```
f(x::A, y::A) = ...
f(x::A, y::B) = ...
f(x::B, y::A) = ...
f(x::B, y::B) = ...
```

정의하는 것을 고려할 수 있습니다.

```
f(x::A, y::A) = ...
f(x, y) = f(g(x), g(y))
```

where `g` converts the argument to type A. This is a very specific example of the more general principle of [orthogonal design](#), in which separate concepts are assigned to separate methods. Here, `g` will most likely need a fallback definition 여기서 `g` 는 인수를 타입 A로 변환합니다. 이것은 별도의 개념이 별도의 방법으로 할당 된 [직교 설계](#)의 보다 일반적인 매우 구체적인 예입니다. 여기서 `g` 는 대개 대체 정의를 요구합니다.

A related strategy은  $x$ 와  $y$ 를 일반적인 유형으로 유도하기 위해 `promote`을 이용합니다 :

```
f(x::T, y::T) where {T} = ...  
f(x, y) = f(promote(x, y)...)
```

이 디자인의 한 가지 위험은  $x$  와  $y$  를 같은 유형으로 변환하는 적절한 프로모션 방법이 없으면 두 번째 방법이 무한히 반복되어 스택 오버플로가 트리거 될 수 있다는 것입니다. exported 안된 함수 `Base.promote_noncircular`는 대안으로 사용할 수 있습니다. 프로모션이 실패하면 여전히 오류가 발생하지만 더 구체적인 오류 메시지로 인해 더 빨리 오류가 발생합니다.

### 한 번에 하나의 인수로 디스패치

If you need to dispatch on multiple arguments, and there are many fallbacks with too many combinations to make it practical to define all possible variants, then consider introducing a "name cascade" where (for example) you dispatch on the first argument and then call an internal method:

여러 인수를 전달해야하거나 가능한 많은 변형을 정의하기 위해 너무 많은 조합을 사용하는 많은 대체 시스템이 있는 경우에는 "name cascade"를 도입하여 (예를 들어) 첫 번째 인수로 전달한 다음 내부 메소드를 호출하는 것을 고려하십시오.

```
f(x::A, y) = _fA(x, y)  
f(x::B, y) = _fB(x, y)
```

그러면 내부 메소드 `_fA` 와 `_fB` 는  $x$  에 대해 서로 모호한 점을 고려하지 않고  $y$  에 디스패치 할 수 있습니다.

이 전략에는 최소 하나이상의 주요한 단점이 있습니다 : 많은 경우, 사용자가 exported한 함수 `f` 의 추가 특수화를 정의하여 `f` 의 동작을 추가로 사용자정의 할 수 없습니다. 대신,

~~CHAPTER 6 메소드와~~  
CHAPTER 6 메소드와  
exported한 메소드 사이의 줄을 흐리게 만듭니다.

## 추상 컨테이너 및 요소 유형

가능한, 파견하는 방법을 정의하지 않도록하십시오. 추상 컨테이너의 특정 요소 유형은 예를 들어,

```
| -(A::AbstractArray{T}, b::Date) where {T<:Date}
```

메소드를 정의할 누군가를 위해 모호한 메소드를 정의합니다.

```
| -(A::MyArrayType{T}, b::T) where {T}
```

가장 좋은 방법은 다음중 하나의 방법을 정의하는 것을 피하는 것입니다. 대신, 일반적인 메소드- A::AbstractArray, b) 이 메소드가 각 컨테이너 유형과 요소 유형에 대해 올바른 작업을 수행하는 일반 호출(예 :similar and -)로 구현되었는지 확인하십시오. 이것은 당신의 메소드를 [직교화](#) 하는 조언의 좀 더 복잡한 변형입니다.

이 접근법이 가능하지 않을 때 모호성을 해결하는 것에 대해 다른 개발자와 토론을 시작하는 것이 좋습니다. 처음에 하나의 방법이 정의되었다고해서 그것이 반드시 수정되거나 제거 될 수 없다는 것을 의미하지는 않습니다. 최후의 수단으로 한 개발자는 "반창고"방법을 정의 할 수 있습니다

```
| -(A::MyArrayType{T}, b::Date) where {T<:Date} = ...
```

그것은 무차별한 힘에 의한 모호성을 해결합니다.

## 복잡한 인수 "cascades"와 기본 인수

기본값을 제공하는 "cascades"메서드를 정의하는 경우 잠재적 기본값에 해당하는 인수를 삭제하는 데 주의해야합니다. 예를 들어, 디지털 필터링 알고리즘을 작성 중이며 패딩을 적용하여 신호의 가장자리를 처리하는 방법이 있다고 가정합니다. :

## 16.10. 방법 선택 및 모호한 방지

257

```
function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel) # "" .
end
```

이것은 디폴트 패딩을 제공하는 메소드와 충돌 할 것이다. :

```
myfilter(A, kernel) = myfilter(A, kernel, Replicate()) # replicate
→ the edge by default
```

이 두 가지 방법은 A가 계속 균지면서 무한 재귀를 생성합니다.

더 나은 디자인은 다음과 같이 호출 계층 구조를 정의하는 것입니다.

```
struct NoPad end # .

myfilter(A, kernel) = myfilter(A, kernel, Replicate()) #

function myfilter(A, kernel, ::Replicate)
    Apadded = replicate_edges(A, size(kernel))
    myfilter(Apadded, kernel, NoPad()) # .
end

# .

function myfilter(A, kernel, ::NoPad)
    # "" .
end
```

NoPad is supplied in the same argument position as any other kind of padding, so it keeps the dispatch hierarchy well organized and with reduced likelihood of ambiguities. Moreover, it extends the "public" `myfilter` interface: a user who wants to control the padding explicitly can call the `NoPad` variant directly.

`NoPad`는 다른 종류의 패딩과 같은 인수 위치에서 제공되기 때문에 `API`를 정리하고 애매하게 만들 가능성이 낮습니다. 또한, “public” `myfilter` 인터페이스를 확장합니다. 패딩을 명시적으로 제어하려는 사용자는 `NoPad` 변형을 직접 호출 할 수 있습니다.

---

<sup>1</sup>Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

# Chapter 17

## Constructors

Constructors<sup>1</sup> are functions that create new objects – specifically, instances of [Composite Types](#). In Julia, type objects also serve as constructor functions: they create new instances of themselves when applied to an argument tuple as a function. This much was already mentioned briefly when composite types were introduced. For example:

```
julia> struct Foo  
  
        bar  
  
        baz  
  
    end  
  
julia> foo = Foo(1, 2)  
Foo(1, 2)  
  
julia> foo.bar  
1  
  
julia> foo.baz
```

For many types, forming new objects by binding their field values together is all that is ever needed to create instances. There are, however, cases where more functionality is required when creating composite objects. Sometimes invariants must be enforced, either by checking arguments or by transforming them. [Recursive data structures](#), especially those that may be self-referential, often cannot be constructed cleanly without first being created in an incomplete state and then altered programmatically to be made whole, as a separate step from object creation. Sometimes, it's just convenient to be able to construct objects with fewer or different types of parameters than they have fields. Julia's system for object construction addresses all of these cases and more.

## 17.1 Outer Constructor Methods

A constructor is just like any other function in Julia in that its overall behavior is defined by the combined behavior of its methods. Accordingly, you can add functionality to a constructor by simply defining new methods. For example, let's say you want to add a constructor method for `Foo` objects that takes only one argument and uses the given value for both the `bar` and `baz` fields. This is simple:

```
julia> Foo(x) = Foo(x, x)
```

```
Foo
```

```
julia> Foo(1)
```

```
Foo(1, 1)
```

<sup>1</sup>Nomenclature: while the term "constructor" generally refers to the entire function which constructs objects of a type, it is common to abuse terminology slightly and refer to specific constructor methods as "constructors". In such situations, it is generally clear from context that the term is used to mean "constructor method" rather than "constructor function", especially as it is often used in the sense of singling out a particular method of the constructor from all of the others.

~~Y3.2.1.1 INNER CONSTRUCTOR METHODS~~ Foo constructor method that supplies default values for both of the bar and baz fields:

```
julia> Foo() = Foo(0)
```

```
Foo
```

```
julia> Foo()
```

```
Foo(0, 0)
```

Here the zero-argument constructor method calls the single-argument constructor method, which in turn calls the automatically provided two-argument constructor method. For reasons that will become clear very shortly, additional constructor methods declared as normal methods like this are called outer constructor methods. Outer constructor methods can only ever create a new instance by calling another constructor method, such as the automatically provided default ones.

## 17.2 Inner Constructor Methods

While outer constructor methods succeed in addressing the problem of providing additional convenience methods for constructing objects, they fail to address the other two use cases mentioned in the introduction of this chapter: enforcing invariants, and allowing construction of self-referential objects. For these problems, one needs inner constructor methods. An inner constructor method is much like an outer constructor method, with two differences:

1. It is declared inside the block of a type declaration, rather than outside of it like normal methods.
2. It has access to a special locally existent function called `new` that creates objects of the block's type.

example, suppose one wants to declare a type at CHAPTER 17. CONSTRUCTORS numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

```
julia> struct OrderedPair  
    x::Real  
    y::Real  
    OrderedPair(x,y) = x > y ? error("out of order") :  
    → new(x,y)  
end
```

Now `OrderedPair` objects can only be constructed such that  $x \leq y$ :

```
julia> OrderedPair(1, 2)  
OrderedPair(1, 2)  
  
julia> OrderedPair(2,1)  
ERROR: out of order  
Stacktrace:  
[1] OrderedPair(::Int64, ::Int64) at ./none:4
```

If the type were declared `mutable`, you could reach in and directly change the field values to violate this invariant, but messing around with an object's internals uninvited is considered poor form. You (or someone else) can also provide additional outer constructor methods at any later point, but once a type is declared, there is no way to add more inner constructor methods. Since outer constructor methods can only create objects by calling other constructor methods, ultimately, some inner constructor must be called to create an object.

This guide on [inner constructor methods](#) declared type must come into existence by a call to one of the inner constructor methods provided with the type, thereby giving some degree of enforcement of a type's invariants.

If any inner constructor method is defined, no default constructor method is provided: it is presumed that you have supplied yourself with all the inner constructors you need. The default constructor is equivalent to writing your own inner constructor method that takes all of the object's fields as parameters (constrained to be of the correct type, if the corresponding field has a type), and passes them to `new`, returning the resulting object:

```
julia> struct Foo  
  
    bar  
  
    baz  
  
    Foo(bar,baz) = new(bar,baz)  
  
end
```

This declaration has the same effect as the earlier definition of the `Foo` type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

```
julia> struct T1  
  
    x::Int64  
  
end
```

264  
julia> struct T2

CHAPTER 17. CONSTRUCTORS

x::Int64

T2(x) = new(x)

end

julia> T1(1)

T1(1)

julia> T2(1)

T2(1)

julia> T1(1.0)

T1(1)

julia> T2(1.0)

T2(1)

It is considered good form to provide as few inner constructor methods as possible: only those taking all arguments explicitly and enforcing essential error checking and transformation. Additional convenience constructor methods, supplying default values or auxiliary transformations, should be provided as outer constructors that call the inner constructors to do the heavy lifting. This separation is typically quite natural.

### 17.3 Incomplete Initialization

The final problem which has still not been addressed is construction of self-referential objects, or more generally, recursive data structures. Since the

IMMEDIATE INITIALIZATION 265  
Immediately obvious, let us briefly explore it. Consider the following recursive type declaration:

```
julia> mutable struct SelfReferential  
|     obj::SelfReferential  
|  
| end
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `a` is an instance of `SelfReferential`, then a second instance can be created by the call:

```
julia> b = SelfReferential(a)
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `obj` field? The only solution is to allow creating an incompletely initialized instance of `SelfReferential` with an unassigned `obj` field, and using that incomplete instance as a valid value for the `obj` field of another instance, such as, for example, itself.

To allow for the creation of incompletely initialized objects, Julia allows the `new` function to be called with fewer than the number of fields that the type has, returning an object with the unspecified fields uninitialized. The inner constructor method can then use the incomplete object, finishing its initialization before returning it. Here, for example, we take another crack at defining the `SelfReferential` type, with a zero-argument inner constructor returning instances having `obj` fields pointing to themselves:

```
julia> mutable struct SelfReferential  
|     obj::SelfReferential  
|  
| end
```

```
SelfReferential() = (x = new(); x.obj = x)
```

```
end
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
julia> x = SelfReferential();
```

```
julia> x === x
```

```
true
```

```
julia> x === x.obj
```

```
true
```

```
julia> x === x.obj.obj
```

```
true
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, incompletely initialized objects can be returned:

```
julia> mutable struct Incomplete
```

```
xx
```

```
Incomplete() = new()
```

```
end
```

```
julia> z = Incomplete();
```

When you incomplete objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

```
julia> z.xx  
ERROR: UndefRefError: access to undefined reference
```

This avoids the need to continually check for `null` values. However, not all object fields are references. Julia considers some types to be "plain data", meaning all of their data is self-contained and does not reference other objects. The plain data types consist of primitive types (e.g. `Int`) and immutable structs of other plain data types. The initial contents of a plain data type is undefined:

```
julia> struct HasPlain  
  
    n::Int  
  
    HasPlain() = new()  
  
end  
  
julia> HasPlain()  
HasPlain(438103441441)
```

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

```
julia> mutable struct Lazy  
  
    xx
```

```
end
```

As with incomplete objects returned from constructors, if `complete_me` or any of its callees try to access the `xx` field of the `Lazy` object before it has been initialized, an error will be thrown immediately.

## 17.4 Parametric Constructors

Parametric types add a few wrinkles to the constructor story. Recall from [Parametric Types](#) that, by default, instances of parametric composite types can be constructed either with explicitly given type parameters or with type parameters implied by the types of the arguments given to the constructor. Here are some examples:

```
julia> struct Point{T<:Real}

    x::T

    y::T

end

julia> Point(1,2) ## implicit T ##
Point{Int64}(1, 2)

julia> Point(1.0,2.5) ## implicit T ##
Point{Float64}(1.0, 2.5)

julia> Point(1,2.5) ## implicit T ##
ERROR: MethodError: no method matching Point(::Int64, ::Float64)
```

```
Point(::T<:Real, !Matched::T<:Real) where T<:Real at none:2

julia> Point{Int64}(1, 2) ## explicit T ##
Point{Int64}(1, 2)

julia> Point{Int64}(1.0, 2.5) ## explicit T ##
ERROR: InexactError: convert(Int64, 2.5)
Stacktrace:
 [1] convert at ./float.jl:703 [inlined]
 [2] Point{Int64}(:Float64, ::Float64) at ./none:2

julia> Point{Float64}(1.0, 2.5) ## explicit T ##
Point{Float64}(1.0, 2.5)

julia> Point{Float64}(1, 2) ## explicit T ##
Point{Float64}(1.0, 2.0)
```

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: `Point{Int64}(1, 2)` works, but `Point{Int64}(1.0, 2.5)` raises an `InexactError` when converting 2.5 to `Int64`. When the type is implied by the arguments to the constructor call, as in `Point(1, 2)`, then the types of the arguments must agree – otherwise the T cannot be determined – but any pair of real arguments with matching type may be given to the generic `Point` constructor.

What's really going on here is that `Point`, `Point{Float64}` and `Point{Int64}` are all different constructor functions. In fact, `Point{T}` is a distinct constructor function for each type T. Without any explicitly provided inner constructors, the declaration of the composite type `Point{T<:Real}` automatically provides an inner constructor, `Point{T}`, for each possible type `T<:Real`,

It behaves just like non-parametric default constructor provides a single general outer `Point` constructor that takes pairs of real arguments, which must be of the same type. This automatic provision of constructors is equivalent to the following explicit declaration:

```
julia> struct Point{T<:Real}

    x::T

    y::T

    Point{T}(x,y) where {T<:Real} = new(x,y)

end

julia> Point(x::T, y::T) where {T<:Real} = Point{T}(x,y);
```

Notice that each definition looks like the form of constructor call that it handles. The call `Point{Int64}(1,2)` will invoke the definition `Point{T}(x,y)` inside the `type` block. The outer constructor declaration, on the other hand, defines a method for the general `Point` constructor which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `Point(1,2)` and `Point(1.0,2.5)`, work. Since the method declaration restricts the arguments to being of the same type, calls like `Point(1,2.5)`, with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `Point(1,2.5)` work by "promoting" the integer value `1` to the floating-point value `1.0`. The simplest way to achieve this is to define the following additional outer constructor method:

```
julia> Point(x::Int64, y::Float64) = Point(convert(Float64,x),y);
```

This method uses the [convert](#) function to explicitly convert `x` to `Float64` and then delegates construction to the general constructor for the case where both arguments are `Float64`. With this method definition what was previously a `MethodError` now successfully creates a point of type `Point{Float64}`:

```
julia> Point(1,2.5)
Point{Float64}(1.0, 2.5)
```

```
julia> typeof(ans)
Point{Float64}
```

However, other similar calls still don't work:

```
julia> Point(1.5,2)
ERROR: MethodError: no method matching Point(::Float64, ::Int64)
Closest candidates are:
  Point(::T<:Real, !Matched::T<:Real) where T<:Real at none:1
```

For a more general way to make all such calls work sensibly, see [Conversion and Promotion](#). At the risk of spoiling the suspense, we can reveal here that all it takes is the following outer method definition to make all calls to the general `Point` constructor work as one would expect:

```
julia> Point(x::Real, y::Real) = Point(promote(x,y)...);
```

The `promote` function converts all its arguments to a common type – in this case `Float64`. With this method definition, the `Point` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
julia> Point(1.5,2)
Point{Float64}(1.5, 2.0)
```

```
julia> Point(1,1//2)
Point{Rational{Int64}}(1//1, 1//2)

julia> Point(1.0,1//2)
Point{Float64}(1.0, 0.5)
```

Thus, while the implicit type parameter constructors provided by default in Julia are fairly strict, it is possible to make them behave in a more relaxed but sensible manner quite easily. Moreover, since constructors can leverage all of the power of the type system, methods, and multiple dispatch, defining sophisticated behavior is typically quite simple.

## 17.5 Case Study: Rational

Perhaps the best way to tie all these pieces together is to present a real world example of a parametric composite type and its constructor methods. To that end, here is the (slightly modified) beginning of `rational.jl`, which implements Julia's [Rational Numbers](#):

```
julia> struct OurRational{T<:Integer} <: Real

    num::T

    den::T

    function OurRational{T}(num::T, den::T) where
        T<:Integer

        if num == 0 && den == 0

            error("invalid rational: 0//0")
```

```
    end

    g = gcd(den, num)

    num = div(num, g)

    den = div(den, g)

    new(num, den)

end

end

julia> OurRational(n::T, d::T) where {T<:Integer} =
    OurRational{T}(n,d)
OurRational

julia> OurRational(n::Integer, d::Integer) =
    OurRational(promote(n,d)...)

OurRational

julia> OurRational(n::Integer) = OurRational(n,one(n))
OurRational

julia> //(n::Integer, d::Integer) = OurRational(n,d)
// (generic function with 1 method)

julia> //(x::OurRational, y::Integer) = x.num // (x.den*y)
// (generic function with 2 methods)
```

```
julia> //(x::Integer, y::OurRational) = (x*y.den) // y.num
// (generic function with 3 methods)

julia> //(x::Complex, y::Real) = complex(real(x)//y, imag(x)//y)
// (generic function with 4 methods)

julia> //(x::Real, y::Complex) = x*y'//real(y*y')
// (generic function with 5 methods)

julia> function //(x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end
// (generic function with 6 methods)
```

The first line – `struct OurRational{T<:Integer} <: Real` – declares that `OurRational` takes one type parameter of an integer type, and is itself a real type. The field declarations `num::T` and `den::T` indicate that the data held in a `OurRational{T}` object are a pair of integers of type `T`, one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. `OurRational` has a single inner constructor method which checks that both of `num` and `den` aren't zero and ensures that every rational is constructed in "lowest terms" with a non-negative denominator. This is accomplished by dividing the given numerator and denominator values by

~~the 5. greatest common divisor~~ of ~~our rational~~ computed using the `gcd` function. Since `gcd` turns the greatest common divisor of its arguments with sign matching the first argument (`den` here), after this division the new value of `den` is guaranteed to be non-negative. Because this is the only inner constructor for `OurRational`, we can be certain that `OurRational` objects are always constructed in this normalized form.

`OurRational` also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter `T` from the type of the numerator and denominator when they have the same type. The second applies when the given numerator and denominator values have different types: it promotes them to a common type and then delegates construction to the outer constructor for arguments of matching type. The third outer constructor turns integer values into rationals by supplying a value of 1 as the denominator.

Following the outer constructor definitions, we have a number of methods for the `//` operator, which provides a syntax for writing rationals. Before these definitions, `//` is a completely undefined operator with only syntax and no meaning. Afterwards, it behaves just as described in [Rational Numbers](#) – its entire behavior is defined in these few lines. The first and most basic definition just makes `a//b` construct a `OurRational` by applying the `OurRational` constructor to `a` and `b` when they are integers. When one of the operands of `//` is already a rational number, we construct a new rational for the resulting ratio slightly differently; this behavior is actually identical to division of a rational with an integer. Finally, applying `//` to complex integral values creates an instance of `Complex{OurRational}` – a complex number whose real and imaginary parts are rationals:

```
julia> ans = (1 + 2im)//(1 - 2im);
```

276  
Julia> `typeof(ans)`

`Complex{OurRational{Int64}}`

Julia> `ans <: Complex{OurRational}`

`false`

## CHAPTER 17. CONSTRUCTORS

Thus, although the `//` operator usually returns an instance of `OurRational`, if either of its arguments are complex integers, it will return an instance of `Complex{OurRational}` instead. The interested reader should consider perusing the rest of `rational.jl`: it is short, self-contained, and implements an entire basic Julia type.

## 17.6 Constructors and Conversion

Constructors `T(args...)` in Julia are implemented like other callable objects: methods are added to their types. The type of a type is `Type`, so all constructor methods are stored in the method table for the `Type` type. This means that you can declare more flexible constructors, e.g. constructors for abstract types, by explicitly defining methods for the appropriate types.

However, in some cases you could consider adding methods to `Base.convert` instead of defining a constructor, because Julia falls back to calling `convert` if no matching constructor is found. For example, if no constructor `T(args...) = ...` exists `Base.convert(::Type{T}, args...) = ...` is called.

`convert` is used extensively throughout Julia whenever one type needs to be converted to another (e.g. in assignment, `ccall`, etcetera), and should generally only be defined (or successful) if the conversion is lossless. For example, `convert(Int, 3.0)` produces 3, but `convert(Int, 3.2)` throws an `InexactError`. If you want to define a constructor for a lossless conversion from one type to another, you should probably define a `convert` method instead.

On the other hand if `convert` does not represent a lossless conversion, or doesn't represent "conversion" at all, it is better to leave it as a constructor rather than a `convert` method. For example, the `Array{Int}()` constructor creates a zero-dimensional `Array` of the type `Int`, but is not really a "conversion" from `Int` to an `Array`.

## 17.7 Outer-only constructors

As we have seen, a typical parametric type has inner constructors that are called when type parameters are known; e.g. they apply to `Point{Int}` but not to `Point`. Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a `Point{Int}` from the call `Point(1, 2)`. Outer constructors call inner constructors to do the core work of making an instance. However, in some cases one would rather not provide inner constructors, so that specific type parameters cannot be requested manually.

For example, say we define a type that stores a vector along with an accurate representation of its sum:

```
julia> struct SummedArray{T<:Number,S<:Number}

    data::Vector{T}

    sum::S

end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
SummedArray{Int32,Int32}(Int32[1, 2, 3], 6)
```

The problem is that we want `S` to be a larger type than `T`, so that we can sum

many elements with less information loss. For example, when CHAPTER 17. CONSTRUCTORS would like `S` to be `Int64`. Therefore we want to avoid an interface that allows the user to construct instances of the type `SummedArray{Int32, Int32}`. One way to do this is to provide a constructor only for `SummedArray`, but inside the type definition block to suppress generation of default constructors:

```
julia> struct SummedArray{T<:Number, S<:Number}

    data::Vector{T}

    sum::S

    function SummedArray(a::Vector{T}) where T

        S = widen(T)

        new{T,S}(a, sum(S, a))

    end

end

julia> SummedArray(Int32[1; 2; 3], Int32(6))
ERROR: MethodError: no method matching
    SummedArray(::Array{Int32,1}, ::Int32)
Closest candidates are:
    SummedArray(::Array{T,1}) where T at none:5
```

This constructor will be invoked by the syntax `SummedArray(a)`. The syntax `new{T, S}` allows specifying parameters for the type to be constructed, i.e. this call will return a `SummedArray{T, S}`. `new{T, S}` can be used in any construc-

~~for. Definition b ONLY CONSTRUCTORS~~ parameters to new{} are automatically derived from the type being constructed when possible.



## Chapter 18

# Conversion and Promotion

Julia has a system for promoting arguments of mathematical operators to a common type, which has been mentioned in various other sections, including [Integers and Floating-Point Numbers](#), [Mathematical Operations and Elementary Functions](#), [Types](#), and [Methods](#). In this section, we explain how this promotion system works, as well as how to extend it to new types and apply it to functions besides built-in mathematical operators. Traditionally, programming languages fall into two camps with respect to promotion of arithmetic arguments:

Automatic promotion for built-in arithmetic types and operators. In most languages, built-in numeric types, when used as operands to arithmetic operators with infix syntax, such as `+`, `-`, `*`, and `/`, are automatically promoted to a common type to produce the expected results. C, Java, Perl, and Python, to name a few, all correctly compute the sum `1 + 1.5` as the floating-point value `2.5`, even though one of the operands to `+` is an integer. These systems are convenient and designed carefully enough that they are generally all-but-invisible to the programmer: hardly anyone consciously thinks of this promotion taking place when writing such an expression, but compilers and interpreters must perform conversion be-

as-is. Complex rules for such automatic conversions are thus inevitably part of specifications and implementations for such languages.

No automatic promotion. This camp includes Ada and ML – very "strict" statically typed languages. In these languages, every conversion must be explicitly specified by the programmer. Thus, the example expression `1 + 1.5` would be a compilation error in both Ada and ML. Instead one must write `real(1) + 1.5`, explicitly converting the integer `1` to a floating-point value before performing addition. Explicit conversion everywhere is so inconvenient, however, that even Ada has some degree of automatic conversion: integer literals are promoted to the expected integer type automatically, and floating-point literals are similarly promoted to appropriate floating-point types.

In a sense, Julia falls into the "no automatic promotion" category: mathematical operators are just functions with special syntax, and the arguments of functions are never automatically converted. However, one may observe that applying mathematical operations to a wide variety of mixed argument types is just an extreme case of polymorphic multiple dispatch – something which Julia's dispatch and type systems are particularly well-suited to handle. "Automatic" promotion of mathematical operands simply emerges as a special application: Julia comes with pre-defined catch-all dispatch rules for mathematical operators, invoked when no specific implementation exists for some combination of operand types. These catch-all rules first promote all operands to a common type using user-definable promotion rules, and then invoke a specialized implementation of the operator in question for the resulting values, now of the same type. User-defined types can easily participate in this promotion system by defining methods for conversion to and from other types, and providing a handful of promotion rules defining what types they

## 18.1 Conversion

Conversion of values to various types is performed by the `convert` function. The `convert` function generally takes two arguments: the first is a type object while the second is a value to convert to that type; the returned value is the value converted to an instance of given type. The simplest way to understand this function is to see it in action:

```
julia> x = 12
12

julia> typeof(x)
Int64

julia> convert(UInt8, x)
0x0c

julia> typeof(ans)
UInt8

julia> convert(AbstractFloat, x)
12.0

julia> typeof(ans)
Float64

julia> a = Any[1 2 3; 4 5 6]
2×3 Array{Any,2}:
 1  2  3
 4  5  6
```

```
julia> convert(Array{Float64}, a)
2×3 Array{Float64,2}:
 1.0  2.0  3.0
 4.0  5.0  6.0
```

Conversion isn't always possible, in which case a no method error is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert(AbstractFloat, "foo")
ERROR: MethodError: Cannot `convert` an object of type String to
         ↳ an object of type AbstractFloat
This may have arisen from a call to the constructor
         ↳ AbstractFloat(...),
since type constructors fall back to convert methods.
```

Some languages consider parsing strings as numbers or formatting numbers as strings to be conversions (many dynamic languages will even perform conversion for you automatically), however Julia does not: even though some strings can be parsed as numbers, most strings are not valid representations of numbers, and only a very limited subset of them are. Therefore in Julia the dedicated `parse` function must be used to perform this operation, making it more explicit.

## Defining New Conversions

To define a new conversion, simply provide a new method for `convert`. That's really all there is to it. For example, the method to convert a real number to a boolean is this:

```
convert(::Type{Bool}, x::Real) = x==0 ? false : x==1 ? true :
         ↳ throw(InexactError())
```

The type `Bool` argument of this method is a [singleton type](#), `Type{Bool}`,<sup>285</sup> the only instance of which is `Bool`. Thus, this method is only invoked when the first argument is the type value `Bool`. Notice the syntax used for the first argument: the argument name is omitted prior to the `::` symbol, and only the type is given. This is the syntax in Julia for a function argument whose type is specified but whose value is never used in the function body. In this example, since the type is a singleton, there would never be any reason to use its value within the body. When invoked, the method determines whether a numeric value is true or false as a boolean, by comparing it to one and zero:

```
julia> convert(Bool, 1)
true

julia> convert(Bool, 0)
false

julia> convert(Bool, 1im)
ERROR: InexactError: convert(Bool, 0 + 1im)
Stacktrace:
 [1] convert(::Type{Bool}, ::Complex{Int64}) at ./complex.jl:37

julia> convert(Bool, 0im)
false
```

The method signatures for conversion methods are often quite a bit more involved than this example, especially for parametric types. The example above is meant to be pedagogical, and is not the actual Julia behaviour. This is the actual implementation in Julia:

```
convert(::Type{T}, z::Complex) where {T<:Real} =
    (imag(z) == 0 ? convert(T, real(z)) : throw(InexactError()))
```

To continue our case study of Julia's `Rational` type, here are the conversions declared in `rational.jl`, right after the declaration of the type and its constructors:

```
convert(::Type{Rational{T}}, x::Rational) where {T<:Integer} =
    Rational(convert(T,x.num),convert(T,x.den))
convert(::Type{Rational{T}}, x::Integer) where {T<:Integer} =
    Rational(convert(T,x), convert(T,1))

function convert(::Type{Rational{T}}, x::AbstractFloat, tol::Real)
    where T<:Integer
        if isnan(x); return zero(T)//zero(T); end
        if isinf(x); return sign(x)//zero(T); end
        y = x
        a = d = one(T)
        b = c = zero(T)
        while true
            f = convert(T,round(y)); y -= f
            a, b, c, d = f*a+c, f*b+d, a, b
            if y == 0 || abs(a/b-x) <= tol
                return a//b
            end
            y = 1/y
        end
    end
convert(rt::Type{Rational{T}}, x::AbstractFloat) where
    {T<:Integer} = convert(rt,x,eps(x))

convert(::Type{T}, x::Rational) where {T<:AbstractFloat} =
    convert(T,x.num)/convert(T,x.den)
```

```
|18.2 PROMOTION, convert{Type{T}}, x::Rational) where {T<:Integer} =  
|   ↳ div(convert(T,x.num),convert(T,x.den))
```

287

The initial four convert methods provide conversions to rational types. The first method converts one type of rational to another type of rational by converting the numerator and denominator to the appropriate integer type. The second method does the same conversion for integers by taking the denominator to be 1. The third method implements a standard algorithm for approximating a floating-point number by a ratio of integers to within a given tolerance, and the fourth method applies it, using machine epsilon at the given value as the threshold. In general, one should have `a//b == convert(Rational{Int64}, a/b)`.

The last two convert methods provide conversions from rational types to floating-point and integer types. To convert to floating point, one simply converts both numerator and denominator to that floating point type and then divides. To convert to integer, one can use the `div` operator for truncated integer division (rounded towards zero).

## 18.2 Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values. In this sense, the term "promotion" is appropriate since the values are converted to a "greater" type – i.e. one which can represent all of the input values in a single common type. It is important, however, not to confuse this with object-oriented (structural) super-typing, or Julia's notion of abstract super-types: promotion has nothing to do with the type hierarchy, and everything to do with converting between alternate representations. For instance, although every `Int32` value can also be represented as a `Float64`

Promotion to a common "greater" type is performed in Julia by the `promote` function, which takes any number of arguments, and returns a tuple of the same number of values, converted to a common type, or throws an exception if promotion is not possible. The most common use case for promotion is to convert numeric arguments to a common type:

```
julia> promote(1, 2.5)
(1.0, 2.5)

julia> promote(1, 2.5, 3)
(1.0, 2.5, 3.0)

julia> promote(2, 3//4)
(2//1, 3//4)

julia> promote(1, 2.5, 3, 3//4)
(1.0, 2.5, 3.0, 0.75)

julia> promote(1.5, im)
(1.5 + 0.0im, 0.0 + 1.0im)

julia> promote(1 + 2im, 3//4)
(1//1 + 2//1*im, 3//4 + 0//1*im)
```

Floating-point values are promoted to the largest of the floating-point argument types. Integer values are promoted to the larger of either the native machine word size or the largest integer argument type. Mixtures of integers and floating-point values are promoted to a floating-point type big enough to hold all the values. Integers mixed with rationals are promoted to rationals.

R8.2 PROMOTION With floats are promoted to floats. Complex values mixed with real values are promoted to the appropriate kind of complex value.

That is really all there is to using promotions. The rest is just a matter of clever application, the most typical "clever" application being the definition of catch-all methods for numeric operations like the arithmetic operators `+`, `-`, `*` and `/`. Here are some of the catch-all method definitions given in `promotion.jl`:

```
+ (x::Number, y::Number) = +(promote(x,y)...)
-(x::Number, y::Number) = -(promote(x,y)...)
*(x::Number, y::Number) = *(promote(x,y)...)
/(x::Number, y::Number) = /(promote(x,y)...)
```

These method definitions say that in the absence of more specific rules for adding, subtracting, multiplying and dividing pairs of numeric values, promote the values to a common type and then try again. That's all there is to it: nowhere else does one ever need to worry about promotion to a common numeric type for arithmetic operations – it just happens automatically. There are definitions of catch-all promotion methods for a number of other arithmetic and mathematical functions in `promotion.jl`, but beyond that, there are hardly any calls to `promote` required in the Julia standard library. The most common usages of `promote` occur in outer constructors methods, provided for convenience, to allow constructor calls with mixed types to delegate to an inner type with fields promoted to an appropriate common type. For example, recall that `rational.jl` provides the following outer constructor method:

```
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)
```

This allows calls like the following to work:

```
julia> Rational(Int8(15), Int32(-5))
-3 // 1
```

```
julia> typeof(ans)  
Rational{Int32}
```

For most user-defined types, it is better practice to require programmers to supply the expected types to constructor functions explicitly, but sometimes, especially for numeric problems, it can be convenient to do promotion automatically.

## Defining Promotion Rules

Although one could, in principle, define methods for the `promote` function directly, this would require many redundant definitions for all possible permutations of argument types. Instead, the behavior of `promote` is defined in terms of an auxiliary function called `promote_rule`, which one can provide methods for. The `promote_rule` function takes a pair of type objects and returns another type object, such that instances of the argument types will be promoted to the returned type. Thus, by defining the rule:

```
promote_rule(::Type{Float64}, ::Type{Float32}) = Float64
```

one declares that when 64-bit and 32-bit floating-point values are promoted together, they should be promoted to 64-bit floating-point. The promotion type does not need to be one of the argument types, however; the following promotion rules both occur in Julia's standard library:

```
promote_rule(::Type{UInt8}, ::Type{Int8}) = Int  
promote_rule(::Type{BigInt}, ::Type{Int8}) = BigInt
```

In the latter case, the result type is `BigInt` since `BigInt` is the only type large enough to hold integers for arbitrary-precision integer arithmetic. Also note that one does not need to define both `promote_rule(::Type{A},`

18.2 Type Promotion

`promote_rule(::Type{B}, ::Type{A})` – the symmetry is implied by the way `promote_rule` is used in the promotion process.<sup>201</sup>

The `promote_rule` function is used as a building block to define a second function called `promote_type`, which, given any number of type objects, returns the common type to which those values, as arguments to `promote` should be promoted. Thus, if one wants to know, in absence of actual values, what type a collection of values of certain types would promote to, one can use `promote_type`:

```
julia> promote_type(Int8, Int64)
Int64
```

Internally, `promote_type` is used inside of `promote` to determine what type argument values should be converted to for promotion. It can, however, be useful in its own right. The curious reader can read the code in [promotion.jl](#), which defines the complete promotion mechanism in about 35 lines.

## Case Study: Rational Promotions

Finally, we finish off our ongoing case study of Julia's rational number type, which makes relatively sophisticated use of the promotion mechanism with the following promotion rules:

```
promote_rule(::Type{Rational{T}}, ::Type{S}) where
    {T<:Integer,S<:Integer} = Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{Rational{S}}) where
    {T<:Integer,S<:Integer} = Rational{promote_type(T,S)}
promote_rule(::Type{Rational{T}}, ::Type{S}) where
    {T<:Integer,S<:AbstractFloat} = promote_type(T,S)
```

The first rule says that promoting a rational number with any other integer type promotes to a rational type whose numerator/denominator type is the result of

second rule applies the same logic to two different types of rational numbers, resulting in a rational of the promotion of their respective numerator/denominator types. The third and final rule dictates that promoting a rational with a float results in the same type as promoting the numerator/denominator type with the float.

This small handful of promotion rules, together with the [conversion methods discussed above](#), are sufficient to make rational numbers interoperate completely naturally with all of Julia's other numeric types – integers, floating-point numbers, and complex numbers. By providing appropriate conversion methods and promotion rules in the same manner, any user-defined numeric type can interoperate just as naturally with Julia's predefined numerics.

# Chapter 19

## Interfaces

A lot of the power and extensibility in Julia comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors.

### 19.1 Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`. Instead of mutating objects as they are iterated over, Julia provides these three methods to keep track of the iteration state externally from the object. The `start(iter)` method returns the initial state for the iterable object `iter`. That state gets passed along to `done(iter, state)`, which tests if there are any elements remaining, and `next(iter, state)`, which returns a tuple containing the current element and an updated `state`. The `state` object can be anything, and is generally considered to be an implementation detail private to the iterable object.

Any object defines these three methods is iterable and can be used in the many functions that rely upon `iteration`. It can also be used directly in a `for`

Required methods		Brief description	CHAPTER 19. INTERFACES
<code>start(iter)</code>		Returns the initial iteration state	
<code>next(iter, state)</code>		Returns the current item and the next state	
<code>done(iter, state)</code>		Tests if there are any items remaining	
Important optional methods	Default definition	Brief description	
<code>iterator-size(IterType)</code>	<code>HasLength()</code>	One of <code>HasLength()</code> , <code>HasShape()</code> , <code>IsInfinite()</code> , or <code>SizeUnknown()</code> as appropriate	
<code>iteratorel-type(IterType)</code>	<code>HasEl-type()</code>	Either <code>EltypeUnknown()</code> or <code>HasEltype()</code> as appropriate	
<code>eltype(IterType)</code>	<code>Any</code>	The type of the items returned by <code>next()</code>	
<code>length(iter)</code>	( <code>undefined</code> )	The number of items, if known	
<code>size(iter, [dim...])</code>	( <code>undefined</code> )	The number of items in each dimension, if known	

Value returned by <code>iterator-size(IterType)</code>	Required Methods
<code>HasLength()</code>	<code>length(iter)</code>
<code>HasShape()</code>	<code>length(iter)</code> and <code>size(iter, [dim...])</code>
<code>IsInfinite()</code>	(none)
<code>SizeUnknown()</code>	(none)

Value returned by <code>iteratorel-type(IterType)</code>	Required Methods
<code>HasEl-type()</code>	<code>eltype(IterType)</code>
<code>EltypeUnknown()</code>	(none)

loop since the syntax:

```
for i in iter  # or "for i = iter"
    # body
end
```

is translated into:

```
state = start(iter)
while !done(iter, state)
```

```
19.1. (i, state) = next(iter, state)
    # body
end
```

295

A simple example is an iterable sequence of square numbers with a defined length:

```
julia> struct Squares

    count::Int

end

julia> Base.start(::Squares) = 1

julia> Base.next(S::Squares, state) = (state*state, state+1)

julia> Base.done(S::Squares, state) = state > S.count

julia> Base.eltype(::Type{Squares}) = Int # Note that this is
→ defined for the type

julia> Base.length(S::Squares) = S.count
```

With only `start`, `next`, and `done` definitions, the `Squares` type is already pretty powerful. We can iterate over all the elements:

```
julia> for i in Squares(7)

    println(i)

end
```

```
4  
9  
16  
25  
36  
49
```

We can use many of the builtin methods that work with iterables, like `in`, `mean` and `std`:

```
julia> 25 in Squares(10)  
true
```

```
julia> mean(Squares(100))  
3383.5
```

```
julia> std(Squares(100))  
3024.355854282583
```

There are a few more methods we can extend to give Julia more information about this iterable collection. We know that the elements in a `Squares` sequence will always be `Int`. By extending the `eltype` method, we can give that information to Julia and help it make more specialized code in the more complicated methods. We also know the number of elements in our sequence, so we can extend `length`, too.

Now, when we ask Julia to `collect` all the elements into an array it can preallocate a `Vector{Int}` of the right size instead of blindly `push!`ing each element into a `Vector{Any}`:

```
julia> collect(Squares(10))' # transposed to save space
```

```
| 1×10 RowVector{Int64,Array{Int64,1}}:
```

```
| 1  4  9  16 25 36 49 64 81 100
```

While we can rely upon generic implementations, we can also extend specific methods where we know there is a simpler algorithm. For example, there's a formula to compute the sum of squares, so we can override the generic iterative version with a more performant solution:

```
julia> Base.sum(S::Squares) = (n = S.count; return  
→   n*(n+1)*(2n+1)÷6)
```

```
julia> sum(Squares(1803))
```

```
1955361914
```

This is a very common pattern throughout the Julia standard library: a small set of required methods define an informal interface that enable many fancier behaviors. In some cases, types will want to additionally specialize those extra behaviors when they know a more efficient algorithm can be used in their specific case.

It is also often useful to allow iteration over a collection in reverse order by iterating over `Iterators.reverse(iterator)`. To actually support reverse-order iteration, however, an iterator type `T` needs to implement `start`, `next`, and `done` methods for `Iterators.Reverse{T}`. (Given `r::Iterators.Reverse{T}`, the underling iterator of type `T` is `r.itr`.) In our `Squares` example, we would implement `Iterators.Reverse{Squares}` methods:

```
julia> Base.start(rS::Iterators.Reverse{Squares}) = rS.itr.count
```

```
julia> Base.next(::Iterators.Reverse{Squares}, state) =  
→   (state*state, state-1)
```

298

CHAPTER 19 INTERFACES

```
julia> Base.done(::Iterators.Reverse{Squares}, state) = state <=
julia> collect(Iterators.reverse(Squares(10)))' # transposed to
   → save space
1×10 RowVector{Int64,Array{Int64,1}}:
 100  81  64  49  36  25  16  9  4  1
```

## 19.2 Indexing

Methods to implement	Brief description
<code>getindex(X, i)</code>	<code>X[i]</code> , indexed element access
<code>setindex!(X, v, i)</code>	<code>X[i] = v</code> , indexed assignment
<code>endof(X)</code>	The last index, used in <code>X[end]</code>

For the `Squares` iterable above, we can easily compute the `i`th element of the sequence by squaring it. We can expose this as an indexing expression `S[i]`. To opt into this behavior, `Squares` simply needs to define `getindex`:

```
julia> function Base.getindex(S::Squares, i::Int)
           1 <= i <= S.count || throw(BoundsError(S, i))
           return i*i
       end

julia> Squares(100)[23]
529
```

Additionally, to support the syntax `S[end]`, we must define `endof` to specify the last valid index:

```
julia> Base.endof(S::Squares) = length(S)
```

Note, though, that the above only defines `getindex` with one integer index. Indexing with anything other than an `Int` will throw a `MethodError` saying that there was no matching method. In order to support indexing with ranges or vectors of `Ints`, separate methods must be written:

```
julia> Base.getindex(S::Squares, i::Number) = S[convert(Int, i)]
```

```
julia> Base.getindex(S::Squares, I) = [S[i] for i in I]
```

```
julia> Squares(10)[[3,4..,5]]
```

```
3-element Array{Int64,1}:
```

```
 9
```

```
16
```

```
25
```

While this is starting to support more of the [indexing operations supported by some of the builtin types](#), there's still quite a number of behaviors missing. This `Squares` sequence is starting to look more and more like a vector as we've added behaviors to it. Instead of defining all these behaviors ourselves, we can officially define it as a subtype of an [AbstractArray](#).

## 19.3 Abstract Arrays

If a type is defined as a subtype of `AbstractArray`, it inherits a very large set of rich behaviors including iteration and multidimensional indexing built on top of single-element access. See the [arrays manual page](#) and [standard library section](#) for more supported methods.

A key part in defining an `AbstractArray` subtype is `IndexStyle`. Since indexing is such an important part of an array and often occurs in hot loops, it's

Important to make both indexing and indexed assignment efficient and safe. Array data structures are typically defined in one of two ways: either it most efficiently accesses its elements using just one index (linear indexing) or it intrinsically accesses the elements with indices specified for every dimension. These two modalities are identified by Julia as `IndexLinear()` and `IndexCartesian()`. Converting a linear index to multiple indexing subscripts is typically very expensive, so this provides a traits-based mechanism to enable efficient generic code for all array types.

This distinction determines which scalar indexing methods the type must define. `IndexLinear()` arrays are simple: just define `getindex(A::ArrayType, i::Int)`. When the array is subsequently indexed with a multidimensional set of indices, the fallback `getindex(A::AbstractArray, I...)( )` efficiently converts the indices into one linear index and then calls the above method. `IndexCartesian()` arrays, on the other hand, require methods to be defined for each supported dimensionality with `ndims(A)` `Int` indices. For example, the built-in `SparseMatrixCSC` type only supports two dimensions, so it just defines `getindex(A::SparseMatrixCSC, i::Int, j::Int)`. The same holds for `setindex!`.

Returning to the sequence of squares from above, we could instead define it as a subtype of an `AbstractArray{Int, 1}`:

```
julia> struct SquaresVector <: AbstractArray{Int, 1}

    count::Int

end

julia> Base.size(S::SquaresVector) = (S.count,)

julia> Base.IndexStyle(::Type{<:SquaresVector}) = IndexLinear()
```

```
julia> Base.getindex(S::SquaresVector, i::Int) = i*i
```

Note that it's very important to specify the two parameters of the `AbstractArray`; the first defines the `eltype`, and the second defines the `ndims`. That supertype and those three methods are all it takes for `SquaresVector` to be an iterable, indexable, and completely functional array:

```
julia> s = SquaresVector(7)
```

```
7-element SquaresVector:
```

```
1  
4  
9  
16  
25  
36  
49
```

```
julia> s[s .> 20]
```

```
3-element Array{Int64,1}:
```

```
25  
36  
49
```

```
julia> s \ [1 2; 3 4; 5 6; 7 8; 9 10; 11 12; 13 14]
```

```
1×2 RowVector{Float64,Array{Float64,1}}:
```

```
0.305389  0.335329
```

```
julia> s * s # dot(s, s)
```

```
4676
```

As a more complicated example, let's define our own toy N-dimensional sparse-

```
julia> struct SparseArray{T,N} <: AbstractArray{T,N}

    data::Dict{NTuple{N,Int}, T}

    dims::NTuple{N, Int}

end

julia> SparseArray(::Type{T}, dims::Int...) where {T} =
→ SparseArray(T, dims);

julia> SparseArray(::Type{T}, dims::NTuple{N,Int}) where {T,N} =
→ SparseArray{T,N}(Dict{NTuple{N,Int}}, T()(), dims);

julia> Base.size(A::SparseArray) = A.dims

julia> Base.similar(A::SparseArray, ::Type{T}, dims::Dims) where
→ {T} = SparseArray(T, dims)

julia> Base.getindex(A::SparseArray{T,N}, I::Vararg{Int,N}) where
→ {T,N} = get(A.data, I, zero(T))

julia> Base.setindex!(A::SparseArray{T,N}, v, I::Vararg{Int,N})
→ where {T,N} = (A.data[I] = v)
```

Notice that this is an `IndexCartesian` array, so we must manually define `getindex` and `setindex!` at the dimensionality of the array. Unlike the `SquaresVector`, we are able to define `setindex!`, and so we can mutate the array:

1.9.3 ABSTRACT ARRAYS  
**julia>** A = SparseArray(Float64, 3, 3)

303

3×3 SparseArray{Float64,2}:

```
0.0 0.0 0.0
0.0 0.0 0.0
0.0 0.0 0.0
```

**julia>** fill!(A, 2)

3×3 SparseArray{Float64,2}:

```
2.0 2.0 2.0
2.0 2.0 2.0
2.0 2.0 2.0
```

**julia>** A[:] = 1:length(A); A

3×3 SparseArray{Float64,2}:

```
1.0 4.0 7.0
2.0 5.0 8.0
3.0 6.0 9.0
```

The result of indexing an `AbstractArray` can itself be an array (for instance when indexing by an `AbstractRange`). The `AbstractArray` fallback methods use `similar` to allocate an `Array` of the appropriate size and element type, which is filled in using the basic indexing method described above. However, when implementing an array wrapper you often want the result to be wrapped as well:

**julia>** A[1:2,:]

2×3 SparseArray{Float64,2}:

```
1.0 4.0 7.0
2.0 5.0 8.0
```

In this example it is accomplished by defining `Base.similar{T}(A::SparseArray, ::Type{T}, dims::Dims)` to create the appropriate wrapped array.

Note that while `similar` supports 1- and 2-arg forms, in these cases you only need to specialize the 3-argument form.) For this to work it's important that `SparseArray` is mutable (supports `setindex!`). Defining `similar`, `getindex` and `setindex!` for `SparseArray` also makes it possible to `copy` the array:

```
julia> copy(A)
3×3 SparseArray{Float64,2}:
 1.0  4.0  7.0
 2.0  5.0  8.0
 3.0  6.0  9.0
```

In addition to all the iterable and indexable methods from above, these types can also interact with each other and use most of the methods defined in the standard library for `AbstractArrays`:

```
julia> A[SquaresVector(3)]
3-element SparseArray{Float64,1}:
 1.0
 4.0
 9.0

julia> dot(A[:,1],A[:,2])
32.0
```

If you are defining an array type that allows non-traditional indexing (indices that start at something other than 1), you should specialize `indices`. You should also specialize `similar` so that the `dims` argument (ordinarily a `Dims` size-tuple) can accept `AbstractUnitRange` objects, perhaps range-types `Ind` of your own design. For more information, see [Arrays with custom indices](#).

19 Methods ABSTRACT ARRAYS implement		Brief description	305
<code>size(A)</code>		Returns a tuple containing the dimensions of A	
<code>getindex(A, i::Int)</code>		(if IndexLinear) Linear scalar indexing	
<code>getindex(A, I::Vararg{Int, N})</code>		(if IndexCartesian, where N = <code>ndims(A)</code> ) N-dimensional scalar indexing	
<code>setindex!(A, v, i::Int)</code>		(if IndexLinear) Scalar indexed assignment	
<code>setindex!(A, v, I::Vararg{Int, N})</code>		(if IndexCartesian, where N = <code>ndims(A)</code> ) N-dimensional scalar indexed assignment	
Optional methods	Default definition	Brief description	
<code>IndexCartesian(::Type)</code>	<code>IndexCartesian()</code>	Returns either <code>IndexLinear()</code> or <code>IndexCartesian()</code> . See the description below.	
<code>getindex(A, I...)</code>	defined in terms of scalar <code>getindex</code>	Multidimensional and nonscalar indexing	
<code>setindex!(A, I...)</code>	defined in terms of scalar <code>setindex!</code>	Multidimensional and nonscalar indexed assignment	
<code>start/next/done</code>	defined in terms of scalar <code>getindex</code>	Iteration	
<code>length(A)</code>	<code>prod(size(A))</code>	Number of elements	
<code>similar(A)</code>	<code>similar(A, eltype(A), size(A))</code>	Return a mutable array with the same shape and element type	
<code>similar(A, ::Type{S})</code>	<code>similar(A, S, size(A))</code>	Return a mutable array with the same shape and the specified element type	
<code>similar(A, dims::NTuple{Int})</code>	<code>similar(A, eltype(A), dims)</code>	Return a mutable array with the same element type and size dims	
<code>similar(A, ::Type{S}, dims::NTuple{Int})</code>	<code>Ar-ray{S}(dims)</code>	Return a mutable array with the specified element type and size	
Non-traditional indices	Default definition	Brief description	
<code>indices(A)</code>	<code>map(OneTo, size(A))</code>	Return the <code>AbstractUnitRange</code> of valid indices	
<code>Base.similar(A, ::Type{S}, inds::NTuple{Int})</code>	<code>similar(A, S, Base.to_shape(inds))</code>	Return a mutable array with the specified indices <code>inds</code> (see below)	
<code>Base.similar(T::Union{Type, Function},</code>	<code>T(Base.to_shape(inds))</code>	array similar to T with the specified indices <code>inds</code> (see below)	



# Chapter 20

## Modules

Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope. They are delimited syntactically, inside `module Name ... end`. Modules allow you to create top-level definitions (aka global variables) without worrying about name conflicts when your code is used together with somebody else's. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting).

The following example demonstrates the major features of modules. It is not meant to be run, but is shown for illustrative purposes:

```
module MyModule
using Lib

using BigLib: thing1, thing2

import Base.show

export MyType, foo

struct MyType
```

```
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1

show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

Note that the style is not to indent the body of the module, since that would typically lead to whole files being indented.

This module defines a type `MyType`, and two functions. Function `foo` and type `MyType` are exported, and so will be available for importing into other modules. Function `bar` is private to `MyModule`.

The statement `using Lib` means that a module called `Lib` will be available for resolving names as needed. When a global variable is encountered that has no definition in the current module, the system will search for it among variables exported by `Lib` and import it if it is found there. This means that all uses of that global within the current module will resolve to the definition of that variable in `Lib`.

The statement `using BigLib: thing1, thing2` is a syntactic shortcut for `using BigLib.thing1, BigLib.thing2`.

The `import` keyword supports all the same syntax as `using`, but only operates on a single name at a time. It does not add modules to be searched the way `using` does. `import` also differs from `using` in that functions must be imported using `import` to be extended with new methods.

In `MyModule` above we wanted to add a method to the standard `show` function, so we had to write `import Base.show`. Functions whose names are only visible via `using` cannot be extended.

**One Summary Of Module Usage** using or import, a module may 309  
create its own variable with the same name. Imported variables are read-  
only; assigning to a global variable always affects a variable owned by the  
current module, or else raises an error.

## 20.1 Summary of module usage

To load a module, two main keywords can be used: `using` and `import`. To understand their differences, consider the following example:

```
module MyModule

export x, y

x() = "x"
y() = "y"
p() = "p"

end
```

In this module we export the `x` and `y` functions (with the keyword `export`), and also have the non-exported function `p`. There are several different ways to load the Module and its inner functions into the current workspace:

### Modules and files

Files and file names are mostly unrelated to modules; modules are associated only with module expressions. One can have multiple files per module, and multiple modules per file:

```
module Foo

include("file1.jl")
```

Import Command	What is brought into scope	CHARACTERISTICS OF MODULES
<code>using MyModule</code>	All exported names (x and y), <code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>
<code>using MyModule.x, MyModule.p</code>	x and p	
<code>using MyModule: x, p</code>	x and p	
<code>import MyModule</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>	<code>MyModule.x</code> , <code>MyModule.y</code> and <code>MyModule.p</code>
<code>import MyModule.x, MyModule.p</code>	x and p	x and p
<code>import MyModule: x, p</code>	x and p	x and p

```
include("file2.jl")
```

```
end
```

Including the same code in different modules provides mixin-like behavior. One could use this to run the same code with different base definitions, for example testing code by running it with "safe" versions of some operators:

```
module Normal
include("mycode.jl")
end

module Testing
include("safe_operators.jl")
include("mycode.jl")
end
```

There are three important standard modules: Main, Core, and Base.

Main is the top-level module, and Julia starts with Main set as the current module. Variables defined at the prompt go in Main, and `whos()` lists variables in Main.

Core contains all identifiers considered "built in" to the language, i.e. part of the core language and not libraries. Every module implicitly specifies `using Core`, since you can't do anything without those definitions.

Base is the standard library (the contents of `base/`). All modules implicitly contain `using Base`, since this is needed in the vast majority of cases.

### Default top-level definitions and bare modules

In addition to `using Base`, modules also automatically contain a definition of the `eval` function, which evaluates expressions within the context of that module.

If these default definitions are not wanted, modules can be defined using the keyword `baremodule` instead (note: Core is still imported, as per above). In terms of `baremodule`, a standard module looks like this:

```
baremodule Mod

    using Base

    eval(x) = Core.eval(Mod, x)
    eval(m, x) = Core.eval(m, x)

    ...

end
```

Given the statement `using Foo`, the system consults an internal table of top-level modules to look for one named `Foo`. If the module does not exist, the system attempts to `require(:Foo)`, which typically results in loading code from an installed package.

However, some modules contain submodules, which means you sometimes need to access a non-top-level module. There are two ways to do this. The first is to use an absolute path, for example `using Base.Sort`. The second is to use a relative path, which makes it easier to import submodules of the current module or any of its enclosing modules:

```
module Parent

  module Utils
    ...
  end

  using .Utils

  ...
end
```

Here module `Parent` contains a submodule `Utils`, and code in `Parent` wants the contents of `Utils` to be visible. This is done by starting the `using` path with a period. Adding more leading periods moves up additional levels in the module hierarchy. For example `using ..Utils` would look for `Utils` in `Parent`'s enclosing module rather than in `Parent` itself.

Note that relative-import qualifiers are only valid in `using` and `import` statements.

The global variable `LOAD_PATH` contains the directories Julia searches for modules when calling `require`. It can be extended using `push!`:

```
|push!(LOAD_PATH, "/Path/To/My/Module/")
```

Putting this statement in the file `~/.juliarc.jl` will extend `LOAD_PATH` on every Julia startup. Alternatively, the module load path can be extended by defining the environment variable `JULIA_LOAD_PATH`.

## Namespace miscellanea

If a name is qualified (e.g. `Base.sin`), then it can be accessed even if it is not exported. This is often useful when debugging. It can also have methods added to it by using the qualified name as the function name. However, due to syntactic ambiguities that arise, if you wish to add methods to a function in a different module whose name contains only symbols, such as an operator, `Base.+` for example, you must use `Base.:+` to refer to it. If the operator is more than one character in length you must surround it in brackets, such as: `Base.:(==)`.

Macro names are written with `@` in import and export statements, e.g. `import Mod.@mac`. Macros in other modules can be invoked as `Mod.@mac` or `@Mod.mac`.

The syntax `M.x = y` does not work to assign a global in another module; global assignment is always module-local.

A variable name can be "reserved" without assigning to it by declaring it as `global x`. This prevents name conflicts for globals initialized after load time.

Large modules can take several seconds to load because executing all of the statements in a module often involves compiling a large amount of code. Julia provides the ability to create precompiled versions of modules to reduce this time.

To create an incremental precompiled module file, add `__precompile__()` at the top of your module file (before the `module` starts). This will cause it to be automatically compiled the first time it is imported. Alternatively, you can manually call `Base.compilecache(modulename)`. The resulting cache files will be stored in `Base.LOAD_CACHE_PATH[1]`. Subsequently, the module is automatically recompiled upon `import` whenever any of its dependencies change; dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by `include_dependency(path)` in the module file(s).

For file dependencies, a change is determined by examining whether the modification time (`mtime`) of each file loaded by `include` or added explicitly by `include_dependency` is unchanged, or equal to the modification time truncated to the nearest second (to accommodate systems that can't copy `mtime` with sub-second accuracy). It also takes into account whether the path to the file chosen by the search logic in `require` matches the path that had created the precompile file.

It also takes into account the set of dependencies already loaded into the current process and won't recompile those modules, even if their files change or disappear, in order to avoid creating incompatibilities between the running system and the precompile cache. If you want to have changes to the source reflected in the running system, you should call `reload("Module")` on the module you changed, and any module that depended on it in which you want to see the change reflected.

**2.1. SUMMARY OF MODULE USAGE** By precompiling any modules that are imported therein. If you know that it is not safe to precompile your module (for the reasons described below), you should put `__precompile__(false)` in the module file to cause `Base.compilecache` to throw an error (and thereby prevent the module from being imported by any other precompiled module).

`__precompile__()` should not be used in a module unless all of its dependencies are also using `__precompile__()`. Failure to do so can result in a runtime error when loading the module.

In order to make your module work with precompilation, however, you may need to change your module to explicitly separate any initialization steps that must occur at runtime from steps that can occur at compile time. For this purpose, Julia allows you to define an `__init__()` function in your module that executes any initialization steps that must occur at runtime. This function will not be called during compilation (`--output-*` or `__precompile__()`). You may, of course, call it manually if necessary, but the default is to assume this function deals with computing state for the local machine, which does not need to be – or even should not be – captured in the compiled image. It will be called after the module is loaded into a process, including if it is being loaded into an incremental compile (`--output-incremental=yes`), but not if it is being loaded into a full-compilation process.

In particular, if you define a function `__init__()` in a module, then Julia will call `__init__()` immediately after the module is loaded (e.g., by `import`, `using`, or `require`) at runtime for the first time (i.e., `__init__` is only called once, and only after all statements in the module have been executed). Because it is called after the module is fully imported, any submodules or other imported modules have their `__init__` functions called before the `__init__` of the enclosing module.

Two typical uses of `__init__` are calling runtime initialization functions of ex-

816 CHAPTER 20. MODULES

Refer C libraries and initializing global constants that are pointers by external libraries. For example, suppose that we are calling a C library `libfoo` that requires us to call a `foo_init()` initialization function at runtime. Suppose that we also want to define a global constant `foo_data_ptr` that holds the return value of a `void *foo_data()` function defined by `libfoo` – this constant must be initialized at runtime (not at compile time) because the pointer address will change from run to run. You could accomplish this by defining the following `__init__` function in your module:

```
const foo_data_ptr = Ref{Ptr{Void}}(0)
function __init__()
    ccall((:foo_init, :libfoo), Void, ())
    foo_data_ptr[] = ccall((:foo_data, :libfoo), Ptr{Void}, ())
    nothing
end
```

Notice that it is perfectly possible to define a global inside a function like `__init__`; this is one of the advantages of using a dynamic language. But by making it a constant at global scope, we can ensure that the type is known to the compiler and allow it to generate better optimized code. Obviously, any other globals in your module that depends on `foo_data_ptr` would also have to be initialized in `__init__`.

Constants involving most Julia objects that are not produced by `ccall` do not need to be placed in `__init__`: their definitions can be precompiled and loaded from the cached module image. This includes complicated heap-allocated objects like arrays. However, any routine that returns a raw pointer value must be called at runtime for precompilation to work (Ptr objects will turn into null pointers unless they are hidden inside an `isbits` object). This includes the return values of the Julia functions `cfunction` and `pointer`.

**Dictionary Summary of `Method` and `Edge`** anything that depends on the output of a `hash(key)` method, are a trickier case. In the common case where the keys are numbers, strings, symbols, ranges, `Expr`, or compositions of these types (via arrays, tuples, sets, pairs, etc.) they are safe to precompile. However, for a few other key types, such as `Function` or `DataType` and generic user-defined types where you haven't defined a `hash` method, the fallback hash method depends on the memory address of the object (via its `object_id`) and hence may change from run to run. If you have one of these key types, or if you aren't sure, to be safe you can initialize this dictionary from within your `__init__` function. Alternatively, you can use the `ObjectIdDict` dictionary type, which is specially handled by precompilation so that it is safe to initialize at compile-time.

When using precompilation, it is important to keep a clear sense of the distinction between the compilation phase and the execution phase. In this mode, it will often be much more clearly apparent that Julia is a compiler which allows execution of arbitrary Julia code, not a standalone interpreter that also generates compiled code.

Other known potential failure scenarios include:

1. Global counters (for example, for attempting to uniquely identify objects)  
Consider the following code snippet:

```
mutable struct UniquedById
    myid::Int
    let counter = 0
        UniquedById() = new(counter += 1)
    end
end
```

while the intent of this code was to give every instance a unique id, the counter value is recorded at the end of compilation. All subsequent

318 usages of this incrementally compiled module CHAPTER 20. MODULES  
counter value.

Note that `object_id` (which works by hashing the memory pointer) has similar issues (see notes on `Dict` usage below).

One alternative is to use a macro to capture `@__MODULE__` and store it alone with the current `counter` value, however, it may be better to redesign the code to not depend on this global state.

2. Associative collections (such as `Dict` and `Set`) need to be re-hashed in `__init__`. (In the future, a mechanism may be provided to register an initializer function.)
3. Depending on compile-time side-effects persisting through load-time. Example include: modifying arrays or other variables in other Julia modules; maintaining handles to open files or devices; storing pointers to other system resources (including memory);
4. Creating accidental "copies" of global state from another module, by referencing it directly instead of via its lookup path. For example, (in global scope):

```
#mystdout = Base.STDOUT #= will not work correctly, since this
→ will copy Base.STDOUT into this module =#
# instead use accessor functions:
getstdout() = Base.STDOUT #= best option =#
# or move the assignment into the runtime:
__init__() = global mystdout = Base.STDOUT #= also works =#
```

Several additional restrictions are placed on the operations that can be done while precompiling code to help the user avoid other wrong-behavior situations:

20.1 Calling `eval` to Embed Usage

in another module. This will also cause a warning to be emitted when the incremental precompile flag is set.

2. `global const` statements from local scope after `__init__( )` has been started (see issue #12010 for plans to add an error for this)
3. Replacing a module (or calling `workspace( )`) is a runtime error while doing an incremental precompile.

A few other points to be aware of:

1. No code reload / cache invalidation is performed after changes are made to the source files themselves, (including by `Pkg.update`), and no cleanup is done after `Pkg.rm`
2. The memory sharing behavior of a reshaped array is disregarded by pre-compilation (each view gets its own copy)
3. Expecting the filesystem to be unchanged between compile-time and runtime e.g. `@__FILE__/source_path( )` to find resources at runtime, or the BinDeps `@checked_lib` macro. Sometimes this is unavoidable. However, when possible, it can be good practice to copy resources into the module at compile-time so they won't need to be found at runtime.
4. `WeakRef` objects and finalizers are not currently handled properly by the serializer (this will be fixed in an upcoming release).
5. It is usually best to avoid capturing references to instances of internal metadata objects such as `Method`, `MethodInstance`, `MethodTable`, `TypeMapLevel`, `TypeMapEntry` and fields of those objects, as this can confuse the serializer and may not lead to the outcome you desire. It is not necessarily an error to do this, but you simply need to be prepared that the system will try to copy some of these and to create a single unique instance of others.

320sometimes helpful during module development to CHAPTER 20 MODULES compilation. The command line flag `--compiled-modules={yes|no}` enables you to toggle module precompilation on and off. When Julia is started with `--compiled-modules=no` the serialized modules in the compile cache are ignored when loading modules and module dependencies. `Base.compilecache` can still be called manually and it will respect `__precompile__()` directives for the module. The state of this command line flag is passed to `Pkg.build` to disable automatic precompilation triggering when installing, updating, and explicitly building packages.

# Chapter 21

## Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system since Julia 0.4.

The basic syntax is very simple: any string appearing at the top-level right before an object (function, macro, type or instance) will be interpreted as documenting it (these are called docstrings). Here is a very simple example:

```
"Tell whether there are too foo items in the array."  
foo(xs::Array) = ...
```

Documentation is interpreted as [Markdown](#), so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the `@doc` macro just as well.

Here is a more complex example, still using Markdown:

```
"""  
bar(x[, y])  
  
Compute the Bar index between `x` and `y`. If `y` is missing,  
→ compute
```

```
# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
...
```
function bar(x, y) ...
```

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

This can be identical to the signature present in the Julia code (like `mean(x::AbstractArray)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `f(x, y=1)`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `f(x[, y])` and `f(x[, y[, z]])`). An alternative solution is to use several lines: one without optional arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a `<keyword arguments>` placeholder in the signature (i.e. `f(x; <keyword arguments>)`), and give the complete list under an `# Arguments` section (see point 4 below).

2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

The one-line sentence should use the imperative form ("Do this", "Return that") instead of the third person (do not write "Returns the length...") when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).

### 3. Do not repeat yourself.

Since the function name is given by the signature, there is no need to start the documentation with "The function **bar**...": go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.

### 4. Only provide an argument list when really necessary.

For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function's purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular keyword arguments). In that case, insert it after the general description of the function, under an **# Arguments** header, with one - bullet for each argument. The list should mention the types and default values (if any) of the arguments:

```
"""
...
# Arguments
- `n::Integer`: the number of elements to compute.
- `dim::Integer=1`: the dimensions along which to perform the
  computation.
...
"""
```

Sometimes there are functions of related functionality. To increase discoverability please provide a short list of these in a `See also:` paragraph.

```
| See also: [ `bar!` ](@ref), [ `baz` ](@ref), [ `baaz` ](@ref)
```

## 6. Include any code examples in an `# Examples` section.

Examples should, whenever possible, be written as doctests. A doctest is a fenced code block (see [Code blocks](#)) starting with ````jldoctest` and contains any number of `julia>` prompts together with inputs and expected outputs that mimic the Julia REPL.

For example in the following docstring a variable `a` is defined and the expected result, as printed in a Julia REPL, appears afterwards:

```
"""
Some nice documentation here.

# Examples
```jldoctest
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```
"""


```

### Warning

Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly

construct and seed your own [MersenneTwister](#) (or other pseudorandom number generator) and pass it to the functions you are doctesting.

Operating system word size ([Int32](#) or [Int64](#)) as well as path separator differences (/ or \) will also affect the reproducibility of some doctests.

Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

You can then run `make -C doc doctest` to run all the doctests in the Julia Manual, which will ensure that your example works.

To indicate that the output result is truncated, you may write [...] at the line where checking should stop. This is useful to hide a stacktrace (which contains non-permanent references to lines of julia code) when the doctest shows that an exception is thrown, for example:

```
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
[...]
```

```

Examples that are untestable should be written within fenced code blocks starting with ```julia so that they are highlighted correctly in the generated documentation.

### Tip

Wherever possible examples should be self-contained and runnable so that readers are able to try them out without having to include any dependencies.

Julia identifiers and code excerpts should always appear between backticks ` to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks ``. Use Unicode characters rather than their LaTeX escape sequence, i.e. `` $a = 1$ `` rather than ``\\alpha = 1``.

8. Place the starting and ending " " " characters on lines by themselves.

That is, write:

```
"""
...
...
...
f(x, y) = ...
```

rather than:

```
""" ...
...
f(x, y) = ...
```

This makes it more clear where docstrings start and end.

9. Respect the line length limit used in the surrounding code.

Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It is advised to add line breaks after 92 characters.

Documentation can be accessed at the REPL or in [IJulia](#) by typing `? followed by the name of a function or macro, and pressing Enter.` For example,

```
?cos  
?@time  
?r""
```

will bring up docs for the relevant function, macro or string macro respectively. In [Juno](#) using `Ctrl-J`, `Ctrl-D` will bring up documentation for the object under the cursor.

## 21.2 Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general, only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by `function bar end`). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere.

For example:

```
"""  
*(x, y, z...)  
  
Multiplication operator. `x * y * z *...` calls this function with  
→ multiple  
arguments, i.e. `*(x, y, z...)`.  
"""  
function *(x, y, z...)
```

```
328 # ... [implementation sold separately] ...
end
```

```
"""

```

```
*(x::AbstractString, y::AbstractString, z::AbstractString...)
```

When applied to strings, concatenates them.

```
"""

```

```
function *(x::AbstractString, y::AbstractString,
→ z::AbstractString...)
    # ... [insert secret sauce here] ...
end
```

```
help?> *
```

```
search: * .*
```

```
*(x, y, z...)
```

Multiplication operator.  $x * y * z * \dots$  calls this **function** with  
→ multiple  
arguments, i.e. `*(x,y,z...)`.

```
*(x::AbstractString, y::AbstractString, z::AbstractString...)
```

When applied to strings, concatenates them.

When retrieving documentation for a generic function, the metadata for each method is concatenated with the `catdoc` function, which can of course be overridden for custom types.

The `@doc` macro associates its first argument with its second in a per-module dictionary called `META`. By default, documentation is expected to be written in Markdown, and the `doc""` string macro simply creates an object representing the Markdown content. In the future it is likely to do more advanced things such as allowing for relative image or link paths.

When used for retrieving documentation, the `@doc` macro (or equally, the `doc` function) will search all `META` dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example) will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

```
|@doc "..."\n|@doc (@doc foo!) foo
```

Or for use with Julia's metaprogramming functionality:

```
|for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*),\n|  →  (:divide, :/))\n|  @eval begin\n|    $f(a,b) = $op(a,b)\n|  end\nend\n\n@doc "`add(a,b)` adds `a` and `b` together" add\n@doc "`subtract(a,b)` subtracts `b` from `a`" subtract
```

Documentation written in non-toplevel blocks, such as `begin`, `if`, `for`, and `let`, is added to the documentation system as blocks are evaluated. For example:

```
330 if condition()
    ...
    f(x) = x
end
```

## CHAPTER 21. DOCUMENTATION

will add documentation to `f(x)` when `condition()` is `true`. Note that even if `f(x)` goes out of scope at the end of the block, its documentation will remain.

### Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to `Docs.getdoc` for your custom type that returns the documentation on a per-instance basis. For instance,

```
struct MyType
    value::String
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value
                           → $(t.value)"

x = MyType("x")
y = MyType("y")
```

?`x` will display "Documentation for MyType with value x" while ?`y` will display "Documentation for MyType with value y".

## 21.4 Syntax Guide

A comprehensive overview of all documentable Julia syntax.

In the following examples "... " is used to illustrate an arbitrary docstring which may be one of the follow four variants and contain arbitrary text:

```
doc"..."
```

```
"..."
```

```
..."
```

```
"..."
```

```
doc"""
```

```
..."
```

```
"..."
```

@doc\_str should only be used when the docstring contains \$ or \ characters that should not be parsed by Julia such as LaTeX syntax or Julia source code examples containing interpolation.

## Functions and Methods

```
"..."  
function f end  
  
"..."  
f
```

Adds docstring "..." to function f. The first version is the preferred syntax, however both are equivalent.

```
"..."  
f(x) = x  
  
"..."  
function f(x)  
    x
```

332

**end**

"..."

f(x)

## CHAPTER 21. DOCUMENTATION

Adds docstring "..." to the **Method** f(:Any).

"..."

f(x, y = 1) = x + y

Adds docstring "..." to two **Methods**, namely f(:Any) and f(:Any, ::Any).

## Macros

"..."

**macro** m(x) **end**

Adds docstring "..." to the @m(:Any) macro definition.

"..."

:(@m)

Adds docstring "..." to the macro named @m.

## Types

"..."

**abstract type** T1 **end**

"..."

**mutable struct** T2

...

**end**

```
"..."  
struct T3  
...  
end
```

Adds the docstring "..." to types T1, T2, and T3.

```
"..."  
struct T  
    "x"  
    x  
    "y"  
    y  
end
```

Adds docstring "..." to type T, "x" to field T.x and "y" to field T.y. Also applicable to `mutable struct` types.

## Modules

```
"..."  
module M end  
  
module M  
    "..."  
    M  
  
end
```

Adds docstring "..." to the `ModuleM`. Adding the docstring above the `Module` is the preferred syntax, however both are equivalent.

```
...
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

Documenting a **baremodule** by placing a docstring above the expression automatically imports `@doc` into the module. These imports must be done manually when the module expression is not documented. Empty **baremodules** cannot be documented.

## Global Variables

```
"..."
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring "..." to the Bindings a, b, and c.

**Binding Syntax Guide** To store a reference to a particular `Symbol` in a `Module` without storing the referenced value itself.

### Note

When a `const` definition is only used to define an alias of another definition, such as is the case with the function `div` and its alias `÷` in `Base`, do not document the alias and instead document the actual function.

If the alias is documented and not the real definition then the doc-system (`? mode`) will not return the docstring attached to the alias when the real definition is searched for.

For example you should write

```
"..."  
f(x) = x + 1  
const alias = f
```

rather than

```
f(x) = x + 1  
"..."  
const alias = f
```

Adds docstring "..." to the value associated with `sym`. Users should prefer documenting `sym` at its definition.

### Multiple Objects

```
"..."  
a, b
```

Adds docstring "..." to **a** and **b** each of which is a block expression. This syntax is equivalent to

```
"..."  
| a  
|  
"..."  
| b
```

Any number of expressions may be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions **f** and **f!**.

Macro-generated code

```
"..."  
| @m expression
```

Adds docstring "..." to expression generated by expanding **@m expression**. This allows for expressions decorated with **@inline**, **@noinline**, **@generated**, or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the **@@doc\_\_** macro.

The **@enum** macro makes use of **@@doc\_\_** to allow for documenting Enums. Examining its definition should serve as an example of how to use **@@doc\_\_** correctly.

[Core.@@doc\\_\\_](#) – Macro.

```
| @@doc__(ex)
```

21.5 MARKDOWN SYNTAX

should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @_doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

`source`

## 21.5 Markdown syntax

The following markdown syntax is supported in Julia.

### Inline elements

Here “inline” refers to elements that can be found within blocks of text, i.e. paragraphs. These include the following elements.

#### Bold

Surround words with two asterisks, `**`, to display the enclosed text in boldface.

A paragraph containing a `**bold**` word.

#### Italics

Surround words with one asterisk, `*`, to display the enclosed text in italics.

A paragraph containing an `*emphasised*` word.

Surround text that should be displayed exactly as written with single backticks,

A paragraph containing a `literal` word.

Literals should be used when writing text that refers to names of variables, functions, or other parts of a Julia program.

### Tip

To include a backtick character within literal text use three backticks rather than one to enclose the text.

A paragraph containing a ```` `backtick` character ````.

By extension any odd number of backticks may be used to enclose a lesser number of backticks.

### \LaTeX

Surround text that should be displayed as mathematics using \LaTeX syntax with double backticks, `` .

A paragraph containing some ``\LaTeX`` markup.

### Tip

As with literals in the previous section, if literal backticks need to be written within double backticks use an even number greater than two. Note that if a single literal backtick needs to be included within \LaTeX markup then two enclosing backticks is sufficient.

Links to either external or internal addresses can be written using the following syntax, where the text enclosed in square brackets, [ ], is the name of the link and the text enclosed in parentheses, ( ), is the URL.

```
A paragraph containing a link to [Julia](http://www.julialang.org)
```

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

```
"""
    eigvals!(A,[irange,][vl,][vu]) -> values

Same as [`eigvals`](@ref), but saves space by overwriting the
→ input `A`, instead of creating a copy.
"""
```

This will create a link in the generated docs to the `eigvals` documentation (which has more information about what this function actually does). It's good to include cross references to mutating/non-mutating versions of a function, or to highlight a difference between two similar-seeming functions.

### Note

The above cross referencing is not a Markdown feature, and relies on [Documenter.jl](#), which is used to build base Julia's documentation.

### Footnote references

Named and numbered footnote references can be written using the following syntax. A footnote name must be a single alphanumeric word containing no punctuation.

340 A paragraph containing a numbered footnote [^1] and a named one [^2] and a named one [^3].

## Note

The text associated with a footnote can be written anywhere within the same page as the footnote reference. The syntax used to define the footnote text is discussed in the [Footnotes](#) section below.

## Toplevel elements

The following elements can be written either at the "toplevel" of a document or within another "toplevel" element.

### Paragraphs

A paragraph is a block of plain text, possibly containing any number of inline elements defined in the [Inline elements](#) section above, with one or more blank lines above and below it.

This is a paragraph.

And this is \*another\* one containing some emphasised text.

A new line, but still part of the same paragraph.

### Headers

A document can be split up into different sections using headers. Headers use the following syntax:

```
# Level One  
## Level Two  
### Level Three  
#### Level Four
```

```
##### Level Five
```

```
##### Level Six
```

A header line can contain any inline syntax in the same way as a paragraph can.

Tip

Try to avoid using too many levels of header within a single document. A heavily nested document may be indicative of a need to restructure it or split it into several pages covering separate topics.

## Code blocks

Source code can be displayed as a literal block using an indent of four spaces as shown in the following example.

```
This is a paragraph.
```

```
function func(x)
    # ...
end
```

```
Another paragraph.
```

Additionally, code blocks can be enclosed using triple backticks with an optional "language" to specify how a block of code should be highlighted.

```
A code block without a "language":
```

```
```
```

```
function func(x)
    # ...
end
```
```

and another one with the "language" specified as `julia`:

```
```julia
function func(x)
    # ...
end
...``
```

### Note

"Fenced" code blocks, as shown in the last example, should be preferred over indented code blocks since there is no way to specify what language an indented code block is written in.

### Block quotes

Text from external sources, such as quotations from books or websites, can be quoted using > characters prepended to each line of the quote as follows.

Here's a quote:

```
> Julia is a high-level, high-performance dynamic programming
   language for
> technical computing, with syntax that is familiar to users of
   other
> technical computing environments.
```

Note that a single space must appear after the > character on each line. Quoted blocks may themselves contain other toplevel or inline elements.

### Images

The syntax for images is similar to the link syntax mentioned above. Prepending a ! character to a link will display an image from the specified URL rather than

```
! [alternative text] (link/to/image.png)
```

## Lists

Unordered lists can be written by prepending each item in a list with either \*, +, or -.

```
A list of items:
```

- \* item one
- \* item two
- \* item three

Note the two spaces before each \* and the single space after each one.

Lists can contain other nested toplevel elements such as lists, code blocks, or quoteblocks. A blank line should be left between each list item when including any toplevel elements within a list.

```
Another list:
```

- \* item one

- \* item two

- ...

- f(x) = x

- ...

- \* And a sublist:

- + sub-item one
- + sub-item two

The contents of each item in the list must line up with the first line of the item. In the above example the fenced code block must be indented by four spaces to align with the `i` in `item two`.

Ordered lists are written by replacing the "bullet" character, either `*`, `+`, or `-`, with a positive integer followed by either `.` or `)`.

Two ordered lists:

1. item one
2. item two
3. item three
  
- 5) item five
- 6) item six
- 7) item seven

An ordered list may start from a number other than one, as in the second list of the above example, where it is numbered from five. As with unordered lists, ordered lists can contain nested toplevel elements.

Display equations

Large  $\text{\LaTeX}$ equations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `math` as in the example below.

```
```math
f(a) = \frac{1}{2\pi}\int_{-\pi}^{\pi} (\alpha+R\cos(\theta))d\theta
```

```

This syntax is paired with the inline syntax for [Footnote references](#). Make sure to read that section as well.

Footnote text is defined using the following syntax, which is similar to footnote reference syntax, aside from the : character that is appended to the footnote label.

```
[^1]: Numbered footnote text.
```

```
[^note]:
```

Named footnote text containing several toplevel elements.

```
* item one
* item two
* item three
```

```
```julia
function func(x)
    # ...
end
...``
```

## Note

No checks are done during parsing to make sure that all footnote references have matching footnotes.

## Horizontal rules

The equivalent of an `<hr>` HTML tag can be written using the following syntax:

```
Text above the line.
```

And text below the line.

## Tables

Basic tables can be written using the syntax described below. Note that markdown tables have limited features and cannot contain nested toplevel elements unlike other elements discussed above – only inline elements are allowed. Tables must always contain a header row with column names. Cells cannot span multiple rows or columns of the table.

Column One	Column Two	Column Three
:	-----	:-----:
Row `1`	Column `2`	
*Row* 2	**Row** 2	Column ``3``

### Note

As illustrated in the above example each column of | characters must be aligned vertically.

A : character on either end of a column's header separator (the row containing - characters) specifies whether the row is left-aligned, right-aligned, or (when : appears on both ends) center-aligned.

Providing no : characters will default to right-aligning the column.

## Admonitions

Specially formatted blocks with titles such as "Notes", "Warning", or "Tips" are known as admonitions and are used when some part of a document needs special attention. They can be defined using the following !!! syntax:

!!! note

```
!!! warning "Beware!"
```

And this is another one.

This warning admonition has a custom title: ``"Beware!"``.

Admonitions, like most other toplevel elements, can contain other toplevel elements. When no title text, specified after the admonition type in double quotes, is included then the title used will be the type of the block, i.e. "Note" in the case of the `note` admonition.

## 21.6 Markdown Syntax Extensions

Julia's markdown supports interpolation in a very similar way to basic string literals, with the difference that it will store the object itself in the Markdown tree (as opposed to converting it to a string). When the Markdown content is rendered the usual `show` methods will be called, and these can be overridden as usual. This design allows the Markdown to be extended with arbitrarily complex features (such as references) without cluttering the basic syntax.

In principle, the Markdown parser itself can also be arbitrarily extended by packages, or an entirely custom flavour of Markdown can be used, but this should generally be unnecessary.



## Chapter 22

# Metaprogramming

The strongest legacy of Lisp in the Julia language is its metaprogramming support. Like Lisp, Julia represents its own code as a data structure of the language itself. Since code is represented by objects that can be created and manipulated from within the language, it is possible for a program to transform and generate its own code. This allows sophisticated code generation without extra build steps, and also allows true Lisp-style macros operating at the level of [abstract syntax trees](#). In contrast, preprocessor "macro" systems, like that of C and C++, perform textual manipulation and substitution before any actual parsing or interpretation occurs. Because all data types and code in Julia are represented by Julia data structures, powerful [reflection](#) capabilities are available to explore the internals of a program and its types just like any other data.

### 22.1 Program representation

Every Julia program starts life as a string:

```
julia> prog = "1 + 1"  
"1 + 1"
```

What happens next?

The next step is to [parse](#) each string in [CHAPTER 22. METaprogramming](#), represented by the Julia type `Expr`:

```
julia> ex1 = Meta.parse(prog)
:(1 + 1)

julia> typeof(ex1)
Expr
```

`Expr` objects contain two parts:

a `Symbol` identifying the kind of expression. A symbol is an [interned string](#) identifier (more discussion below).

```
julia> ex1.head
:call
```

the expression arguments, which may be symbols, other expressions, or literal values:

```
julia> ex1.args
3-element Array{Any,1}:
:+
1
1
```

Expressions may also be constructed directly in [prefix notation](#):

```
julia> ex2 = Expr(:call, :+, 1, 1)
:(1 + 1)
```

The two expressions constructed above – by parsing and by direct construction – are equivalent:

```
julia> ex1 == ex2
```

true

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

The `dump` function provides indented and annotated display of `Expr` objects:

```
julia> dump(ex2)
```

Expr  
  head: Symbol call  
  args: Array{Any}((3,))  
    1: Symbol +  
    2: Int64 1  
    3: Int64 1  
  typ: Any

`Expr` objects may also be nested:

```
julia> ex3 = Meta.parse("(4 + 4) / 2")  
:( (4 + 4) / 2 )
```

Another way to view expressions is with `Meta.show_sexpr`, which displays the **S-expression** form of a given `Expr`, which may look very familiar to users of Lisp. Here's an example illustrating the display on a nested `Expr`:

```
julia> Meta.show_sexpr(ex3)  
(:call, :/, (:call, :+, 4, 4), 2)
```

## Symbols

The `:` character has two syntactic purposes in Julia. The first form creates a **Symbol**, an **interned string** used as one building-block of expressions:

352

julia> :foo

:foo

julia> typeof(ans)

Symbol

## CHAPTER 22. METAPROGRAMMING

The `Symbol` constructor takes any number of arguments and creates a new symbol by concatenating their string representations together:

julia> :foo == Symbol("foo")

true

julia> Symbol("func", 10)

:func10

julia> Symbol(:var, '\_ ', "sym")

:var\_sym

In the context of an expression, symbols are used to indicate access to variables; when an expression is evaluated, a symbol is replaced with the value bound to that symbol in the appropriate `scope`.

Sometimes extra parentheses around the argument to `:` are needed to avoid ambiguity in parsing.:

julia> :(::)

:(::)

julia> :(::)

:(::)

## Quoting

The second syntactic purpose of the `:` character is to create expression objects without using the explicit `Expr` constructor. This is referred to as quoting. The `:` character, followed by paired parentheses around a single statement of Julia code, produces an `Expr` object based on the enclosed code. Here is example of the short form used to quote an arithmetic expression:

```
julia> ex = :(a+b*c+1)  
:(a + b * c + 1)
```

```
julia> typeof(ex)  
Expr
```

(to view the structure of this expression, try `ex.head` and `ex.args`, or use `dump` as above or `Meta.@dump`)

Note that equivalent expressions may be constructed using `Meta.parse` or the direct `Expr` form:

```
julia>      :(a + b*c + 1)      ==  
  
          Meta.parse("a + b*c + 1") ==  
  
          Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)  
true
```

Expressions provided by the parser generally only have symbols, other expressions, and literal values as their args, whereas expressions constructed by Julia code can have arbitrary run-time values without literal forms as args. In this specific example, `+` and `a` are symbols, `*(b, c)` is a subexpression, and `1` is a literal 64-bit signed integer.

Here is a second syntactic form of quoting code enclosed in `quote ... end`.

```
julia> ex = quote  
        x = 1  
  
        y = 2  
  
        x + y  
  
    end  
quote  
#= none:2 =#  
x = 1  
#= none:3 =#  
y = 2  
#= none:4 =#  
x + y  
end  
  
julia> typeof(ex)  
Expr
```

## Interpolation

Direct construction of `Expr` objects with value arguments is powerful, but `Expr` constructors can be tedious compared to "normal" Julia syntax. As an alternative, Julia allows interpolation of literals or expressions into quoted expressions. Interpolation is indicated by a prefix `$`.

In this example, the value of variable `a` is interpolated:

```
julia> ex = :($a + b)
:(1 + b)
```

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

```
julia> $a + b
ERROR: unsupported or misplaced expression $
```

In this example, the tuple `(1,2,3)` is interpolated as an expression into a conditional test:

```
julia> ex = :(a in $:(1,2,3) )
:(a in (1, 2, 3))
```

The use of `$` for expression interpolation is intentionally reminiscent of [string interpolation](#) and [command interpolation](#). Expression interpolation allows convenient, readable programmatic construction of complex Julia expressions.

### Splatting interpolation

Notice that the `$` interpolation syntax allows inserting only a single expression into an enclosing expression. Occasionally, you have an array of expressions and need them all to become arguments of the surrounding expression. This can be done with the syntax `$(xs...)`. For example, the following code generates a function call where the number of arguments is determined programmatically:

```
julia> args = [:x, :y, :z];
julia> :(f(1, $(args...)))
:(f(1, x, y, z))
```

Naturally, it is possible for quote expressions to contain other quote expressions. Understanding how interpolation works in these cases can be a bit tricky. Consider this example:

```
julia> x = :(1 + 2);

julia> e = quote quote $x end end
quote
#= none:1 =#
$(Expr(:quote, quote
#= none:1 =#
$(Expr(:$, :x))
end))
end
```

Notice that the result contains `Expr(:$, :x)`, which means that `x` has not been evaluated yet. In other words, the `$` expression “belongs to” the inner quote expression, and so its argument is only evaluated when the inner quote expression is:

```
julia> eval(e)
quote
#= none:1 =#
1 + 2
end
```

However, the outer `quote` expression is able to interpolate values inside the `$` in the inner quote. This is done with multiple `$`s:

```
julia> e = quote quote $$x end end
quote
```

```
$(Expr(:quote, quote
#= none:1 =#
$(Expr(:$, :(1 + 2)))
end))
end
```

Notice that `:(1 + 2)` now appears in the result instead of the symbol `:x`. Evaluating this expression yields an interpolated 3:

```
julia> eval(e)
quote
#= none:1 =#
3
end
```

The intuition behind this behavior is that `x` is evaluated once for each `$`: one `$` works similarly to `eval(:x)`, giving `x`'s value, while two `$`s do the equivalent of `eval(eval(:x))`.

## QuoteNode

The usual representation of a `quote` form in an AST is an `Expr` with head `:quote`:

```
julia> dump(Meta.parse(":(1+2)"))
Expr
  head: Symbol quote
  args: Array{Any}((1,))
    1: Expr
  typ: Any
```

As we have seen, such expressions support interpolation with `$`. However, in some situations it is necessary to quote code without performing interpolation.

358 kind of quoting does not yet have Syntax, But 22s Metaprogramming as an object of type `QuoteNode`. The parser yields `QuoteNodes` for simple quoted items like symbols:

```
julia> dump(Meta.parse(":x"))
QuoteNode
  value: Symbol x
```

`QuoteNode` can also be used for certain advanced metaprogramming tasks.

## **eval** and effects

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using `eval`:

```
julia> :(1 + 2)
:(1 + 2)

julia> eval(ans)
3

julia> ex = :(a + b)
:(a + b)

julia> eval(ex)
ERROR: UndefVarError: b not defined
[...]

julia> a = 1; b = 2;

julia> eval(ex)
3
```

22.2 EXPRESSIONS AND EVALUATION that evaluates expressions in its global scope. Expressions passed to `eval` are not limited to returning values – they can also have side-effects that alter the state of the enclosing module's environment:

```
julia> ex = :(x = 1)
:(x = 1)

julia> x
ERROR: UndefVarError: x not defined

julia> eval(ex)
1

julia> x
1
```

Here, the evaluation of an expression object causes a value to be assigned to the global variable `x`.

Since expressions are just `Expr` objects which can be constructed programmatically and then evaluated, it is possible to dynamically generate arbitrary code which can then be run using `eval`. Here is a simple example:

```
julia> a = 1;

julia> ex = Expr(:call, :+, a, :b)
:(1 + b)

julia> a = 0; b = 2;

julia> eval(ex)
3
```

The value of `a` is used to construct the function to the value 1 and the variable `b`. Note the important distinction between the way `a` and `b` are used:

The value of the variable `a` at expression construction time is used as an immediate value in the expression. Thus, the value of `a` when the expression is evaluated no longer matters: the value in the expression is already 1, independent of whatever the value of `a` might be.

On the other hand, the symbol `:b` is used in the expression construction, so the value of the variable `b` at that time is irrelevant – `:b` is just a symbol and the variable `b` need not even be defined. At expression evaluation time, however, the value of the symbol `:b` is resolved by looking up the value of the variable `b`.

## Functions on `Expr`s

As hinted above, one extremely useful feature of Julia is the capability to generate and manipulate Julia code within Julia itself. We have already seen one example of a function returning `Expr` objects: the `parse` function, which takes a string of Julia code and returns the corresponding `Expr`. A function can also take one or more `Expr` objects as arguments, and return another `Expr`. Here is a simple, motivating example:

```
julia> function math_expr(op, op1, op2)

    expr = Expr(:call, op, op1, op2)

    return expr

end

math_expr (generic function with 1 method)
```

```
julia> ex = math_expr(:+, 1, Expr(:call, :*, 4, 5))  
:(1 + 4 * 5)  
  
julia> eval(ex)  
21
```

As another example, here is a function that doubles any numeric argument, but leaves expressions alone:

```
julia> function make_expr2(op, opr1, opr2)  
    opr1f, opr2f = map(x -> isa(x, Number) ? 2*x : x,  
    ↵ (opr1, opr2))  
  
    retexpr = Expr(:call, op, opr1f, opr2f)  
  
    return retexpr  
  
end  
make_expr2 (generic function with 1 method)  
  
julia> make_expr2(:+, 1, 2)  
:(2 + 4)  
  
julia> ex = make_expr2(:+, 1, Expr(:call, :*, 5, 8))  
:(2 + 5 * 8)  
  
julia> eval(ex)
```

Macros provide a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned expression, and the resulting expression is compiled directly rather than requiring a runtime `eval` call. Macro arguments may include expressions, literal values, and symbols.

## Basics

Here is an extraordinarily simple macro:

```
julia> macro sayhello()  
         return :( println("Hello, world!") )  
  
end  
@sayhello (macro with 1 method)
```

Macros have a dedicated character in Julia's syntax: the @ (at-sign), followed by the unique name declared in a `macro NAME ... end` block. In this example, the compiler will replace all instances of `@sayhello` with:

```
:( println("Hello, world!") )
```

When `@sayhello` is entered in the REPL, the expression executes immediately, thus we only see the evaluation result:

```
julia> @sayhello()  
Hello, world!
```

Now, consider a slightly more complex macro:

```
        return :( println("Hello, ", $name) )  
  
    end  
@sayhello (macro with 1 method)
```

This macro takes one argument: `name`. When `@sayhello` is encountered, the quoted expression is expanded to interpolate the value of the argument into the final expression:

```
julia> @sayhello("human")  
Hello, human
```

We can view the quoted return expression using the function `macroexpand` (important note: this is an extremely useful tool for debugging macros):

```
julia> ex = macroexpand(Main, :(@sayhello("human")) )  
:(Main.println)("Hello, ", "human")  
  
julia> typeof(ex)  
Expr
```

We can see that the "human" literal has been interpolated into the expression.

There also exists a macro `@macroexpand` that is perhaps a bit more convenient than the `macroexpand` function:

```
julia> @macroexpand @sayhello "human"  
:((println)("Hello, ", "human"))
```

Hold up: why macros?

We have already seen a function `f(::Expr...)` → `Expr` in a previous section. In fact, `macroexpand` is also such a function. So, why do macros exist?

Macros are necessary because they execute when code is parsed, therefore, macros allow the programmer to generate and include fragments of customized code before the full program is run. To illustrate the difference, consider the following example:

```
julia> macro twostep(arg)
           println("I execute at parse time. The argument is: ",
           arg)
           return :(println("I execute at runtime. The argument is
           : ", $arg))
       end
@twostep (macro with 1 method)

julia> ex = macroexpand(Main, :(@twostep :(1, 2, 3)) );
I execute at parse time. The argument is: $(Expr(:quote, :((1, 2,
3))))
```

The first call to `println` is executed when `macroexpand` is called. The resulting expression contains only the second `println`:

```
julia> typeof(ex)
Expr

julia> ex
:((println)("I execute at runtime. The argument is: ", $(Expr(:copyast, :($(QuoteNode(:((1, 2, 3))))))))))

julia> eval(ex)
I execute at runtime. The argument is: (1, 2, 3)
```

## Macro invocation

Macros are invoked with the following general syntax:

22.3 MACROS  
| @name expr1 expr2 ...  
| @name(expr1, expr2, ...)

365

Note the distinguishing @ before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after @name in the second form. The two styles should not be mixed. For example, the following syntax is different from the examples above; it passes the tuple (expr1, expr2, ...) as one argument to the macro:

| @name (expr1, expr2, ...)

An alternative way to invoke a macro over an array literal (or comprehension) is to juxtapose both without using parentheses. In this case, the array will be the only expression fed to the macro. The following syntax is equivalent (and different from @name [a b] \* v):

| @name[a b] \* v  
| @name([a b]) \* v

It is important to emphasize that macros receive their arguments as expressions, literals, or symbols. One way to explore macro arguments is to call the `show` function within the macro body:

```
julia> macro showarg(x)  
  
    show(x)  
  
    # ... remainder of macro, returning an expression  
  
  end  
@showarg (macro with 1 method)
```

```
julia> @showarg(a)
```

```
:a
```

```
julia> @showarg(1+1)
```

```
:(1 + 1)
```

```
julia> @showarg(println("Yo!"))
```

```
:(println("Yo!"))
```

In addition to the given argument list, every macro is passed extra arguments named `__source__` and `__module__`.

The argument `__source__` provides information (in the form of a `LineNumberNode` object) about the parser location of the `@` sign from the macro invocation. This allows macros to include better error diagnostic information, and is commonly used by logging, string-parser macros, and docs, for example, as well as to implement the `__LINE__`, `__FILE__`, and `__DIR__` macros.

The location information can be accessed by referencing `__source__.line` and `__source__.file`:

```
julia> macro __LOCATION__(); return QuoteNode(__source__); end
```

```
@__LOCATION__ (macro with 1 method)
```

```
julia> dump(
```

```
    __LOCATION__(
```

```
))
```

```
LineNumberNode
```

```
  line: Int64 2
```

```
  file: Symbol none
```

`__MACROS_MODULE__` provides information (in the form of a `Module` object) about the expansion context of the macro invocation. This allows macros to look up contextual information, such as existing bindings, or to insert the value as an extra argument to a runtime function call doing self-reflection in the current module.

## Building an advanced macro

Here is a simplified definition of Julia's `@assert` macro:

```
julia> macro assert(ex)
           return :( $ex ? nothing :
→   throw(AssertionError($(string(ex)))) )
           end
@assert (macro with 1 method)
```

This macro can be used like this:

```
julia> @assert 1 == 1.0
julia> @assert 1 == 0
ERROR: AssertionError: 1 == 0
```

In place of the written syntax, the macro call is expanded at parse time to its returned result. This is equivalent to writing:

```
1 == 1.0 ? nothing : throw(AssertionError("1 == 1.0"))
1 == 0 ? nothing : throw(AssertionError("1 == 0"))
```

That is, in the first call, the expression `:(1 == 1.0)` is spliced into the test condition slot, while the value of `string(:(1 == 1.0))` is spliced into the

the syntax tree where the `@assert` macro call occurs. Then at execution time, if the test expression evaluates to true, then `nothing` is returned, whereas if the test is false, an error is raised indicating the asserted expression that was false. Notice that it would not be possible to write this as a function, since only the value of the condition is available and it would be impossible to display the expression that computed it in the error message.

The actual definition of `@assert` in the standard library is more complicated. It allows the user to optionally specify their own error message, instead of just printing the failed expression. Just like in functions with a variable number of arguments, this is specified with an ellipses following the last argument:

```
julia> macro assert(ex, msgs...)
    msg_body = isempty(msgs) ? ex : msgs[1]
    msg = string(msg_body)
    return :($ex ? nothing : throw(AssertionError($msg)))
end
@assert (macro with 1 method)
```

Now `@assert` has two modes of operation, depending upon the number of arguments it receives! If there's only one argument, the tuple of expressions captured by `msgs` will be empty and it will behave the same as the simpler definition above. But now if the user specifies a second argument, it is printed in the message body instead of the failing expression. You can inspect the result of a macro expansion with the aptly named `@macroexpand` macro:

```
julia> @macroexpand @assert a == b
```

22.3 MACROS  
?:  
if Main.a == Main.b  
    Main.nothing  
else  
    (Main.throw)((Main.AssertionError)("a == b"))  
end)

369

```
julia> @macroexpand @assert a==b "a should equal b!"  
:(if Main.a == Main.b  
    Main.nothing  
else  
    (Main.throw)((Main.AssertionError)("a should equal b!"))  
end)
```

There is yet another case that the actual `@assert` macro handles: what if, in addition to printing “a should equal b,” we wanted to print their values? One might naively try to use string interpolation in the custom message, e.g., `@assert a==b "a ($a) should equal b ($b)!"`, but this won’t work as expected with the above macro. Can you see why? Recall from [string interpolation](#) that an interpolated string is rewritten to a call to `string`. Compare:

```
julia> typeof(:("a should equal b"))  
String  
  
julia> typeof(:("a ($a) should equal b ($b)!"))  
Expr  
  
julia> dump(:("a ($a) should equal b ($b)!"))  
Expr  
head: Symbol string  
args: Array{Any}((5, ))  
1: String "a ("  
2: Symbol a
```

```
|370 3: String ") should equal b (" CHAPTER 22. METAPROGRAMMING
| 4: Symbol b
| 5: String ")!""
| typ: Any
```

So now instead of getting a plain string in `msg_body`, the macro is receiving a full expression that will need to be evaluated in order to display as expected. This can be spliced directly into the returned expression as an argument to the `string` call; see [error.jl](#) for the complete implementation.

The `@assert` macro makes great use of splicing into quoted expressions to simplify the manipulation of expressions inside the macro body.

## Hygiene

An issue that arises in more complex macros is that of [hygiene](#). In short, macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. Conversely, the expressions that are passed into a macro as arguments are often expected to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. Another concern arises from the fact that a macro may be called in a different module from where it was defined. In this case we need to ensure that all global variables are resolved to the correct module. Julia already has a major advantage over languages with textual macro expansion (like C) in that it only needs to consider the returned expression. All the other variables (such as `msg` in `@assert` above) follow the [normal scoping block behavior](#).

To demonstrate these issues, let us consider writing a `@time` macro that takes an expression as its argument, records the time, evaluates the expression, records the time again, prints the difference between the before and after

在这种情况下，`MACROS` 有表达式的值作为其最终值。宏 `macro`<sup>270</sup> 可能看起来像这样：

```
macro time(ex)
    return quote
        local t0 = time()
        local val = $ex
        local t1 = time()
        println("elapsed time: ", t1-t0, " seconds")
        val
    end
end
```

在这里，我们希望 `t0`、`t1` 和 `val` 成为私有临时变量，我们希望 `time` 指向标准库中的 `time` 函数，而不是用户可能具有的任何 `time` 变量（对 `println` 也是如此）。想象一下如果用户表达式 `ex` 也包含对名为 `t0` 的变量的赋值，或定义了自己的 `time` 变量，可能会出现的问题。我们可能会得到错误，或者神秘地得到不正确的行为。

Julia 的宏展开器通过以下方式解决这些问题。首先，宏结果中的变量被分类为本地或全局。如果变量被赋值（且未声明全局），或声明为本地，或用作函数参数名，则认为是本地的。否则，认为是全局的。本地变量随后被重命名以使其唯一（使用 `gensym` 函数，该函数生成新符号），全局变量在宏定义环境中解析。因此，上述所有担忧都得到了处理：宏的本地将不会与任何用户变量冲突，并且 `time` 和 `println` 将指向标准库定义。

然而，一个问题仍然存在。考虑以下使用此宏的方式：

```
372 module MyModule  
import Base.@time  
  
time() = ... # compute something  
  
@time time()  
end
```

## CHAPTER 22. METAPROGRAMMING

Here the user expression `ex` is a call to `time`, but not the same `time` function that the macro uses. It clearly refers to `MyModule.time`. Therefore we must arrange for the code in `ex` to be resolved in the macro call environment. This is done by "escaping" the expression with `esc`:

```
macro time(ex)  
    ...  
    local val = $(esc(ex))  
    ...  
end
```

An expression wrapped in this manner is left alone by the macro expander and simply pasted into the output verbatim. Therefore it will be resolved in the macro call environment.

This escaping mechanism can be used to "violate" hygiene when necessary, in order to introduce or manipulate user variables. For example, the following macro sets `x` to zero in the call environment:

```
julia> macro zerox()  
        return esc(:(x = 0))  
end
```

```
julia> function foo()  
  
    x = 1  
  
    @zerox  
  
    return x # is zero  
  
end  
foo (generic function with 1 method)  
  
julia> foo()  
0
```

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

Getting the hygiene rules correct can be a formidable challenge. Before using a macro, you might want to consider whether a function closure would be sufficient. Another useful strategy is to defer as much work as possible to run-time. For example, many macros simply wrap their arguments in a `QuoteNode` or other similar `Expr`. Some examples of this include `@task body` which simply returns `schedule(Task(() -> $body))`, and `@eval expr`, which simply returns `eval(QuoteNode(expr))`.

To demonstrate, we might rewrite the `@time` example above as:

```
macro time(expr)  
    return :(timeit(() -> $(esc(expr))))  
end  
function timeit(f)
```

```
374 t0 = time()
    val = f()
    t1 = time()
    println("elapsed time: ", t1-t0, " seconds")
    return val
end
```

## CHAPTER 22. METAPROGRAMMING

However, we don't do this for a good reason: wrapping the `expr` in a new scope block (the anonymous function) also slightly changes the meaning of the expression (the scope of any variables in it), while we want `@time` to be usable with minimum impact on the wrapped code.

### Macros and dispatch

Macros, just like Julia functions, are generic. This means they can also have multiple method definitions, thanks to multiple dispatch:

```
julia> macro m end
@m (macro with 0 methods)

julia> macro m(args...)
        println(length(args)) arguments"
        end
@m (macro with 1 method)

julia> macro m(x,y)
        println("Two arguments")
        end
```

```
julia> @m "asd"
```

```
1 arguments
```

```
julia> @m 1 2
```

```
Two arguments
```

However one should keep in mind, that macro dispatch is based on the types of AST that are handed to the macro, not the types that the AST evaluates to at runtime:

```
julia> macro m(::Int)
```

```
    println("An Integer")
```

```
end
```

```
@m (macro with 3 methods)
```

```
julia> @m 2
```

```
An Integer
```

```
julia> x = 2
```

```
2
```

```
julia> @m x
```

```
1 arguments
```

## 22.4 Code Generation

When a significant amount of repetitive boilerplate code is required, it is common to generate it programmatically to avoid redundancy. In most languages,

876 requires an extra build step, and a CHAPTER 22. METAPROGRAMMING separate code generation repetitive code. In Julia, expression interpolation and `eval` allow such code generation to take place in the normal course of program execution. For example, the following code defines a series of operators on three arguments in terms of their 2-argument forms:

```
for op = (:+, :*, :&, :|, :$)
    eval(quote
        ($op)(a,b,c) = ($op)((($op)(a,b),c))
    end)
end
```

In this manner, Julia acts as its own `preprocessor`, and allows code generation from inside the language. The above code could be written slightly more tersely using the `:` prefix quoting form:

```
for op = (:+, :*, :&, :|, :$)
    eval(:((($op)(a,b,c) = ($op)((($op)(a,b),c))))
end
```

This sort of in-language code generation, however, using the `eval(quote(...))` pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

```
for op = (:+, :*, :&, :|, :$)
    @eval ($op)(a,b,c) = ($op)((($op)(a,b),c))
end
```

The `@eval` macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to `@eval` can be a block:

```
@eval begin  
    # multiple lines  
end
```

## 22.5 Non-Standard String Literals

Recall from [Strings](#) that string literals prefixed by an identifier are called non-standard string literals, and can have different semantics than un-prefixed string literals. For example:

`r"^\s*(?:#|\$)"` produces a regular expression object rather than a string

`b"DATA\xff\u2200"` is a byte array literal for `[68, 65, 84, 65, 255, 226, 136, 128]`.

Perhaps surprisingly, these behaviors are not hard-coded into the Julia parser or compiler. Instead, they are custom behaviors provided by a general mechanism that anyone can use: prefixed string literals are parsed as calls to specially-named macros. For example, the regular expression macro is just the following:

```
macro r_str(p)  
    Regex(p)  
end
```

That's all. This macro says that the literal contents of the string literal `r"^\s*(?:#|\$)"` should be passed to the `@r_str` macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression `r"^\s*(?:#|\$)"` is equivalent to placing the following object directly into the syntax tree:

```
Regex("^\s*(?:#|\$)")
```

CHAPTER 22. METAPROGRAMMING

878 only is the string literal form shorter and also more efficient: since the regular expression is compiled and the `Regex` object is actually created when the code is compiled, the compilation occurs only once, rather than every time the code is executed. Consider if the regular expression occurs in a loop:

```
for line = lines
  m = match(r"\s*(?:#|\$)", line)
  if m === nothing
    # non-comment
  else
    # comment
  end
end
```

Since the regular expression `r"\s*(?:#|\$)"` is compiled and inserted into the syntax tree when this code is parsed, the expression is only compiled once instead of each time the loop is executed. In order to accomplish this without macros, one would have to write this loop like this:

```
re = Regex("\s*(?:#|\$)")
for line = lines
  m = match(re, line)
  if m === nothing
    # non-comment
  else
    # comment
  end
end
```

Moreover, if the compiler could not determine that the regex object was constant over all loops, certain optimizations might not be possible, making this

course, there are still situations where the non-literal form is more convenient: if one needs to interpolate a variable into the regular expression, one must take this more verbose approach; in cases where the regular expression pattern itself is dynamic, potentially changing upon each loop iteration, a new regular expression object must be constructed on each iteration. In the vast majority of use cases, however, regular expressions are not constructed based on run-time data. In this majority of cases, the ability to write regular expressions as compile-time values is invaluable.

Like non-standard string literals, non-standard command literals exist using a prefixed variant of the command literal syntax. The command literal `custom`literal`` is parsed as `@custom_cmd "literal"`. Julia itself does not contain any non-standard command literals, but packages can make use of this syntax. Aside from the different syntax and the `_cmd` suffix instead of the `_str` suffix, non-standard command literals behave exactly like non-standard string literals.

In the event that two modules provide non-standard string or command literals with the same name, it is possible to qualify the string or command literal with a module name. For instance, if both `Foo` and `Bar` provide non-standard string literal `@x_str`, then one can write `Foo.x"literal"` or `Bar.x"literal"` to disambiguate between the two.

The mechanism for user-defined string literals is deeply, profoundly powerful. Not only are Julia's non-standard literals implemented using it, but also the command literal syntax (``echo "Hello, $person"``) is implemented with the following innocuous-looking macro:

```
macro cmd(str)
    :(cmd_gen($(shell_parse(str)[1])))
end
```

Of course, a large amount of complexity is hidden behind this macro definition, but they are just functions, written entirely in Julia. You can read their source and see precisely what they do – and all they do is construct expression objects to be inserted into your program’s syntax tree.

## 22.6 Generated functions

A very special macro is `@generated`, which allows you to define so-called generated functions. These have the capability to generate specialized code depending on the types of their arguments with more flexibility and/or less code than what can be achieved with multiple dispatch. While macros work with expressions at parse time and cannot access the types of their inputs, a generated function gets expanded at a time when the types of the arguments are known, but the function is not yet compiled.

Instead of performing some calculation or action, a generated function declaration returns a quoted expression which then forms the body for the method corresponding to the types of the arguments. When a generated function is called, the expression it returns is compiled and then run. To make this efficient, the result is usually cached. And to make this inferable, only a limited subset of the language is usable. Thus, generated functions provide a flexible way to move work from run time to compile time, at the expense of greater restrictions on allowed constructs.

When defining generated functions, there are four main differences to ordinary functions:

1. You annotate the function declaration with the `@generated` macro. This adds some information to the AST that lets the compiler know that this is a generated function.
2. In the body of the generated function you only have access to the types

The arguments to their values – and any function that was defined before the definition of the generated function.

3. Instead of calculating something or performing some action, you return a quoted expression which, when evaluated, does what you want.
4. Generated functions must not mutate or observe any non-constant global state (including, for example, IO, locks, non-local dictionaries, or using `method_exists`). This means they can only read global constants, and cannot have any side effects. In other words, they must be completely pure. Due to an implementation limitation, this also means that they currently cannot define a closure or generator.

It's easiest to illustrate this with an example. We can declare a generated function `foo` as

```
julia> @generated function foo(x)

    Core.println(x)

    return :(x * x)

end
foo (generic function with 1 method)
```

Note that the body returns a quoted expression, namely `:(x * x)`, rather than just the value of `x * x`.

From the caller's perspective, this is identical to a regular function; in fact, you don't have to know whether you're calling a regular or generated function. Let's see how `foo` behaves:

382 CHAPTER 22. METAPROGRAMMING

```
julia> x = foo(2); # note: output is from println() statement in
   ↳ the body
Int64

julia> x           # now we print x
4

julia> y = foo("bar");
String

julia> y
"barbar"
```

So, we see that in the body of the generated function, `x` is the type of the passed argument, and the value returned by the generated function, is the result of evaluating the quoted expression we returned from the definition, now with the value of `x`.

What happens if we evaluate `foo` again with a type that we have already used?

```
julia> foo(4)
16
```

Note that there is no printout of `Int64`. We can see that the body of the generated function was only executed once here, for the specific set of argument types, and the result was cached. After that, for this example, the expression returned from the generated function on the first invocation was re-used as the method body. However, the actual caching behavior is an implementation-defined performance optimization, so it is invalid to depend too closely on this behavior.

The generated function is generated might be only once,<sup>383</sup> it might also be more often, or appear to not happen at all. As a consequence, you should never write a generated function with side effects – when, and how often, the side effects occur is undefined. (This is true for macros too – and just like for macros, the use of `eval` in a generated function is a sign that you’re doing something the wrong way.) However, unlike macros, the runtime system cannot correctly handle a call to `eval`, so it is disallowed.

It is also important to see how `@generated` functions interact with method redefinition. Following the principle that a correct `@generated` function must not observe any mutable state or cause any mutation of global state, we see the following behavior. Observe that the generated function cannot call any method that was not defined prior to the definition of the generated function itself.

Initially `f(x)` has one definition

```
julia> f(x) = "original definition";
```

Define other operations that use `f(x)`:

```
julia> g(x) = f(x);
```

```
julia> @generated gen1(x) = f(x);
```

```
julia> @generated gen2(x) = :(f(x));
```

We now add some new definitions for `f(x)`:

```
julia> f(x::Int) = "definition for Int";
```

```
julia> f(x::Type{Int}) = "definition for Type{Int}";
```

and compare how these results differ:

384  
julia> f(1)

"definition for Int"

julia> g(1)

"definition for Int"

julia> gen1(1)

"original definition"

julia> gen2(1)

"definition for Int"

## CHAPTER 22. METAPROGRAMMING

Each method of a generated function has its own view of defined functions:

julia> @generated gen1(x::Real) = f(x);

julia> gen1(1)

"definition for Type{Int}"

The example generated function `foo` above did not do anything a normal function `foo(x) = x * x` could not do (except printing the type on the first invocation, and incurring higher overhead). However, the power of a generated function lies in its ability to compute different quoted expressions depending on the types passed to it:

julia> @generated **function** bar(x)

**if** x <: Integer

**return** :(x ^ 2)

**else**

```
    return :(x)

  end

end
bar (generic function with 1 method)
```

```
julia> bar(4)
```

```
16
```

```
julia> bar("baz")
```

```
"baz"
```

(although of course this contrived example would be more easily implemented using multiple dispatch...)

Abusing this will corrupt the runtime system and cause undefined behavior:

```
julia> @generated function baz(x)

  if rand() < .9

    return :(x^2)

  else

    return :( "boo! " )

  end

end
baz (generic function with 1 method)
```

See the body of the generated function, and the behavior of all subsequent code is undefined.

Don't copy these examples!

These examples are hopefully helpful to illustrate how generated functions work, both in the definition end and at the call site; however, don't copy them, for the following reasons:

the `foo` function has side-effects (the call to `Core.println`), and it is undefined exactly when, how often or how many times these side-effects will occur

the `bar` function solves a problem that is better solved with multiple dispatch - defining `bar(x) = x` and `bar(x::Integer) = x ^ 2` will do the same thing, but it is both simpler and faster.

the `baz` function is pathological

Note that the set of operations that should not be attempted in a generated function is unbounded, and the runtime system can currently only detect a subset of the invalid operations. There are many other operations that will simply corrupt the runtime system without notification, usually in subtle ways not obviously connected to the bad definition. Because the function generator is run during inference, it must respect all of the limitations of that code.

Some operations that should not be attempted include:

1. Caching of native pointers.
2. Interacting with the contents or methods of `Core.Inference` in any way.
3. Observing any mutable state.
  - Inference on the generated function may be run at any time, including while your code is attempting to observe or mutate this state.

## 24.6. Calling any lock functions you call out to may use locks internally, 387

example, it is not problematic to call `malloc`, even though most implementations require locks internally) but don't attempt to hold or acquire any while executing Julia code.

5. Calling any function that is defined after the body of the generated function. This condition is relaxed for incrementally-loaded precompiled modules to allow calling any function in the module.

Alright, now that we have a better understanding of how generated functions work, let's use them to build some more advanced (and valid) functionality...

An advanced example

Julia's base library has a `sub2ind` function to calculate a linear index into an n-dimensional array, based on a set of n multilinear indices – in other words, to calculate the index `i` that can be used to index into an array `A` using `A[i]`, instead of `A[x, y, z, ...]`. One possible implementation is the following:

```
julia> function sub2ind_loop(dims::NTuple{N}, I::Integer...) where
→   N

    ind = I[N] - 1

    for i = N-1:-1:1

        ind = I[i]-1 + dims[i]*ind

    end

    return ind + 1

end
```

```
julia> sub2ind_loop((3, 5), 1, 2)
4
```

The same thing can be done using recursion:

```
julia> sub2ind_rec(dims::Tuple{}) = 1;

julia> sub2ind_rec(dims::Tuple{}, i1::Integer, I::Integer...) =
           i1 == 1 ? sub2ind_rec(dims, I...) :
→   throw(BoundsError());

julia> sub2ind_rec(dims::Tuple{Integer}, Vararg{Integer}|,
→   i1::Integer) = i1;

julia> sub2ind_rec(dims::Tuple{Integer}, Vararg{Integer}|,
→   i1::Integer, I::Integer...) =
           i1 + dims[1] * (sub2ind_rec(Base.tail(dims), I...) -
→   1);

julia> sub2ind_rec((3, 5), 1, 2)
4
```

Both these implementations, although different, do essentially the same thing: a runtime loop over the dimensions of the array, collecting the offset in each dimension into the final index.

However, all the information we need for the loop is embedded in the type information of the arguments. Thus, we can utilize generated functions to

226 GENERATING FUNCTIONS 389  
functions to manually unroll the loop. The body becomes almost identical, but instead of calculating the linear index, we build up an expression that calculates the index:

```
julia> @generated function sub2ind_gen(dims::NTuple{N},  
→ I::Integer...) where N  
  
    ex = :(I[$N] - 1)  
  
    for i = (N - 1):-1:1  
  
        ex = :(I[$i] - 1 + dims[$i] * $ex)  
  
    end  
  
    return :($ex + 1)  
  
end  
  
sub2ind_gen (generic function with 1 method)  
  
julia> sub2ind_gen((3, 5), 1, 2)  
4
```

What code will this generate?

An easy way to find out is to extract the body into another (regular) function:

```
julia> @generated function sub2ind_gen(dims::NTuple{N},  
→ I::Integer...) where N  
  
    return sub2ind_gen_impl(dims, I...)
```

390       end

## CHAPTER 22. METAPROGRAMMING

```
sub2ind_gen (generic function with 1 method)

julia> function sub2ind_gen_impl(dims::Type{T}, I...) where T <:
    → NTuple{N,Any} where N

            length(I) == N || return :(error("partial indexing is
    → unsupported"))

            ex = :(I[$N] - 1)

            for i = (N - 1):-1:1

              ex = :(I[$i] - 1 + dims[$i] * $ex)

            end

            return :($ex + 1)

    end

sub2ind_gen_impl (generic function with 1 method)
```

We can now execute `sub2ind_gen_impl` and examine the expression it returns:

```
julia> sub2ind_gen_impl(Tuple{Int,Int}, Int, Int)
:(((I[1] - 1) + dims[1] * (I[2] - 1)) + 1)
```

So, the method body that will be used here doesn't include a loop at all – just indexing into the two tuples, multiplication and addition/subtraction. All the looping is performed compile-time, and we avoid looping during execution entirely. Thus, we only loop once per type, in this case once per `N` (except in

22.6 GENERATED FUNCTIONS is generated more than once - see disclaimer above).

## Optionally-generated functions

Generated functions can achieve high efficiency at run time, but come with a compile time cost: a new function body must be generated for every combination of concrete argument types. Typically, Julia is able to compile "generic" versions of functions that will work for any arguments, but with generated functions this is impossible. This means that programs making heavy use of generated functions might be impossible to statically compile.

To solve this problem, the language provides syntax for writing normal, non-generated alternative implementations of generated functions. Applied to the `sub2ind` example above, it would look like this:

```
function sub2ind_gen(dims::NTuple{N}, I::Integer...) where N
    if N != length(I)
        throw(ArgumentError("Number of dimensions must match
                           → number of indices."))
    end
    if @generated
        ex = :(I[$N] - 1)
        for i = (N - 1):-1:1
            ex = :(I[$i] - 1 + dims[$i] * $ex)
        end
        return :($ex + 1)
    else
        ind = I[N] - 1
        for i = (N - 1):-1:1
            ind = I[i] - 1 + dims[i]*ind
        end
        return ind + 1
    end
end
```

```
392 end  
end
```

## CHAPTER 22. METAPROGRAMMING

Internally, this code creates two implementations of the function: a generated one where the first block in `if @generated` is used, and a normal one where the `else` block is used. Inside the `then` part of the `if @generated` block, code has the same semantics as other generated functions: argument names refer to types, and the code should return an expression. Multiple `if @generated` blocks may occur, in which case the generated implementation uses all of the `then` blocks and the alternate implementation uses all of the `else` blocks.

Notice that we added an error check to the top of the function. This code will be common to both versions, and is run-time code in both versions (it will be quoted and returned as an expression from the generated version). That means that the values and types of local variables are not available at code generation time -- the code-generation code can only see the types of arguments.

In this style of definition, the code generation feature is essentially an optional optimization. The compiler will use it if convenient, but otherwise may choose to use the normal implementation instead. This style is preferred, since it allows the compiler to make more decisions and compile programs in more ways, and since normal code is more readable than code-generating code. However, which implementation is used depends on compiler implementation details, so it is essential for the two implementations to behave identically.

## Chapter 23

# Multi-dimensional Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia. As such, it's also possible to define custom array types by inheriting from `AbstractArray`. See the [manual section on the AbstractArray interface](#) for more details on implementing a custom array type.

An array is a collection of objects stored in a multi-dimensional grid. In the most general case, an array may contain objects of type `Any`. For most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`.

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions passed by reference are arrays. In computing languages pass arrays by value, and this is convenient in many cases. In Julia, modifications made to input arrays within a function will be visible in the parent function. The entire Julia array library ensures that inputs are not modified by library functions. User code, if it needs to exhibit similar behavior, should take care to create a copy of inputs that it may modify.

## 23.1 Arrays

### Basic Functions

Function	Description
<code>eltype(A)</code>	the type of the elements contained in A
<code>length(A)</code>	the number of elements in A
<code>ndims(A)</code>	the number of dimensions of A
<code>size(A)</code>	a tuple containing the dimensions of A
<code>size(A, n)</code>	the size of A along dimension n
<code>in-dices(A)</code>	a tuple containing the valid indices of A
<code>in-dices(A, n)</code>	a range expressing the valid indices along dimension n
<code>eachindex(A)</code>	an efficient iterator for visiting each position in A
<code>stride(A, k)</code>	the stride (linear index distance between adjacent elements) along dimension k
<code>strides(A)</code>	a tuple of the strides in each dimension

### Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a `dims...` argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments. Most of these functions also accept a first input T, which is the element type of the array. If the type T is omitted it will default to `Float64`.

---

<sup>1</sup>iid, independently and identically distributed.

The syntax `[A, B, C, ...]` constructs a 1-d array (vector) of its arguments.<sup>395</sup>  
If all arguments have a common [promotion type](#) then they get converted to that type using `convert`.

## Concatenation

Arrays can be constructed and also concatenated using the following functions:

Scalar values passed to these functions are treated as 1-element arrays.

The concatenation functions are used so often that they have special syntax:

`hvcat` concatenates in both dimension 1 (with semicolons) and dimension 2 (with spaces).

## Typed array initializers

An array with a specific element type can be constructed using the syntax `T[A, B, C, ...]`. This will construct a 1-d array with element type `T`, initialized to contain elements `A, B, C`, etc. For example `Any[x, y, z]` constructs a heterogeneous array that can contain any values.

Concatenation syntax can similarly be prefixed with a type to specify the element type of the result.

```
julia> [[1 2] [3 4]]  
1×4 Array{Int64,2}:  
 1 2 3 4  
  
julia> Int8[[1 2] [3 4]]  
1×4 Array{Int8,2}:  
 1 2 3 4
```

Comprehensions provide a general and powerful way to construct arrays.

Comprehension syntax is similar to set construction notation in mathematics:

```
| A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

The meaning of this form is that  $F(x, y, \dots)$  is evaluated with the variables  $x$ ,  $y$ , etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like  $1:n$  or  $2:(n-1)$ , or explicit arrays of values like  $[1.2, 3.4, 5.7]$ . The result is an  $N$ -d dense array with dimensions that are the concatenation of the dimensions of the variable ranges  $rx$ ,  $ry$ , etc. and each  $F(x, y, \dots)$  evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbor along a 1-d grid. :

```
julia> x = rand(8)
8-element Array{Float64,1}:
 0.843025
 0.869052
 0.365105
 0.699456
 0.977653
 0.994953
 0.41084
 0.809411

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1
→   ]
6-element Array{Float64,1}:
 0.736559
 0.57468
```

```
0.912429
```

```
0.8446
```

```
0.656511
```

The resulting array type depends on the types of the computed elements. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1  
→ ]
```

## Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance (see [Iteration](#)). For example, the following expression sums a series without allocating memory:

```
julia> sum(1/n^2 for n=1:1000)  
1.6439345666815615
```

When writing a generator expression with multiple dimensions inside an argument list, parentheses are needed to separate the generator from subsequent arguments:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])  
ERROR: syntax: invalid iteration specification
```

All comma-separated expressions after `for` are interpreted as ranges. Adding parentheses lets us add a third argument to `map`:

```
398 CHAPTER 23: MULTI-DIMENSIONAL ARRAYS  
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])  
2×2 Array{Tuple{Float64, Int64}, 2}:  
 (0.5, 1)      (0.333333, 3)  
 (0.333333, 2) (0.25, 4)
```

Ranges in generators and comprehensions can depend on previous ranges by writing multiple **for** keywords:

```
julia> [(i, j) for i=1:3 for j=1:i]  
6-element Array{Tuple{Int64, Int64}, 1}:  
 (1, 1)  
 (2, 1)  
 (2, 2)  
 (3, 1)  
 (3, 2)  
 (3, 3)
```

In such cases, the result is always 1-d.

Generated values can be filtered using the **if** keyword:

```
julia> [(i, j) for i=1:3 for j=1:i if i+j == 4]  
2-element Array{Tuple{Int64, Int64}, 1}:  
 (2, 2)  
 (3, 1)
```

## Indexing

The general syntax for indexing into an n-dimensional array A is:

```
X = A[I_1, I_2, ..., I_n]
```

where each **I\_k** may be a scalar integer, an array of integers, or any other supported index. This includes **Colon** (**:**) to select all indices within the entire

~~Advanced ARRAYS~~ ~~399~~  
Ranges of the form `a:c` or `a:b:c` to select contiguous or stride subsections, and arrays of booleans to select elements at their `true` indices.

If all the indices are scalars, then the result `X` is a single element from the array `A`. Otherwise, `X` is an array with the same number of dimensions as the sum of the dimensionalities of all the indices.

If all indices are vectors, for example, then the shape of `X` would be `(length(I_1), length(I_2), ..., length(I_n))`, with location `(i_1, i_2, ..., i_n)` of `X` containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`. If `I_1` is changed to a two-dimensional matrix, then `X` becomes an `n+1`-dimensional array of shape `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`. The matrix adds a dimension. The location `(i_1, i_2, i_3, ..., i_{n+1})` contains the value at `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]`. All dimensions indexed with scalars are dropped. For example, the result of `A[2, I, 3]` is an array with size `size(I)`. Its `i`th element is populated by `A[2, I[i], 3]`.

As a special part of this syntax, the `end` keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the innermost array being indexed. Indexing syntax without the `end` keyword is equivalent to a call to `getindex`:

```
| X = getindex(A, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16
```

400  
**julia>** x[2:3, 2:end-1]

## CHAPTER 23. MULTI-DIMENSIONAL ARRAYS

2×2 Array{Int64,2}:

6 10

7 11

**julia>** x[1, [2 3; 4 1]]

2×2 Array{Int64,2}:

5 9

13 1

Empty ranges of the form `n:n-1` are sometimes used to indicate the inter-index location between `n-1` and `n`. For example, the `searchsorted` function uses this convention to indicate the insertion point of a value not found in a sorted array:

**julia>** a = [1, 2, 5, 6, 7];

**julia>** searchsorted(a, 3)

3:2

## Assignment

The general syntax for assigning values in an  $n$ -dimensional array `A` is:

`A[I_1, I_2, ..., I_n] = X`

where each `I_k` may be a scalar integer, an array of integers, or any other [supported index](#). This includes [Colon](#) (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If `X` is an array, it must have the same number of elements as the product of the lengths of the indices: `prod(length(I_1), length(I_2), ..., length(I_n))`. The value in location `I_1[i_1], I_2[i_2], ..., I_n[i_n]`

~~A is an array~~ is written with the value  $X[i_1, i_2, \dots, i_n]$ . If  $X$  is not an array, its value is written to all referenced locations of  $A$ .

Just as in [Indexing](#), the `end` keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the array being assigned into. Indexed assignment syntax without the `end` keyword is equivalent to a call to [`setindex!`](#):

```
| setindex!(A, X, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[1:2, 2:3] = -1
-1

julia> x
3×3 Array{Int64,2}:
 1  -1  -1
 2  -1  -1
 3   6   9
```

## Supported index types

In the expression  $A[I_1, I_2, \dots, I_n]$ , each  $I_k$  may be a scalar index, an array of scalar indices, or an object that represents an array of scalar indices and can be converted to such by [`to\_indices`](#):

1. A scalar index. By default this includes:

- `CartesianIndex{N}`s, which behave like an  $N$ -tuple of integers spanning multiple dimensions (see below for more details)

2. An array of scalar indices. This includes:

- Vectors and multidimensional arrays of integers
- Empty arrays like `[ ]`, which select no elements
- Ranges like `a:c` or `a:b:c`, which select contiguous or strided subsections from `a` to `c` (inclusive)
- Any custom array of scalar indices that is a subtype of `AbstractArray`
- Arrays of `CartesianIndex{N}` (see below for more details)

3. An object that represents an array of scalar indices and can be converted to such by `to_indices`. By default this includes:

- `Colon() (:)`, which represents all indices within an entire dimension or across the entire array
- Arrays of booleans, which select elements at their `true` indices (see below for more details)

## Cartesian indices

The special `CartesianIndex{N}` object represents a scalar index that behaves like an  $N$ -tuple of integers spanning multiple dimensions. For example:

```
julia> A = reshape(1:32, 4, 4, 2);
```

```
julia> A[3, 2, 1]
```

```
julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7  
true
```

Considered alone, this may seem relatively trivial; `CartesianIndex` simply gathers multiple integers together into one object that represents a single multidimensional index. When combined with other indexing forms and iterators that yield `CartesianIndex`es, however, this can lead directly to very elegant and efficient code. See [Iteration](#) below, and for some more advanced examples, see [this blog post on multidimensional algorithms and iteration](#).

Arrays of `CartesianIndex{N}` are also supported. They represent a collection of scalar indices that each span `N` dimensions, enabling a form of indexing that is sometimes referred to as pointwise indexing. For example, it enables accessing the diagonal elements from the first "page" of `A` from above:

```
julia> page = A[:, :, 1]  
4×4 Array{Int64, 2}:  
 1  5   9  13  
 2  6  10  14  
 3  7  11  15  
 4  8  12  16  
  
julia> page[[CartesianIndex(1, 1),  
           CartesianIndex(2, 2),  
           CartesianIndex(3, 3),  
           CartesianIndex(4, 4)]]  
4-element Array{Int64, 1}:  
 1  
 6
```

This can be expressed much more simply with [dot broadcasting](#) and by combining it with a normal integer index (instead of extracting the first page from A as a separate step). It can even be combined with a `:` to extract both diagonals from the two pages at the same time:

```
julia> A[CartesianIndex.(indices(A, 1), indices(A, 2)), 1]
4-element Array{Int64,1}:
 1
 6
11
16

julia> A[CartesianIndex.(indices(A, 1), indices(A, 2)), :]
4×2 Array{Int64,2}:
 1  17
 6  22
11  27
16  32
```

### Warning

`CartesianIndex` and arrays of `CartesianIndex` are not compatible with the `end` keyword to represent the last index of a dimension. Do not use `end` in indexing expressions that may contain either `CartesianIndex` or arrays thereof.

### Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are

**13.1e. ARRAYS** Indexing by a boolean vector `B` is effectively the same as indexing<sup>405</sup> the vector of integers that is returned by `find(B)`. Similarly, indexing by a `N`-dimensional boolean array is effectively the same as indexing by the vector of `CartesianIndex{N}`s where its values are `true`. A logical index must be a vector of the same length as the dimension it indexes into, or it must be the only index provided and match the size and dimensionality of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling `find`.

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[[false, true, true, false], :]
2×4 Array{Int64,2}:
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Array{Bool,2}:
 true  false  false  false
 true  false  false  false
 false  false  false  false
 true  true  false  true

julia> x[mask]
5-element Array{Int64,1}:
 1
```

```
406  
2  
4  
8  
16
```

## CHAPTER 23. MULTI-DIMENSIONAL ARRAYS

### Iteration

The recommended ways to iterate over a whole array are

```
for a in A  
    # Do something with the element a  
end  
  
for i in eachindex(A)  
    # Do something with i and/or A[i]  
end
```

The first construct is used when you need the value, but not index, of each element. In the second construct, `i` will be an `Int` if `A` is an array type with fast linear indexing; otherwise, it will be a `CartesianIndex`:

```
julia> A = rand(4,3);  
  
julia> B = view(A, 1:3, 2:3);  
  
julia> for i in eachindex(B)  
        @show i  
  
    end  
i = CartesianIndex(1, 1)  
i = CartesianIndex(2, 1)  
i = CartesianIndex(3, 1)
```

23.1. ARRAYS

```
i = CartesianIndex(1, 2)
i = CartesianIndex(2, 2)
i = CartesianIndex(3, 2)
```

407

In contrast with `for i = 1:length(A)`, iterating with `eachindex` provides an efficient way to iterate over any array type.

## Array traits

If you write a custom `AbstractArray` type, you can specify that it has fast linear indexing using

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

This setting will cause `eachindex` iteration over a `MyArray` to use integers. If you don't specify this trait, the default value `IndexCartesian()` is used.

## Array and Vectorized Operators and Functions

The following operators are supported for arrays:

1. Unary arithmetic – `-`, `+`
2. Binary arithmetic – `-`, `+`, `*`, `/`, `\`, `^`
3. Comparison – `==`, `!=`, `≈` (`isapprox`),

Most of the binary arithmetic operators listed above also operate elementwise when one argument is scalar: `-`, `+`, and `*` when either argument is scalar, and `/` and `\` when the denominator is scalar. For example, `[1, 2] + 3 == [4, 5]` and `[6, 4] / 2 == [3, 2]`.

Additionally, to enable convenient vectorization of mathematical and other operations, Julia [provides the dot syntax](#) `f.(args...)`, e.g. `sin.(x)` or `min.(x, y)`, for elementwise operations over arrays or mixtures of arrays and

scalars (a [Broadcasting](#) operation) have the same behavior as arrays.

Also, every binary operator supports a [dot version](#) that can be applied to arrays (and combinations of arrays and scalars) in such [fused broadcasting operations](#), e.g. `z .== sin.(x .* y)`.

Note that comparisons such as `==` operate on whole arrays, giving a single boolean answer. Use dot operators like `.==` for elementwise comparisons. (For comparison operations like `<`, only the elementwise `.<` version is applicable to arrays.)

Also notice the difference between `max.(a, b)`, which [broadcasts](#) `max` elementwise over `a` and `b`, and `maximum(a)`, which finds the largest value within `a`. The same relationship holds for `min.(a, b)` and `minimum(a)`.

## Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

```
julia> a = rand(2,1); A = rand(2,3);
```

```
julia> repmat(a,1,3)+A
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

This is wasteful when dimensions get large, so Julia offers [broadcast](#), which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given function elementwise:

```
Julia> broadcast(+, a, A)
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1×2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2×2 Array{Float64,2}:
 1.71056  0.847604
 1.73659  0.873631
```

Dotted operators such as `.+` and `.*` are equivalent to `broadcast` calls (except that they fuse, as described below). There is also a `broadcast!` function to specify an explicit destination (which can also be accessed in a fusing fashion by `.=` assignment), and functions `broadcast_getindex` and `broadcast_setindex!` that broadcast the indices before indexing. Moreover, `f.(args...)` is equivalent to `broadcast(f, args...)`, providing a convenient syntax to broadcast any function ([dot syntax](#)). Nested "dot calls" `f.(...)` (including calls to `.+` etcetera) [automatically fuse](#) into a single `broadcast` call.

Additionally, `broadcast` is not limited to arrays (see the function documentation), it also handles tuples and treats any argument that is not an array, tuple or `Ref` (except for `Ptr`) as a "scalar".

```
julia> convert.(Float32, [1, 2])
2-element Array{Float32,1}:
 1.0f0
 2.0f0
```

```
julia> ceil.((UInt8,), [1.2 3.4; 5.6 6.7])
2×2 Array{UInt8,2}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Array{String,1}:
 "1. First"
 "2. Second"
 "3. Third"
```

## Implementation

The base array type in Julia is the abstract type `AbstractArray{T,N}`. It is parametrized by the number of dimensions `N` and the element type `T`. `AbstractVector` and `AbstractMatrix` are aliases for the 1-d and 2-d cases. Operations on `AbstractArray` objects are defined using higher level operators and functions, in a way that is independent of the underlying storage. These operations generally work correctly as a fallback for any specific array implementation.

The `AbstractArray` type includes anything vaguely array-like, and implementations of it might be quite different from conventional arrays. For example, elements might be computed on request rather than stored. However, any concrete `AbstractArray{T,N}` type should generally implement at least `size(A)` (returning an `Int` tuple), `getindex(A,i)` and `getindex(A,i1,...,in)`; mutable arrays should also implement `setindex!`. It is recommended that these operations have nearly constant time complexity, or technically (1) complexity, as otherwise some array functions may be unexpectedly slow. Concrete types should also typically provide a `similar(A,T=eltype(A),dims=size(A))`

`AbstractArray` is used to allocate a similar array for `copy` and other out-of-place operations. No matter how an `AbstractArray{T,N}` is represented internally, `T` is the type of object returned by integer indexing (`A[1, ..., 1]`, when `A` is not empty) and `N` should be the length of the tuple returned by `size`.

`DenseArray` is an abstract subtype of `AbstractArray` intended to include all arrays that are laid out at regular offsets in memory, and which can therefore be passed to external C and Fortran functions expecting this memory layout. Subtypes should provide a method `stride(A,k)` that returns the "stride" of dimension `k`: increasing the index of dimension `k` by 1 should increase the index `i` of `getindex(A,i)` by `stride(A,k)`. If a pointer conversion method `Base.unsafe_convert(Ptr{T}, A)` is provided, the memory layout should correspond in the same way to these strides.

The `Array` type is a specific instance of `DenseArray` where elements are stored in column-major order (see additional notes in [Performance Tips](#)). `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Specific operations such as scalar indexing, assignment, and a few other basic storage-specific operations are all that have to be implemented for `Array`, so that the rest of the array library can be implemented in a generic manner.

`SubArray` is a specialization of `AbstractArray` that performs indexing by reference rather than by copying. A `SubArray` is created with the `view` function, which is called the same way as `getindex` (with an array and a series of index arguments). The result of `view` looks the same as the result of `getindex`, except the data is left in place. `view` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly. By putting the `@views` macro in front of an expression or block of code, any `array[...]` slice in that expression will be converted to create a `SubArray` view instead.

## StridedVector and StridedMatrix

### CHAPTER 23. MULTIDIMENSIONAL ARRAYS

it possible for Julia to call a wider range of BLAS and LAPACK functions by passing them either [Array](#) or [SubArray](#) objects, and thus saving inefficiencies from memory allocation and copying.

The following example computes the QR decomposition of a small section of a larger array, without creating any temporaries, and by calling the appropriate LAPACK function with the right leading dimension size and stride parameters.

```
julia> a = rand(10,10)
10×10 Array{Float64,2}:
 0.561255   0.226678   0.203391   0.308912   ...  0.750307   0.235023
  ↵  0.217964
 0.718915   0.537192   0.556946   0.996234       0.666232   0.509423
  ↵  0.660788
 0.493501   0.0565622  0.118392   0.493498       0.262048   0.940693
  ↵  0.252965
 0.0470779  0.736979   0.264822   0.228787       0.161441   0.897023
  ↵  0.567641
 0.343935   0.32327    0.795673   0.452242       0.468819   0.628507
  ↵  0.511528
 0.935597   0.991511   0.571297   0.74485    ...  0.84589   0.178834
  ↵  0.284413
 0.160706   0.672252   0.133158   0.65554    ...  0.371826   0.770628
  ↵  0.0531208
 0.306617   0.836126   0.301198   0.0224702  ...  0.39344   0.0370205
  ↵  0.536062
 0.890947   0.168877   0.32002    0.486136  ...  0.096078   0.172048
  ↵  0.77672
 0.507762   0.573567   0.220124   0.165816  ...  0.211049   0.433277
  ↵  0.539476
```

```
4×2 SubAr-
```

```
→ Array{Float64, 2, Array{Float64, 2}, Tuple{StepRange{Int64}, Int64}, StepRange{Int64}}:  
0.537192  0.996234  
0.736979  0.228787  
0.991511  0.74485  
0.836126  0.0224702
```

```
julia> (q, r) = qr(b);
```

```
julia> q
```

```
4×2 Array{Float64, 2}:  
-0.338809  0.78934  
-0.464815  -0.230274  
-0.625349  0.194538  
-0.527347  -0.534856
```

```
julia> r
```

```
2×2 Array{Float64, 2}:  
-1.58553  -0.921517  
0.0        0.866567
```

## 23.2 Sparse Vectors and Matrices

Julia has built-in support for sparse vectors and [sparse matrices](#). Sparse arrays are arrays that contain enough zeros that storing them in a special data structure leads to savings in space and execution time, compared to dense arrays.

## CHAPTER 23: SPARSE MATRIX STORAGE

In Julia, sparse matrices are stored in the Compressed Sparse Column (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC{Tv,Ti}`, where `Tv` is the type of the stored values, and `Ti` is the integer type for storing column pointers and row indices. The internal representation of `SparseMatrixCSC` is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <:  
    AbstractSparseMatrix{Tv,Ti}  
    m::Int          # Number of rows  
    n::Int          # Number of columns  
    colptr::Vector{Ti}      # Column i is in  
    ↳ colptr[i]:(colptr[i+1]-1)  
    rowval::Vector{Ti}      # Row indices of stored values  
    nzval::Vector{Tv}        # Stored values, typically nonzeros  
end
```

The compressed sparse column storage makes it easy and quick to access the elements in the column of a sparse matrix, whereas accessing the sparse matrix by rows is considerably slower. Operations such as insertion of previously unstored entries one at a time in the CSC structure tend to be slow. This is because all elements of the sparse matrix that are beyond the point of insertion have to be moved one place over.

All operations on sparse matrices are carefully implemented to exploit the CSC data structure for performance, and to avoid expensive operations.

If you have data in CSC format from a different application or library, and wish to import it in Julia, make sure that you use 1-based indexing. The row indices in every column need to be sorted. If your `SparseMatrixCSC` object contains unsorted row indices, one quick way to sort them is by doing a double transpose.

MatrixCSC. These are accepted by functions in `Base` (but there is no guarantee that they will be preserved in mutating operations). Such explicitly stored zeros are treated as structural nonzeros by many routines. The `nnz` function returns the number of elements explicitly stored in the sparse data structure, including structural nonzeros. In order to count the exact number of numerical nonzeros, use `count(!iszero, x)`, which inspects every stored element of a sparse matrix. `dropzeros`, and the in-place `dropzeros!`, can be used to remove stored zeros from the sparse matrix.

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [0, 2, 0])
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 [1, 1] = 0
 [2, 2] = 2
 [3, 3] = 0

julia> dropzeros(A)
3×3 SparseMatrixCSC{Int64, Int64} with 1 stored entry:
 [2, 2] = 2
```

## Sparse Vector Storage

Sparse vectors are stored in a close analog to compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type `SparseVector{Tv, Ti}` where `Tv` is the type of the stored values and `Ti` the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
    n::Int          # Length of the sparse vector
    nzind::Vector{Ti} # Indices of stored values
    nzval::Vector{Tv} # Stored values, typically nonzeros
end
```

A<sup>16</sup>for `SparseMatrixCSC`, the `SparseVector` type that stores only non-zero entries, and `nzval` stores zeros. (See [Sparse Matrix Storage](#)).

## Sparse Vector and Matrix Constructors

The simplest way to create a sparse array is to use a function equivalent to the `zeros` function that Julia provides for working with dense arrays. To produce a sparse array instead, you can use the same name with an `sp` prefix:

```
julia> spzeros(3)
3-element SparseVector{Float64,Int64} with 0 stored entries
```

The `sparse` function is often a handy way to construct sparse arrays. For example, to construct a sparse matrix we can input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of stored values (this is also known as the [COO \(coordinate\) format](#)). `sparse(I,J,V)` then constructs a sparse matrix such that  $S[I[k], J[k]] = V[k]$ . The equivalent sparse vector constructor is `sparsevec`, which takes the (row) index vector `I` and the vector `V` with the stored values and constructs a sparse vector `R` such that  $R[I[k]] = V[k]$ .

```
julia> I = [1, 4, 3, 5]; J = [4, 7, 18, 9]; V = [1, 2, -5, 3];
```

```
julia> S = sparse(I,J,V)
5×18 SparseMatrixCSC{Int64,Int64} with 4 stored entries:
 [1 ,  4]  =  1
 [4 ,  7]  =  2
 [5 ,  9]  =  3
 [3 , 18]  = -5
```

```
julia> R = sparsevec(I,V)
5-element SparseVector{Int64,Int64} with 4 stored entries:
```

```
[1] [3] = -5  
[4] = 2  
[5] = 3
```

The inverse of the `sparse` and `sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array. There is also a `findn` function which only returns the index vectors.

```
julia> findnz(S)  
([1, 4, 5, 3], [4, 7, 9, 18], [1, 2, 3, -5])  
  
julia> findn(S)  
([1, 4, 5, 3], [4, 7, 9, 18])  
  
julia> findnz(R)  
([1, 3, 4, 5], [1, -5, 2, 3])  
  
julia> find(!iszero, R)  
4-element Array{Int64,1}:  
 1  
 3  
 4  
 5
```

Another way to create a sparse array is to convert a dense array into a sparse array using the `sparse` function:

```
julia> sparse(Matrix(1.0I, 5, 5))  
5×5 SparseMatrixCSC{Float64, Int64} with 5 stored entries:  
 [1, 1] = 1.0  
 [2, 2] = 1.0
```

```
418 [3, 3] = 1.0  
[4, 4] = 1.0  
[5, 5] = 1.0
```

## CHAPTER 23. MULTI-DIMENSIONAL ARRAYS

```
julia> sparse([1.0, 0.0, 1.0])  
3-element SparseVector{Float64,Int64} with 2 stored entries:  
[1] = 1.0  
[3] = 1.0
```

You can go in the other direction using the [Array](#) constructor. The [issparse](#) function can be used to query if a matrix is sparse.

```
julia> issparse(spzeros(5))  
true
```

### Sparse matrix operations

Arithmetic operations on sparse matrices also work as they do on dense matrices. Indexing of, assignment into, and concatenation of sparse matrices work in the same way as dense matrices. Indexing operations, especially assignment, are expensive, when carried out one element at a time. In many cases it may be better to convert the sparse matrix into  $(I, J, V)$  format using [findnz](#), manipulate the values or the structure in the dense vectors  $(I, J, V)$ , and then reconstruct the sparse matrix.

### Correspondence of dense and sparse methods

The following table gives a correspondence between built-in methods on sparse matrices and their corresponding methods on dense matrix types. In general, methods that generate sparse matrices differ from their dense counterparts in that the resulting matrix follows the same sparsity pattern as a

## 23.2.2 SPARSE VECTORS AND MATRICES

Creating sparse matrix has density  $d$ ,  
each matrix element has a probability  $d$  of being non-zero.

Details can be found in the [Sparse Vectors and Matrices](#) section of the standard library reference.

Function	Description
<code>Ar-ray{T}(dims...)</code>	an uninitialized dense <code>Array</code>
<code>zeros(T, dims...)</code>	an <code>Array</code> of all zeros
<code>zeros(A)</code>	an array of all zeros with the same type, element type and shape as <code>A</code>
<code>ones(T, dims...)</code>	an <code>Array</code> of all ones
<code>ones(A)</code>	an array of all ones with the same type, element type and shape as <code>A</code>
<code>trues(dims...)</code>	a <code>BitArray</code> with all values <code>true</code>
<code>trues(A)</code>	a <code>BitArray</code> with all values <code>true</code> and the same shape as <code>A</code>
<code>falses(dims...)</code>	a <code>BitArray</code> with all values <code>false</code>
<code>falses(A)</code>	a <code>BitArray</code> with all values <code>false</code> and the same shape as <code>A</code>
<code>reshape(A, dims...)</code>	an array containing the same data as <code>A</code> , but with different dimensions
<code>copy(A)</code>	copy <code>A</code>
<code>deepcopy(A)</code>	copy <code>A</code> , recursively copying its elements
<code>similar(A, T, dims...)</code>	an uninitialized array of the same type as <code>A</code> (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and dimensions of <code>A</code> if omitted.
<code>reinterpret(T, A)</code>	an array with the same binary data as <code>A</code> , but with element type <code>T</code>
<code>rand(T, dims...)</code>	an <code>Array</code> with random, iid <sup>1</sup> and uniformly distributed values in the half-open interval $[0, 1)$
<code>randn(T, dims...)</code>	an <code>Array</code> with random, iid and standard normally distributed values
<code>Matrix{T}{I, m, n}</code>	<code>m</code> -by- <code>n</code> identity matrix
<code>linspace(start, stop, n)</code>	range of <code>n</code> linearly spaced elements from <code>start</code> to <code>stop</code>
<code>fill!(A, x)</code>	fill the array <code>A</code> with the value <code>x</code>
<code>fill(x, dims...)</code>	an <code>Array</code> filled with the value <code>x</code>

Function	Description
<code>cat(k, A...)</code>	concatenate input $n$ -d arrays along the dimension <code>k</code>
<code>vcat(A...)</code>	shorthand for <code>cat(1, A...)</code>
<code>hcat(A...)</code>	shorthand for <code>cat(2, A...)</code>

Expression	Calls
[A; B; C; ...]	<a href="#">vcat</a>
[A B C ...]	<a href="#">hcat</a>
[A B; C D; ...]	<a href="#">hvcat</a>

Sparse	Dense	Description
<a href="#">spzeros(m, n)</a>	<a href="#">zeros(m, n)</a>	Creates a m-by-n matrix of zeros. ( <a href="#">spzeros(m, n)</a> is empty.)
<a href="#">spones(S)</a>	<a href="#">ones(m, n)</a>	Creates a matrix filled with ones. Unlike the dense version, <a href="#">spones</a> has the same sparsity pattern as S.
<a href="#">sparse(I, J, S)</a>	<a href="#">Matrix(I, J, S)</a>	Creates a n-by-n identity matrix.
<a href="#">array(S)</a>	<a href="#">sparse(A)</a>	Interconverts between dense and sparse formats.
<a href="#">sprand(m, n, d)</a>	<a href="#">rand(m, n)</a>	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed uniformly on the half-open interval [0,1).
<a href="#">sprandn(m, n, d)</a>	<a href="#">randn(m, n)</a>	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
<a href="#">sprandn(m, n, d, X)</a>	<a href="#">randn(m, n, X)</a>	Creates a m-by-n random matrix (of density d) with iid non-zero elements distributed according to the X distribution. (Requires the <a href="#">Distributions</a> package.)



# Chapter 24

## Linear algebra

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations. Basic operations, such as `trace`, `det`, and `inv` are all supported:

```
julia> A = [1 2 3; 4 1 6; 7 8 1]
3×3 Array{Int64,2}:
 1  2  3
 4  1  6
 7  8  1

julia> trace(A)
3

julia> det(A)
104.0

julia> inv(A)
3×3 Array{Float64,2}:
 -0.451923   0.211538   0.0865385
  0.365385  -0.192308   0.0576923
  0.240385   0.0576923  -0.0673077
```

```
julia> A = [-4. -17.; 2. 2.]  
2×2 Array{Float64,2}:  
-4.0 -17.0  
2.0 2.0  
  
julia> eigvals(A)  
2-element Array{Complex{Float64},1}:  
-1.0 + 5.0im  
-1.0 - 5.0im  
  
julia> eigvecs(A)  
2×2 Array{Complex{Float64},2}:  
0.945905+0.0im 0.945905-0.0im  
-0.166924-0.278207im -0.166924+0.278207im
```

In addition, Julia provides many [factorizations](#) which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. See the documentation on [factorize](#) for more information. As an example:

```
julia> A = [1.5 2 -4; 3 -1 -6; -10 2.3 4]  
3×3 Array{Float64,2}:  
1.5 2.0 -4.0  
3.0 -1.0 -6.0  
-10.0 2.3 4.0  
  
julia> factorize(A)  
Base.LinAlg.LU{Float64,Array{Float64,2}} with factors L and U:
```

```
[1.0 0.0 0.0; -0.15 1.0 0.0; -0.3 -0.132196 1.0]  
[-10.0 2.3 4.0; 0.0 2.345 -3.4; 0.0 0.0 -5.24947]
```

425

Since A is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]  
3×3 Array{Float64,2}:  
 1.5  2.0  -4.0  
 2.0  -1.0  -3.0  
 -4.0  -3.0   5.0  
  
julia> factorize(B)  
Base.LinAlg.BunchKaufman{Float64,Array{Float64,2}}  
D factor:  
3×3 Tridiagonal{Float64,Array{Float64,1}}:  
 -1.64286    0.0  
  0.0       -2.8  0.0  
          0.0  5.0  
U factor:  
3×3 Base.LinAlg.UnitUpperTriangular{Float64,Array{Float64,2}}:  
 1.0  0.142857  -0.8  
 0.0  1.0        -0.6  
 0.0  0.0        1.0  
permutation:  
3-element Array{Int64,1}:  
 1  
 2  
 3
```

Here, Julia was able to detect that B is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for

426 matrix that is known to have certain properties. For example, if you want to store a triangular matrix in memory, you can use a triangular matrix type. This is called **type tagging**. In Julia, a matrix is considered triangular if it is either upper triangular or lower triangular. A matrix is considered symmetric if it is equal to its transpose. A matrix is considered Hermitian if it is equal to its conjugate transpose. A matrix is considered unitary if its inverse is equal to its conjugate transpose. A matrix is considered orthogonal if its inverse is equal to its transpose. A matrix is considered normal if it is equal to its conjugate transpose. A matrix is considered positive definite if all of its eigenvalues are positive. A matrix is considered negative definite if all of its eigenvalues are negative. A matrix is considered semidefinite if all of its eigenvalues are non-negative. A matrix is considered indefinite if all of its eigenvalues are non-positive. A matrix is considered singular if its determinant is zero. A matrix is considered nonsingular if its determinant is non-zero. A matrix is considered invertible if it has an inverse. A matrix is considered non-invertible if it does not have an inverse. A matrix is considered sparse if it has many zero entries. A matrix is considered dense if it has few zero entries. A matrix is considered symmetric if it is equal to its transpose. A matrix is considered Hermitian if it is equal to its conjugate transpose. A matrix is considered unitary if its inverse is equal to its conjugate transpose. A matrix is considered orthogonal if its inverse is equal to its transpose. A matrix is considered normal if it is equal to its conjugate transpose. A matrix is considered positive definite if all of its eigenvalues are positive. A matrix is considered negative definite if all of its eigenvalues are negative. A matrix is considered semidefinite if all of its eigenvalues are non-negative. A matrix is considered indefinite if all of its eigenvalues are non-positive. A matrix is considered singular if its determinant is zero. A matrix is considered nonsingular if its determinant is non-zero. A matrix is considered invertible if it has an inverse. A matrix is considered non-invertible if it does not have an inverse. A matrix is considered sparse if it has many zero entries. A matrix is considered dense if it has few zero entries.

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
 1.5  2.0  -4.0
 2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

```
julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
 1.5  2.0  -4.0
 2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

sB has been tagged as a matrix that's (real) symmetric, so for later operations we might perform on it, such as eigenfactorization or computing matrix–vector products, efficiencies can be found by only referencing half of it. For example:

```
julia> B = [1.5 2 -4; 2 -1 -3; -4 -3 5]
3×3 Array{Float64,2}:
 1.5  2.0  -4.0
 2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

```
julia> sB = Symmetric(B)
3×3 Symmetric{Float64,Array{Float64,2}}:
 1.5  2.0  -4.0
 2.0  -1.0  -3.0
 -4.0  -3.0   5.0
```

```
julia> x = [1, 2, 3]
```

```
3-element Array{Int64,1}:
```

```
1
2
3
```

```
julia> sB\x
```

```
3-element Array{Float64,1}:
```

```
-1.7391304347826084
-1.1086956521739126
-1.4565217391304346
```

The `\` operation here performs the linear solution. Julia's parser provides convenient dispatch to specialized methods for the transpose of a matrix left-divided by a vector, or for the various combinations of transpose operations in matrix-matrix solutions. Many of these are further specialized for certain special matrix types. For example, `A\B` will end up calling `Base.LinAlg.A_ldiv_B!` while `A'\B` will end up calling `Base.LinAlg.Ac_ldiv_B`, even though we used the same left-division operator. This works for matrices too: `A.\B.'` would call `Base.LinAlg.At_ldiv_Bt`. The left-division operator is pretty powerful and it's easy to write compact, readable code that is flexible enough to solve all sorts of systems of linear equations.

## 24.1 Special matrices

Matrices with special symmetries and structures arise often in linear algebra and are frequently associated with various matrix factorizations. Julia features a rich collection of special matrix types, which allow for fast computation with specialized routines that are specially developed for particular matrix types.

The following tables summarize the types of special matrices that have been implemented in Julia, as well as whether hooks to various optimized methods

128 them in LAPACK are available.

## CHAPTER 24. LINEAR ALGEBRA

Type	Description
Symmetric	Symmetric matrix
Hermitian	Hermitian matrix
UpperTriangular	Upper triangular matrix
LowerTriangular	Lower triangular matrix
Tridiagonal	Tridiagonal matrix
SymTridiagonal	Symmetric tridiagonal matrix
Bidiagonal	Upper/lower bidiagonal matrix
Diagonal	Diagonal matrix
UniformScaling	Uniform scaling operator

### Elementary operations

Matrix type	+	-	*	\	Other functions with optimized methods
Symmetric				MV	inv, sqrt, exp
Hermitian				MV	inv, sqrt, exp
UpperTriangular			MV	MV	inv, det
LowerTriangular			MV	MV	inv, det
SymTridiagonal	M	M	MS	MV	eigmax, eigmin
Tridiagonal	M	M	MS	MV	
Bidiagonal	M	M	MS	MV	
Diagonal	M	M	MV	MV	inv, det, logdet, /
UniformScaling	M	M	MVS	MVS	/

### Legend:

Key	Description
M (matrix)	An optimized method for matrix-matrix operations is available
V (vector)	An optimized method for matrix-vector operations is available
S (scalar)	An optimized method for matrix-scalar operations is available

### Matrix factorizations

### Legend:

MATRIX TYPE	ESPACK	eig	eigvals	eigvecs	svd	svdvals	1429
Symmetric	SY		ARI				
Hermitian	HE		ARI				
UpperTriangular	TR	A	A	A			
LowerTriangular	TR	A	A	A			
SymTridiagonal	ST	A	ARI	AV			
Tridiagonal	GT						
Bidiagonal	BD				A	A	
Diagonal	DI		A				

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g. eigvals(M)
R (range)	An optimized method to find the $i_l$ th through the $i_h$ th characteristic values are available	eigvals(M, $i_l, i_h$ )
I (interval)	An optimized method to find the characteristic values in the interval $[v_l, v_h]$ is available	eigvals(M, $v_l, v_h$ )
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values $x=[x_1, x_2, \dots]$ is available	eigvecs(M, $x$ )

## The uniform scaling operator

A **UniformScaling** operator represents a scalar times the identity operator,  $\lambda * I$ . The identity operator  $I$  is defined as a constant and is an instance of **UniformScaling**. The size of these operators are generic and match the other matrix in the binary operations `+`, `-`, `*` and `\`. For  $A+I$  and  $A-I$  this means that  $A$  must be square. Multiplication with the identity operator  $I$  is a noop (except for checking that the scaling factor is one) and therefore almost without overhead.

To see the **UniformScaling** operator in action:

```
julia> U = UniformScaling(2);
```

```
julia> a = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
1 2
3 4
```

```
julia> a + U
2×2 Array{Int64,2}:
 3  2
 3  6

julia> a * U
2×2 Array{Int64,2}:
 2  4
 6  8

julia> [a U]
2×4 Array{Int64,2}:
 1  2  2  0
 3  4  0  2

julia> b = [1 2 3; 4 5 6]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6

julia> b - U
ERROR: DimensionMismatch("matrix is not square: dimensions are (2,
→  3)")
Stacktrace:
 [1] checksquare at ./linalg/linalg.jl:220 [inlined]
 [2] -(::Array{Int64,2}, ::UniformScaling{Int64}) at
→  ./linalg/uniformscaling.jl:156
 [3] top-level scope
```

Matrix factorizations (a.k.a. matrix decompositions) compute the factorization of a matrix into a product of matrices, and are one of the central concepts in linear algebra.

The following table summarizes the types of matrix factorizations that have been implemented in Julia. Details of their associated methods can be found in the [Linear Algebra](#) section of the standard library documentation.

Type	Description
<code>Cholesky</code>	<code>Cholesky</code> factorization
<code>CholeskyPivoted</code>	Pivoted Cholesky factorization
<code>LU</code>	<code>LU</code> factorization
<code>LUTridiagonal</code>	LU factorization for <code>Tridiagonal</code> matrices
<code>UmfpackLU</code>	LU factorization for sparse matrices (computed by UMFPack)
<code>QR</code>	<code>QR</code> factorization
<code>QRCompactWY</code>	Compact WY form of the QR factorization
<code>QRPivoted</code>	Pivoted <code>QR</code> factorization
<code>Hessenberg</code>	<code>Hessenberg</code> decomposition
<code>Eigen</code>	Spectral decomposition
<code>SVD</code>	Singular value decomposition
<code>GeneralizedSVD</code>	Generalized SVD



## Chapter 25

# Networking and Streams

Julia provides a rich interface to deal with streaming I/O objects such as terminals, pipes and TCP sockets. This interface, though asynchronous at the system level, is presented in a synchronous manner to the programmer and it is usually unnecessary to think about the underlying asynchronous operation. This is achieved by making heavy use of Julia cooperative threading ([coroutine](#)) functionality.

### 25.1 Basic Stream I/O

All Julia streams expose at least a `read` and a `write` method, taking the stream as their first argument, e.g.:

```
julia> write(STDOUT, "Hello World"); # suppress return value 11
→ with ;
Hello World
julia> read(STDIN, Char)

'\n': ASCII/Unicode U+000a (category Cc: Other, control)
```

Note that `write` returns 11, the number of bytes (in "Hello World") written to `STDOUT`, but this return value is suppressed with the `;`.

434 Enter was pressed again so CHAPTER 25 NETWORKING AND STREAMS you can see from this example, `write` takes the data to write as its second argument, while `read` takes the type of the data to be read as the second argument.

For example, to read a simple byte array, we could do:

```
julia> x = zeros(UInt8, 4)
4-element Array{UInt8,1}:
 0x00
 0x00
 0x00
 0x00

julia> read!(STDIN, x)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```
julia> read(STDIN, 4)
abcd
4-element Array{UInt8,1}:
 0x61
 0x62
 0x63
 0x64
```

```
julia> readline(STDIN)  
abcd  
"abcd"
```

Note that depending on your terminal settings, your TTY may be line buffered and might thus require an additional enter before the data is sent to Julia.

To read every line from `STDIN` you can use `eachline`:

```
for line in eachline(STDIN)  
    print("Found $line")  
end
```

or `read` if you wanted to read by character instead:

```
while !eof(STDIN)  
    x = read(STDIN, Char)  
    println("Found: $x")  
end
```

## 25.2 Text I/O

Note that the `write` method mentioned above operates on binary streams. In particular, values do not get converted to any canonical text representation but are written out as is:

```
julia> write(STDOUT, 0x61); # suppress return value 1 with ;  
a
```

Note that `a` is written to `STDOUT` by the `write` function and that the returned value is 1 (since `0x61` is one byte).

For text I/O, use the `print` or `show` functions. See [CHAPTER 25. NETWORKING AND STREAMS](#) for the standard library reference for a detailed discussion of the difference between the two):

```
julia> print(STDOUT, 0x61)  
97
```

## 25.3 IO Output Contextual Properties

Sometimes IO output can benefit from the ability to pass contextual information into show methods. The `IOContext` object provides this framework for associating arbitrary metadata with an IO object. For example, `showcompact` adds a hinting parameter to the IO object that the invoked show method should print a shorter output (if applicable).

## 25.4 Working with Files

Like many other environments, Julia has an `open` function, which takes a filename and returns an `IOStream` object that you can use to read and write things from the file. For example if we have a file, `hello.txt`, whose contents are `Hello, World!`:

```
julia> f = open("hello.txt")  
IOStream(<file hello.txt>)  
  
julia> readlines(f)  
1-element Array{String,1}:  
"Hello, World!"
```

If you want to write to a file, you can open it with the write ("w") flag:

```
julia> f = open("hello.txt", "w")  
IOStream(<file hello.txt>)
```

```
julia> write(f, "Hello again.")  
12
```

If you examine the contents of `hello.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IOStream` must be closed before the write is actually flushed to disk:

```
julia> close(f)
```

Examining `hello.txt` again will show its contents have been changed.

Opening a file, doing something to its contents, and closing it again is a very common pattern. To make this easier, there exists another invocation of `open` which takes a function as its first argument and filename as its second, opens the file, calls the function with the file as an argument, and then closes it again.

For example, given a function:

```
function read_and_capitalize(f::IOStream)  
    return uppercase(read(f, String))  
end
```

You can call:

```
julia> open(read_and_capitalize, "hello.txt")  
"HELLO AGAIN."
```

to open `hello.txt`, call `read_and_capitalize` on it, close `hello.txt` and return the capitalized contents.

To avoid even having to define a named function, you can use the `do` syntax, which creates an anonymous function on the fly:

438

CHAPTER 25. NETWORKING AND STREAMS

```
julia> open("hello.txt") do f
           uppercase(read(f, String))
       end
"HELLO AGAIN."
```

## 25.5 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. Let's first create a simple server:

```
julia> @async begin
           server = listen(2000)
           while true
               sock = accept(server)
               println("Hello World\n")
           end
       end
Task (runnable) @0x00007fd31dc11ae0
```

To those familiar with the Unix socket API, the method names will feel familiar, though their usage is somewhat simpler than the raw Unix socket API. The first call to `listen` will create a server waiting for incoming connections on the specified port (2000) in this case. The same function may also be used to create various other kinds of servers:

```
Base.TCPServer(active)
```

```
julia> listen(ip"127.0.0.1",2000) # Equivalent to the first
```

```
Base.TCPServer(active)
```

```
julia> listen(ip":1",2000) # Listens on localhost:2000 (IPv6)
```

```
Base.TCPServer(active)
```

```
julia> listen(IPv4(0),2001) # Listens on port 2001 on all IPv4
```

```
→ interfaces
```

```
Base.TCPServer(active)
```

```
julia> listen(IPv6(0),2001) # Listens on port 2001 on all IPv6
```

```
→ interfaces
```

```
Base.TCPServer(active)
```

```
julia> listen("testsocket") # Listens on a UNIX domain socket
```

```
Base.PipeServer(active)
```

```
julia> listen("\\\\.\\pipe\\testsocket") # Listens on a Windows
```

```
→ named pipe
```

```
Base.PipeServer(active)
```

Note that the return type of the last invocation is different. This is because this server does not listen on TCP, but rather on a named pipe (Windows) or UNIX domain socket. Also note that Windows named pipe format has to be a specific pattern such that the name prefix (\.\.\pipe\) uniquely identifies the *file type*. The difference between TCP and named pipes or UNIX domain sockets is subtle and has to do with the `accept` and `connect` methods. The `accept` method retrieves a connection to the client that is connecting on the

server we just created, while the `connect` function returns a socket object for the specified method. The `connect` function takes the same arguments as `listen`, so, assuming the environment (i.e. host, cwd, etc.) is the same you should be able to pass the same arguments to `connect` as you did to listen to establish the connection. So let's try that out (after having created the server above):

```
julia> connect(2000)
TCPSocket(open, 0 bytes waiting)

julia> Hello World
```

As expected we saw "Hello World" printed. So, let's actually analyze what happened behind the scenes. When we called `connect`, we connect to the server we had just created. Meanwhile, the `accept` function returns a server-side connection to the newly created socket and prints "Hello World" to indicate that the connection was successful.

A great strength of Julia is that since the API is exposed synchronously even though the I/O is actually happening asynchronously, we didn't have to worry about callbacks or even making sure that the server gets to run. When we called `connect` the current task waited for the connection to be established and only continued executing after that was done. In this pause, the server task resumed execution (because a connection request was now available), accepted the connection, printed the message and waited for the next client. Reading and writing works in the same way. To see this, consider the following simple echo server:

```
julia> @async begin
           server = listen(2001)
```

```
    sock = accept(server)

    @async while isopen(sock)

        write(sock, readline(sock))

    end

end

end

Task (runnable) @0x00007fd31dc12e60

julia> clientside = connect(2001)
TCPSocket(RawFD(28) open, 0 bytes waiting)

julia> @async while true

        write(STDOUT, readline(clientside))

    end

Task (runnable) @0x00007fd31dc11870

julia> println(clientside, "Hello World from the Echo Server")
Hello World from the Echo Server
```

As with other streams, use `close` to disconnect the socket:

```
julia> close(clientside)
```

## 25.6 Resolving IP Addresses

### CHAPTER 25. NETWORKING AND STREAMS

One of the `connect` methods that does not follow the `listen` methods is `connect(host::String, port)`, which will attempt to connect to the host given by the `host` parameter on the port given by the `port` parameter. It allows you to do things like:

```
julia> connect("google.com", 80)
TCP Socket(RawFD(30) open, 0 bytes waiting)
```

At the base of this functionality is `getaddrinfo`, which will do the appropriate address resolution:

```
julia> getaddrinfo("google.com")
ip"74.125.226.225"
```

## Chapter 26

# Parallel Computing

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly. There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the [cache](#). Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once.

Julia's implementation of message passing is different from other environments such as MPI <sup>1</sup>. Communication in Julia is generally "one-sided", meaning that the programmer needs to explicitly manage only one process in a two-process operation. Furthermore, these operations typically do not look like "message send" and "message receive" but rather resemble higher-level operations like calls to user functions.

Parallel programming in Julia is built on two primitives: remote references and

44 CHAPTER 4. PARALLEL COMPUTING

Remote calls. A remote reference is a process to refer to an object stored on a particular process. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Remote references come in two flavors: [Future](#) and [RemoteChannel](#).

A remote call returns a [Future](#) to its result. Remote calls return immediately; the process that made the call proceeds to its next operation while the remote call happens somewhere else. You can wait for a remote call to finish by calling [wait](#) on the returned [Future](#), and you can obtain the full value of the result using [fetch](#).

On the other hand, [RemoteChannel](#)s are rewritable. For example, multiple processes can co-ordinate their processing by referencing the same remote Channel.

Each process has an associated identifier. The process providing the interactive Julia prompt always has an [id](#) equal to 1. The processes used by default for parallel operations are referred to as "workers". When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

Let's try this out. Starting with `julia -p n` provides `n` worker processes on the local machine. Generally it makes sense for `n` to equal the number of CPU cores on the machine.

```
$ ./julia -p 2

julia> r = remotecall(rand, 2, 2, 2)
Future(2, 1, 4, Nullable{Any}())

julia> s = @spawnat 2 1 .+ fetch(r)
Future(2, 1, 5, Nullable{Any}())
```

```
julia> fetch(s)
2×2 Array{Float64,2}:
 1.18526  1.50912
 1.16296  1.60607
```

The first argument to `remotecall` is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but `remotecall` is considered a low-level interface providing finer control. The second argument to `remotecall` is the `id` of the process that will do the work, and the remaining arguments will be passed to the function being called.

As you can see, in the first line we asked process 2 to construct a 2-by-2 random matrix, and in the second line we asked it to add 1 to it. The result of both calculations is available in the two futures, `r` and `s`. The `@spawnat` macro evaluates the expression in the second argument on the process specified by the first argument.

Occasionally you might want a remotely-computed value immediately. This typically happens when you read from a remote object to obtain data needed by the next local operation. The function `remotecall_fetch` exists for this purpose. It is equivalent to `fetch(remotecall(...))` but is more efficient.

```
julia> remotecall_fetch(getindex, 2, r, 1, 1)
0.18526337335308085
```

Remember that `getindex(r, 1, 1)` is equivalent to `r[1, 1]`, so this call fetches the first element of the future `r`.

The syntax of `remotecall` is not especially convenient. The macro `@spawn` makes things easier. It operates on an expression rather than a function, and picks where to do the operation for you:

```
Julia> r = @spawn rand(2,2)
Future(2, 1, 4, Nullable{Any}())
```

```
julia> s = @spawn 1 .+ fetch(r)
Future(3, 1, 5, Nullable{Any}())
```

```
julia> fetch(s)
2×2 Array{Float64,2}:
 1.38854  1.9098
 1.20939  1.57158
```

Note that we used `1 .+ fetch(r)` instead of `1 .+ r`. This is because we do not know where the code will run, so in general a `fetch` might be required to move `r` to the process doing the addition. In this case, `@spawn` is smart enough to perform the computation on the process that owns `r`, so the `fetch` will be a no-op (no work is done).

(It is worth noting that `@spawn` is not built-in but defined in Julia as a `macro`. It is possible to define your own such constructs.)

An important thing to remember is that, once fetched, a `Future` will cache its value locally. Further `fetch` calls do not entail a network hop. Once all referencing `Futures` have fetched, the remote stored value is deleted.

## 26.1 Code Availability and Loading Packages

Your code must be available on any process that runs it. For example, type the following into the Julia prompt:

```
julia> function rand2(dims...)
           return 2*rand(dims...)
end
```

```
julia> rand2(2,2)
2×2 Array{Float64,2}:
 0.153756  0.368514
 1.15119   0.918912

julia> fetch(@spawn rand2(2,2))
ERROR: RemoteException(2,
→ CapturedException(UndefVarError(Symbol("#rand2")))
Stacktrace:
[...]
```

Process 1 knew about the function `rand2`, but process 2 did not.

Most commonly you'll be loading code from files or packages, and you have a considerable amount of flexibility in controlling which processes load code. Consider a file, `DummyModule.jl`, containing the following code:

```
module DummyModule

export MyType, f

mutable struct MyType
    a::Int
end

f(x) = x^2+1

println("loaded")

end
```

include("DummyModule.jl") loads the file on just a single process (whichever one executes the statement).

using DummyModule causes the module to be loaded on all processes; however, the module is brought into scope only on the one executing the statement.

As long as DummyModule is loaded on process 2, commands like

```
| rr = RemoteChannel(2)
| put!(rr, MyType(7))
```

allow you to store an object of type MyType on process 2 even if DummyModule is not in scope on process 2.

You can force a command to run on all processes using the @everywhere macro. For example, @everywhere can also be used to directly define a function on all processes:

```
julia> @everywhere id = myid()
julia> remotecall_fetch(()->id, 2)
2
```

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

```
julia -p <n> -L file1.jl -L file2.jl driver.jl
```

The Julia process running the driver script in the example above has an id equal to 1, just like a process providing an interactive prompt.

The base Julia installation has in-built support for two types of clusters:

A cluster spanning machines using the `--machinefile` option. This uses a passwordless `ssh` login to start Julia worker processes (from the same path as the current host) on the specified machines.

Functions `addprocs`, `rmprocs`, `workers`, and others are available as a programmatic means of adding, removing and querying the processes in a cluster.

Note that workers do not run a `.juliarc.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

Other types of clusters can be supported by writing your own custom `ClusterManager`, as described below in the [ClusterManagers](#) section.

## 26.2 Data Movement

Sending messages and moving data constitute most of the overhead in a parallel program. Reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. To this end, it is important to understand the data movement performed by Julia's various parallel programming constructs.

`fetch` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `@spawn` (and a few related constructs) also moves data, but this is not as obvious, hence it can be called an implicit data movement operation. Consider these two approaches to constructing and squaring a random matrix:

Method 1:

```
julia> A = rand(1000,1000);
```

450  
**julia>** Bref = @spawn A^2;

CHAPTER 26. PARALLEL COMPUTING

[ ... ]

**julia>** fetch(Bref);

Method 2:

**julia>** Bref = @spawn rand(1000,1000)^2;

[ ... ]

**julia>** fetch(Bref);

The difference seems trivial, but in fact is quite significant due to the behavior of `@spawn`. In the first method, a random matrix is constructed locally, then sent to another process where it is squared. In the second method, a random matrix is both constructed and squared on another process. Therefore the second method sends much less data than the first.

In this toy example, the two methods are easy to distinguish and choose from. However, in a real program designing data movement might require more thought and likely some measurement. For example, if the first process needs matrix `A` then the first method might be better. Or, if computing `A` is expensive and only the current process has it, then moving it to another process might be unavoidable. Or, if the current process has very little to do between the `@spawn` and `fetch(Bref)`, it might be better to eliminate the parallelism altogether. Or imagine `rand(1000,1000)` is replaced with a more expensive operation. Then it might make sense to add another `@spawn` statement just for this step.

# Chapter 27

## Global variables

Expressions executed remotely via `@spawn`, or closures specified for remote execution using `remotecall` may refer to global variables. Global bindings under module `Main` are treated a little differently compared to global bindings in other modules. Consider the following code snippet:

```
A = rand(10,10)
remotecall_fetch(()->norm(A), 2)
```

In this case `norm` is a function that takes 2D array as a parameter, and MUST be defined in the remote process. You could use any function other than `norm` as long as it is defined in the remote process and accepts the appropriate parameter.

Note that `A` is a global variable defined in the local workspace. Worker 2 does not have a variable called `A` under `Main`. The act of shipping the closure `(())->norm(A)` to worker 2 results in `Main.A` being defined on 2. `Main.A` continues to exist on worker 2 even after the call `remotecall_fetch` returns. Remote calls with embedded global references (under `Main` module only) manage globals as follows:

New global bindings are created on destination workers if they are ref-

Global constants are declared as constants on remote nodes too.

Globals are re-sent to a destination worker only in the context of a remote call, and then only if its value has changed. Also, the cluster does not synchronize global bindings across nodes. For example:

```
A = rand(10,10)
remotecall_fetch(()->norm(A), 2) # worker 2
A = rand(10,10)
remotecall_fetch(()->norm(A), 3) # worker 3
A = nothing
```

Executing the above snippet results in `Main.A` on worker 2 having a different value from `Main.A` on worker 3, while the value of `Main.A` on node 1 is set to `nothing`.

As you may have realized, while memory associated with globals may be collected when they are reassigned on the master, no such action is taken on the workers as the bindings continue to be valid. `clear!` can be used to manually reassign specific globals on remote nodes to `nothing` once they are no longer required. This will release any memory associated with them as part of a regular garbage collection cycle.

Thus programs should be careful referencing globals in remote calls. In fact, it is preferable to avoid them altogether if possible. If you must reference globals, consider using `let` blocks to localize global variables.

For example:

```
julia> A = rand(10,10);

julia> remotecall_fetch(()->A, 2);
```

```
julia> B = rand(10,10);

julia> let B = B

                    remotecall_fetch(()->B, 2)

    end;

julia> @spawnat 2 whos();

julia> From worker 2:                                A      800
→   bytes  10×10 Array{Float64,2}

          From worker 2:                                Base
→   Module

          From worker 2:                                Core
→   Module

          From worker 2:                                Main
→   Module
```

As can be seen, global variable A is defined on worker 2, but B is captured as a local variable and hence a binding for B does not exist on worker 2.

## 27.1 Parallel Map and Loops

Fortunately, many useful parallel computations do not require data movement. A common example is a Monte Carlo simulation, where multiple processes can handle independent simulation trials simultaneously. We can use `@spawn` to flip coins on two processes. First, write the following function in `count_heads.jl`:

```
454 function count_heads(n)
    c::Int = 0
    for i = 1:n
        c += rand(Bool)
    end
    c
end
```

## CHAPTER 27. GLOBAL VARIABLES

The function `count_heads` simply adds together `n` random bits. Here is how we can perform some trials on two machines, and add together the results:

```
julia> @everywhere include_string(Main, $(read("count_heads.jl",
    String)), "count_heads.jl")

julia> a = @spawn count_heads(100000000)
Future(2, 1, 6, Nullable{Any}())

julia> b = @spawn count_heads(100000000)
Future(3, 1, 7, Nullable{Any}())

julia> fetch(a)+fetch(b)
100001564
```

This example demonstrates a powerful and often-used parallel programming pattern. Many iterations run independently over several processes, and then their results are combined using some function. The combination process is called a reduction, since it is generally tensor-rank-reducing: a vector of numbers is reduced to a single number, or a matrix is reduced to a single row or column, etc. In code, this typically looks like the pattern `x = f(x, v[i])`, where `x` is the accumulator, `f` is the reduction function, and the `v[i]` are the elements being reduced. It is desirable for `f` to be associative, so that it does not matter what order the operations are performed in.

We used two explicit `@spawn` statements, which limits the parallelism to two processes. To run on any number of processes, we can use a parallel for loop, which can be written in Julia using `@parallel` like this:

```
nheads = @parallel (+) for i = 1:200000000
    Int(rand(Bool))
end
```

This construct implements the pattern of assigning iterations to multiple processes, and combining them with a specified reduction (in this case `(+)`). The result of each iteration is taken as the value of the last expression inside the loop. The whole parallel loop expression itself evaluates to the final answer.

Note that although parallel for loops look like serial for loops, their behavior is dramatically different. In particular, the iterations do not happen in a specified order, and writes to variables or arrays will not be globally visible since iterations run on different processes. Any variables used inside the parallel loop will be copied and broadcast to each process.

For example, the following code will not work as intended:

```
a = zeros(100000)
@parallel for i = 1:100000
    a[i] = i
end
```

This code will not initialize all of `a`, since each process will have a separate copy of it. Parallel for loops like these must be avoided. Fortunately, [Shared Arrays](#) can be used to get around this limitation:

```
a = SharedArray{Float64}(10)
@parallel for i = 1:10
```

```
456 a[i] = i  
end
```

## CHAPTER 27. GLOBAL VARIABLES

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
a = randn(1000)  
@parallel (+) for i = 1:100000  
    f(a[rand(1:end)])  
end
```

Here each iteration applies `f` to a randomly-chosen sample from a vector `a` shared by all processes.

As you could see, the reduction operator can be omitted if it is not needed. In that case, the loop executes asynchronously, i.e. it spawns independent tasks on all available workers and returns an array of `Future` immediately without waiting for completion. The caller can wait for the `Future` completions at a later point by calling `fetch` on them, or wait for completion at the end of the loop by prefixing it with `@sync`, like `@sync @parallel for`.

In some cases no reduction operator is needed, and we merely wish to apply a function to all integers in some range (or, more generally, to all elements in some collection). This is another useful operation called parallel map, implemented in Julia as the `pmap` function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
julia> M = Matrix{Float64}[rand(1000,1000) for i = 1:10];  
  
julia> pmap(svd, M);
```

Julia's `pmap` is designed for the case where each function call does a large amount of work. In contrast, `@parallel for` can handle situations where

27.2 SYNCHRONIZATION WITH REMOTE REFERENCES 457  
cesses are used by both `pmap` and `@parallel for` for the parallel computation. In case of `@parallel for`, the final reduction is done on the calling process.

## 27.2 Synchronization With Remote References

### 27.3 Scheduling

Julia's parallel programming platform uses [Tasks \(aka Coroutines\)](#) to switch among multiple computations. Whenever code performs a communication operation like `fetch` or `wait`, the current task is suspended and a scheduler picks another task to run. A task is restarted when the event it is waiting for completes.

For many problems, it is not necessary to think about tasks directly. However, they can be used to wait for multiple events at the same time, which provides for dynamic scheduling. In dynamic scheduling, a program decides what to compute or where to compute it based on when other jobs finish. This is needed for unpredictable or unbalanced workloads, where we want to assign more work to processes only when they finish their current tasks.

As an example, consider computing the singular values of matrices of different sizes:

```
julia> M = Matrix{Float64}[rand(800,800), rand(600,600),
           ↪   rand(800,800), rand(600,600)];
julia> pmap(svd, M);
```

If one process handles both  $800 \times 800$  matrices and another handles both  $600 \times 600$  matrices, we will not get as much scalability as we could. The solution

458 To make a local task to "feed" work to each process when it completes its current task. For example, consider a simple `pmap` implementation:

```
function pmap(f, lst)
    np = nprocs() # determine the number of processes available
    n = length(lst)
    results = Vector{Any}(n)
    i = 1
    # function to produce the next work item from the queue.
    # in this case it's just an index.
    nextidx() = (idx=i; i+=1; idx)
    @sync begin
        for p=1:np
            if p != myid() || np == 1
                @async begin
                    while true
                        idx = nextidx()
                        if idx > n
                            break
                        end
                        results[idx] = remotecall_fetch(f, p,
                            ↳ lst[idx])
                    end
                end
            end
        end
    end
    results
end
```

`@async` is similar to `@spawn`, but only runs tasks on the local process. We use it to create a "feeder" task for each process. Each task picks the next index that

~~27.4 Channels~~ 459

we run out of indexes. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `@sync` block, at which point it surrenders control and waits for all the local tasks to complete before returning from the function. The feeder tasks are able to share state via `nextidx` because they all run on the same process. No locking is required, since the threads are scheduled cooperatively and not preemptively. This means context switches only occur at well-defined points: in this case, when `remotecall_fetch` is called.

## 27.4 Channels

The section on [Tasks](#) in [Control Flow](#) discussed the execution of multiple functions in a co-operative manner. [Channel](#)s can be quite useful to pass data between running tasks, particularly those involving I/O operations.

Examples of operations involving I/O include reading/writing to files, accessing web services, executing external programs, etc. In all these cases, overall execution time can be improved if other tasks can be run while a file is being read, or while waiting for an external service/program to complete.

A channel can be visualized as a pipe, i.e., it has a write end and read end.

Multiple writers in different tasks can write to the same channel concurrently via `put!` calls.

Multiple readers in different tasks can read data concurrently via `take!` calls.

As an example:

```
# Given Channels c1 and c2,  
c1 = Channel(32)  
c2 = Channel(32)
```

```

# and a function `foo` which reads items from from c1, processes
→ the item read
# and writes a result to c2,
function foo()
    while true
        data = take!(c1)
        [...]                      # process data
        put!(c2, result)          # write out result
    end
end

# we can schedule `n` instances of `foo` to be active
→ concurrently.
for _ in 1:n
    @schedule foo()
end

```

Channels are created via the `Channel{T}(sz)` constructor. The channel will only hold objects of type T. If the type is not specified, the channel can hold objects of any type. sz refers to the maximum number of elements that can be held in the channel at any time. For example, `Channel(32)` creates a channel that can hold a maximum of 32 objects of any type. A `Channel{MyType}(64)` can hold up to 64 objects of MyType at any time.

If a `Channel` is empty, readers (on a `take!` call) will block until data is available.

If a `Channel` is full, writers (on a `put!` call) will block until space becomes available.

`isready` tests for the presence of any object in the channel, while `wait` waits for an object to become available.

27.4A `Channel` is in an open state initially. This means that it can be read from and written to freely via `take!` and `put!` calls. `close` closes a `Channel`. On a closed `Channel`, `put!` will fail. For example:

```
julia> c = Channel(2);

julia> put!(c, 1) # `put!` on an open channel succeeds
1

julia> close(c);

julia> put!(c, 2) # `put!` on a closed channel throws an exception.
ERROR: InvalidStateException("Channel is closed.", :closed)
Stacktrace:
[...]
```

`take!` and `fetch` (which retrieves but does not remove the value) on a closed channel successfully return any existing values until it is emptied. Continuing the above example:

```
julia> fetch(c) # Any number of `fetch` calls succeed.
1

julia> fetch(c)
1

julia> take!(c) # The first `take!` removes the value.
1

julia> take!(c) # No more data available on a closed channel.
ERROR: InvalidStateException("Channel is closed.", :closed)
```

462  
Stacktrace:  
[...]

## CHAPTER 27. GLOBAL VARIABLES

A `Channel` can be used as an iterable object in a `for` loop, in which case the loop runs as long as the `Channel` has data or is open. The loop variable takes on all values added to the `Channel`. The `for` loop is terminated once the `Channel` is closed and emptied.

For example, the following would cause the `for` loop to wait for more data:

```
julia> c = Channel{Int}(10);  
  
julia> foreach(i->put!(c, i), 1:3) # add a few entries  
  
julia> data = [i for i in c]
```

while this will return after reading all data:

```
julia> c = Channel{Int}(10);  
  
julia> foreach(i->put!(c, i), 1:3); # add a few entries  
  
julia> close(c); # `for` loops can exit  
  
julia> data = [i for i in c]  
3-element Array{Int64,1}:  
1  
2  
3
```

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single `jobs` channel. Jobs, identified by an id (`job_id`), are written to the channel. Each task in this simulation reads a

`job_id` channel for a random amount of time and writes back a tuple of `job_id` and the simulated time to the results channel. Finally all the `results` are printed out.

```
julia> const jobs = Channel{Int}(32);

julia> const results = Channel{Tuple}((32);

julia> function do_work()
    for job_id in jobs
        exec_time = rand()
        sleep(exec_time)           # simulates elapsed
        → time doing actual work
        → performed externally.
        put!(results, (job_id, exec_time))
    end
end;

julia> function make_jobs(n)
    for i in 1:n
        put!(jobs, i)
```

```
        end

    end;

julia> n = 12;

julia> @schedule make_jobs(n); # feed the jobs channel with "n"
→   jobs

julia> for i in 1:4 # start 4 tasks to process requests in parallel

        @schedule do_work()

    end

julia> @elapsed while n > 0 # print out results

        job_id, exec_time = take!(results)

        println("$job_id finished in $(round(exec_time,2))
→   seconds")

        n = n - 1

    end

4 finished in 0.22 seconds
3 finished in 0.45 seconds
1 finished in 0.5 seconds
7 finished in 0.14 seconds
2 finished in 0.78 seconds
5 finished in 0.9 seconds
```

```
6 finished in 0.87 seconds
8 finished in 0.79 seconds
10 finished in 0.64 seconds
12 finished in 0.5 seconds
11 finished in 0.97 seconds
0.029772311
```

The current version of Julia multiplexes all tasks onto a single OS thread. Thus, while tasks involving I/O operations benefit from parallel execution, compute bound tasks are effectively executed sequentially on a single OS thread. Future versions of Julia may support scheduling of tasks on multiple threads, in which case compute bound tasks will see benefits of parallel execution too.

## 27.5 Remote References and AbstractChannels

Remote references always refer to an implementation of an `AbstractChannel`.

A concrete implementation of an `AbstractChannel` (like `Channel`), is required to implement `put!`, `take!`, `fetch`, `isready` and `wait`. The remote object referred to by a `Future` is stored in a `Channel{Any}(1)`, i.e., a `Channel` of size 1 capable of holding objects of `Any` type.

`RemoteChannel`, which is rewritable, can point to any type and size of channels, or any other implementation of an `AbstractChannel`.

The constructor `RemoteChannel(f::Function, pid)()` allows us to construct references to channels holding more than one value of a specific type. `f` is a function executed on `pid` and it must return an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10. The channel exists on worker `pid`.

proxied onto the backing store on the remote process.

`RemoteChannel` can thus be used to refer to user implemented `AbstractChannel` objects. A simple example of this is provided in `examples/dictchannel.jl` which uses a dictionary as its remote store.

## 27.6 Channels and RemoteChannels

A `Channel` is local to a process. Worker 2 cannot directly refer to a `Channel` on worker 3 and vice-versa. A `RemoteChannel`, however, can put and take values across workers.

A `RemoteChannel` can be thought of as a handle to a `Channel`.

The process id, `pid`, associated with a `RemoteChannel` identifies the process where the backing store, i.e., the backing `Channel` exists.

Any process with a reference to a `RemoteChannel` can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a `RemoteChannel` is associated with.

Serializing a `Channel` also serializes any data present in the channel. Deserializing it therefore effectively makes a copy of the original object.

On the other hand, serializing a `RemoteChannel` only involves the serialization of an identifier that identifies the location and instance of `Channel` referred to by the handle. A deserialized `RemoteChannel` object (on any worker), therefore also points to the same backing store as the original.

The channels example from above can be modified for interprocess communication, as shown below.

We start 4 workers to process a single `jobs` remote channel. Jobs, identified by an id (`job_id`), are written to the channel. Each remotely executing task in

THIS SECTION IS ADOPTED FROM THE JULIA DOCS

163

CHANNELS AND REMOTE CHANNELS

random amount of time and writes back a tuple of `job_id`, time taken and its own `pid` to the results channel. Finally all the `results` are printed out on the master process.

```
julia> addprocs(4); # add worker processes

julia> const jobs = RemoteChannel(()->Channel{Int}(32));

julia> const results = RemoteChannel(()->Channel{Tuple}(32));

julia> @everywhere function do_work(jobs, results) # define work
  ↪   function everywhere

    while true

      job_id = take!(jobs)

      exec_time = rand()

      sleep(exec_time) # simulates elapsed time doing
  ↪   actual work

      put!(results, (job_id, exec_time, myid()))

    end

  end

julia> function make_jobs(n)

  for i in 1:n
```

```
    end

    end;

julia> n = 12;

julia> @schedule make_jobs(n); # feed the jobs channel with "n"
→   jobs

julia> for p in workers() # start tasks on the workers to process
→   requests in parallel

    remote_do(do_work, p, jobs, results)

end

julia> @elapsed while n > 0 # print out results

    job_id, exec_time, where = take!(results)

        println("$job_id finished in $(round(exec_time,2))
→  seconds on worker $where")

    n = n - 1

end

1 finished in 0.18 seconds on worker 4
2 finished in 0.26 seconds on worker 5
6 finished in 0.12 seconds on worker 4
```

```
7 finished in 0.18 seconds on worker 4
5 finished in 0.35 seconds on worker 5
4 finished in 0.68 seconds on worker 2
3 finished in 0.73 seconds on worker 3
11 finished in 0.01 seconds on worker 3
12 finished in 0.02 seconds on worker 3
9 finished in 0.26 seconds on worker 5
8 finished in 0.57 seconds on worker 4
10 finished in 0.58 seconds on worker 2
0.055971741
```

## 27.7 Remote References and Distributed Garbage Collection

Objects referred to by remote references can be freed only when all held references in the cluster are deleted.

The node where the value is stored keeps track of which of the workers have a reference to it. Every time a [RemoteChannel](#) or a (unfetched) [Future](#) is serialized to a worker, the node pointed to by the reference is notified. And every time a [RemoteChannel](#) or a (unfetched) [Future](#) is garbage collected locally, the node owning the value is again notified. This is implemented in an internal cluster aware serializer. Remote references are only valid in the context of a running cluster. Serializing and deserializing references to and from regular [I0](#) objects is not supported.

The notifications are done via sending of "tracking" messages – an "add reference" message when a reference is serialized to a different process and a "delete reference" message when a reference is locally garbage collected.

Since [Futures](#) are write-once and cached locally, the act of [fetching](#) a [Future](#) also updates reference tracking information on the node owning the value.

With [Future](#)s, serializing an already fetched [Future](#) to a different node also sends the value since the original remote store may have collected the value by this time.

It is important to note that when an object is locally garbage collected depends on the size of the object and the current memory pressure in the system.

In case of remote references, the size of the local reference object is quite small, while the value stored on the remote node may be quite large. Since the local object may not be collected immediately, it is a good practice to explicitly call [finalize](#) on local instances of a [RemoteChannel](#), or on unfetched [Future](#)s. Since calling [fetch](#) on a [Future](#) also removes its reference from the remote store, this is not required on fetched [Future](#)s. Explicitly calling [finalize](#) results in an immediate message sent to the remote node to go ahead and remove its reference to the value.

Once finalized, a reference becomes invalid and cannot be used in any further calls.

## 27.8 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a [DArray](#), the behavior of a [SharedArray](#) is quite different. In a [DArray](#), each process has local access to just a chunk of the data, and no two processes share the same chunk; in contrast, in a [SharedArray](#) each "participating" process has access to the entire array. A [SharedArray](#) is a good choice when you want to have a large amount of data jointly accessible to two or more processes on the same machine.

[SharedArray](#) indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available

**27.8. Shared Arrays** Therefore, most algorithms work naturally on `SharedArray`s, albeit in single-process mode. In cases where an algorithm insists on an `Array` input, the underlying array can be retrieved from a `SharedArray` by calling `sdata`. For other `AbstractArray` types, `sdata` just returns the object itself, so it's safe to use `sdata` on any `Array`-type object.

The constructor for a shared array is of the form:

```
| SharedArray{T,N}(dims::NTuple; init=false, pids=Int[])
```

which creates an  $N$ -dimensional shared array of a bits type `T` and size `dims` across the processes specified by `pids`. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `pids` named argument (and the creating process too, if it is on the same host).

If an `init` function, of signature `initfn(S::SharedArray)`, is specified, it is called on all the participating workers. You can specify that each worker runs the `init` function on a distinct portion of the array, thereby parallelizing initialization.

Here's a brief example:

```
julia> addprocs(3)
3-element Array{Int64,1}:
 2
 3
 4

julia> @everywhere using SharedArrays

julia> S = SharedArray{Int,2}((3,4), init = S ->
   → S[localindexes(S)] = myid())
3×4 SharedArray{Int64,2}:
```

```
472 2 3 4  
2 3 3 4  
2 3 4 4
```

## CHAPTER 27. GLOBAL VARIABLES

```
julia> S[3,2] = 7  
7  
  
julia> S  
3×4 SharedArray{Int64,2}:  
2 2 3 4  
2 3 3 4  
2 7 4 4
```

`SharedArrays.localindexes` provides disjoint one-dimensional ranges of indexes, and is sometimes convenient for splitting up tasks among processes. You can, of course, divide the work any way you wish:

```
julia> S = SharedArray{Int,2}((3,4), init = S ->  
→ S[indexpids(S):length(procs(S)):length(S)] = myid()  
3×4 SharedArray{Int64,2}:  
2 2 2 2  
3 3 3 3  
4 4 4 4
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
@sync begin  
    for p in procs(S)  
        @async begin  
            remotecall_wait(fill!, p, S, p)  
        end  
    end
```

```
end
```

would result in undefined behavior. Because each process fills the entire array with its own `pid`, whichever process is the last to execute (for any particular element of `S`) will have its `pid` retained.

As a more extended and complex example, consider running the following “kernel” in parallel:

```
| q[i,j,t+1] = q[i,j,t] + u[i,j,t]
```

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if `q[i, j, t]` is near the end of the block assigned to one worker and `q[i, j, t+1]` is near the beginning of the block assigned to another, it’s very likely that `q[i, j, t]` will not be ready at the time it’s needed for computing `q[i, j, t+1]`. In such cases, one is better off chunking the array manually. Let’s split along the second dimension. Define a function that returns the (`irange`, `jranging`) indexes assigned to this worker:

```
julia> @everywhere function myrange(q::SharedArray)

    idx = indexpids(q)

    if idx == 0 # This worker is not assigned a piece

        return 1:0, 1:0

    end

    nchunks = length(procs(q))
```

```
474         splits = [round(Int, s) for s in CHAPTER 27. GLOBAL VARIABLES
→   linspace(0, size(q,2), nchunks+1)]
→
→   1:size(q,1), splits[idx]+1:splits[idx+1]
→
end
```

Next, define the kernel:

```
julia> @everywhere function advection_chunk!(q, u, irange, jrange,
→   trange)
→
→       @show (irange, jrange, trange) # display so we can see
→   what's happening
→
→       for t in trange, j in jrange, i in irange
→
→           q[i,j,t+1] = q[i,j,t] + u[i,j,t]
→
→       end
→
→       q
→
end
```

We also define a convenience wrapper for a SharedArray implementation

```
julia> @everywhere advection_shared_chunk!(q, u) =
→
→       advection_chunk!(q, u, myrange(q)..., 1:size(q,3)-1)
```

Now let's compare three different versions, one that runs in a single process:

```
Julia> advection_serial!(q, u) = advection_chunk!(q, u,
    ↵ 1:size(q,1), 1:size(q,2), 1:size(q,3)-1);
```

one that uses `@parallel`:

```
julia> function advection_parallel!(q, u)
    for t = 1:size(q,3)-1
        @sync @parallel for j = 1:size(q,2)
            for i = 1:size(q,1)
                q[i,j,t+1] = q[i,j,t] + u[i,j,t]
            end
        end
    end
    q
end;
```

and one that delegates in chunks:

```
julia> function advection_shared!(q, u)
    @sync begin
        for p in procs(q)
```

476

CHAPTER 27. GLOBAL VARIABLES

```
    → p, q, u)
```

```
    end
```

```
end
```

```
q
```

```
end;
```

If we create `SharedArrays` and time these functions, we get the following results (with `julia -p 4`):

```
julia> q = SharedArray{Float64,3}((500,500,500));
```

```
julia> u = SharedArray{Float64,3}((500,500,500));
```

Run the functions once to JIT-compile and `@time` them on the second run:

```
julia> @time advection_serial!(q, u);
```

```
(irange,jrange,trange) = (1:500,1:500,1:499)
```

```
830.220 milliseconds (216 allocations: 13820 bytes)
```

```
julia> @time advection_parallel!(q, u);
```

```
2.495 seconds      (3999 k allocations: 289 MB, 2.09% gc time)
```

```
julia> @time advection_shared!(q,u);
```

```
From worker 2:      (irange,jrange,trange) =
```

```
→ (1:500,1:125,1:499)
```

```
From worker 4: (irange, jrange, trange) =  
→ (1:500, 251:375, 1:499)
```

```
From worker 3: (irange, jrange, trange) =  
→ (1:500, 126:250, 1:499)
```

```
From worker 5: (irange, jrange, trange) =  
→ (1:500, 376:500, 1:499)
```

```
238.119 milliseconds (2264 allocations: 169 KB)
```

The biggest advantage of `advection_shared!` is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

## 27.9 Shared Arrays and Distributed Garbage Collection

Like remote references, shared arrays are also dependent on garbage collection on the creating node to release references from all participating workers. Code which creates many short lived shared array objects would benefit from explicitly finalizing these objects as soon as possible. This results in both memory and file handles mapping the shared segment being released sooner.

## 27.10 ClusterManagers

The launching, management and networking of Julia processes into a logical cluster is done via cluster managers. A `ClusterManager` is responsible for

- launching worker processes in a cluster environment

- managing events during the lifetime of each worker

- optionally, providing data transport

A Julia cluster has the following characteristics:

478 The initial Julia process, also called the **master** process, has an **id** variable of 1.

Only the **master** process can add or remove worker processes.

All processes can directly communicate with each other.

Connections between workers (using the in-built TCP/IP transport) is established in the following manner:

**addprocs** is called on the master process with a **ClusterManager** object.

**addprocs** calls the appropriate **launch** method which spawns required number of worker processes on appropriate machines.

Each worker starts listening on a free port and writes out its host and port information to **STDOUT**.

The cluster manager captures the **STDOUT** of each worker and makes it available to the master process.

The master process parses this information and sets up TCP/IP connections to each worker.

Every worker is also notified of other workers in the cluster.

Each worker connects to all workers whose **id** is less than the worker's own **id**.

In this way a mesh network is established, wherein every worker is directly connected with every other worker.

While the default transport layer uses plain **TCPSocket**, it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

27.1 LocalManager<sup>479</sup> when `addprocs()` or `addprocs(np::Integer)`

are called

`SSHManager`, used when `addprocs(hostnames::Array)` is called with a list of hostnames

`LocalManager` is used to launch additional workers on the same host, thereby leveraging multi-core and multi-processor hardware.

Thus, a minimal cluster manager would need to:

be a subtype of the abstract `ClusterManager`

implement `launch`, a method responsible for launching new workers

implement `manage`, which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

`addprocs(manager::FooManager)` requires `FooManager` to implement:

```
function launch(manager::FooManager, params::Dict,
    ↳ launched::Array, c::Condition)
    [ ... ]
end

function manage(manager::FooManager, id::Integer,
    ↳ config::WorkerConfig, op::Symbol)
    [ ... ]
end
```

As an example let us see how the `LocalManager`, the manager responsible for starting workers on the same host, is implemented:

```
struct LocalManager <: ClusterManager
    np::Integer
```

```
function launch(manager::LocalManager, params::Dict,
    ↳ launched::Array, c::Condition)
    [...]
end

function manage(manager::LocalManager, id::Integer,
    ↳ config::WorkerConfig, op::Symbol)
    [...]
end
```

The `launch` method takes the following arguments:

`manager::ClusterManager`: the cluster manager that `addprocs` is called with

`params::Dict`: all the keyword arguments passed to `addprocs`

`launched::Array`: the array to append one or more `WorkerConfig` objects to

`c::Condition`: the condition variable to be notified as and when workers are launched

The `launch` method is called asynchronously in a separate task. The termination of this task signals that all requested workers have been launched. Hence the `launch` function MUST exit as soon as all the requested workers have been launched.

Newly launched workers are connected to each other and the master process in an all-to-all manner. Specifying the command line argument `--worker[=<cookie>]` results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets.

All Workers ~~and Cluster Managers~~<sup>181</sup> share the same cookie as the master. When the code is unspecified, i.e., with the --worker option, the worker tries to read it from its standard input. LocalManager and SSHManager both pass the cookie to newly launched workers via their standard inputs.

By default a worker will listen on a free port at the address returned by a call to `getipaddr()`. A specific address to listen on may be specified by optional argument `--bind-to bind_addr[:port]`. This is useful for multi-homed hosts.

As an example of a non-TCP/IP transport, an implementation may choose to use MPI, in which case --worker must NOT be specified. Instead, newly launched workers should call `init_worker(cookie)` before using any of the parallel constructs.

For every worker launched, the `launch` method must add a `WorkerConfig` object (with appropriate fields initialized) to `launched`

```
mutable struct WorkerConfig
    # Common fields relevant to all cluster managers
    io::Nullable{IO}
    host::Nullable{AbstractString}
    port::Nullable{Integer}

    # Used when launching additional workers at a host
    count::Nullable{Union{Int, Symbol}}
    exename::Nullable{AbstractString}
    exeflags::Nullable{Cmd}

    # External cluster managers can use this to store information
    # → at a per-worker level
    # Can be a dict if multiple fields need to be stored.
    userdata::Nullable{Any}
```

```

# SSHManager / SSH tunnel connections to workers
tunnel::Nullable{Bool}
bind_addr::Nullable{AbstractString}
sshflags::Nullable{Cmd}
max_parallel::Nullable{Integer}

connect_at::Nullable{Any}

[ ... ]

end

```

Most of the fields in `WorkerConfig` are used by the inbuilt managers. Custom cluster managers would typically specify only `io` or `host` / `port`:

If `io` is specified, it is used to read host/port information. A Julia worker prints out its bind address and port at startup. This allows Julia workers to listen on any free port available instead of requiring worker ports to be configured manually.

If `io` is not specified, `host` and `port` are used to connect.

`count`, `exename` and `exeflags` are relevant for launching additional workers from a worker. For example, a cluster manager may launch a single worker per node, and use that to launch additional workers.

- `count` with an integer value `n` will launch a total of `n` workers.
- `count` with a value of `:auto` will launch as many workers as the number of cores on that machine.
- `exename` is the name of the `julia` executable including the full path.
- `exeflags` should be set to the required command line arguments for new workers.

## 27.1 TUNNELS FOR MANAGERS AND CUSTOM TRANSPORT

tunnel is required to connect to the workers from the master process.

`userdata` is provided for custom cluster managers to store their own worker-specific information.

`manage(manager::FooManager, id::Integer, config::WorkerConfig, op::Symbol)` is called at different times during the worker's lifetime with appropriate `op` values:

with `:register/:deregister` when a worker is added / removed from the Julia worker pool.

with `:interrupt` when `interrupt(workers)` is called. The Cluster-Manager should signal the appropriate worker with an interrupt signal.

with `:finalize` for cleanup purposes.

## 27.11 Cluster Managers with Custom Transports

Replacing the default TCP/IP all-to-all socket connections with a custom transport layer is a little more involved. Each Julia process has as many communication tasks as the workers it is connected to. For example, consider a Julia cluster of 32 processes in an all-to-all mesh network:

Each Julia process thus has 31 communication tasks.

Each task handles all incoming messages from a single remote worker in a message-processing loop.

The message-processing loop waits on an `I0` object (for example, a `TCP-Socket` in the default implementation), reads an entire message, processes it and waits for the next one.

just communication tasks – again, via the appropriate `I0` object.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate `I0` objects that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

```
connect(manager::FooManager, pid::Integer, config::WorkerConfig)  
kill(manager::FooManager, pid::Int, config::WorkerConfig)
```

The default implementation (which uses TCP/IP sockets) is implemented as

```
connect(manager::ClusterManager, pid::Integer, config::WorkerConfig).
```

`connect` should return a pair of `I0` objects, one for reading data sent from worker `pid`, and the other to write data that needs to be sent to worker `pid`. Custom cluster managers can use an in-memory `BufferStream` as the plumbing to proxy data between the custom, possibly non-`I0` transport and Julia's in-built parallel infrastructure.

A `BufferStream` is an in-memory `I0Buffer` which behaves like an `I0` – it is a stream which can be handled asynchronously.

Folder `examples/clustermanager/0mq` contains an example of using ZeroMQ to connect Julia workers in a star topology with a 0MQ broker in the middle. Note: The Julia processes are still all logically connected to each other – any worker can message any other worker directly without any awareness of 0MQ being used as the transport layer.

When using custom transports:

Julia workers must NOT be started with `--worker`. Starting with `--worker` will result in the newly launched workers defaulting to the TCP/IP socket

For every incoming logical connection with a worker, `Base.process_messages(rd::IO, wr::IO)()` must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the `IO` objects.

`init_worker(cookie, manager::FooManager)` MUST be called as part of worker process initialization.

Field `connect_at::Any` in `WorkerConfig` can be set by the cluster manager when `launch` is called. The value of this field is passed in in all `connect` callbacks. Typically, it carries information on how to connect to a worker. For example, the TCP/IP socket transport uses this field to specify the `(host, port)` tuple at which to connect to a worker.

`kill(manager, pid, config)` is called to remove a worker from the cluster. On the master process, the corresponding `IO` objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an `exit()` call on the specified remote worker.

`examples/clustermanager/simple` is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

## 27.12 Network Requirements for LocalManager and SSHManager

Julia clusters are designed to be executed on already secured environments on infrastructure such as local laptops, departmental clusters, or even the cloud. This section covers network security requirements for the inbuilt `LocalManager` and `SSHManager`:

486 The master process does not listen on [CHAPTER 27 GLOBAL VARIABLES](#)  
the workers.

Each worker binds to only one of the local interfaces and listens on an ephemeral port number assigned by the OS.

**LocalManager**, used by `addprocs(N)`, by default binds only to the loopback interface. This means that workers started later on remote hosts (or by anyone with malicious intentions) are unable to connect to the cluster. An `addprocs(4)` followed by an `addprocs(["remote_host"])` will fail. Some users may need to create a cluster comprising their local system and a few remote systems. This can be done by explicitly requesting **LocalManager** to bind to an external network interface via the `restrict` keyword argument: `addprocs(4; restrict=false)`.

**SSHManager**, used by `addprocs(list_of_remote_hosts)`, launches workers on remote hosts via SSH. By default SSH is only used to launch Julia workers. Subsequent master-worker and worker-worker connections use plain, unencrypted TCP/IP sockets. The remote hosts must have passwordless login enabled. Additional SSH flags or credentials may be specified via keyword argument `sshflags`.

`addprocs(list_of_remote_hosts; tunnel=true, sshflags=<ssh keys and other flags>)` is useful when we wish to use SSH connections for master-worker too. A typical scenario for this is a local laptop running the Julia REPL (i.e., the master) with the rest of the cluster on the cloud, say on Amazon EC2. In this case only port 22 needs to be opened at the remote cluster coupled with SSH client authenticated via public key infrastructure (PKI). Authentication credentials can be supplied via `sshflags`, for example `sshflags=' -e <keyfile>'`.

In an all-to-all topology (the default), all workers connect to each other via plain TCP sockets. The security policy on the cluster nodes must thus

27.1 Ensuring Cluster Connectivity between workers for the ephemeral port range (varies by OS).<sup>487</sup>

Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom ClusterManager.

## 27.13 Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

`Base.cluster_cookie()` returns the cookie, while `Base.cluster_cookie(cookie)` sets it and returns the new cookie.

All connections are authenticated on both sides to ensure that only workers started by the master are allowed to connect to each other.

The cookie may be passed to the workers at startup via argument `--worker=<cookie>`. If argument `--worker` is specified without the cookie, the worker tries to read the cookie from its standard input (STDIN). The STDIN is closed immediately after the cookie is retrieved.

ClusterManagers can retrieve the cookie on the master by calling `Base.cluster_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `--worker`) must call `init_worker(cookie, manager)` with the same cookie as on the master.

Note that environments requiring higher levels of security can implement this via a custom ClusterManager. For example, cookies can be pre-shared and hence not specified as a startup argument.

## 28.14 Specifying Network Topology

CHAPTER 27 GLOBAL VARIABLES

Experimental

The keyword argument `topology` passed to `addprocs` is used to specify how the workers must be connected to each other:

`:all_to_all`, the default: all workers are connected to each other.

`:master_slave`: only the driver process, i.e. `pid 1`, has connections to the workers.

`:custom`: the `launch` method of the cluster manager specifies the connection topology via the fields `ident` and `connect_ids` in `Worker-Config`. A worker with a cluster-manager-provided identity `ident` will connect to all workers specified in `connect_ids`.

Keyword argument `lazy=true|false` only affects `topology` option `:all_to_all`. If `true`, the cluster starts off with the master connected to all workers. Specific worker-worker connections are established at the first remote invocation between two workers. This helps in reducing initial resources allocated for intra-cluster communication. Connections are setup depending on the run-time requirements of a parallel program. Default value for `lazy` is `true`.

Currently, sending a message between unconnected workers results in an error. This behaviour, as with the functionality and interface, should be considered experimental in nature and may change in future releases.

## 27.15 Multi-Threading (Experimental)

In addition to tasks, remote calls, and remote references, Julia from v0.5 forwards will natively support multi-threading. Note that this section is experimental and the interfaces may change in the future.

By default, Julia starts up with a single thread of execution. This can be verified by using the command `Threads.nthreads()`:

```
julia> Threads.nthreads()  
1
```

The number of threads Julia starts up with is controlled by an environment variable called `JULIA_NUM_THREADS`. Now, let's start up Julia with 4 threads:

```
export JULIA_NUM_THREADS=4
```

(The above command works on bourne shells on Linux and OSX. Note that if you're using a C shell on these platforms, you should use the keyword `set` instead of `export`. If you're on Windows, start up the command line in the location of `julia.exe` and use `set` instead of `export`.)

Let's verify there are 4 threads at our disposal.

```
julia> Threads.nthreads()  
4
```

But we are currently on the master thread. To check, we use the function `Threads.threadid`

```
julia> Threads.threadid()  
1
```

## The `@threads` Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
julia> a = zeros(10)  
10-element Array{Float64,1}:
```

```
490  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0  
0.0
```

## CHAPTER 27. GLOBAL VARIABLES

Let us operate on this array simultaneously using 4 threads. We'll have each thread write its thread ID into each location.

Julia supports parallel loops using the `Threads.@threads` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
julia> Threads.@threads for i = 1:10  
           a[i] = Threads.threadid()  
       end
```

The iteration space is split amongst the threads, after which each thread writes its thread ID to its assigned locations:

```
julia> a  
10-element Array{Float64,1}:  
1.0  
1.0  
1.0  
2.0
```

2.0
3.0
3.0
4.0
4.0

Note that `Threads.@threads` does not have an optional reduction parameter like `@parallel`.

## 27.16 @threadcall (Experimental)

All I/O tasks, timers, REPL commands, etc are multiplexed onto a single OS thread via an event loop. A patched version of libuv (<http://docs.libuv.org/en/v1.x/>) provides this functionality. Yield points provide for co-operatively scheduling multiple tasks onto the same OS thread. I/O tasks and timers yield implicitly while waiting for the event to occur. Calling `yield` explicitly allows for other tasks to be scheduled.

Thus, a task executing a `ccall` effectively prevents the Julia scheduler from executing any other tasks till the call returns. This is true for all calls into external libraries. Exceptions are calls into custom C code that call back into Julia (which may then yield) or C code that calls `jl_yield()` (C equivalent of `yield`).

Note that while Julia code runs on a single thread (by default), libraries used by Julia may launch their own internal threads. For example, the BLAS library may start as many threads as there are cores on a machine.

The `@threadcall` macro addresses scenarios where we do not want a `ccall` to block the main Julia event loop. It schedules a C function for execution in a separate thread. A threadpool with a default size of 4 is used for this. The size of the threadpool is controlled via environment variable `UV_THREAD-`

`POOL_SIZE`. While waiting for a free thread, `@threadcall` yields<sup>1</sup> once a thread is available, the requesting task (on the main Julia event loop) yields to other tasks. Note that `@threadcall` does not return till the execution is complete. From a user point of view, it is therefore a blocking call like other Julia APIs.

It is very important that the called function does not call back into Julia.

`@threadcall` may be removed/changed in future versions of Julia.

---

<sup>1</sup>In this context, MPI refers to the MPI-1 standard. Beginning with MPI-2, the MPI standards committee introduced a new set of communication mechanisms, collectively referred to as Remote Memory Access (RMA). The motivation for adding RMA to the MPI standard was to facilitate one-sided communication patterns. For additional information on the latest MPI standard, see <http://mpi-forum.org/docs>.

## Chapter 28

# Date and DateTime

The `Dates` module provides two types for working with dates: `Date` and `DateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TimeType`. The motivation for distinct types is simple: some operations are much simpler, both in terms of code and mental reasoning, when the complexities of greater precision don't have to be dealt with. For example, since the `Date` type only resolves to the precision of a single date (i.e. no hours, minutes, or seconds), normal considerations for time zones, daylight savings/summer time, and leap seconds are unnecessary and avoided.

Both `Date` and `DateTime` are basically immutable `Int64` wrappers. The single `instant` field of either type is actually a `UTInstant{P}` type, which represents a continuously increasing machine timeline based on the UT second <sup>1</sup>. The `DateTime` type is not aware of time zones (naive, in Python parlance), analogous to a `LocalDateTime` in Java 8. Additional time zone functionality can be added through the [TimeZones.jl package](#), which compiles the [IANA time zone database](#). Both `Date` and `DateTime` are based on the [ISO 8601](#) standard, which follows the proleptic Gregorian calendar. One note is that the ISO 8601 standard is particular about BC/BCE dates. In general, the last day of the BC/BCE era, 1-12-31 BC/BCE, was followed by 1-1-1 AD/CE, thus

A year zero exists. The ISO standard, however, CHAPTER 28 DATE AND DATETIME zero, so 0000-12-31 is the day before 0001-01-01, and year -0001 (yes, negative one for the year) is 2 BC/BCE, year -0002 is 3 BC/BCE, etc.

## 28.1 Constructors

`Date` and `DateTime` types can be constructed by integer or `Period` types, by parsing, or through adjusters (more on those later):

```
julia> DateTime(2013)
```

```
2013-01-01T00:00:00
```

```
julia> DateTime(2013, 7)
```

```
2013-07-01T00:00:00
```

```
julia> DateTime(2013, 7, 1)
```

```
2013-07-01T00:00:00
```

```
julia> DateTime(2013, 7, 1, 12)
```

```
2013-07-01T12:00:00
```

```
julia> DateTime(2013, 7, 1, 12, 30)
```

```
2013-07-01T12:30:00
```

```
julia> DateTime(2013, 7, 1, 12, 30, 59)
```

```
2013-07-01T12:30:59
```

---

<sup>1</sup>The notion of the UT second is actually quite fundamental. There are basically two different notions of time generally accepted, one based on the physical rotation of the earth (one full rotation = 1 day), the other based on the SI second (a fixed, constant value). These are radically different! Think about it, a "UT second", as defined relative to the rotation of the earth, may have a different absolute length depending on the day! Anyway, the fact that `Date` and `DateTime` are based on UT seconds is a simplifying, yet honest assumption so that things like leap seconds and all their complexity can be avoided. This basis of time is formally called `UT` or `UT1`. Basing types on the UT second basically means that every minute has 60 seconds and every day has 24 hours and leads to more natural calculations when working with calendar dates.

```
julia> DateTime(2013,7,1,12,30,59,1)  
2013-07-01T12:30:59.001
```

```
julia> Date(2013)  
2013-01-01
```

```
julia> Date(2013,7)  
2013-07-01
```

```
julia> Date(2013,7,1)  
2013-07-01
```

```
julia> Date(Dates.Year(2013),Dates.Month(7),Dates.Day(1))  
2013-07-01
```

```
julia> Date(Dates.Month(7),Dates.Year(2013))  
2013-07-01
```

Date or DateTime parsing is accomplished by the use of format strings. Format strings work by the notion of defining delimited or fixed-width "slots" that contain a period to parse and passing the text to parse and format string to a Date or DateTime constructor, of the form Date("2015-01-01", "y-m-d") or DateTime("20150101", "yyyymmdd").

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so "y-m-d" lets the parser know that between the first and second slots in a date string like "2014-07-16", it should find the - character. The y, m, and d characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number

So "yyyymmdd" would correspond to a date string like "20140716". The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition "yyyymm" from one period character to the next.

Support for text-form month parsing is also supported through the u and U characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so u corresponds to "Jan", "Feb", "Mar", etc. And U corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions `dayname` and `monthname`, custom locales can be loaded by passing in the `locale=>Dict{String, Int}` mapping to the `MONTHTOVALUEABBR` and `MONTHTOVALUE` dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the `Date(date_string, format_string)` function is fine if only called a few times. If there are many similarly formatted date strings to parse however, it is much more efficient to first create a `Dates.DateFormat`, and pass it instead of a raw format string.

```
julia> df = DateFormat("y-m-d");
julia> dt = Date("2015-01-01",df)
2015-01-01

julia> dt2 = Date("2015-01-02",df)
2015-01-02
```

You can also use the `dateformat""` string macro. This macro creates the `DateFormat` object once when the macro is expanded and uses the same `DateFormat` object even if a code snippet is run multiple times.

```
julia> for i = 1:10^5
```

```
end
```

A full suite of parsing and formatting tests and examples is available in `tests/dates/io.jl`.

## 28.2 Durations/Comparisons

Finding the length of time between two `Date` or `DateTime` is straightforward given their underlying representation as `UTInstant{Day}` and `UTInstant{Millisecond}`, respectively. The difference between `Date` is returned in the number of `Day`, and `DateTime` in the number of `Millisecond`. Similarly, comparing `TimeType` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `Int64` values).

```
julia> dt = Date(2012,2,29)
2012-02-29

julia> dt2 = Date(2000,2,1)
2000-02-01

julia> dump(dt)
Date
instant: Dates.UTInstant{Dates.Day}
periods: Dates.Day

value: Int64 734562

julia> dump(dt2)
Date
instant: Dates.UTInstant{Dates.Day}
```

```
      value: Int64 730151
julia> dt > dt2
true

julia> dt != dt2
true

julia> dt + dt2
ERROR: MethodError: no method matching +(::Date, ::Date)
[...]

julia> dt * dt2
ERROR: MethodError: no method matching *(::Date, ::Date)
[...]

julia> dt / dt2
ERROR: MethodError: no method matching /(::Date, ::Date)
[...]

julia> dt - dt2
4411 days

julia> dt2 - dt
-4411 days

julia> dt = DateTime(2012,2,29)
2012-02-29T00:00:00
```

```
2000-02-01T00:00:00
```

```
julia> dt - dt2  
381110400000 milliseconds
```

## 28.3 Accessor Functions

Because the `Date` and `DateTime` types are stored as single `Int64` values, date parts or fields can be retrieved through accessor functions. The lower-case accessors return the field as an integer:

```
julia> t = Date(2014, 1, 31)  
2014-01-31
```

```
julia> Dates.year(t)  
2014
```

```
julia> Dates.month(t)  
1
```

```
julia> Dates.week(t)  
5
```

```
julia> Dates.day(t)  
31
```

While propercase return the same value in the corresponding `Period` type:

```
julia> Dates.Year(t)  
2014 years
```

```
|500  
julia> Dates.Day(t)  
31 days
```

## CHAPTER 28. DATE AND DATETIME

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

```
julia> Dates.yearmonth(t)  
(2014, 1)  
  
julia> Dates.monthday(t)  
(1, 31)  
  
julia> Dates.yeарmonthday(t)  
(2014, 1, 31)
```

One may also access the underlying `UTInstant` or integer value:

```
julia> dump(t)  
Date  
  instant: Dates.UTInstant{Dates.Day}  
  periods: Dates.Day  
  
  value: Int64 735264  
julia> t.instant  
Dates.UTInstant{Dates.Day}(735264 days)  
  
julia> Dates.value(t)  
735264
```

Query functions provide calendrical information about a [TimeType](#). They include information about the day of the week:

```
julia> t = Date(2014, 1, 31)
2014-01-31

julia> Dates.dayofweek(t)
5

julia> Dates.dayname(t)
"Friday"

julia> Dates.dayofweekofmonth(t) # 5th Friday of January
5
```

Month of the year:

```
julia> Dates.monthname(t)
"January"

julia> Dates.daysinmonth(t)
31
```

As well as information about the [TimeType](#)'s year and quarter:

```
julia> Dates.isleapyear(t)
false

julia> Dates.dayofyear(t)
31
```

```
502 julia> Dates.quarterofyear(t)
```

```
1
```

```
julia> Dates.dayofquarter(t)
```

```
31
```

## CHAPTER 28. DATE AND DATETIME

The `dayname` and `monthname` methods can also take an optional `locale` keyword that can be used to return the name of the day or month of the year for other languages/locales. There are also versions of these functions returning the abbreviated names, namely `dayabbr` and `monthabbr`. First the mapping is loaded into the `LOCALES` variable:

```
julia> french_months = ["janvier", "février", "mars", "avril",
→   "mai", "juin",
→
→   "juillet", "août", "septembre", "octobre",
→   "novembre", "décembre"];
```

  

```
julia> french_monts_abbrev =
→   ["janv", "févr", "mars", "avril", "mai", "juin",
→
→   "juil", "août", "sept", "oct", "nov", "déc"];
```

  

```
julia> french_days =
→   ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi", "dimanche"];
```

  

```
julia> Dates.LOCALES["french"] = Dates.DateLocale(french_months,
→   french_monts_abbrev, french_days, [""]);
```

The above mentioned functions can then be used to perform the queries:

28.5. TIMETYPE-PERIOD ARITHMETIC  
**Julia>** Dates.dayname(t;Locale="french")  
"vendredi"

503

**julia>** Dates.monthname(t;locale="french")  
"janvier"

**julia>** Dates.monthabbr(t;locale="french")  
"janv"

Since the abbreviated versions of the days are not loaded, trying to use the function `dayabbr` will error.

```
julia> Dates.dayabbr(t;locale="french")  
ERROR: BoundsError: attempt to access 1-element Array{String,1} at  
    → index [5]  
Stacktrace:  
[...]
```

## 28.5 TimeType-Period Arithmetic

It's good practice when using any language/date framework to be familiar with how date-period arithmetic is handled as there are some [tricky issues](#) to deal with (though much less so for day-precision types).

The `Dates` module approach tries to follow the simple principle of trying to change as little as possible when doing `Period` arithmetic. This approach is also often known as calendrical arithmetic or what you would probably guess if someone were to ask you the same calculation in a conversation. Why all the fuss about this? Let's take a classic example: add 1 month to January 31st, 2014. What's the answer? Javascript will say [March 3](#) (assumes 31 days). PHP says [March 2](#) (assumes 30 days). The fact is, there is no right answer. In the

**Dates** module, it gives the result of February 28. How does this work out? I like to think of the classic 7-7-7 gambling game in casinos.

Now just imagine that instead of 7-7-7, the slots are Year-Month-Day, or in our example, 2014-01-31. When you ask to add 1 month to this date, the month slot is incremented, so now we have 2014-02-31. Then the day number is checked if it is greater than the last valid day of the new month; if it is (as in the case above), the day number is adjusted down to the last valid day (28). What are the ramifications with this approach? Go ahead and add another month to our date, `2014-02-28 + Month(1) == 2014-03-28`. What? Were you expecting the last day of March? Nope, sorry, remember the 7-7-7 slots. As few slots as possible are going to change, so we first increment the month slot by 1, 2014-03-28, and boom, we're done because that's a valid date. On the other hand, if we were to add 2 months to our original date, 2014-01-31, then we end up with 2014-03-31, as expected. The other ramification of this approach is a loss in associativity when a specific ordering is forced (i.e. adding things in different orders results in different outcomes). For example:

```
julia> (Date(2014,1,29)+Dates.Day(1)) + Dates.Month(1)  
2014-02-28
```

```
julia> (Date(2014,1,29)+Dates.Month(1)) + Dates.Day(1)  
2014-03-01
```

What's going on there? In the first line, we're adding 1 day to January 29th, which results in 2014-01-30; then we add 1 month, so we get 2014-02-30, which then adjusts down to 2014-02-28. In the second example, we add 1 month first, where we get 2014-02-29, which adjusts down to 2014-02-28, and then add 1 day, which results in 2014-03-01. One design principle that helps in this case is that, in the presence of multiple Periods, the operations will be ordered by the Periods' types, not their value or positional order; this

285ansTWELVE-YEAR-PERIOD-DATES-METHOD first, then Month, then Week, etc. Hence the following does result in associativity and Just Works:

```
julia> Date(2014,1,29) + Dates.Day(1) + Dates.Month(1)
```

```
2014-03-01
```

```
julia> Date(2014,1,29) + Dates.Month(1) + Dates.Day(1)
```

```
2014-03-01
```

Tricky? Perhaps. What is an innocent **Dates** user to do? The bottom line is to be aware that explicitly forcing a certain associativity, when dealing with months, may lead to some unexpected results, but otherwise, everything should work as expected. Thankfully, that's pretty much the extent of the odd cases in date-period arithmetic when dealing with time in UT (avoiding the "joys" of dealing with daylight savings, leap seconds, etc.).

As a bonus, all period arithmetic objects work directly with ranges:

```
julia> dr = Date(2014,1,29):Date(2014,2,3)
```

```
2014-01-29:1 day:2014-02-03
```

```
julia> collect(dr)
```

```
6-element Array{Date,1}:
```

```
2014-01-29
```

```
2014-01-30
```

```
2014-01-31
```

```
2014-02-01
```

```
2014-02-02
```

```
2014-02-03
```

```
julia> dr = Date(2014,1,29):Dates.Month(1):Date(2014,07,29)
```

```
2014-01-29:1 month:2014-07-29
```

506  
**julia>** collect(dr)

CHAPTER 28. DATE AND DATETIME

```
7-element Array{Date,1}:
2014-01-29
2014-02-28
2014-03-29
2014-04-29
2014-05-29
2014-06-29
2014-07-29
```

## 28.6 Adjuster Functions

As convenient as date-period arithmetics are, often the kinds of calculations needed on dates take on a calendrical or temporal nature rather than a fixed number of periods. Holidays are a perfect example; most follow rules such as "Memorial Day = Last Monday of May", or "Thanksgiving = 4th Thursday of November". These kinds of temporal expressions deal with rules relative to the calendar, like first or last of the month, next Tuesday, or the first and third Wednesdays, etc.

The **Dates** module provides the adjuster API through several convenient methods that aid in simply and succinctly expressing temporal rules. The first group of adjuster methods deal with the first and last of weeks, months, quarters, and years. They each take a single **TimeType** as input and return or adjust to the first or last of the desired period relative to the input.

```
julia> Dates.firstdayofweek(Date(2014,7,16)) # Adjusts the input
→ to the Monday of the input's week
2014-07-14
```

```
julia> Dates.lastdayofmonth(Date(2014,7,16)) # Adjusts to the last
→ day of the input's month
```

```
julia> Dates.lastdayofquarter(Date(2014,7,16)) # Adjusts to the
→ last day of the input's quarter
2014-09-30
```

The next two higher-order methods, `tonext`, and `toprev`, generalize working with temporal expressions by taking a `DateFunction` as first argument, along with a starting `TimeType`. A `DateFunction` is just a function, usually anonymous, that takes a single `TimeType` as input and returns a `Bool`, `true` indicating a satisfied adjustment criterion. For example:

```
julia> istuesday = x->Dates.dayofweek(x) == Dates.Tuesday; #
→ Returns true if the day of the week of x is Tuesday

julia> Dates.tonext(istuesday, Date(2014,7,13)) # 2014-07-13 is a
→ Sunday
2014-07-15

julia> Dates.tonext(Date(2014,7,13), Dates.Tuesday) # Convenience
→ method provided for day of the week adjustments
2014-07-15
```

This is useful with the `do`-block syntax for more complex temporal expressions:

```
julia> Dates.tonext(Date(2014,7,13)) do x
          # Return true on the 4th Thursday of November
→ (Thanksgiving)
          Dates.dayofweek(x) == Dates.Thursday &&
```

```
Dates.month(x) == Dates.November  
  
end  
2014-11-27
```

The `Base.filter` method can be used to obtain all valid dates/moments in a specified range:

```
# Pittsburgh street cleaning; Every 2nd Tuesday from April to  
→ November  
# Date range from January 1st, 2014 to January 1st, 2015  
julia> dr = Dates.Date(2014):Dates.Date(2015);  
  
julia> filter(dr) do x  
  
    Dates.dayofweek(x) == Dates.Tue &&  
  
    Dates.April <= Dates.month(x) <= Dates.Nov &&  
  
    Dates.dayofweekofmonth(x) == 2  
  
end  
8-element Array{Date,1}:  
2014-04-08  
2014-05-13  
2014-06-10  
2014-07-08  
2014-08-12  
2014-09-09
```

```
2014-11-11
```

Additional examples and tests are available in [test/dates/adjusters.jl](#).

## 28.7 Period Types

Periods are a human view of discrete, sometimes irregular durations of time. Consider 1 month; it could represent, in days, a value of 28, 29, 30, or 31 depending on the year and month context. Or a year could represent 365 or 366 days in the case of a leap year. `Period` types are simple `Int64` wrappers and are constructed by wrapping any `Int64` convertible type, i.e. `Year(1)` or `Month(3.0)`. Arithmetic between `Period` of the same type behave like integers, and limited `Period`-`Real` arithmetic is available.

```
julia> y1 = Dates.Year(1)
```

```
1 year
```

```
julia> y2 = Dates.Year(2)
```

```
2 years
```

```
julia> y3 = Dates.Year(10)
```

```
10 years
```

```
julia> y1 + y2
```

```
3 years
```

```
julia> div(y3,y2)
```

```
5
```

```
julia> y3 - y2
```

```
8 years
```

```
julia> y3 % y2  
0 years  
  
julia> div(y3,3) # mirrors integer division  
3 years
```

## 28.8 Rounding

`Date` and `DateTime` values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with `floor`, `ceil`, or `round`:

```
julia> floor(Date(1985, 8, 16), Dates.Month)  
1985-08-01  
  
julia> ceil(DateTime(2013, 2, 13, 0, 31, 20), Dates.Minute(15))  
2013-02-13T00:45:00  
  
julia> round(DateTime(2016, 8, 6, 20, 15), Dates.Day)  
2016-08-07T00:00:00
```

Unlike the numeric `round` method, which breaks ties toward the even number by default, the `TimeType``round` method uses the `RoundNearestTiesUp` rounding mode. (It's difficult to guess what breaking ties to nearest "even" `TimeType` would entail.) Further details on the available `RoundingMode`s can be found in the [API reference](#).

Rounding should generally behave as expected, but there are a few cases in which the expected behaviour is not obvious.

In many cases, the resolution specified for rounding (e.g., `Dates.Second(30)`) divides evenly into the next largest period (in this case, `Dates.Minute(1)`). But rounding behaviour in cases in which this is not true may lead to confusion. What is the expected result of rounding a `DateTime` to the nearest 10 hours?

```
julia> round(DateTime(2016, 7, 17, 11, 55), Dates.Hour(10))  
2016-07-17T12:00:00
```

That may seem confusing, given that the hour (12) is not divisible by 10. The reason that `2016-07-17T12:00:00` was chosen is that it is 17,676,660 hours after `0000-01-01T00:00:00`, and 17,676,660 is divisible by 10.

As Julia `Date` and `DateTime` values are represented according to the ISO 8601 standard, `0000-01-01T00:00:00` was chosen as base (or "rounding epoch") from which to begin the count of days (and milliseconds) used in rounding calculations. (Note that this differs slightly from Julia's internal representation of `Date`s using Rata Die notation; but since the ISO 8601 standard is most visible to the end user, `0000-01-01T00:00:00` was chosen as the rounding epoch instead of the `0000-12-31T00:00:00` used internally to minimize confusion.)

The only exception to the use of `0000-01-01T00:00:00` as the rounding epoch is when rounding to weeks. Rounding to the nearest week will always return a Monday (the first day of the week as specified by ISO 8601). For this reason, we use `0000-01-03T00:00:00` (the first day of the first week of year 0000, as defined by ISO 8601) as the base when rounding to a number of weeks.

Here is a related case in which the expected behaviour is not necessarily obvious: What happens when we round to the nearest  $P(2)$ , where  $P$  is a

**Period** type? In so  
the answer is clear:

es (specifically CHAPTER 28: DATES AND PERIODS)

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Hour(2))
2016-07-17T08:00:00
```

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Minute(2))
2016-07-17T08:56:00
```

This seems obvious, because two of each of these periods still divides evenly into the next larger order period. But in the case of two months (which still divides evenly into one year), the answer may be surprising:

```
julia> round(DateTime(2016, 7, 17, 8, 55, 30), Dates.Month(2))
2016-07-01T00:00:00
```

Why round to the first day in July, even though it is month 7 (an odd number)? The key is that months are 1-indexed (the first month is assigned 1), unlike hours, minutes, seconds, and milliseconds (the first of which are assigned 0).

This means that rounding a `DateTime` to an even multiple of seconds, minutes, hours, or years (because the ISO 8601 specification includes a year zero) will result in a `DateTime` with an even value in that field, while rounding a `DateTime` to an even multiple of months will result in the months field having an odd value. Because both months and years may contain an irregular number of days, whether rounding to an even number of days will result in an even value in the days field is uncertain.

See the [API reference](#) for additional information on methods exported from the `Dates` module.

## Chapter 29

# Interacting With Julia

Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the `julia` executable. In addition to allowing quick and easy evaluation of Julia statements, it has a searchable history, tab-completion, many helpful keybindings, and dedicated help and shell modes. The REPL can be started by simply calling `julia` with no arguments or double-clicking on the executable:

```
$ julia
      -
      - _(_)_ | A fresh approach to technical computing
      (_)_ | (_)_ | Documentation: https://docs.julialang.org
      org
      - - -| |- -- -| Type "?help" for help.
      | | | | | | / _` | |
      | | | -| | | | (-| | | Version 0.6.0-dev.2493 (2017-01-31 18:53
      UTC)
      -/ | \--'_-|-|-|\--'_-| Commit c99e12c* (0 days old master)
      |__/_ | x86_64-linux-gnu

julia>
```

To exit the interactive session, type `Ctrl-D` together with the `key` on a blank line – or type `quit()` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

## 29.1 The different prompt modes

### The Julian mode

The REPL has four main modes of operation. The first and most common is the Julian prompt. It is the default mode of operation; each new line initially starts with `julia>`. It is here that you can enter Julia expressions. Hitting return or enter after a complete expression has been entered will evaluate the entry and show the result of the last expression.

```
julia> string(1 + 2)  
"3"
```

There are a number useful features unique to interactive work. In addition to showing the result, the REPL also binds the result to the variable `ans`. A trailing semicolon on the line can be used as a flag to suppress showing the result.

```
julia> string(3 * 4);  
  
julia> ans  
"12"
```

In Julia mode, the REPL supports something called prompt pasting. This activates when pasting text that starts with `julia>` into the REPL. In that case, only expressions starting with `julia>` are parsed, others are removed. This makes it is possible to paste a chunk of code that has been copied from a REPL session without having to scrub away prompts and outputs. This feature is enabled by default but can be disabled or enabled at will with `Base.REPL.enable_promptpaste(::Bool)`. If it is enabled, you can try it out by pasting

~~291 code~~ ~~THE DIFFERENT PROMPT MODES~~ straight into the REPL. This feature ~~does~~ <sup>15</sup> not work on the standard Windows command prompt due to its limitation at detecting when a paste occurs.

## Help mode

When the cursor is at the beginning of the line, the prompt can be changed to a help mode by typing ?. Julia will attempt to print help or documentation for anything entered in help mode:

```
julia> ? # upon typing ?, the prompt changes (in place) to: help?>

help?> string
search: string String stringmime Cstring Cwstring RevString
  ↳ randstring bytestring SubString

string(xs...)

Create a string from any values using the print function.
```

Macros, types and variables can also be queried:

```
help?> @time
@time

A macro to execute an expression, printing the time it took to
execute, the number of allocations,
and the total number of bytes its execution caused to be
allocated, before returning the value of the
expression.

See also @timed, @elapsed, and @allocated.
```

```
| 516 Help?> AbstractString
```

## CHAPTER 29. INTERACTING WITH JULIA

```
search: AbstractString AbstractSparseMatrix AbstractSparseVector  
AbstractSet
```

No documentation found.

### Summary

```
abstract type AbstractString <: Any
```

### Subtypes

```
Base.SubstitutionString
```

```
String
```

```
SubString
```

```
Test.GenericString
```

Help mode can be exited by pressing backspace at the beginning of the line.

### Shell mode

Just as help mode is useful for quick access to documentation, another common task is to use the system shell to execute system commands. Just as ? entered help mode when at the beginning of the line, a semicolon (;) will enter the shell mode. And it can be exited by pressing backspace at the beginning of the line.

```
julia> ; # upon typing ;, the prompt changes (in place) to: shell>
```

```
shell> echo hello  
hello
```

In all of the above modes, the executed lines get saved to a history file, which can be searched. To initiate an incremental search through the previous history, type `^R` – the control key together with the `r` key. The prompt will change to `(reverse-i-search) ``:`, and as you type the search query will appear in the quotes. The most recent result that matches the query will dynamically update to the right of the colon as more is typed. To find an older result using the same query, simply type `^R` again.

Just as `^R` is a reverse search, `^S` is a forward search, with the prompt `(i-search) ``:`. The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

## 29.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (`^D` to exit, `^R` and `^S` for searching), but there are many more. In addition to the control-key, there are also meta-key bindings. These vary more by platform, but most terminals default to using alt- or option- held down with a key to send the meta-key (or can be configured to do so).

### Customizing keybindings

Julia's REPL keybindings may be fully customized to a user's preferences by passing a dictionary to `REPL.setup_interface()`. The keys of this dictionary may be characters or strings. The key `'*'` refers to the default action. Control plus character `x` bindings are indicated with `"^x"`. Meta plus `x` can be written `"\\Mx"`. The values of the custom keymap must be `nothing` (indicating that the input should be ignored) or functions that accept the signature `(PromptState, AbstractREPL, Char)`. The `REPL.setup_interface()`

function must be called before the REPL is initialized by registering the application with `atreplinit()`. For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in `.juliarc.jl`:

```
import Base: LineEdit, REPL

const mykeys = Dict{Any,Any}(
    # Up Arrow
    "\e[A" => (s,o...)->(LineEdit.edit_move_up(s) ||
                           LineEdit.history_prev(s, LineEdit.mode(s).hist)),
    # Down Arrow
    "\e[B" => (s,o...)->(LineEdit.edit_move_up(s) ||
                           LineEdit.history_next(s, LineEdit.mode(s).hist))
)

function customize_keys(repl)
    repl.interface = REPL.setup_interface(repl; extra_repl_keymap
                                           = mykeys)
end

atreplinit(customize_keys)
```

Users should refer to `LineEdit.jl` to discover the available actions on key input.

### 29.3 Tab completion

In both the Julian and help modes of the REPL, one can enter the first few characters of a function or type and then press the tab key to get a list all matches:

```
stride      strides      string      stringmime  strip

julia> Stri[TAB]
StridedArray   StridedMatrix   StridedVecOrMat  StridedVector
→ String
```

The tab key can also be used to substitute LaTeX math symbols with their Unicode equivalents, and get a list of LaTeX matches as well:

```
julia> \pi[TAB]
julia> π
π = 3.1415926535897...

julia> e\_-1[TAB] = [1,0]
julia> e = [1,0]
2-element Array{Int64,1}:
 1
 0

julia> e\^1[TAB] = [1 0]
julia> e¹ = [1 0]
1×2 Array{Int64,2}:
 1  0

julia> \sqrt[TAB]2      # √ is equivalent to the sqrt() function
julia> √2
1.4142135623730951

julia> \hbar[TAB](h) = h / 2\pi[TAB]
julia> ħ(h) = h / 2π
ħ (generic function with 1 method)
```

```
julia> \h[TAB]
\hat          \hermitconjmatrix  \hkswarow
↪ \hrectangle
\hatapprox   \hexagon        \hookleftarrow
↪ \hrectangleblack
\hbar         \hexagonblack   \hookrightarrow    \hslash
\heartsuit    \hksearrow      \house         \hspace

julia> a = "\alpha[TAB]"    # LaTeX completion also works in strings
julia> a = "a"
```

A full list of tab-completions can be found in the [Unicode Input](#) section of the manual.

Completion of paths works for strings and julia's shell mode:

```
julia> path="/[TAB]"
.dockerenv .juliabox/ boot/ etc/ lib/
↪ media/     opt/     root/    sbin/    sys/
↪ usr/
.dockerinit bin/ dev/ home/ lib64/
↪ mnt/       proc/    run/     srv/    tmp/
↪ var/
shell> /[TAB]
.dockerenv .juliabox/ boot/ etc/ lib/
↪ media/     opt/     root/    sbin/    sys/
↪ usr/
.dockerinit bin/ dev/ home/ lib64/
↪ mnt/       proc/    run/     srv/    tmp/
↪ var/
```

`julia> TAB` completion with investigation of the available methods matching the input arguments:

```
julia> max([TAB] # All methods are displayed, not shown here due to
   ↳ size of the list

julia> max([1, 2], [TAB] # All methods where `Vector{Int}` matches
   ↳ as first argument
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281

julia> max([1, 2], max(1, 2), [TAB] # All methods matching the
   ↳ arguments.
max(x, y) in Base at operators.jl:215
max(a, b, c, xs...) in Base at operators.jl:281
```

Keywords are also displayed in the suggested methods, see second line after ; where `limit` and `keep` are keyword arguments:

```
julia> split("1 1 1", [TAB]
split(str::AbstractString) in Base at strings/util.jl:302
split(str::T, splitter; limit, keep) where T<:AbstractString in
   ↳ Base at strings/util.jl:277
```

The completion of the methods uses type inference and can therefore see if the arguments match even if the arguments are output from functions. The function needs to be type stable for the completion to be able to remove non-matching methods.

Tab completion can also help completing fields:

```
julia> Pkg.a[TAB]
add      available
```

```
julia> split("", "")[1].[TAB]  
endof offset string
```

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

## 29.4 Customizing Colors

The colors used by Julia and the REPL can be customized, as well. To change the color of the Julia prompt you can add something like the following to your `.juliarc.jl` file, which is to be placed inside your home directory:

```
function customize_colors(repl)  
    repl.prompt_color = Base.text_colors[:cyan]  
end  
  
atreplinit(customize_colors)
```

The available color keys can be seen by typing `Base.text_colors` in the help mode of the REPL. In addition, the integers 0 to 255 can be used as color keys for terminals with 256 color support.

You can also change the colors for the help and shell prompts and input and answer text by setting the appropriate field of `repl` in the `customize_colors` function above (respectively, `help_color`, `shell_color`, `input_color`, and `answer_color`). For the latter two, be sure that the `envcolors` field is also set to false.

It is also possible to apply boldface formatting by using `Base.text_colors[:bold]` as a color. For instance, to print answers in boldface font, one can use the following as a `.juliarc.jl`:

```
function customize_colors(repl)
    repl.envcolors = false
    repl.answer_color = Base.text_colors[:bold]
end

atreplinit(customize_colors)
```

You can also customize the color used to render warning and informational messages by setting the appropriate environment variables. For instance, to render error, warning, and informational messages respectively in magenta, yellow, and cyan you can add the following to your `.juliarc.jl` file:

```
ENV["JULIA_ERROR_COLOR"] = :magenta
ENV["JULIA_WARN_COLOR"] = :yellow
ENV["JULIA_INFO_COLOR"] = :cyan
```

## 524 Keybinding CHAPTER 29. INTERACTING WITH JULIA

Keybinding	Description
Program control	
<b>^D</b>	Exit (when buffer is empty)
<b>^C</b>	Interrupt or cancel
<b>^L</b>	Clear console screen
Return/Enter, <b>^J</b>	New line, executing if it is complete
meta-Return/Enter	Insert new line without executing it
? or ;	Enter help or shell mode (when at start of a line)
<b>^R, ^S</b>	Incremental history search, described above
Cursor movement	
Right arrow, <b>^F</b>	Move right one character
Left arrow, <b>^B</b>	Move left one character
ctrl-Right, <b>meta-F</b>	Move right one word
ctrl-Left, <b>meta-B</b>	Move left one word
Home, <b>^A</b>	Move to beginning of line
End, <b>^E</b>	Move to end of line
Up arrow, <b>^P</b>	Move up one line (or change to the previous history entry that matches the text before the cursor)
Down arrow, <b>^N</b>	Move down one line (or change to the next history entry that matches the text before the cursor)
Page-up, <b>meta-P</b>	Change to the previous history entry
Page-down, <b>meta-N</b>	Change to the next history entry
<b>meta-&lt;</b>	Change to the first history entry (of the current session if it is before the current position in history)
<b>meta-&gt;</b>	Change to the last history entry
<b>^-Space</b>	Set the "mark" in the editing region
<b>^X^X</b>	Exchange the current position with the mark
Editing	
Backspace, <b>^H</b>	Delete the previous character
Delete, <b>^D</b>	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
<b>meta-d</b>	Forward delete the next word
<b>^W</b>	Delete previous text up to the nearest whitespace
<b>meta-w</b>	Copy the current region in the kill ring
<b>meta-W</b>	"Kill" the current region, placing the text in the kill ring
<b>^K</b>	"Kill" to end of line, placing the text in the kill ring
<b>^Y</b>	"Yank" insert the text from the kill ring
<b>meta-y</b>	Replace a previously yanked text with an older entry from the kill ring
<b>^T</b>	Transpose the characters about the cursor
<b>meta-u</b>	Change the next word to uppercase

## Chapter 30

# Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```
julia> `echo hello`  
`echo hello`
```

differs in several aspects from the behavior in various shells, Perl, or Ruby:

Instead of immediately running the command, backticks create a `Cmd` object to represent the command. You can use this object to connect the command to others via pipes, run it, and read or write to it.

When the command is run, Julia does not capture its output unless you specifically arrange for it to. Instead, the output of the command by default goes to `STDOUT` as it would using `libc's system call`.

The command is never run with a shell. Instead, Julia parses the command syntax directly, appropriately interpolating variables and splitting on words as the shell would, respecting shell quoting syntax. The command is run as `julia's` immediate child process, using `fork` and `exec` calls.

Here's a simple example of running an external program.

```
julia> mycommand = `echo hello`  
`echo hello`
```

```
julia> typeof(mycommand)  
Cmd
```

```
julia> run(mycommand)  
hello
```

The `hello` is the output of the `echo` command, sent to `STDOUT`. The `run` method itself returns `nothing`, and throws an `ErrorException` if the external command fails to run successfully.

If you want to read the output of the external command, `read` can be used instead:

```
julia> a = read(`echo hello`, String)  
"hello\n"
```

```
julia> chomp(a) == "hello"  
true
```

More generally, you can use `open` to read from or write to an external command.

```
julia> open(`less`, "w", STDOUT) do io  
    for i = 1:3  
        println(io, i)
```

```
    end  
1  
2  
3
```

The program name and the individual arguments in a command can be accessed and iterated over as if the command were an array of strings:

```
julia> collect(`echo "foo bar"`)  
2-element Array{String,1}:  
  "echo"  
  "foo bar"  
  
julia> `echo "foo bar"``[2]  
"foo bar"
```

## 30.1 Interpolation

Suppose you want to do something a bit more complicated and use the name of a file in the variable `file` as an argument to a command. You can use `$` for interpolation much as you would in a string literal (see [Strings](#)):

```
julia> file = "/etc/passwd"  
"/etc/passwd"  
  
julia> `sort $file`  
`sort /etc/passwd`
```

A common pitfall when running external programs via a shell is that if a file name contains characters that are special to the shell, they may cause undesirable behavior. Suppose, for example, rather than `/etc/passwd`, we wanted

528 sort the contents of the file `data.csv` it:

```
julia> file = "/Volumes/External HD/data.csv"
"/Volumes/External HD/data.csv"

julia> `sort $file`
`sort '/Volumes/External HD/data.csv'`
```

How did the file name get quoted? Julia knows that `file` is meant to be interpolated as a single argument, so it quotes the word for you. Actually, that is not quite accurate: the value of `file` is never interpreted by a shell, so there's no need for actual quoting; the quotes are inserted only for presentation to the user. This will even work if you interpolate a value as part of a shell word:

```
julia> path = "/Volumes/External HD"
"/Volumes/External HD"

julia> name = "data"
"data"

julia> ext = "csv"
"csv"

julia> `sort $path/$name.$ext`
`sort '/Volumes/External HD/data.csv'`
```

As you can see, the space in the `path` variable is appropriately escaped. But what if you want to interpolate multiple words? In that case, just use an array (or any other iterable container):

30.1 INTERPOLATION **julia>** files = [ "/etc/passwd", "/Volumes/External HD/data.csv" ] 529

```
2-element Array{String,1}:
"/etc/passwd"
"/Volumes/External HD/data.csv"
```

```
julia> `grep foo $files`
`grep foo /etc/passwd '/Volumes/External HD/data.csv'`
```

If you interpolate an array as part of a shell word, Julia emulates the shell's {a,b,c} argument generation:

```
julia> names = [ "foo", "bar", "baz" ]
3-element Array{String,1}:
"foo"
"bar"
"baz"

julia> `grep xylophone $names.txt`
`grep xylophone foo.txt bar.txt baz.txt`
```

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

```
julia> names = [ "foo", "bar", "baz" ]
3-element Array{String,1}:
"foo"
"bar"
"baz"

julia> exts = [ "aux", "log" ]
2-element Array{String,1}:
"aux"
```

```
julia> `rm -f $names.$exts`  
`rm -f foo.aux foo.log bar.aux bar.log baz.aux baz.log`
```

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
julia> `rm -rf ["foo", "bar", "baz", "qux"].["aux", "log", "pdf"]`  
`rm -rf foo.aux foo.log bar.aux bar.log bar.pdf baz.aux  
→ baz.log baz.pdf qux.aux qux.log qux.pdf`
```

## 30.2 Quoting

Inevitably, one wants to write commands that aren't quite so simple, and it becomes necessary to use quotes. Here's a simple example of a Perl one-liner at a shell prompt:

```
sh$ perl -le '$|=1; for (0..3) { print }'  
0  
1  
2  
3
```

The Perl expression needs to be in single quotes for two reasons: so that spaces don't break the expression into multiple shell words, and so that uses of Perl variables like `$|` (yes, that's the name of a variable in Perl), don't cause interpolation. In other instances, you may want to use double quotes so that interpolation does occur:

```
sh$ first="A"  
sh$ second="B"  
sh$ perl -le '$|=1; print for @ARGV' "1: $first" "2: $second"  
1: A
```

In general, the Julia backtick syntax is carefully designed so that you can just cut-and-paste shell commands as is into backticks and they will work: the escaping, quoting, and interpolation behaviors are the same as the shell's. The only difference is that the interpolation is integrated and aware of Julia's notion of what is a single string value, and what is a container for multiple values. Let's try the above two examples in Julia:

```
julia> A = `perl -le '$|=1; for (0..3) { print }'`  
`perl -le '$|=1; for (0..3) { print }'`  
  
julia> run(A)  
0  
1  
2  
3  
  
julia> first = "A"; second = "B";  
  
julia> B = `perl -le 'print for @ARGV' "1: $first" "2: $second"`  
`perl -le 'print for @ARGV' '1: A' '2: B'`  
  
julia> run(B)  
1: A  
2: B
```

The results are identical, and Julia's interpolation behavior mimics the shell's with some improvements due to the fact that Julia supports first-class iterable objects while most shells use strings split on spaces for this, which introduces ambiguities. When trying to port shell commands to Julia, try cut and pasting

532. Since Julia shows command-line programs as external programs,  
CHAPTER 30: RUNNING EXTERNAL PROGRAMS  
and safely just examine its interpretation without doing any damage.

### 30.3 Pipelines

Shell metacharacters, such as |, &, and >, need to be quoted (or escaped) inside of Julia's backticks:

```
julia> run(`echo hello '| sort`)
hello | sort

julia> run(`echo hello \| sort`)
hello | sort
```

This expression invokes the `echo` command with three words as arguments: `hello`, `|`, and `sort`. The result is that a single line is printed: `hello | sort`. How, then, does one construct a pipeline? Instead of using '`|`' inside of backticks, one uses `pipeline`:

```
julia> run(pipeline(`echo hello`, `sort`))
hello
```

This pipes the output of the `echo` command to the `sort` command. Of course, this isn't terribly interesting since there's only one line to sort, but we can certainly do much more interesting things:

```
julia> run(pipeline(`cut -d: -f3 /etc/passwd`, `sort -n`, `tail
→ -n5`))
210
211
212
213
214
```

~~363.PIPELINES~~ highest five user IDs on a UNIX system. The `cut`, `sort` and `tail` commands are all spawned as immediate children of the current `julia` process, with no intervening shell process. Julia itself does the work to setup pipes and connect file descriptors that is normally done by the shell. Since Julia does this itself, it retains better control and can do some things that shells cannot.

Julia can run multiple commands in parallel:

```
julia> run(`echo hello` & `echo world`)
world
hello
```

The order of the output here is non-deterministic because the two `echo` processes are started nearly simultaneously, and race to make the first write to the `STDOUT` descriptor they share with each other and the `julia` parent process. Julia lets you pipe the output from both of these processes to another program:

```
julia> run(pipeline(`echo world` & `echo hello`, `sort`))
hello
world
```

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both `echo` processes, and the other end of the pipe is read from by the `sort` command.

I/O redirection can be accomplished by passing keyword arguments `stdin`, `stdout`, and `stderr` to the `pipeline` function:

```
pipeline(`do_work`, stdout=pipeline(`sort`, "out.txt"),
        stderr="errs.txt")
```

When reading and writing to both ends of a pipeline from a single process, it is important to avoid forcing the kernel to buffer all of the data.

For example, when reading all of the output from a command, call `read(out, String)`, not `wait(process)`, since the former will actively consume all of the data written by the process, whereas the latter will attempt to store the data in the kernel's buffers while waiting for a reader to be connected.

Another common solution is to separate the reader and writer of the pipeline into separate Tasks:

```
writer = @async write(process, "data")
reader = @async do_compute(read(process, String))
wait(process)
fetch(reader)
```

## Complex Example

The combination of a high-level programming language, a first-class command abstraction, and automatic setup of pipes between processes is a powerful one. To give some sense of the complex pipelines that can be created easily, here are some more sophisticated examples, with apologies for the excessive use of Perl one-liners:

```
julia> prefixer(prefix, sleep) = `perl -nle '$|=1; print
→   "'$prefix' ", $_; sleep '$sleep';'`;
julia> run(pipeline(`perl -le '$|=1; for(0..9){ print; sleep 1
→   }``,
→   prefixer("A",2) & prefixer("B",2)))
A 0
B 1
A 2
```

```
A 4  
B 5  
A 6  
B 7  
A 8  
B 9
```

This is a classic example of a single producer feeding two concurrent consumers: one `perl` process generates lines with the numbers 0 through 9 on them, while two parallel processes consume that output, one prefixing lines with the letter "A", the other with the letter "B". Which consumer gets the first line is non-deterministic, but once that race has been won, the lines are consumed alternately by one process and then the other. (Setting `$|=1` in Perl causes each print statement to flush the `STDOUT` handle, which is necessary for this example to work. Otherwise all the output is buffered and printed to the pipe at once, to be read by just one consumer process.)

Here is an even more complex multi-stage producer-consumer example:

```
julia> run(pipeline(`perl -le '$|=1; for(0..9){ print; sleep 1
→   }' `,
                           prefixer("X",3) & prefixer("Y",3) & prefixer("Z",3),
                           prefixer("A",2) & prefixer("B",2)))
A X 0
B Y 1
A Z 2
B X 3
A Y 4
B Z 5
```

536  
A X 6

B Y 7

A Z 8

B X 9

## CHAPTER 30. RUNNING EXTERNAL PROGRAMS

This example is similar to the previous one, except there are two stages of consumers, and the stages have different latency so they use a different number of parallel workers, to maintain saturated throughput.

We strongly encourage you to try all these examples to see how they work.

# Chapter 31

## Calling C and Fortran Code

Though most code can be written in Julia, there are many high-quality, mature libraries for numerical computing already written in C and Fortran. To allow easy use of this existing code, Julia makes it simple and efficient to call C and Fortran functions. Julia has a “no boilerplate” philosophy: functions can be called directly from Julia without any “glue” code, code generation, or compilation – even from the interactive prompt. This is accomplished just by making an appropriate call with `ccall` syntax, which looks like an ordinary function call.

The code to be called must be available as a shared library. Most C and Fortran libraries ship compiled as shared libraries already, but if you are compiling the code yourself using GCC (or Clang), you will need to use the `-shared` and `-fPIC` options. The machine instructions generated by Julia’s JIT are the same as a native C call would be, so the resulting overhead is the same as calling a library function from C code. (Non-library function calls in both C and Julia can be inlined and thus may have even less overhead than calls to shared library functions. When both libraries and executables are generated by LLVM, it is possible to perform whole-program optimizations that can even optimize across this boundary, but Julia does not yet support that. In the future, however, it may do so, yielding even greater performance gains.)

tion, "library") or ("function", "library") where function is the C-exported function name. library refers to the shared library name: shared libraries available in the (platform-specific) load path will be resolved by name, and if necessary a direct path may be specified.

A function name may be used alone in place of the tuple (just :function or "function"). In this case the name is resolved within the current process. This form can be used to call C library functions, functions in the Julia runtime, or functions in an application linked to Julia.

By default, Fortran compilers generate mangled names (for example, converting function names to lowercase or uppercase, often appending an underscore), and so to call a Fortran function via `ccall` you must pass the mangled identifier corresponding to the rule followed by your Fortran compiler. Also, when calling a Fortran function, all inputs must be passed by reference.

Finally, you can use `ccall` to actually generate a call to the library function. Arguments to `ccall` are as follows:

1. A (:function, "library") pair, which must be written as a literal constant,

OR

a function pointer (for example, from `dlsym`).
2. Return type (see below for mapping the declared C type to Julia)
  - This argument will be evaluated at compile-time, when the containing method is defined.
3. A tuple of input types. The input types must be written as a literal tuple, not a tuple-valued variable or expression.

- This argument will be evaluated at compile-time, when the containing method is defined.

4. The following arguments, if any, are the actual argument values passed to the function.

As a complete but simple example, the following calls the `clock` function from the standard C library:

```
julia> t = ccall(:clock, "libc", Int32, ())
2292761

julia> t
2292761

julia> typeof(ans)
Int32
```

`clock` takes no arguments and returns an `Int32`. One common gotcha is that a 1-tuple must be written with a trailing comma. For example, to call the `getenv` function to get a pointer to the value of an environment variable, one makes a call like this:

```
julia> path = ccall(:getenv, "libc", Cstring, (Cstring,),,
                     "SHELL")
Cstring(@0x00007fff5fbfffc45)

julia> unsafe_string(path)
"/bin/bash"
```

Note that the argument type tuple must be written as `(Cstring, )`, rather than `(Cstring)`. This is because `(Cstring)` is just the expression `Cstring` surrounded by parentheses, rather than a 1-tuple containing `Cstring`:

540  
Julia> (Cstring)

## CHAPTER 31. CALLING C AND FORTRAN CODE

Cstring

```
julia> (Cstring,)  
(Cstring, )
```

In practice, especially when providing reusable functionality, one generally wraps `ccall` uses in Julia functions that set up arguments and then check for errors in whatever manner the C or Fortran function indicates them, propagating to the Julia caller as exceptions. This is especially important since C and Fortran APIs are notoriously inconsistent about how they indicate error conditions. For example, the `getenv` C library function is wrapped in the following Julia function, which is a simplified version of the actual definition from `env.jl`:

```
function getenv(var::AbstractString)  
    val = ccall((:getenv, "libc"),  
                Cstring, (Cstring,), var)  
    if val == C_NULL  
        error("getenv: undefined variable: ", var)  
    end  
    unsafe_string(val)  
end
```

The C `getenv` function indicates an error by returning `NULL`, but other standard C functions indicate errors in various different ways, including by returning `-1`, `0`, `1` and other special values. This wrapper throws an exception clearly indicating the problem if the caller tries to get a non-existent environment variable:

```
julia> getenv("SHELL")  
"/bin/bash"
```

```
julia> getenv("FOOBAR")
getenv: undefined variable: FOOBAR
```

Here is a slightly more complex example that discovers the local machine's hostname:

```
function gethostname()
    hostname = Vector{UInt8}(128)
    ccall((:gethostname, "libc"), Int32,
          (Ptr{UInt8}, Csize_t),
          hostname, sizeof(hostname))
    hostname[end] = 0; # ensure null-termination
    return unsafe_string(pointer(hostname))
end
```

This example first allocates an array of bytes, then calls the C library function `gethostname` to fill the array in with the hostname, takes a pointer to the hostname buffer, and converts the pointer to a Julia string, assuming that it is a NUL-terminated C string. It is common for C libraries to use this pattern of requiring the caller to allocate memory to be passed to the callee and filled in. Allocation of memory from Julia like this is generally accomplished by creating an uninitialized array and passing a pointer to its data to the C function. This is why we don't use the `Cstring` type here: as the array is uninitialized, it could contain NUL bytes. Converting to a `Cstring` as part of the `ccall` checks for contained NUL bytes and could therefore throw a conversion error.

## 31.1 Creating C-Compatible Julia Function Pointers

It is possible to pass Julia functions to native C functions that accept function pointer arguments. For example, to match C prototypes of the form:

| 542 CHAPTER 31. CALLING C AND FORTRAN CODE

The function `cfunction` generates the C-compatible function pointer for a call to a Julia function. Arguments to `cfunction` are as follows:

1. A Julia Function
2. Return type
3. A tuple type of input types

Only platform-default C calling convention is supported. `cfunction`-generated pointers cannot be used in calls where WINAPI expects `stdcall` function on 32-bit windows, but can be used on WIN64 (where `stdcall` is unified with C calling convention).

A classic example is the standard C library `qsort` function, declared as:

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compare)(const void *a, const void *b));
```

The `base` argument is a pointer to an array of length `nmemb`, with elements of `size` bytes each. `compare` is a callback function which takes pointers to two elements `a` and `b` and returns an integer less/greater than zero if `a` should appear before/after `b` (or zero if any order is permitted). Now, suppose that we have a 1d array `A` of values in Julia that we want to sort using the `qsort` function (rather than Julia's built-in `sort` function). Before we worry about calling `qsort` and passing arguments, we need to write a comparison function that works for some arbitrary type `T`:

```
julia> function mycompare(a::T, b::T) where T
           return convert(Cint, a < b ? -1 : a > b ? +1 : 0)::Cint
```

```
| mycompare (generic function with 1 method)
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a C `int`, so we must be sure to return `Cint` via a call to `convert` and a `typeassert`.

In order to pass this function to C, we obtain its address using the function `cfunction`:

```
| julia> const mycompare_c = cfunction(mycompare, Cint,
|   ~ Tuple{Ref{Cdouble}, Ref{Cdouble}});
```

`cfunction` accepts three arguments: the Julia function (`mycompare`), the return type (`Cint`), and a tuple type of the input argument types, in this case to sort an array of `Cdouble` (`Float64`) elements.

The final call to `qsort` looks like this:

```
| julia> A = [1.3, -2.7, 4.4, 3.1]
| 4-element Array{Float64,1}:
|   1.3
|   -2.7
|   4.4
|   3.1
|
| julia> ccall(:qsort, Void, (Ptr{Cdouble}, Csize_t, Csize_t,
|   ~ Ptr{Void}),
|
|   A, length(A), sizeof(eltype(A)), mycompare_c)
|
| julia> A
| 4-element Array{Float64,1}:
```

544  
-2.7  
1.3  
3.1  
4.4

## CHAPTER 31. CALLING C AND FORTRAN CODE

As can be seen, A is changed to the sorted array [-2.7, 1.3, 3.1, 4.4]. Note that Julia knows how to convert an array into a `Ptr{Cdouble}`, how to compute the size of a type in bytes (identical to C's `sizeof` operator), and so on. For fun, try inserting a `println("mycompare($a,$b)")` line into `mycompare`, which will allow you to see the comparisons that `qsort` is performing (and to verify that it is really calling the Julia function that you passed to it).

## 31.2 Mapping C Types to Julia

It is critical to exactly match the declared C type with its declaration in Julia. Inconsistencies can cause code that works correctly on one system to fail or produce indeterminate results on a different system.

Note that no C header files are used anywhere in the process of calling C functions: you are responsible for making sure that your Julia types and call signatures accurately reflect those in the C header file. (The [Clang package](#) can be used to auto-generate Julia code from a C header file.)

Auto-conversion:

Julia automatically inserts calls to the `Base.cconvert` function to convert each argument to the specified type. For example, the following call:

```
|ccall((:foo, "libfoo"), Void, (Int32, Float64), x, y)
```

will behave as if the following were written:

31.2 MAPPING C TYPES TO JULIA  
ccall(.foo,"libfoo"),Void,(Int32, Float64),

545

```
Base.unsafe_convert(Int32, Base.cconvert(Int32, x)),  
Base.unsafe_convert(Float64, Base.cconvert(Float64, y)))
```

`Base.cconvert` normally just calls `convert`, but can be defined to return an arbitrary new object more appropriate for passing to C. This should be used to perform all allocations of memory that will be accessed by the C code. For example, this is used to convert an `Array` of objects (e.g. strings) to an array of pointers.

`Base.unsafe_convert` handles conversion to `Ptr` types. It is considered unsafe because converting an object to a native pointer can hide the object from the garbage collector, causing it to be freed prematurely.

Type Correspondences:

First, a review of some relevant Julia type terminology:

Bits Types:

There are several special types to be aware of, as no other type can be defined to behave the same:

`Float32`

Exactly corresponds to the `float` type in C (or `REAL*4` in Fortran).

`Float64`

Exactly corresponds to the `double` type in C (or `REAL*8` in Fortran).

`Complex64`

Exactly corresponds to the `complex float` type in C (or `COMPLEX*8` in Fortran).

`Complex128`

Syntax / Keyword	Example	CHAPTER 81 CALLING C AND FORTRAN CODE Description
<code>mutable struct</code>	<code>String</code>	"Leaf Type" :: A group of related data that includes a type-tag, is managed by the Julia GC, and is defined by object-identity. The type parameters of a leaf type must be fully defined (no <code>TypeVars</code> are allowed) in order for the instance to be constructed.
<code>ab-stract type</code>	<code>Any, AbstractArray{T, N}, Complex{T}</code>	"Super Type" :: A super-type (not a leaf-type) that cannot be instantiated, but can be used to describe a group of types.
<code>T{A}</code>	<code>Vector{Int}</code>	"Type Parameter" :: A specialization of a type (typically used for dispatch or storage optimization).
		"TypeVar" :: The T in the type parameter declaration is referred to as a TypeVar (short for type variable).
<code>primi-tive type</code>	<code>Int, Float64</code>	"Primitive Type" :: A type with no fields, but a size. It is stored and defined by-value.
<code>struct</code>	<code>Pair{Int, Int}</code>	"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	<code>Complex128 (isbits)</code>	"Is-Bits" :: A primitive type, or a struct type where all fields are other isbits types. It is defined by-value, and is stored without a type-tag.
<code>struct ...; end</code>	<code>nothing</code>	"Singleton" :: a Leaf Type or Struct with no fields.
<code>(...)</code> or <code>tu-ple(...)</code>	<code>(1, 2, 3)</code>	"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Exactly corresponds to the `complex double` type in C (or `COMPLEX*16` in Fortran).

### Signed

Exactly corresponds to the `signed` type annotation in C (or any `INTEGER` type in Fortran). Any Julia type that is not a subtype of `Signed` is assumed to be `unsigned`.

### `Ref{T}`

## Array{T,N}

When an array is passed to C as a `Ptr{T}` argument, it is not reinterpret-cast: Julia requires that the element type of the array matches `T`, and the address of the first element is passed.

Therefore, if an `Array` contains data in the wrong format, it will have to be explicitly converted using a call such as `trunc(Int32, a)`.

To pass an array `A` as a pointer of a different type without converting the data beforehand (for example, to pass a `Float64` array to a function that operates on uninterpreted bytes), you can declare the argument as `Ptr{Void}`.

If an array of eltype `Ptr{T}` is passed as a `Ptr{Ptr{T}}` argument, `Base.cconvert` will attempt to first make a null-terminated copy of the array with each element replaced by its `Base.cconvert` version. This allows, for example, passing an `argv` pointer array of type `Vector{String}` to an argument of type `Ptr{Ptr{Cchar}}`.

On all systems we currently support, basic C/C++ value types may be translated to Julia types as follows. Every C type also has a corresponding Julia type with the same name, prefixed by C. This can help for writing portable code (and remembering that an `int` in C is not the same as an `Int` in Julia).

System Independent:

The `Cstring` type is essentially a synonym for `Ptr{UInt8}`, except the conversion to `Cstring` throws an error if the Julia string contains any embedded NUL characters (which would cause the string to be silently truncated if the C routine treats NUL as the terminator). If you are passing a `char*` to a C routine that does not assume NUL termination (e.g. because you pass an explicit string length), or if you know for certain that your Julia string does not contain

~~CHAPTER 21. USING C AND FORTRAN CODE~~  
and want to skip the checks, use `CAPTURED` type. `Cstring` can also be used as the `ccall` return type, but in that case it obviously does not introduce any extra checks and is only meant to improve readability of the call.

System-dependent:

#### Note

When calling a Fortran function, all inputs must be passed by reference, so all type correspondences above should contain an additional `Ptr{..}` or `Ref{..}` wrapper around their type specification.

#### Warning

For string arguments (`char*`) the Julia type should be `Cstring` (if NUL-terminated data is expected) or either `Ptr{Cchar}` or `Ptr{UInt8}` otherwise (these two pointer types have the same effect), as described above, not `String`. Similarly, for array arguments (`T[]` or `T*`), the Julia type should again be `Ptr{T}`, not `Vector{T}`.

#### Warning

Julia's `Char` type is 32 bits, which is not the same as the wide character type (`wchar_t` or `wint_t`) on all platforms.

#### Warning

A return type of `Union{}` means the function will not return i.e. C++11 `[[noreturn]]` or C11 `_Noreturn` (e.g. `jl_throw` or `longjmp`). Do not use this for functions that return no value (`void`) but do return, use `Void` instead.

#### Note

31.2 For MAPPING C TYPES TO JULIA, the Julia type should be `Cwstring` (if the 549 C routine expects a NUL-terminated string) or `Ptr{Cwchar_t}` otherwise. Note also that UTF-8 string data in Julia is internally NUL-terminated, so it can be passed to C functions expecting NUL-terminated data without making a copy (but using the `Cwstring` type will cause an error to be thrown if the string itself contains NUL characters).

### Note

C functions that take an argument of the type `char**` can be called by using a `Ptr{Ptr{UInt8}}` type within Julia. For example, C functions of the form:

```
| int main(int argc, char **argv);
```

can be called via the following Julia code:

```
argc = [ "a.out", "arg1", "arg2" ]  
ccall(:main, Int32, (Int32, Ptr{Ptr{UInt8}}), length(argc),  
      → argv)
```

### Note

A C function declared to return `Void` will return the value `nothing` in Julia.

## Struct Type correspondences

Composite types, aka `struct` in C or `TYPE` in Fortran90 (or `STRUCTURE / RECORD` in some variants of F77), can be mirrored in Julia by creating a `struct` definition with the same field layout.

When used recursively, `isbits` types are stored inline. All other types are stored as a pointer to the data. When mirroring a struct used by-value inside another struct in C, it is imperative that you do not attempt to manually copy the fields over, as this will not preserve the correct field alignment. Instead,

Declare an `isbits` struct type that is flat and aligned. It is not possible in the translation to Julia.

Packed structs and union declarations are not supported by Julia.

You can get a near approximation of a `union` if you know, a priori, the field that will have the greatest size (potentially including padding). When translating your fields to Julia, declare the Julia field to be only of that type.

Arrays of parameters can be expressed with `NTuple`:

```
in C:
struct B {
    int A[3];
};

b_a_2 = B.A[2];

in Julia:
struct B
    A::NTuple{3, CInt}
end
b_a_2 = B.A[3] # note the difference in indexing (1-based in
                 Julia, 0-based in C)
```

Arrays of unknown size (C99-compliant variable length structs specified by `[ ]` or `[ 0 ]`) are not directly supported. Often the best way to deal with these is to deal with the byte offsets directly. For example, if a C library declared a proper string type and returned a pointer to it:

```
struct String {
    int strlen;
    char data[];
};
```

In Julia, we can access the parts independently to make a copy of that string:

```
31.2. =MAPPING.C TYPES TO JULIA
str = from_c::Ptr{Void}
len = unsafe_load(Ptr{Cint}(str))
unsafe_string(str + Core.sizeof(Cint), len)
```

551

## Type Parameters

The type arguments to `ccall` are evaluated statically, when the method containing the `ccall` is defined. They therefore must take the form of a literal tuple, not a variable, and cannot reference local variables.

This may sound like a strange restriction, but remember that since C is not a dynamic language like Julia, its functions can only accept argument types with a statically-known, fixed signature.

However, while the type layout must be known statically to compute the `ccall` ABI, the static parameters of the function are considered to be part of this static environment. The static parameters of the function may be used as type parameters in the `ccall` signature, as long as they don't affect the layout of the type. For example, `f(x::T) where {T} = ccall(:valid, Ptr{T}, (Ptr{T},), x)` is valid, since `Ptr` is always a word-size primitive type. But, `g(x::T) where {T} = ccall(:notvalid, T, (T,), x)` is not valid, since the type layout of `T` is not known statically.

## SIMD Values

Note: This feature is currently implemented on 64-bit x86 and AArch64 platforms only.

If a C/C++ routine has an argument or return value that is a native SIMD type, the corresponding Julia type is a homogeneous tuple of `VecElement` that naturally maps to the SIMD type. Specifically:

The tuple must be the same size as the SIMD type. For example, a tuple representing an `_m128` on x86 must have a size of 16

The element type of the tuple must be an instance of `VecElement{T}` where `T` is a primitive type that is 1, 2, 4 or 8 bytes.

For instance, consider this C routine that uses AVX intrinsics:

```
#include <immintrin.h>

__m256 dist( __m256 a, __m256 b ) {
    return _mm256_sqrt_ps(_mm256_add_ps(_mm256_mul_ps(a, a),
                                         _mm256_mul_ps(b, b)));
}
```

The following Julia code calls `dist` using `ccall`:

```
const m256 = NTuple{8, VecElement{Float32}}

a = m256(ntuple(i -> VecElement(sin(Float32(i))), 8))
b = m256(ntuple(i -> VecElement(cos(Float32(i))), 8))

function call_dist(a::m256, b::m256)
    ccall((:dist, "libdist"), m256, (m256, m256), a, b)
end

println(call_dist(a,b))
```

The host machine must have the requisite SIMD registers. For example, the code above will not work on hosts without AVX support.

## Memory Ownership

`malloc/free`

Memory allocation and deallocation of such objects must be handled by calls to the appropriate cleanup routines in the libraries being used, just like in

~~31.3 MAPPING C FUNCTIONS TO JULIA~~ Object received from a C library ~~will be freed by~~ ~~Libc.free~~ in Julia, as this may result in the `free` function being called via the wrong `libc` library and cause Julia to crash. The reverse (passing an object allocated in Julia to be freed by an external library) is equally invalid.

### When to use T, Ptr{T} and Ref{T}

In Julia code wrapping calls to external C routines, ordinary (non-pointer) data should be declared to be of type T inside the `ccall`, as they are passed by value. For C code accepting pointers, `Ref{T}` should generally be used for the types of input arguments, allowing the use of pointers to memory managed by either Julia or C through the implicit call to `Base.cconvert`. In contrast, pointers returned by the C function called should be declared to be of output type `Ptr{T}`, reflecting that the memory pointed to is managed by C only. Pointers contained in C structs should be represented as fields of type `Ptr{T}` within the corresponding Julia struct types designed to mimic the internal structure of corresponding C structs.

In Julia code wrapping calls to external Fortran routines, all input arguments should be declared as of type `Ref{T}`, as Fortran passes all variables by reference. The return type should either be `Void` for Fortran subroutines, or a T for Fortran functions returning the type T.

## 31.3 Mapping C Functions to Julia

### **ccall/cfunction** argument translation guide

For translating a C argument list to Julia:

T, where T is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents

- T, where T is an equivalent Julia Bits Type (per the table above)

**Cuint**

- argument value will be copied (passed by value)

**struct T** (including typedef to a struct)

- T, where T is a Julia leaf type
- argument value will be copied (passed by value)

**void\***

- depends on how this parameter is used, first translate this to the intended pointer type, then determine the Julia equivalent using the remaining rules in this list
- this argument may be declared as **Ptr{Void}**, if it really is just an unknown pointer

**jl\_value\_t\***

- Any
- argument value must be a valid Julia object
- currently unsupported by **cfunction**

**jl\_value\_t\*\***

- **Ref{Any}**
- argument value must be a valid Julia object (or C\_NULL)
- currently unsupported by **cfunction**

**T\***

- **Ref{T}**, where T is the Julia type corresponding to T
- argument value will be copied if it is an **isbits** type otherwise, the value must be a valid Julia object

- `Ptr{Void}` (you may need to use `cfunction` explicitly to create this pointer)

... (e.g. a vararg)

- `T...`, where `T` is the Julia type

`va_arg`

- not supported

## **ccall/cfunction** return type translation guide

For translating a C return type to Julia:

`void`

- `Void` (this will return the singleton instance `nothing::Void`)

`T`, where `T` is one of the primitive types: `char`, `int`, `long`, `short`, `float`, `double`, `complex`, `enum` or any of their `typedef` equivalents

- `T`, where `T` is an equivalent Julia Bits Type (per the table above)
- if `T` is an `enum`, the argument type should be equivalent to `Cint` or `Cuint`
- argument value will be copied (returned by-value)

`struct T` (including `typedef` to a struct)

- `T`, where `T` is a Julia Leaf Type
- argument value will be copied (returned by-value)

`void*`

- 556 – depends on how this parameter is used, first translate FORTRAN CODE  
tended pointer type, then determine the Julia equivalent using the remaining rules in this list
- this argument may be declared as `Ptr{Void}`, if it really is just an unknown pointer

`jl_value_t*`

- `Any`
- argument value must be a valid Julia object

`jl_value_t**`

- `Ref{Any}`
- argument value must be a valid Julia object (or `C_NULL`)

`T*`

- If the memory is already owned by Julia, or is an `isbits` type, and is known to be non-null:
  - \* `Ref{T}`, where `T` is the Julia type corresponding to `T`
  - \* a return type of `Ref{Any}` is invalid, it should either be `Any` (corresponding to `jl_value_t*`) or `Ptr{Any}` (corresponding to `Ptr{Any}`)
  - \* C MUST NOT modify the memory returned via `Ref{T}` if `T` is an `isbits` type
- If the memory is owned by C:
  - \* `Ptr{T}`, where `T` is the Julia type corresponding to `T`

`(T*)(...)` (e.g. a pointer to a function)

- `Ptr{Void}` (you may need to use `cfunction` explicitly to create this pointer)

Because C doesn't support multiple return values, often C functions will take pointers to data that the function will modify. To accomplish this within a `ccall`, you need to first encapsulate the value inside an `Ref{T}` of the appropriate type. When you pass this `Ref` object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

```
width = Ref{Cint}(0)
range = Ref{Cfloat}(0)
ccall(:foo, Void, (Ref{Cint}, Ref{Cfloat}), width, range)
```

Upon return, the contents of `width` and `range` can be retrieved (if they were changed by `foo`) by `width[ ]` and `range[ ]`; that is, they act like zero-dimensional arrays.

Special Reference Syntax for `ccall` (deprecated):

The `&` syntax is deprecated, use the `Ref{T}` argument type instead.

A prefix `&` is used on an argument to `ccall` to indicate that a pointer to a scalar argument should be passed instead of the scalar value itself (required for all Fortran function arguments, as noted above). The following example computes a dot product using a BLAS function.

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    @assert length(DX) == length(DY)
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot_, "libLAPACK"),
                    Float64,
                    (Ref{Int32}, Ptr{Float64}, Ref{Int32},
                     → Ptr{Float64}, Ref{Int32}),
```

```

    return product
end

```

The meaning of prefix `&` is not quite the same as in C. In particular, any changes to the referenced variables will not be visible in Julia unless the type is mutable (declared via `type`). However, even for immutable structs it will not cause any harm for called functions to attempt such modifications (that is, writing through the passed pointers). Moreover, `&` may be used with any expression, such as `&0` or `&f(x)`.

When a scalar value is passed with `&` as an argument of type `Ptr{T}`, the value will first be converted to type `T`.

### 31.4 Some Examples of C Wrappers

Here is a simple example of a C wrapper that returns a `Ptr` type:

```

mutable struct gsl_permutation
end

# The corresponding C signature is
#     gsl_permutation * gsl_permutation_alloc (size_t n);
function permutation_alloc(n::Integer)
    output_ptr = ccall(
        (:gsl_permutation_alloc, :libgsl), # name of C function
        → and library
        Ptr{gsl_permutation}, # output type
        (Csize_t,), # tuple of input types
        n # name of Julia
        → variable to pass in
    )

```

```

    if output_ptr == C_NULL # Could not allocate memory
        throw(OutOfMemoryError())
    end
    return output_ptr
end

```

The [GNU Scientific Library](#) (here assumed to be accessible through `:libgsl`) defines an opaque pointer, `gsl_permutation *`, as the return type of the C function `gsl_permutation_alloc`. As user code never has to look inside the `gsl_permutation` struct, the corresponding Julia wrapper simply needs a new type declaration, `gsl_permutation`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `Ptr` type. The return type of the `ccall` is declared as `Ptr{gsl_permutation}`, since the memory allocated and pointed to by `output_ptr` is controlled by C (and not Julia).

The input `n` is passed by value, and so the function's input signature is simply declared as `(Csize_t, )` without any `Ref` or `Ptr` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature should instead be `(Ref{Csize_t}, )`, since Fortran variables are passed by reference.) Furthermore, `n` can be any type that is convertible to a `Csize_t` integer; the `ccall` implicitly calls `Base.cconvert(Csize_t, n)`.

Here is a second example wrapping the corresponding destructor:

```

# The corresponding C signature is
#     void gsl_permutation_free (gsl_permutation * p);
function permutation_free(p::Ref{gsl_permutation})
    ccall(
        (:gsl_permutation_free, :libgsl), # name of C function and
        # library
        Void,                           # output type

```

```
(Ref{gsl_permutation},),) # tuple of input types
    p                      # name of Julia variable
    → to pass in
)
end
```

Here, the input `p` is declared to be of type `Ref{gsl_permutation}`, meaning that the memory that `p` points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type `Ptr{gsl_permutation}`, but it is convertable using `Base.cconvert` and therefore can be used in the same (covariant) context of the input argument to a `ccall`. A pointer to memory allocated by Julia must be of type `Ref{gsl_permutation}`, to ensure that the memory address pointed to is valid and that Julia's garbage collector manages the chunk of memory pointed to correctly. Therefore, the `Ref{gsl_permutation}` declaration allows pointers managed by C or Julia to be used.

If the C wrapper never expects the user to pass pointers to memory managed by Julia, then using `p::Ptr{gsl_permutation}` for the method signature of the wrapper and similarly in the `ccall` is also acceptable.

Here is a third example passing Julia arrays:

```
# The corresponding C signature is
#   int gsl_sf_bessel_Jn_array (int nmin, int nmax, double x,
#                               double result_array[])
function sf_bessel_Jn_array(nmin::Integer, nmax::Integer, x::Real)
    if nmax < nmin
        throw(DomainError())
    end
    result_array = Vector{Cdouble}(nmax - nmin + 1)
    errorcode = ccall(
```

## 31.4. SOME EXAMPLES OF C WRAPPERS 561

```

    Y:gsl_sf_bessel_Jn_array.libgsl), # name of C function
    → and library

    Cint,                                # output type
    (Cint, Cint, Cdouble, Ref{Cdouble}),# tuple of input types
    nmin, nmax, x, result_array         # names of Julia
    → variables to pass in

)
if errorcode != 0
    error("GSL error code $errorcode")
end
return result_array
end

```

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `result_array`. This variable can only be used with corresponding input type declaration `Ref{Cdouble}`, since its memory is allocated and managed by Julia, not C. The implicit call to `Base.cconvert(Ref{Cdouble}, result_array)` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

Note that for this code to work correctly, `result_array` must be declared to be of type `Ref{Cdouble}` and not `Ptr{Cdouble}`. The memory is managed by Julia and the `Ref` signature alerts Julia's garbage collector to keep managing the memory for `result_array` while the `ccall` executes. If `Ptr{Cdouble}` were used instead, the `ccall` may still work, but Julia's garbage collector would not be aware that the memory declared for `result_array` is being used by the external C function. As a result, the code may produce a memory leak if `result_array` never gets freed by the garbage collector, or if the garbage collector prematurely frees `result_array`, the C function may end up throwing an invalid memory access exception.

When passing data to a `ccall`, it is best to avoid using the `pointer` function. Instead define a convert method and pass the variables directly to the `ccall`. `ccall` automatically arranges that all of its arguments will be preserved from garbage collection until the call returns. If a C API will store a reference to memory allocated by Julia, after the `ccall` returns, you must arrange that the object remains visible to the garbage collector. The suggested way to handle this is to make a global variable of type `Array{Ref, 1}` to hold these values, until the C library notifies you that it is finished with them.

Whenever you have created a pointer to Julia data, you must ensure the original data exists until you are done with using the pointer. Many methods in Julia such as `unsafe_load` and `String` make copies of data instead of taking ownership of the buffer, so that it is safe to free (or alter) the original data without affecting Julia. A notable exception is `unsafe_wrap` which, for performance reasons, shares (or can be told to take ownership of) the underlying buffer.

The garbage collector does not guarantee any order of finalization. That is, if `a` contained a reference to `b` and both `a` and `b` are due for garbage collection, there is no guarantee that `b` would be finalized after `a`. If proper finalization of `a` depends on `b` being valid, it must be handled in other ways.

## 31.6 Non-constant Function Specifications

A `(name, library)` function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `eval` as follows:

```
|@eval ccall($(string("a", "b")), "lib"), ...
```

This expression constructs a name using `string`, then substitutes this name

into a `INDIRECT CALL` expression, which is then evaluated. Keep in mind that `eval` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `$`). For this reason, `eval` is typically only used to form top-level definitions, for example when wrapping libraries that contain many similar functions.

If your usage is more dynamic, use indirect calls as described in the next section.

## 31.7 Indirect Calls

The first argument to `ccall` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `Ptr`, which will be used as the address of the native function to call. This behavior occurs when the first `ccall` argument contains references to non-constants, such as local variables, function arguments, or non-constant globals.

For example, you might look up the function via `dlsym`, then cache it in a shared reference for that session. For example:

```
macro dlsym(func, lib)
    z = Ref{Ptr{Void}}(C_NULL)
    quote
        let zlocal = $z[]
        if zlocal == C_NULL
            zlocal = dlsym($(esc(lib))::Ptr{Void},
                           $(esc(func))::Ptr{Void})
        $z[] = $zlocal
    end
    zlocal
end
end
end
```

```
mylibvar = Libdl.dlopen("mylib")
ccall(@dlsym("myfunc", mylibvar), Void, ())
```

## 31.8 Closing a Library

It is sometimes useful to close (unload) a library so that it can be reloaded. For instance, when developing C code for use with Julia, one may need to compile, call the C code from Julia, then close the library, make an edit, recompile, and load in the new changes. One can either restart Julia or use the `Libdl` functions to manage the library explicitly, such as:

```
lib = Libdl.dlopen("./my_lib.so") # Open the library explicitly.
sym = Libdl.dlsym(lib, :my_fcn)   # Get a symbol for the function
#       to call.
ccall(sym, ...) # Use the symbol instead of the (symbol, library)
#       tuple (remaining arguments are the same).
Libdl.dlclose(lib) # Close the library explicitly.
```

Note that when using `ccall` with the tuple input (e.g., `ccall((:my_fcn, "./my_lib.so"), ...)`), the library is opened implicitly and it may not be explicitly closed.

## 31.9 Calling Convention

The second argument to `ccall` can optionally be a calling convention specifier (immediately preceding return type). Without any specifier, the platform-default C calling convention is used. Other supported conventions are: `stdcall`, `cdecl`, `fastcall`, and `thiscall` (no-op on 64-bit Windows). For example (from `base/libc.jl`) we see the same `gethostname``ccall` as above, but with the correct signature for Windows:

```
hn = Vector{UInt8}(256)  
err = ccall(:gethostname, stdcall, Int32, (Ptr{UInt8}, UInt32),  
            hn, length(hn))
```

For more information, please see the [LLVM Language Reference](#).

There is one additional special calling convention `llvmpcall`, which allows inserting calls to LLVM intrinsics directly. This can be especially useful when targeting unusual platforms such as GPGPUs. For example, for [CUDA](#), we need to be able to read the thread index:

```
ccall("llvm.nvvm.read.ptx.sreg.tid.x", llvmpcall, Int32, ())
```

As with any `ccall`, it is essential to get the argument signature exactly correct. Also, note that there is no compatibility layer that ensures the intrinsic makes sense and works on the current target, unlike the equivalent Julia functions exposed by `Core.Intrinsics`.

## 31.10 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `cglobal` function. The arguments to `cglobal` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

```
julia> cglobal(:errno, :libc), Int32)  
Ptr{Int32} @0x00007f418d0816b8
```

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `unsafe_load` and `unsafe_store!`.

## 3611 Accessing Data through Pointers

The following methods are described as "unsafe" because a bad pointer or type declaration can cause Julia to terminate abruptly.

Given a `Ptr{T}`, the contents of type `T` can generally be copied from the referenced memory into a Julia object using `unsafe_load(ptr, [index])`. The index argument is optional (default is 1), and follows the Julia-convention of 1-based indexing. This function is intentionally similar to the behavior of `getindex` and `setindex!` (e.g. `[ ]` access syntax).

The return value will be a new object initialized to contain a copy of the contents of the referenced memory. The referenced memory can safely be freed or released.

If `T` is `Any`, then the memory is assumed to contain a reference to a Julia object (a `jl_value_t*`), the result will be a reference to this object, and the object will not be copied. You must be careful in this case to ensure that the object was always visible to the garbage collector (pointers do not count, but the new reference does) to ensure the memory is not prematurely freed. Note that if the object was not originally allocated by Julia, the new object will never be finalized by Julia's garbage collector. If the `Ptr` itself is actually a `jl_value_t*`, it can be converted back to a Julia object reference by `unsafe_pointer_to_objref(ptr)`. (Julia values `v` can be converted to `jl_value_t*` pointers, as `Ptr{Void}`, by calling `pointer_from_objref(v)`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `unsafe_store!(ptr, value, [index])`. Currently, this is only supported for primitive types or other pointer-free (`isbits`) immutable struct types.

Any operation that throws an error is probably currently unimplemented and should be posted as a bug so that it can be resolved.

If the `ptr` is a plain-data array (primitive type or immutable struct), the function `unsafe_wrap(Array, ptr, dims, [own])` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `free(ptr)` when the returned `Array` object is finalized. If the `own` parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr` type in Julia (e.g. using `+`) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr` in Julia always moves the pointer by some number of bytes, not elements. This way, the address values obtained from pointer arithmetic do not depend on the element types of pointers.

## 31.12 Thread-safety

Some C libraries execute their callbacks from a different thread, and since Julia isn't thread-safe you'll need to take some extra precautions. In particular, you'll need to set up a two-layered system: the C callback should only schedule (via Julia's event loop) the execution of your "real" callback. To do this, create a `AsyncCondition` object and wait on it:

```
|cond = Base.AsyncCondition()  
|wait(cond)
```

The callback you pass to C should only execute a `ccall` to `:uv_async_send`, passing `cond.handle` as the argument, taking care to avoid any allocations or other interactions with the Julia runtime.

Note that events may be coalesced, so multiple calls to `uv_async_send` may result in a single wakeup notification to the condition.

## 36813 More About Callbacks

For more details on how to pass callbacks to C libraries, see this [blog post](#).

## 31.14 C++

For direct C++ interfacing, see the [Cxx](#) package. For tools to create C++ bindings, see the [CxxWrap](#) package.

C/C++ Name	Fortran name	Standard Julia Alias	Julia Base Type	569
unsigned char	CHAR-ACTER	Cuchar	UInt8	
bool (only in C++)		Cuchar	UInt8	
short	INTE-GER*2, LOGI-CAL*2	Cshort	Int16	
unsigned short		Cushort	UInt16	
int, BOOL (C, typical)	INTE-GER*4, LOGI-CAL*4	Cint	Int32	
unsigned int		Cuint	UInt32	
long long	INTE-GER*8, LOGI-CAL*8	Clonglong	Int64	
unsigned long long		Culonglong	UInt64	
intmax_t		Cintmax_t	Int64	
uintmax_t		Cuintmax_t	UInt64	
float	REAL*4i	Cfloat	Float32	
double	REAL*8	Cdouble	Float64	
complex float	COM-PLEX*8	Complex64	Complex{Float32}	
complex double	COM-PLEX*16	Complex128	Complex{Float64}	
ptrdiff_t		Cptrdiff_t	Int	
ssize_t		Cssize_t	Int	
size_t		Csize_t	UInt	
void			Void	
void and [[noreturn]] or _Noreturn			Union{}	
void*			Ptr{Void}	
T* (where T represents an appropriately defined type)			Ref{T}	
char* (or char[], e.g. a string)	CHAR-AC-TER*N		Cstring if NUL-terminated, or Ptr{UInt8} if not	
char** (or *char[])			Ptr{Ptr{UInt8}}	
ii_value_t*			Any	

C name	Standard Julia Alias	Julia Base Type
char	Cchar	Int8 (x86, x86_64), UInt8 (powerpc, arm)
long	Clong	Int (UNIX), Int32 (Windows)
unsigned long	Culong	UInt (UNIX), UInt32 (Windows)
wchar_t	Cwchar_t	Int32 (UNIX), UInt16 (Windows)

## Chapter 32

# Handling Operating System Variation

When dealing with platform libraries, it is often necessary to provide special cases for various platforms. The variable `Sys.KERNEL` can be used to write these special cases. There are several functions in the `Sys` module intended to make this easier: `isunix`, `islinux`, `isapple`, `isbsd`, and `iswindows`. These may be used as follows:

```
|if Sys.iswindows()
|    some_complicated_thing(a)
|end
```

Note that `islinux` and `isapple` are mutually exclusive subsets of `isunix`. Additionally, there is a macro `@static` which makes it possible to use these functions to conditionally hide invalid code, as demonstrated in the following examples.

Simple blocks:

```
|ccall((@static Sys.iswindows() ? :_fopen : :fopen), ...)
```

Complex blocks:

```
|@static if Sys.islinux()
|    some_complicated_thing(a)
```

```
    some_different_thing(a)  
end
```

When chaining conditionals (including `if/elseif/end`), the `@static` must be repeated for each level (parentheses optional, but recommended for readability):

```
@static Sys.iswindows() ? :a : (@static Sys.isapple() ? :b : :c)
```

# Chapter 33

## Environment Variables

Julia may be configured with a number of environment variables, either in the usual way of the operating system, or in a portable way from within Julia. Suppose you want to set the environment variable JULIA\_EDITOR to `vim`, then either type `ENV["JULIA_EDITOR"] = "vim"` for instance in the REPL to make this change on a case by case basis, or add the same to the user configuration file `.juliarc.jl` in the user's home directory to have a permanent effect. The current value of the same environment variable is determined by evaluating `ENV["JULIA_EDITOR"]`.

The environment variables that Julia uses generally start with JULIA. If `Base.versioninfo` is called with `verbose` equal to `true`, then the output will list defined environment variables relevant for Julia, including those for which JULIA appears in the name.

### 33.1 File locations

#### **JULIA\_HOME**

The absolute path of the directory containing the Julia executable, which sets the global variable `Base.JULIA_HOME`. If `$JULIA_HOME` is not set, then Julia determines the value `Base.JULIA_HOME` at run-time.

`julia` executable itself is one of

## CHAPTER 33. ENVIRONMENT VARIABLES

```
| $JULIA_HOME/julia  
| $JULIA_HOME/julia-debug
```

by default.

The global variable `Base.DATAROOTDIR` determines a relative path from `Base.JULIA_HOME` to the data directory associated with Julia. Then the path

```
| $JULIA_HOME/$DATAROOTDIR/julia/base
```

determines the directory in which Julia initially searches for source files (via `Base.find_source_file()`).

Likewise, the global variable `Base.SYSCONFDIR` determines a relative path to the configuration file directory. Then Julia searches for a `juliarc.jl` file at

```
| $JULIA_HOME/$SYSCONFDIR/julia/juliarc.jl  
| $JULIA_HOME/../../etc/julia/juliarc.jl
```

by default (via `Base.load_juliarc()`).

For example, a Linux installation with a Julia executable located at `/bin/julia`, a `DATAROOTDIR` of `../../share`, and a `SYSCONFDIR` of `../../etc` will have `JULIA_HOME` set to `/bin`, a source-file search path of

```
| /share/julia/base
```

and a global configuration search path of

```
| /etc/julia/juliarc.jl
```

## **JULIA\_LOAD\_PATH**

A separated list of absolute paths that are to be appended to the variable `LOAD_PATH`. (In Unix-like systems, the path separator is `:`; in Windows systems, the path separator is `;`.) The `LOAD_PATH` variable is where `Base.require` and `Base.load_in_path()` look for code; it defaults to the absolute paths

| 33.1 FILE LOCATIONS |  
| \$JULIA\_HOME/../../local/share/julia/site/v\$(VERSION.major).\$(VERSION.  
| minor)  
| \$JULIA\_HOME/./share/julia/site/v\$(VERSION.major).\$(VERSION.minor)

so that, e.g., version 0.6 of Julia on a Linux system with a Julia executable at  
`/bin/julia` will have a default LOAD\_PATH of

| /local/share/julia/site/v0.6  
| /share/julia/site/v0.6

## JULIA\_PKGDIR

The path of the parent directory `Pkg.Dir._pkgroot()` for the version-specific  
Julia package repositories. If the path is relative, then it is taken with respect  
to the working directory. If `$JULIA_PKGDIR` is not set, then `Pkg.Dir._pkg-  
root()` defaults to

| \$HOME/.julia

Then the repository location `Pkg.dir` for a given Julia version is

| \$JULIA\_PKGDIR/v\$(VERSION.major).\$(VERSION.minor)

For example, for a Linux user whose home directory is `/home/alice`, the di-  
rectory containing the package repositories would by default be

| /home/alice/.julia

and the package repository for version 0.6 of Julia would be

| /home/alice/.julia/v0.6

## JULIA\_HISTORY

The absolute path `Base.REPL.find_hist_file()` of the REPL's history file.  
If `$JULIA_HISTORY` is not set, then `Base.REPL.find_hist_file()` defaults  
to

## JULIA\_PKGRESOLVE\_ACCURACY

A positive `Int` that determines how much time the max-sum subroutine `MaxSum.maxsum()` of the package dependency resolver `Base.Pkg.resolve` will devote to attempting satisfying constraints before giving up: this value is by default 1, and larger values correspond to larger amounts of time.

Suppose the value of `$JULIA_PKGRESOLVE_ACCURACY` is `n`. Then

the number of pre-decimation iterations is `20*n`,

the number of iterations between decimation steps is `10*n`, and

at decimation steps, at most one in every `20*n` packages is decimated.

## 33.2 External applications

### JULIA\_SHELL

The absolute path of the shell with which Julia should execute external commands (via `Base.repl_cmd()`). Defaults to the environment variable `$SHELL`, and falls back to `/bin/sh` if `$SHELL` is unset.

Note

On Windows, this environment variable is ignored, and external commands are executed directly.

### JULIA\_EDITOR

The editor returned by `Base.editor()` and used in, e.g., `Base.edit`, referring to the command of the preferred editor, for instance `vim`.

`$JULIA_EDITOR` takes precedence over `$VISUAL`, which in turn takes precedence over `$EDITOR`. If none of these environment variables is set, then the

`EDITOR` or `VISUAL` if both are present on Windows and OS X, or `/etc/alternatives/editor` if it exists, or `emacs` otherwise.

### Note

`$JULIA_EDITOR` is not used in the determination of the editor for `Base.Pkg.edit`: this function checks `$VISUAL` and `$EDITOR` alone.

## 33.3 Parallelization

### **JULIA\_CPU\_CORES**

Overrides the global variable `Base.Sys.CPU_CORES`, the number of logical CPU cores available.

### **JULIA\_WORKER\_TIMEOUT**

A `Float64` that sets the value of `Base.worker_timeout()` (default: `60.0`). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

### **JULIA\_NUM\_THREADS**

An unsigned 64-bit integer (`uint64_t`) that sets the maximum number of threads available to Julia. If `$JULIA_NUM_THREADS` exceeds the number of available physical CPU cores, then the number of threads is set to the number of cores. If `$JULIA_NUM_THREADS` is not positive or is not set, or if the number of CPU cores cannot be determined through system calls, then the number of threads is set to 1.

### **JULIA\_THREAD\_SLEEP\_THRESHOLD**

If set to a string that starts with the case-insensitive substring "infinite", then spinning threads never sleep. Otherwise, `$JULIA_THREAD_SLEEP_THRESHOLD`

`JULIA_THREADSLEEP` is interpreted as an unsigned integer giving, in nanoseconds, the amount of time after which spinning threads should sleep.

## **JULIA\_EXCLUSIVE**

If set to anything besides 0, then Julia's thread policy is consistent with running on a dedicated machine: the master thread is on proc 0, and threads are affinitized. Otherwise, Julia lets the operating system handle thread policy.

### 33.4 REPL formatting

Environment variables that determine how REPL output should be formatted at the terminal. Generally, these variables should be set to [ANSI terminal escape sequences](#). Julia provides a high-level interface with much of the same functionality: see the section on [Interacting With Julia](#).

#### **JULIA\_ERROR\_COLOR**

The formatting `Base.error_color()` (default: light red, "\033[91m") that errors should have at the terminal.

#### **JULIA\_WARN\_COLOR**

The formatting `Base.warn_color()` (default: yellow, "\033[93m") that warnings should have at the terminal.

#### **JULIA\_INFO\_COLOR**

The formatting `Base.info_color()` (default: cyan, "\033[36m") that info should have at the terminal.

#### **JULIA\_INPUT\_COLOR**

The formatting `Base.input_color()` (default: normal, "\033[0m") that input should have at the terminal.

The formatting `Base.answer_color()` (default: `normal`, "`\033[0m`") that output should have at the terminal.

## JULIA\_STACKFRAME\_LINEINFO\_COLOR

The formatting `Base.stackframe_lineinfo_color()` (default: `bold`, "`\033[1m`") that line info should have during a stack trace at the terminal.

## JULIA\_STACKFRAME\_FUNCTION\_COLOR

The formatting `Base.stackframe_function_color()` (default: `bold`, "`\033[1m`") that function calls should have during a stack trace at the terminal.

## 33.5 Debugging and profiling

### JULIA\_GC\_ALLOC\_POOL, JULIA\_GC\_ALLOC\_OTHER,

### JULIA\_GC\_ALLOC\_PRINT

If set, these environment variables take strings that optionally start with the character '`r`', followed by a string interpolation of a colon-separated list of three signed 64-bit integers (`int64_t`). This triple of integers `a:b:c` represents the arithmetic sequence `a, a + b, a + 2*b, ... c`.

If it's the `n`th time that `jl_gc_pool_alloc()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_POOL`, then garbage collection is forced.

If it's the `n`th time that `maybe_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_OTHER`, then garbage collection is forced.

If it's the `n`th time that `jl_gc_collect()` has been called, and `n` belongs to the arithmetic sequence represented by `$JULIA_GC_ALLOC_PRINT`,

580 then counts for the number of calls to `jl_gc_collect()` are printed.

If the value of the environment variable begins with the character 'r', then the interval between garbage collection events is randomized.

#### Note

These environment variables only have an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to 1 in the build configuration).

### **JULIA\_GC\_NO\_GENERATIONAL**

If set to anything besides 0, then the Julia garbage collector never performs "quick sweeps" of memory.

#### Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to 1 in the build configuration).

### **JULIA\_GC\_WAIT\_FOR\_DEBUGGER**

If set to anything besides 0, then the Julia garbage collector will wait for a debugger to attach instead of aborting whenever there's a critical error.

#### Note

This environment variable only has an effect if Julia was compiled with garbage-collection debugging (that is, if `WITH_GC_DEBUG_ENV` is set to 1 in the build configuration).

If set to anything besides 0, then the compiler will create and register an event listener for just-in-time (JIT) profiling.

#### Note

This environment variable only has an effect if Julia was compiled with JIT profiling support, using either

Intel's [VTune™ Amplifier](#) (USE\_INTEL\_JITEVENTS set to 1 in the build configuration), or

[OProfile](#) (USE\_OPROFILE\_JITEVENTS set to 1 in the build configuration).

## **JULIA\_LLVM\_ARGS**

Arguments to be passed to the LLVM backend.

#### Note

This environment variable has an effect only if Julia was compiled with JL\_DEBUG\_BUILD set — in particular, the `julia-debug` executable is always compiled with this build variable.

## **JULIA\_DEBUG\_LOADING**

If set, then Julia prints detailed information about the cache in the loading process of [Base.require](#).



# Chapter 34

## Embedding Julia

As we have seen in [Calling C and Fortran Code](#), Julia has a simple and efficient way to call functions written in C. But there are situations where the opposite is needed: calling Julia function from C code. This can be used to integrate Julia code into a larger C/C++ project, without the need to rewrite everything in C/C++. Julia has a C API to make this possible. As almost all programming languages have some way to call C functions, the Julia C API can also be used to build further language bridges (e.g. calling Julia from Python or C#).

### 34.1 High-Level Embedding

We start with a simple C program that initializes Julia and calls some Julia code:

```
#include <julia.h>
JULIA_DEFINE_FAST_TLS() // only define this once, in an executable
// (not in a shared library) if you want fast code.

int main(int argc, char *argv[])
{
    /* required: setup the Julia context */
    jl_init();
```

```
/* run Julia commands */
jl_eval_string("print(sqrt(2.0))");

/* strongly recommended: notify Julia that the
   program is about to terminate. this allows
   Julia time to cleanup pending write requests
   and run all finalizers
*/
jl_atexit_hook(0);
return 0;
}
```

In order to build this program you have to put the path to the Julia header into the include path and link against **libjulia**. For instance, when Julia is installed to **\$JULIA\_DIR**, one can compile the above test program **test.c** with **gcc** using:

```
gcc -o test -fPIC -I$JULIA_DIR/include/julia -L$JULIA_DIR/lib test
     .c -ljulia $JULIA_DIR/lib/julia/libstdc++.so.6
```

Then if the environment variable **JULIA\_HOME** is set to **\$JULIA\_DIR/bin**, the output **test** program can be executed.

Alternatively, look at the **embedding.c** program in the Julia source tree in the **examples/** folder. The file **ui/repl.c** program is another simple example of how to set **jl\_options** options while linking against **libjulia**.

The first thing that has to be done before calling any other Julia C function is to initialize Julia. This is done by calling **jl\_init**, which tries to automatically determine Julia's install location. If you need to specify a custom location, or specify which system image to load, use **jl\_init\_with\_image** instead.

The second statement in the test program evaluates a Julia statement using

Before the program terminates, it is strongly recommended to call `jl_atexit_hook`. The above example program calls this before returning from `main`.

#### Note

Currently, dynamically linking with the `libjulia` shared library requires passing the `RTLD_GLOBAL` option. In Python, this looks like:

```
>>> julia=CDLL('./libjulia.dylib',RTLD_GLOBAL)
>>> julia.jl_init.argtypes = []
>>> julia.jl_init()
250593296
```

#### Note

If the Julia program needs to access symbols from the main executable, it may be necessary to add `-Wl,--export-dynamic` linker flag at compile time on Linux in addition to the ones generated by `julia-config.jl` described below. This is not necessary when compiling a shared library.

### Using `julia-config` to automatically determine build parameters

The script `julia-config.jl` was created to aid in determining what build parameters are required by a program that uses embedded Julia. This script uses the build parameters and system configuration of the particular Julia distribution it is invoked by to export the necessary compiler flags for an embedding program to interact with that distribution. This script is located in the Julia shared data directory.

#### Example

```
#include <julia.h>
```

```
int main(int argc, char *argv[])
{
    jl_init();
    (void)jl_eval_string("println(sqrt(2.0))");
    jl_atexit_hook(0);
    return 0;
}
```

On the command line

A simple use of this script is from the command line. Assuming that `julia-config.jl` is located in `/usr/local/julia/share/julia`, it can be invoked on the command line directly and takes any combination of 3 flags:

```
/usr/local/julia/share/julia/julia-config.jl
Usage: julia-config [--cflags|--ldflags|--ldlibs]
```

If the above example source is saved in the file `embed_example.c`, then the following command will compile it into a running program on Linux and Windows (MSYS2 environment), or if on OS/X, then substitute `clang` for `gcc`:

```
/usr/local/julia/share/julia/julia-config.jl --cflags --ldflags --
ldlibs | xargs gcc embed_example.c
```

Use in Makefiles

But in general, embedding projects will be more complicated than the above, and so the following allows general makefile support as well – assuming GNU make because of the use of the shell macro expansions. Additionally, though many times `julia-config.jl` may be found in the directory `/usr/local`, this is not necessarily the case, but Julia can be used to locate `julia-config.jl` too, and the makefile can be used to take advantage of that. The above example is extended to use a Makefile:

```
JL_SHARE = $(shell julia -e 'print(joinpath(JULIA_HOME, Base.  
    DATAROOTDIR, "julia"))')  
  
CFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)  
CXXFLAGS += $(shell $(JL_SHARE)/julia-config.jl --cflags)  
LDFLAGS += $(shell $(JL_SHARE)/julia-config.jl --ldflags)  
LDLIBS += $(shell $(JL_SHARE)/julia-config.jl --ldlibs)
```

```
all: embed_example
```

Now the build command is simply `make`.

## 34.2 Converting Types

Real applications will not just need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `Float64` in this way is called **boxing**, and extracting the stored primitive data is called **unboxing**. Our improved sample program that calculates the square root of 2 in Julia and reads back the result in C looks as follows:

```
jl_value_t *ret = jl_eval_string("sqrt(2.0));  
  
if (jl_typeis(ret, jl_float64_type)) {  
    double ret_unboxed = jl_unbox_float64(ret);  
    printf("sqrt(2.0) in C: %e \n", ret_unboxed);  
}  
else {  
    printf("ERROR: unexpected return type from sqrt(::Float64)\n")  
};  
}
```

In order to check whether `ret` is of a specific Julia type, we can use the `jl_isa`, `jl_typeis`, or `jl_is_...` functions. By typing `typeof(sqrt(2.0))`

In the Julia shell we can see that the return type is `Float64`:

```
CHAPTER34: EMBEDDING JULIA
julia> 3.0
3.0
```

To convert the boxed Julia value into a C double the `jl_unbox_float64` function is used in the above code snippet.

Corresponding `jl_box_...` functions are used to convert the other way:

```
jl_value_t *a = jl_box_float64(3.0);
jl_value_t *b = jl_box_float32(3.0f);
jl_value_t *c = jl_box_int32(3);
```

As we will see next, boxing is required to call Julia functions with specific arguments.

### 34.3 Calling Julia Functions

While `jl_eval_string` allows C to obtain the result of a Julia expression, it does not allow passing arguments computed in C to Julia. For this you will need to invoke Julia functions directly, using `jl_call`:

```
jl_function_t *func = jl_get_function(jl_base_module, "sqrt");
jl_value_t *argument = jl_box_float64(2.0);
jl_value_t *ret = jl_call1(func, argument);
```

In the first step, a handle to the Julia function `sqrt` is retrieved by calling `jl_get_function`. The first argument passed to `jl_get_function` is a pointer to the `Base` module in which `sqrt` is defined. Then, the double value is boxed using `jl_box_float64`. Finally, in the last step, the function is called using `jl_call1`. `jl_call0`, `jl_call2`, and `jl_call3` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use `jl_call`:

```
jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t
    nargs)
```

**34.4 MEMORY MANAGEMENT** array of `jl_value_t*` arguments and `nargs` is the number of arguments.

## 34.4 Memory Management

As we have seen, Julia objects are represented in C as pointers. This raises the question of who is responsible for freeing these objects.

Typically, Julia objects are freed by a garbage collector (GC), but the GC does not automatically know that we are holding a reference to a Julia value from C. This means the GC can free objects out from under you, rendering pointers invalid.

The GC can only run when Julia objects are allocated. Calls like `jl_box_float64` perform allocation, and allocation might also happen at any point in running Julia code. However, it is generally safe to use pointers in between `jl_...` calls. But in order to make sure that values can survive `jl_...` calls, we have to tell Julia that we hold a reference to a Julia value. This can be done using the `JL_GC_PUSH` macros:

```
| jl_value_t *ret = jl_eval_string("sqrt(2.0)");
| JL_GC_PUSH1(&ret);
| // Do something with ret
| JL_GC_POP();
```

The `JL_GC_POP` call releases the references established by the previous `JL_GC_PUSH`. Note that `JL_GC_PUSH` is working on the stack, so it must be exactly paired with a `JL_GC_POP` before the stack frame is destroyed.

Several Julia values can be pushed at once using the `JL_GC_PUSH2`, `JL_GC_PUSH3`, and `JL_GC_PUSH4` macros. To push an array of Julia values one can use the `JL_GC_PUSHARGS` macro, which can be used as follows:

```
| jl_value_t **args;
```

```

590 JL_GC_PUSHARGS(args, 2); // args can now hold objects
CHAPTER34.jl_Value_t*
objects
args[0] = some_value;
args[1] = some_other_value;
// Do something with args (e.g. call jl_... functions)
JL_GC_POP();

```

The garbage collector also operates under the assumption that it is aware of every old-generation object pointing to a young-generation one. Any time a pointer is updated breaking that assumption, it must be signaled to the collector with the `jl_gc_wb` (write barrier) function like so:

```

jl_value_t *parent = some_old_value, *child = some_young_value;
((some_specific_type*)parent)->field = child;
jl_gc_wb(parent, child);

```

It is in general impossible to predict which values will be old at runtime, so the write barrier must be inserted after all explicit stores. One notable exception is if the `parent` object was just allocated and garbage collection was not run since then. Remember that most `jl_...` functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. For example:

```

jl_array_t *some_array = ...; // e.g. a Vector{Any}
void **data = (void**)jl_array_data(some_array);
jl_value_t *some_value = ...;
data[0] = some_value;
jl_gc_wb(some_array, some_value);

```

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
<code>jl_gc_collect()</code>	Force a GC run
<code>jl_gc_enable(0)</code>	Disable the GC, return previous state as int
<code>jl_gc_enable(1)</code>	Enable the GC, return previous state as int
<code>jl_gc_is_enabled()</code>	Return current state as int

## 34.5 Working with Arrays

Julia and C can share array data without copying. The next example will show how this works.

Julia arrays are represented in C by the datatype `jl_array_t*`. Basically, `jl_array_t` is a struct that contains:

Information about the datatype

A pointer to the data block

Information about the sizes of the array

To keep things simple, we start with a 1D array. Creating an array containing `Float64` elements of length 10 is done by:

```
jl_value_t* array_type = jl_apply_array_type(jl_float64_type, 1);
jl_array_t* x           = jl_alloc_array_1d(array_type, 10);
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
double *existingArray = (double*)malloc(sizeof(double)*10);
jl_array_t *x = jl_ptr_to_array_1d(array_type, existingArray, 10,
                                     0);
```

The last argument is a boolean indicating whether to **EMBEDDING** ownership of the data. If this argument is non-zero, the GC will call `free` on the data pointer when the array is no longer referenced.

In order to access the data of `x`, we can use `jl_array_data`:

```
double *xData = (double*)jl_array_data(x);
```

Now we can fill the array:

```
for(size_t i=0; i<jl_array_len(x); i++)
    xData[i] = i;
```

Now let us call a Julia function that performs an in-place operation on `x`:

```
jl_function_t *func = jl_get_function(jl_base_module, "reverse!");
jl_call1(func, (jl_value_t*)x);
```

By printing the array, one can verify that the elements of `x` are now reversed.

## Accessing Returned Arrays

If a Julia function returns an array, the return value of `jl_eval_string` and `jl_call` can be cast to a `jl_array_t*`:

```
jl_function_t *func = jl_get_function(jl_base_module, "reverse");
jl_array_t *y = (jl_array_t*)jl_call1(func, (jl_value_t*)x);
```

Now the content of `y` can be accessed as before using `jl_array_data`. As always, be sure to keep a reference to the array while it is in use.

## Multidimensional Arrays

Julia's multidimensional arrays are stored in memory in column-major order. Here is some code that creates a 2D array and accesses its properties:

```
// Create 2D array of float64 type
jl_value_t *array_type = jl_apply_array_type(jl_float64_type, 2);
```

```
// Get array pointer
double *p = (double*)jl_array_data(x);
// Get number of dimensions
int ndims = jl_array_ndims(x);
// Get the size of the i-th dim
size_t size0 = jl_array_dim(x,0);
size_t size1 = jl_array_dim(x,1);

// Fill array with data
for(size_t i=0; i<size1; i++)
    for(size_t j=0; j<size0; j++)
        p[j + size0*i] = i + j;
```

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_dim`) in order to read as idiomatic C code.

## 34.6 Exceptions

Julia code can throw exceptions. For example, consider:

```
jl_eval_string("this_function_does_not_exist());
```

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

```
if (jl_exception_occurred())
    printf("%s \n", jl_typeof_str(jl_exception_occurred()));
```

If you are using the Julia C API from a language that supports exceptions (e.g. Python, C#, C++), it makes sense to wrap each call into `libjulia` with

## Throwing Julia Exceptions

When writing Julia callable functions, it might be necessary to validate arguments and throw exceptions to indicate errors. A typical type check looks like:

```
| if (!jl_typeis(val, jl_float64_type)) {  
|     jl_type_error(function_name, (jl_value_t*)jl_float64_type, val  
| );  
| }
```

General exceptions can be raised using the functions:

```
| void jl_error(const char *str);  
| void jl_errorf(const char *fmt, ...);
```

`jl_error` takes a C string, and `jl_errorf` is called like `printf`:

```
| jl_errorf("argument x = %d is too large", x);
```

where in this example `x` is assumed to be an integer.

# Chapter 35

## Packages

Julia has a built-in package manager for installing add-on functionality written in Julia. It can also install external libraries using your operating system's standard system for doing so, or by compiling from source. The list of registered Julia packages can be found at <http://pkg.julialang.org>. All package manager commands are found in the `Pkg` module, included in Julia's Base install.

First we'll go over the mechanics of the `Pkg` family of commands and then we'll provide some guidance on how to get your package registered. Be sure to read the section below on package naming conventions, tagging versions and the importance of a `REQUIRE` file for when you're ready to add your code to the curated METADATA repository.

### 35.1 Package Status

The `Pkg.status()` function prints out a summary of the state of packages you have installed. Initially, you'll have no packages installed:

```
julia> Pkg.status()
INFO: Initializing package repository /Users/someone/.julia/v0.6
```

Your package directory is automatically initialized the first time you run a `Pkg` command that expects it to exist – which includes `Pkg.status()`. Here’s an example non-trivial set of required and additional packages:

```
julia> Pkg.status()  
Required packages:  
- Distributions      0.2.8  
- SHA                0.3.2  
Additional packages:  
- NumericExtensions  0.2.17  
- Stats               0.2.6
```

These packages are all on registered versions, managed by `Pkg`. Packages can be in more complicated states, indicated by annotations to the right of the installed package version; we will explain these states and annotations as we encounter them. For programmatic usage, `Pkg.installed()` returns a dictionary, mapping installed package names to the version of that package which is installed:

```
julia> Pkg.installed()  
Dict{String,VersionNumber} with 4 entries:  
"Distributions"      => v"0.2.8"  
"Stats"               => v"0.2.6"  
"SHA"                 => v"0.3.2"  
"NumericExtensions"  => v"0.2.17"
```

## 35.2 Adding and Removing Packages

Julia’s package manager is a little unusual in that it is declarative rather than imperative. This means that you tell it what you want and it figures out what

VERSIONS ADDING STANDING REMOVING TO PACKAGES Use requirements optimally – [597](#) minimally. So rather than installing a package, you just add it to the list of requirements and then “resolve” what needs to be installed. In particular, this means that if some package had been installed because it was needed by a previous version of something you wanted, and a newer version doesn’t have that requirement anymore, updating will actually remove that package.

Your package requirements are in the file `~/.julia/v0.6/REQUIRE`. You can edit this file by hand and then call `Pkg.resolve()` to install, upgrade or remove packages to optimally satisfy the requirements, or you can do `Pkg.edit()`, which will open `REQUIRE` in your editor (configured via the `EDITOR` or `VISUAL` environment variables), and then automatically call `Pkg.resolve()` afterwards if necessary. If you only want to add or remove the requirement for a single package, you can also use the non-interactive `Pkg.add()` and `Pkg.rm()` commands, which add or remove a single requirement to `REQUIRE` and then call `Pkg.resolve()`.

You can add a package to the list of requirements with the `Pkg.add()` function, and the package and all the packages that it depends on will be installed:

```
julia> Pkg.status()
No packages installed.

julia> Pkg.add("Distributions")
INFO: Cloning cache of Distributions from
  → git://github.com/JuliaStats/Distributions.jl.git
INFO: Cloning cache of NumericExtensions from
  → git://github.com/lindahua/NumericExtensions.jl.git
INFO: Cloning cache of Stats from
  → git://github.com/JuliaStats/Stats.jl.git
INFO: Installing Distributions v0.2.7
INFO: Installing NumericExtensions v0.2.17
```

```
598  
INFO: Installing Stats v0.2.6  
INFO: REQUIRE updated.
```

## CHAPTER 35. PACKAGES

```
julia> Pkg.status()  
Required packages:  
- Distributions 0.2.7  
Additional packages:  
- NumericExtensions 0.2.17  
- Stats 0.2.6
```

What this is doing is first adding `Distributions` to your `~/.julia/v0.6/REQUIRE` file:

```
$ cat ~/.julia/v0.6/REQUIRE  
Distributions
```

It then runs `Pkg.resolve()` using these new requirements, which leads to the conclusion that the `Distributions` package should be installed since it is required but not installed. As stated before, you can accomplish the same thing by editing your `~/.julia/v0.6/REQUIRE` file by hand and then running `Pkg.resolve()` yourself:

```
$ echo SHA >> ~/.julia/v0.6/REQUIRE
```

```
julia> Pkg.resolve()  
INFO: Cloning cache of SHA from  
→ git://github.com/staticfloat/SHA.jl.git  
INFO: Installing SHA v0.3.2
```

```
julia> Pkg.status()  
Required packages:  
- Distributions 0.2.7  
- SHA 0.3.2
```

- NumericExtensions	0.2.17
- Stats	0.2.6

This is functionally equivalent to calling `Pkg.add("SHA")`, except that `Pkg.add()` doesn't change REQUIRE until after installation has completed, so if there are problems, REQUIRE will be left as it was before calling `Pkg.add()`. The format of the REQUIRE file is described in [Requirements Specification](#); it allows, among other things, requiring specific ranges of versions of packages.

When you decide that you don't want to have a package around any more, you can use `Pkg.rm()` to remove the requirement for it from the REQUIRE file:

```
julia> Pkg.rm("Distributions")
INFO: Removing Distributions v0.2.7
INFO: Removing Stats v0.2.6
INFO: Removing NumericExtensions v0.2.17
INFO: REQUIRE updated.
```

```
julia> Pkg.status()
Required packages:
- SHA          0.3.2
```

```
julia> Pkg.rm("SHA")
INFO: Removing SHA v0.3.2
INFO: REQUIRE updated.
```

```
julia> Pkg.status()
No packages installed.
```

Once again, this is equivalent to editing the REQUIRE file to remove the line with each package name on it then running `Pkg.resolve()` to update the set

installed packages to match. While `Pkg.add()` is convenient for adding and removing requirements for a single package, when you want to add or remove multiple packages, you can call `Pkg.edit()` to manually change the contents of REQUIRE and then update your packages accordingly. `Pkg.edit()` does not roll back the contents of REQUIRE if `Pkg.resolve()` fails – rather, you have to run `Pkg.edit()` again to fix the files contents yourself.

Because the package manager uses libgit2 internally to manage the package git repositories, users may run into protocol issues (if behind a firewall, for example), when running `Pkg.add()`. By default, all GitHub-hosted packages will be accessed via 'https'; this default can be modified by calling `Pkg.set-protocol!()`. The following command can be run from the command line in order to tell git to use 'https' instead of the 'git' protocol when cloning all repositories, wherever they are hosted:

```
| git config --global url."https://".insteadOf git://
```

However, this change will be system-wide and thus the use of `Pkg.setprotocol!()` is preferable.

#### Note

The package manager functions also accept the .jl suffix on package names, though the suffix is stripped internally. For example:

```
| Pkg.add("Distributions.jl")
| Pkg.rm("Distributions.jl")
```

### 35.3 Offline Installation of Packages

For machines with no Internet connection, packages may be installed by copying the package root directory (given by `Pkg.dir()`) from a machine with the same operating system and environment.

1. Adds the name of the package to REQUIRE.
2. Downloads the package to `.cache`, then copies the package to the package root directory.
3. Recursively performs step 2 against all the packages listed in the package's REQUIRE file.
4. Runs `Pkg.build()`

#### Warning

Copying installed packages from a different machine is brittle for packages requiring binary external dependencies. Such packages may break due to differences in operating system versions, build environments, and/or absolute path dependencies.

## 35.4 Installing Unregistered Packages

Julia packages are simply git repositories, clonable via any of the [protocols](#) that git supports, and containing Julia code that follows certain layout conventions. Official Julia packages are registered in the [METADATA.jl](#) repository, available at a well-known location <sup>1</sup>. The `Pkg.add()` and `Pkg.rm()` commands in the previous section interact with registered packages, but the package manager can install and work with unregistered packages too. To install an unregistered package, use `Pkg.clone(url)`, where `url` is a git URL from which the package can be cloned:

```
julia> Pkg.clone("git://example.com/path/to/Package.jl.git")
INFO: Cloning Package from
→ git://example.com/path/to/Package.jl.git
Cloning into 'Package'...
```

```
602      Remote: Counting objects: 22, done.          CHAPTER 35. PACKAGES
         remote: Compressing objects: 100% (10/10), done.
         remote: Total 22 (delta 8), reused 22 (delta 8)
Receiving objects: 100% (22/22), 2.64 KiB, done.
Resolving deltas: 100% (8/8), done.
```

By convention, Julia repository names end with `.jl` (the additional `.git` indicates a “bare” git repository), which keeps them from colliding with repositories for other languages, and also makes Julia packages easy to find in search engines. When packages are installed in your `.julia/v0.6` directory, however, the extension is redundant so we leave it off.

If unregistered packages contain a `REQUIRE` file at the top of their source tree, that file will be used to determine which registered packages the unregistered package depends on, and they will automatically be installed. Unregistered packages participate in the same version resolution logic as registered packages, so installed package versions will be adjusted as necessary to satisfy the requirements of both registered and unregistered packages.

## 35.5 Updating Packages

When package developers publish new registered versions of packages that you’re using, you will, of course, want the new shiny versions. To get the latest and greatest versions of all your packages, just do `Pkg.update()`:

```
julia> Pkg.update()
INFO: Updating METADATA...
INFO: Computing changes...
```

<sup>1</sup>The official set of packages is at <https://github.com/JuliaLang/METADATA.jl>, but individuals and organizations can easily use a different metadata repository. This allows control which packages are available for automatic installation. One can allow only audited and approved package versions, and make private packages or forks available. See [Custom METADATA Repository](#) for details.

The first step of updating packages is to pull new changes to `~/.julia/v0.6/METADATA` and see if any new registered package versions have been published. After this, `Pkg.update()` attempts to update packages that are checked out on a branch and not dirty (i.e. no changes have been made to files tracked by git) by pulling changes from the package's upstream repository. Upstream changes will only be applied if no merging or rebasing is necessary – i.e. if the branch can be “[fast-forwarded](#)”. If the branch cannot be fast-forwarded, it is assumed that you're working on it and will update the repository yourself.

Finally, the update process recomputes an optimal set of package versions to have installed to satisfy your top-level requirements and the requirements of “fixed” packages. A package is considered fixed if it is one of the following:

1. Unregistered: the package is not in `METADATA` – you installed it with `Pkg.clone()`.
2. Checked out: the package repo is on a development branch.
3. Dirty: changes have been made to files in the repo.

If any of these are the case, the package manager cannot freely change the installed version of the package, so its requirements must be satisfied by whatever other package versions it picks. The combination of top-level requirements in `~/.julia/v0.6/REQUIRE` and the requirement of fixed packages are used to determine what should be installed.

You can also update only a subset of the installed packages, by providing arguments to the `Pkg.update` function. In that case, only the packages provided as arguments and their dependencies will be updated:

```
604 julia> Pkg.update("Example")
```

## CHAPTER 35. PACKAGES

```
INFO: Updating METADATA...
INFO: Computing changes...
INFO: Upgrading Example: v0.4.0 => 0.4.1
```

This partial update process still computes the new set of package versions according to top-level requirements and “fixed” packages, but it additionally considers all other packages except those explicitly provided, and their dependencies, as fixed.

## 35.6 Checkout, Pin and Free

You may want to use the `master` version of a package rather than one of its registered versions. There might be fixes or functionality on `master` that you need that aren’t yet published in any registered versions, or you may be a developer of the package and need to make changes on `master` or some other development branch. In such cases, you can do `Pkg.checkout(pkg)` to checkout the `master` branch of `pkg` or `Pkg.checkout(pkg, branch)` to checkout some other branch:

```
julia> Pkg.add("Distributions")
INFO: Installing Distributions v0.2.9
INFO: Installing NumericExtensions v0.2.17
INFO: Installing Stats v0.2.7
INFO: REQUIRE updated.
```

```
julia> Pkg.status()
```

Required packages:

- Distributions	0.2.9
-----------------	-------

Additional packages:

- NumericExtensions	0.2.17
---------------------	--------

- Stats	0.2.7
---------	-------

```
julia> Pkg.checkout("Distributions")
INFO: Checking out Distributions master...
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
- Distributions           0.2.9+          master
Additional packages:
- NumericExtensions       0.2.17
- Stats                   0.2.7
```

Immediately after installing `Distributions` with `Pkg.add()` it is on the current most recent registered version – `0.2.9` at the time of writing this. Then after running `Pkg.checkout("Distributions")`, you can see from the output of `Pkg.status()` that `Distributions` is on an unregistered version greater than `0.2.9`, indicated by the “pseudo-version” number `0.2.9+`.

When you checkout an unregistered version of a package, the copy of the `REQUIRE` file in the package repo takes precedence over any requirements registered in `METADATA`, so it is important that developers keep this file accurate and up-to-date, reflecting the actual requirements of the current version of the package. If the `REQUIRE` file in the package repo is incorrect or missing, dependencies may be removed when the package is checked out. This file is also used to populate newly published versions of the package if you use the API that `Pkg` provides for this (described below).

When you decide that you no longer want to have a package checked out on a branch, you can “free” it back to the control of the package manager with `Pkg.free(pkg)`:

```
julia> Pkg.free("Distributions")
```

606

INFO: Freeing Distributions...

CHAPTER 35. PACKAGES

INFO: No packages to install, update or remove.

```
julia> Pkg.status()
```

Required packages:

- Distributions	0.2.9
-----------------	-------

Additional packages:

- NumericExtensions	0.2.17
---------------------	--------

- Stats	0.2.7
---------	-------

After this, since the package is on a registered version and not on a branch, its version will be updated as new registered versions of the package are published.

If you want to pin a package at a specific version so that calling [Pkg.update\(\)](#) won't change the version the package is on, you can use the [Pkg.pin\(\)](#) function:

```
julia> Pkg.pin("Stats")
```

INFO: Creating Stats branch pinned.47c198b1.tmp

```
julia> Pkg.status()
```

Required packages:

- Distributions	0.2.9
-----------------	-------

Additional packages:

- NumericExtensions	0.2.17
---------------------	--------

- Stats	0.2.7
---------	-------

→ pinned.47c198b1.tmp	
-----------------------	--

After this, the **Stats** package will remain pinned at version **0.2.7** – or more specifically, at commit **47c198b1**, but since versions are permanently associated a given git hash, this is the same thing. [Pkg.pin\(\)](#) works by creating a

**85.6 - CHECKOUT, PIN AND FREE** If you want to pin the package at a specific commit you can use `Pkg.pin()`. It will pin the package at that commit and prevent the package manager from checking that branch out. By default, it pins a package at the current commit, but you can choose a different version by passing a second argument:

```
julia> Pkg.pin("Stats", v"0.2.5")
INFO: Creating Stats branch pinned.1fd0983b.tmp
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
- Distributions           0.2.9
Additional packages:
- NumericExtensions       0.2.17
- Stats                   0.2.5
→ pinned.1fd0983b.tmp
```

Now the `Stats` package is pinned at commit `1fd0983b`, which corresponds to version `0.2.5`. When you decide to "unpin" a package and let the package manager update it again, you can use `Pkg.free()` like you would to move off of any branch:

```
julia> Pkg.free("Stats")
INFO: Freeing Stats...
INFO: No packages to install, update or remove.
```

```
julia> Pkg.status()
Required packages:
- Distributions           0.2.9
Additional packages:
- NumericExtensions       0.2.17
- Stats                   0.2.7
```

After this, the `Stats` package is managed by the `Pkg` manager<sup>2</sup>, and future calls to `Pkg.update()` will upgrade it to newer versions when they are published. The throw-away `pinned.1fd0983b.tmp` branch remains in your local `Stats` repo, but since git branches are extremely lightweight, this doesn't really matter; if you feel like cleaning them up, you can go into the repo and delete those branches <sup>2</sup>.

## 35.7 Custom METADATA Repository

By default, Julia assumes you will be using the [official METADATA.jl](#) repository for downloading and installing packages. You can also provide a different metadata repository location. A common approach is to keep your `metadata-v2` branch up to date with the Julia official branch and add another branch with your custom packages. You can initialize your local metadata repository using that custom location and branch and then periodically rebase your custom branch with the official `metadata-v2` branch. In order to use a custom repository and branch, issue the following command:

```
julia> Pkg.init("https://me.example.com/METADATA.jl.git",
    ↵ "branch")
```

The branch argument is optional and defaults to `metadata-v2`. Once initialized, a file named `META_BRANCH` in your `~/.julia/vX.Y/` path will track the branch that your METADATA repository was initialized with. If you want to change branches, you will need to either modify the `META_BRANCH` file directly (be careful!) or remove the `vX.Y` directory and re-initialize your METADATA repository using the `Pkg.init` command.

---

<sup>2</sup>Packages that aren't on branches will also be marked as dirty if you make changes in the repo, but that's a less common thing to do.

## Chapter 36

# Package Development

Julia's package manager is designed so that when you have a package installed, you are already in a position to look at its source code and full development history. You are also able to make changes to packages, commit them using git, and easily contribute fixes and enhancements upstream. Similarly, the system is designed so that if you want to create a new package, the simplest way to do so is within the infrastructure provided by the package manager.

### 36.1 Initial Setup

Since packages are git repositories, before doing any package development you should setup the following standard global git configuration settings:

```
$ git config --global user.name "FULL NAME"  
$ git config --global user.email "EMAIL"
```

where `FULL NAME` is your actual full name (spaces are allowed between the double quotes) and `EMAIL` is your actual email address. Although it isn't necessary to use [GitHub](#) to create or publish Julia packages, most Julia packages as of writing this are hosted on GitHub and the package manager knows how to format origin URLs correctly and otherwise work with the service smoothly.

We recommend that you create a `free` `CHAPTER36 PACKAGE DEVELOPMENT`

```
$ git config --global github.user "USERNAME"
```

where `USERNAME` is your actual GitHub user name. Once you do this, the package manager knows your GitHub user name and can configure things accordingly. You should also [upload](#) your public SSH key to GitHub and set up an [SSH agent](#) on your development machine so that you can push changes with minimal hassle. In the future, we will make this system extensible and support other common git hosting options like [BitBucket](#) and allow developers to choose their favorite. Since the package development functions has been moved to the [PkgDev](#) package, you need to run `Pkg.add("PkgDev"); import PkgDev` to access the functions starting with `PkgDev.` in the document below.

## 36.2 Making changes to an existing package

### Documentation changes

If you want to improve the online documentation of a package, the easiest approach (at least for small changes) is to use GitHub's online editing functionality. First, navigate to the repository's GitHub "home page," find the file (e.g., `README.md`) within the repository's folder structure, and click on it. You'll see the contents displayed, along with a small "pencil" icon in the upper right hand corner. Clicking that icon opens the file in edit mode. Make your changes, write a brief summary describing the changes you want to make (this is your commit message), and then hit "Propose file change." Your changes will be submitted for consideration by the package owner(s) and collaborators.

For larger documentation changes – and especially ones that you expect to have to update in response to feedback – you might find it easier to use the procedure for code changes described below.

### Executive summary

Here we assume you've already set up git on your local machine and have a GitHub account (see above). Let's imagine you're fixing a bug in the Images package:

```
Pkg.checkout("Images")          # check out the master branch
<here, make sure your bug is still a bug and hasn't been fixed
    already>
cd(Pkg.dir("Images"))
;git checkout -b myfixes        # create a branch for your
    changes
<edit code>                  # be sure to add a test for your
    bug
Pkg.test("Images")            # make sure everything works now
;git commit -a -m "Fix foo by calling bar"  # write a descriptive
    message
using PkgDev
PkgDev.submit("Images")
```

The last line will present you with a link to submit a pull request to incorporate your changes.

### Detailed description

If you want to fix a bug or add new functionality, you want to be able to test your changes before you submit them for consideration. You also need to have an easy way to update your proposal in response to the package owner's feedback. Consequently, in this case the strategy is to work locally on your own machine; once you are satisfied with your changes, you submit them for consideration. This process is called a pull request because you are asking to "pull" your changes into the project's main repository. Because the online

repository can't see the code on your machine, you can push changes to a publicly-visible location, your own online fork of the package (hosted on your own personal GitHub account).

Let's assume you already have the `Foo` package installed. In the description below, anything starting with `Pkg.` or `PkgDev.` is meant to be typed at the Julia prompt; anything starting with `git` is meant to be typed in [julia's shell mode](#) (or using the shell that comes with your operating system). Within Julia, you can combine these two modes:

```
julia> cd(Pkg.dir("Foo"))          # go to Foo's folder  
shell> git command arguments...    # command will apply to Foo
```

Now suppose you're ready to make some changes to `Foo`. While there are several possible approaches, here is one that is widely used:

From the Julia prompt, type `Pkg.checkout("Foo")`. This ensures you're running the latest code (the `master` branch), rather than just whatever "official release" version you have installed. (If you're planning to fix a bug, at this point it's a good idea to check again whether the bug has already been fixed by someone else. If it has, you can request that a new official release be tagged so that the fix gets distributed to the rest of the community.) If you receive an error `Foo is dirty, bailing`, see [Dirty packages](#) below.

Create a branch for your changes: navigate to the package folder (the one that Julia reports from `Pkg.dir("Foo")`) and (in shell mode) create a new branch using `git checkout -b <newbranch>`, where `<newbranch>` might be some descriptive name (e.g., `fixbar`). By creating a branch, you ensure that you can easily go back and forth between

(see <https://git-scm.com/book/en/v2/Git-Branching-Banches-in-a-Nutshell>).

If you forget to do this step until after you've already made some changes, don't worry: see [more detail about branching](#) below.

Make your changes. Whether it's fixing a bug or adding new functionality, in most cases your change should include updates to both the `src/` and `test/` folders. If you're fixing a bug, add your minimal example demonstrating the bug (on the current code) to the test suite; by contributing a test for the bug, you ensure that the bug won't accidentally reappear at some later time due to other changes. If you're adding new functionality, creating tests demonstrates to the package owner that you've made sure your code works as intended.

Run the package's tests and make sure they pass. There are several ways to run the tests:

- From Julia, run `Pkg.test("Foo")`: this will run your tests in a separate (new) `julia` process.
- From Julia, `include("runtests.jl")` from the package's `test/` folder (it's possible the file has a different name, look for one that runs all the tests): this allows you to run the tests repeatedly in the same session without reloading all the package code; for packages that take a while to load, this can be much faster. With this approach, you do have to do some extra work to make [changes in the package code](#).
- From the shell, run `julia ./test/runtests.jl` from within the package's `src/` folder.

Commit your changes: see <https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository>.

614 Submit your changes: From the Julia prompt, type `PACKAGE_NAME(PUBLISH)`.

This will push your changes to your GitHub fork, creating it if it doesn't already exist. (If you encounter an error, [make sure you've set up your SSH keys](#).) Julia will then give you a hyperlink; open that link, edit the message, and then click "submit." At that point, the package owner will be notified of your changes and may initiate discussion. (If you are comfortable with git, you can also do these steps manually from the shell.)

The package owner may suggest additional improvements. To respond to those suggestions, you can easily update the pull request (this only works for changes that have not already been merged; for merged pull requests, make new changes by starting a new branch):

- If you've changed branches in the meantime, make sure you go back to the same branch with `git checkout fixbar` (from shell mode) or `Pkg.checkout("Foo", "fixbar")` (from the Julia prompt).
- As above, make your changes, run the tests, and commit your changes.
- From the shell, type `git push`. This will add your new commit(s) to the same pull request; you should see them appear automatically on the page holding the discussion of your pull request.

One potential type of change the owner may request is that you squash your commits. See [Squashing](#) below.

## Dirty packages

If you can't change branches because the package manager complains that your package is dirty, it means you have some changes that have not been committed. From the shell, use `git diff` to see what these changes are; you can either discard them (`git checkout changedfile.jl`) or commit them before switching branches. If you can't easily resolve the problems manually,

26.2. MAKING CHANGES TO AN EXISTING PACKAGE

615  
Delete and reinstall a fresh copy with `Pkg.add("Foo")`. Naturally, this deletes any changes you've made.

## Making a branch post hoc

Especially for newcomers to git, one often forgets to create a new branch until after some changes have already been made. If you haven't yet staged or committed your changes, you can create a new branch with `git checkout -b <newbranch>` just as usual – git will kindly show you that some files have been modified and create the new branch for you. Your changes have not yet been committed to this new branch, so the normal work rules still apply.

However, if you've already made a commit to `master` but wish to go back to the official `master` (called `origin/master`), use the following procedure:

Create a new branch. This branch will hold your changes.

Make sure everything is committed to this branch.

`git checkout master`. If this fails, do not proceed further until you have resolved the problems, or you may lose your changes.

Reset `master` (your current branch) back to an earlier state with `git reset --hard origin/master` (see <https://git-scm.com/blog/2011/07/11/reset.html>).

This requires a bit more familiarity with git, so it's much better to get in the habit of creating a branch at the outset.

## Squashing and rebasing

Depending on the tastes of the package owner (s)he may ask you to "squash" your commits. This is especially likely if your change is quite simple but your commit history looks like this:

WIP: add new 1-line whizbang function (currently breaks package)

Finish whizbang function

Fix typo in variable name

Oops, don't forget to supply default argument

Split into two 1-line functions

Rats, forgot to export the second function

...

This gets into the territory of more advanced git usage, and you're encouraged to do some reading (<https://git-scm.com/book/en/v2/Git-Branching-Rebasing>). However, a brief summary of the procedure is as follows:

To protect yourself from error, start from your **fixbar** branch and create a new branch with `git checkout -b fixbar_backup`. Since you started from **fixbar**, this will be a copy. Now go back to the one you intend to modify with `git checkout fixbar`.

From the shell, type `git rebase -i origin/master`.

To combine commits, change **pick** to **squash** (for additional options, consult other sources). Save the file and close the editor window.

Edit the combined commit message.

If the rebase goes badly, you can go back to the beginning to try again like this:

```
git checkout fixbar  
git reset --hard fixbar_backup
```

Now let's assume you've rebased successfully. Since your **fixbar** repository has now diverged from the one in your GitHub fork, you're going to have to do a force push:

36.3 To ~~CREATE A NEW PACKAGE~~ ~~your GitHub fork, create a "handle" for it with~~ 617

```
git remote add myfork https://github.com/myaccount/Foo.jl.git,
```

where the URL comes from the "clone URL" on your GitHub fork's page.

Force-push to your fork with `git push myfork +fixbar`. The `+` indicates that this should replace the `fixbar` branch found at `myfork`.

### 36.3 Creating a new Package

`REQUIRE` speaks for itself

You should have a `REQUIRE` file in your package repository, with a bare minimum directive of what Julia version you expect your users to be running for the package to work. Putting a floor on what Julia version your package supports is done by simply adding `julia 0.x` in this file. While this line is partly informational, it also has the consequence of whether `Pkg.update()` will update code found in `.julia` version directories. It will not update code found in version directories beneath the floor of what's specified in your `REQUIRE`.

As the development version `0.y` matures, you may find yourself using it more frequently, and wanting your package to support it. Be warned, the development branch of Julia is the land of breakage, and you can expect things to break. When you go about fixing whatever broke your package in the development `0.y` branch, you will likely find that you just broke your package on the stable version.

There is a mechanism found in the `Compat` package that will enable you to support both the stable version and breaking changes found in the development version. Should you decide to use this solution, you will need to add `Compat` to your `REQUIRE` file. In this case, you will still have `julia 0.x` in your `REQUIRE`. The `x` is the floor version of what your package supports.

You might also have no interest in supporting the development version of Julia.

Just as you can add a floor to the version, you can add a ceiling. In this case, you would put `julia 0.x 0.y-` in your REQUIRE file. The `-` at the end of the version number means pre-release versions of that specific version from the very first commit. By setting it as the ceiling, you mean the code supports everything up to but not including the ceiling version.

Another scenario is that you are writing the bulk of the code for your package with Julia `0.y` and do not want to support the current stable version of Julia. If you choose to do this, simply add `julia 0.y-` to your REQUIRE. Just remember to change the `julia 0.y-` to `julia 0.y` in your REQUIRE file once `0.y` is officially released. If you don't edit the dash cruft you are suggesting that you support both the development and stable versions of the same version number! That would be madness. See the [Requirements Specification](#) for the full format of REQUIRE.

Lastly, in many cases you may need extra packages for testing. Additional packages which are only required for tests should be specified in the test/REQUIRE file. This REQUIRE file has the same specification as the standard REQUIRE file.

## Guidelines for naming a package

Package names should be sensible to most Julia users, even to those who are not domain experts. When you submit your package to METADATA, you can expect a little back and forth about the package name with collaborators, especially if it's ambiguous or can be confused with something other than what it is. During this bike-shedding, it's not uncommon to get a range of different name suggestions. These are only suggestions though, with the intent being to keep a tidy namespace in the curated METADATA repository. Since this repository belongs to the entire community, there will likely be a few collaborators who care about your package name. Here are some guidelines

1. Avoid jargon. In particular, avoid acronyms unless there is minimal possibility of confusion.
  - It's ok to say **USA** if you're talking about the USA.
  - It's not ok to say **PMA**, even if you're talking about positive mental attitude.
2. Avoid using **Julia** in your package name.
  - It is usually clear from context and to your users that the package is a Julia package.
  - Having Julia in the name can imply that the package is connected to, or endorsed by, contributors to the Julia language itself.
3. Packages that provide most of their functionality in association with a new type should have pluralized names.
  - **DataFrames** provides the **DataFrame** type.
  - **BloomFilters** provides the **BloomFilter** type.
  - In contrast, **JuliaParser** provides no new type, but instead new functionality in the **JuliaParser.parse()** function.
4. Err on the side of clarity, even if clarity seems long-winded to you.
  - **RandomMatrices** is a less ambiguous name than **RndMat** or **RMT**, even though the latter are shorter.
5. A less systematic name may suit a package that implements one of several possible approaches to its domain.
  - Julia does not have a single comprehensive plotting package. Instead, **Gadfly**, **PyPlot**, **Winston** and other packages each implement a unique approach based on a particular design philosophy.

620 – In contrast, **SortingAlgorithm** uses many well-established sorting algorithms.

6. Packages that wrap external libraries or programs should be named after those libraries or programs.
  - **CPLEX.jl** wraps the CPLEX library, which can be identified easily in a web search.
  - **MATLAB.jl** provides an interface to call the MATLAB engine from within Julia.

## Generating the package

Suppose you want to create a new Julia package called **FooBar**. To get started, do `PkgDev.generate(pkg, license)` where `pkg` is the new package name and `license` is the name of a license that the package generator knows about:

```
julia> PkgDev.generate("FooBar", "MIT")
INFO: Initializing FooBar repo: /Users/someone/.julia/v0.6/FooBar
INFO: Origin: git://github.com/someone/FooBar.jl.git
INFO: Generating LICENSE.md
INFO: Generating README.md
INFO: Generating src/FooBar.jl
INFO: Generating test/runtests.jl
INFO: Generating REQUIRE
INFO: Generating .travis.yml
INFO: Generating appveyor.yml
INFO: Generating .gitignore
INFO: Committing FooBar generated files
```

This creates the directory `~/.julia/v0.6/FooBar`, initializes it as a git repository, generates a bunch of files that all packages should have, and commits

```
$ cd ~/.julia/v0.6/FooBar && git show --stat

commit 84b8e266dae6de30ab9703150b3bf771ec7b6285
Author: Some One <some.one@example.com>
Date:   Wed Oct 16 17:57:58 2013 -0400

  FooBar.jl generated files.

    license: MIT
    authors: Some One
    years:   2013
    user:    someone

Julia Version 0.3.0-prerelease+3217 [5fcfb13*]

.gitignore      |  2 ++
.travis.yml    | 13 ++++++
LICENSE.md     | 22 ++++++
README.md      |  3 ++
REQUIRE        |  1 +
appveyor.yml   | 34 ++++++
src/FooBar.jl   |  5 +++
test/runtests.jl |  5 +++
8 files changed, 85 insertions(+)
```

At the moment, the package manager knows about the MIT "Expat" License, indicated by "MIT", the Simplified BSD License, indicated by "BSD", and version 2.0 of the Apache Software License, indicated by "ASL". If you want to use a different license, you can ask us to add it to the package generator, or just pick one of these three and then modify the `~/.julia/v0.6/PACK-`

If you created a GitHub account and configured git to know about it, `PkgDev.generate()` will set an appropriate origin URL for you. It will also automatically generate a `.travis.yml` file for using the [Travis](#) automated testing service, and an `appveyor.yml` file for using [AppVeyor](#). You will have to enable testing on the Travis and AppVeyor websites for your package repository, but once you've done that, it will already have working tests. Of course, all the default testing does is verify that using `FooBar` in Julia works.

## Loading Static Non-Julia Files

If your package code needs to load static files which are not Julia code, e.g. an external library or data files, and are located within the package directory, use the `@__DIR__` macro to determine the directory of the current source file. For example if `FooBar/src/FooBar.jl` needs to load `FooBar/data/foo.csv`, use the following code:

```
| datapath = joinpath(@__DIR__, "..", "data")
| foo = readdlm(joinpath(datapath, "foo.csv"), ',', ',')
```

## Making Your Package Available

Once you've made some commits and you're happy with how `FooBar` is working, you may want to get some other people to try it out. First you'll need to create the remote repository and push your code to it; we don't yet automatically do this for you, but we will in the future and it's not too hard to figure out<sup>3</sup>. Once you've done this, letting people try out your code is as simple as sending them the URL of the published repo – in this case:

```
| git://github.com/someone/FooBar.jl.git
```

Before you create your package at `NEW PACKAGE`,<sup>3</sup> Hub user name and the name of your package, but you get the idea. People you send this URL to can use `Pkg.clone()` to install the package and try it out:

```
julia> Pkg.clone("git://github.com/someone/FooBar.jl.git")
INFO: Cloning FooBar from git@github.com:someone/FooBar.jl.git
```

## Tagging and Publishing Your Package

### Tip

If you are hosting your package on GitHub, you can use the [attobot integration](#) to handle package registration, tagging and publishing.

Once you've decided that `FooBar` is ready to be registered as an official package, you can add it to your local copy of `METADATA` using `PkgDev.register()`:

```
julia> PkgDev.register("FooBar")
INFO: Registering FooBar at git://github.com/someone/FooBar.jl.git
INFO: Committing METADATA for FooBar
```

This creates a commit in the `~/.julia/v0.6/METADATA` repo:

```
$ cd ~/.julia/v0.6/METADATA && git show

commit 9f71f4becb05cadacb983c54a72eed744e5c019d
Author: Some One <some.one@example.com>
Date:   Wed Oct 16 18:46:02 2013 -0400
```

### Register FooBar

<sup>3</sup>Installing and using GitHub's "hub" tool is highly recommended. It allows you to do things like run `hub create` in the package repo and have it automatically created via GitHub's API.

```
diff --git a/FooBar/url b/FooBar/url
new file mode 100644
index 000000..30e525e
--- /dev/null
+++ b/FooBar/url
@@ -0,0 +1 @@
+git://github.com/someone/FooBar.jl.git
```

This commit is only locally visible, however. To make it visible to the Julia community, you need to merge your local **METADATA** upstream into the official repo. The `PkgDev.publish()` command will fork the **METADATA** repository on GitHub, push your changes to your fork, and open a pull request:

```
julia> PkgDev.publish()
INFO: Validating METADATA
INFO: No new package versions to publish
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to someone
INFO: Pushing changes as branch pull-request/ef45f54b
INFO: To create a pull-request open:

    https://github.com/someone/METADATA.jl/compare/pull-
    ↵ request/ef45f54b
```

### Tip

If `PkgDev.publish()` fails with error:

```
|ERROR: key not found: "token"
```

then you may have encountered an issue from using the GitHub API on multiple systems. The solution is to delete the "Julia Package Manager" personal access token [from your Github account](#) and try again.

36.3 Other Creating A NEW PACKAGE to circumvent `PkgDev.publish()` by 625  
creating a pull request on GitHub. See: Publishing METADATA manually below.

Once the package URL for `FooBar` is registered in the official METADATA repo, people know where to clone the package from, but there still aren't any registered versions available. You can tag and register it with the `PkgDev.tag()` command:

```
julia> PkgDev.tag("FooBar")
INFO: Tagging FooBar v0.0.1
INFO: Committing METADATA for FooBar
```

This tags `v0.0.1` in the `FooBar` repo:

```
$ cd ~/.julia/v0.6/FooBar && git tag
v0.0.1
```

It also creates a new version entry in your local METADATA repo for `FooBar`:

```
$ cd ~/.julia/v0.6/FooBar && git show
commit de77ee4dc0689b12c5e8b574aef7f70e8b311b0e
Author: Some One <some.one@example.com>
Date:   Wed Oct 16 23:06:18 2013 -0400
```

Tag `FooBar v0.0.1`

```
diff --git a/FooBar/versions/0.0.1/sha1 b/FooBar/versions/0.0.1/
sha1
new file mode 100644
index 000000..c1cb1c1
--- /dev/null
+++ b/FooBar/versions/0.0.1/sha1
@@ -0,0 +1 @@
@@
```

The `PkgDev.tag()` command takes an optional second argument that is either an explicit version number object like `v"0.0.1"` or one of the symbols `:patch`, `:minor` or `:major`. These increment the patch, minor or major version number of your package intelligently.

Adding a tagged version of your package will expedite the official registration into `METADATA.jl` by collaborators. It is strongly recommended that you complete this process, regardless if your package is completely ready for an official release.

As a general rule, packages should be tagged `0.0.1` first. Since Julia itself hasn't achieved `1.0` status, it's best to be conservative in your package's tagged versions.

As with `PkgDev.register()`, these changes to `METADATA` aren't available to anyone else until they've been included upstream. Again, use the `PkgDev.publish()` command, which first makes sure that individual package repos have been tagged, pushes them if they haven't already been, and then opens a pull request to `METADATA`:

```
julia> PkgDev.publish()
INFO: Validating METADATA
INFO: Pushing FooBar permanent tags: v0.0.1
INFO: Submitting METADATA changes
INFO: Forking JuliaLang/METADATA.jl to someone
INFO: Pushing changes as branch pull-request/3ef4f5c4
INFO: To create a pull-request open:

    https://github.com/someone/METADATA.jl/compare/pull-
    → request/3ef4f5c4
```

If `PkgDev.publish()` fails you can follow these instructions to manually publish your package.

By "forking" the main METADATA repository, you can create a personal copy (of `METADATA.jl`) under your GitHub account. Once that copy exists, you can push your local changes to your copy (just like any other GitHub project).

1. Create a [fork of `METADATA.jl`](#).
2. Add your fork as a remote repository for the METADATA repository on your local computer (in the terminal where `USERNAME` is your github username):

```
| cd ~/.julia/v0.6/METADATA  
| git remote add USERNAME https://github.com/USERNAME/METADATA.jl  
| .git
```

3. Push your changes to your fork:

```
| git push USERNAME metadata-v2
```

4. If all of that works, then go back to the GitHub page for your fork, and click the "pull request" link.

## 36.4 Fixing Package Requirements

If you need to fix the registered requirements of an already-published package version, you can do so just by editing the metadata for that version, which will still have the same commit hash – the hash associated with a version is permanent:

```
$ cd ~/.julia/v0.6/METADATA/FooBar/versions/0.0.1 && cat requires  
julia 0.3-  
$ vi requires
```

628  
See the commit hash stays the same, the contents of the **REQUIRE** file will be checked out in the repo will not match the requirements in **METADATA** after such a change; this is unavoidable. When you fix the requirements in **METADATA** for a previous version of a package, however, you should also fix the **REQUIRE** file in the current version of the package.

## 36.5 Requirements Specification

The `~/ .julia/v0.6/REQUIRE` file, the **REQUIRE** file inside packages, and the **METADATA** package **requires** files use a simple line-based format to express the ranges of package versions which need to be installed. Package **REQUIRE** and **METADATA requires** files should also include the range of versions of **julia** the package is expected to work with. Additionally, packages can include a `test/REQUIRE` file to specify additional packages which are only required for testing.

Here's how these files are parsed and interpreted.

Everything after a `#` mark is stripped from each line as a comment.

If nothing but whitespace is left, the line is ignored.

If there are non-whitespace characters remaining, the line is a requirement and the is split on whitespace into words.

The simplest possible requirement is just the name of a package name on a line by itself:

### Distributions

This requirement is satisfied by any version of the **Distributions** package. The package name can be followed by zero or more version numbers in ascending order, indicating acceptable intervals of versions of that package.

**3.6.5. VERSION REQUIREMENTS SPECIFICATION** the next closes it, and the next opens a new interval, and so on; if an odd number of version numbers are given, then arbitrarily large versions will satisfy; if an even number of version numbers are given, the last one is an upper limit on acceptable version numbers.

For example, the line:

```
Distributions 0.1
```

is satisfied by any version of **Distributions** greater than or equal to **0.1.0**.

Suffixing a version with **-** allows any pre-release versions as well. For example:

```
Distributions 0.1-
```

is satisfied by pre-release versions such as **0.1-dev** or **0.1-rc1**, or by any version greater than or equal to **0.1.0**.

This requirement entry:

```
Distributions 0.1 0.2.5
```

is satisfied by versions from **0.1.0** up to, but not including **0.2.5**. If you want to indicate that any **0.1.x** version will do, you will want to write:

```
Distributions 0.1 0.2-
```

If you want to start accepting versions after **0.2.7**, you can write:

```
Distributions 0.1 0.2- 0.2.7
```

If a requirement line has leading words that begin with **@**, it is a system-dependent requirement. If your system matches these system conditionals, the requirement is included, if not, the requirement is ignored. For example:

```
@osx Homebrew
```

will require the **Homebrew** package only on systems where the operating system is OS X. The system conditions that are currently supported are (hierarchically):

– @linux

– @bsd

\* @osx

@windows

The @unix condition is satisfied on all UNIX systems, including Linux and BSD. Negated system conditionals are also supported by adding a ! after the leading @. Examples:

@!windows

@unix @!osx

The first condition applies to any system but Windows and the second condition applies to any UNIX system besides OS X.

Runtime checks for the current version of Julia can be made using the built-in VERSION variable, which is of type VersionNumber. Such code is occasionally necessary to keep track of new or deprecated functionality between various releases of Julia. Examples of runtime checks:

```
VERSION < v"0.3-" #exclude all pre-release versions of 0.3

v"0.2-" <= VERSION < v"0.3-" #get all 0.2 versions, including
    ↳ pre-releases, up to the above

v"0.2" <= VERSION < v"0.3-" #To get only stable 0.2 versions (Note
    ↳ v"0.2" == v"0.2.0")

VERSION >= v"0.2.1" #get at least version 0.2.1
```

See the section on [version number literals](#) for a more complete description.

# Chapter 37

## Profiling

The `Profile` module provides tools to help developers improve the performance of their code. When used, it takes measurements on running code, and produces output that helps you understand how much time is spent on individual line(s). The most common usage is to identify "bottlenecks" as targets for optimization.

`Profile` implements what is known as a "sampling" or [statistical profiler](#). It works by periodically taking a backtrace during the execution of any task. Each backtrace captures the currently-running function and line number, plus the complete chain of function calls that led to this line, and hence is a "snapshot" of the current state of execution.

If much of your run time is spent executing a particular line of code, this line will show up frequently in the set of all backtraces. In other words, the "cost" of a given line – or really, the cost of the sequence of function calls up to and including this line – is proportional to how often it appears in the set of all backtraces.

A sampling profiler does not provide complete line-by-line coverage, because the backtraces occur at intervals (by default, 1 ms on Unix systems and 10 ms on Windows, although the actual scheduling is subject to operat-

632 system load). Moreover, as discussed further below, because sampling profilers are collected at a sparse subset of all execution points, the data collected by a sampling profiler is subject to statistical noise.

Despite these limitations, sampling profilers have substantial strengths:

- You do not have to make any modifications to your code to take timing measurements (in contrast to the alternative [instrumenting profiler](#)).

- It can profile into Julia's core code and even (optionally) into C and Fortran libraries.

- By running "infrequently" there is very little performance overhead; while profiling, your code can run at nearly native speed.

For these reasons, it's recommended that you try using the built-in sampling profiler before considering any alternatives.

## 37.1 Basic usage

Let's work with a simple test case:

```
julia> function myfunc()  
        A = rand(200, 200, 400)  
        maximum(A)  
    end
```

It's a good idea to first run the code you intend to profile at least once (unless you want to profile Julia's JIT-compiler):

```
julia> myfunc() # run once to force compilation
```

```
julia> using Profile  
  
julia> @profile myfunc()
```

To see the profiling results, there is a [graphical browser](#) available, but here we'll use the text-based display that comes with the standard library:

```
julia> Profile.print()  
80 ./event.jl:73; (::Base.REPL.##1#2{Base.REPL.REPLBackend})()  
  80 ./REPL.jl:97; macro expansion  
    80 ./REPL.jl:66; eval_user_input(::Any, ::Base.REPL.REPLBackend)  
      80 ./boot.jl:235; eval(::Module, ::Any)  
        80 ./<missing>?:; anonymous  
          80 ./profile.jl:23; macro expansion
```

Each line of this display represents a particular spot (line number) in the code. Indentation is used to indicate the nested sequence of function calls, with more-indented lines being deeper in the sequence of calls. In each line, the first "field" is the number of backtraces (samples) taken at this line or in any functions executed by this line. The second field is the file name and line number and the third field is the function name. Note that the specific line numbers may change as Julia's code changes; if you want to follow along, it's best to run this example yourself.

In this example, we can see that the top level function called is in the file `event.jl`. This is the function that runs the REPL when you launch Julia. If you examine line 97 of `REPL.jl`, you'll see this is where the function `eval_user_input()` is called. This is the function that evaluates what you type at the REPL, and since we're working interactively these functions were invoked when

624 entered @profile myfunc(). The next line reflects backtraces taken inside the @profile macro.

The first line shows that 80 backtraces were taken at line 73 of event.jl, but it's not that this line was "expensive" on its own: the third line reveals that all 80 of these backtraces were actually triggered inside its call to eval\_user\_input, and so on. To find out which operations are actually taking the time, we need to look deeper in the call chain.

The first "important" line in this output is this one:

```
| 52 ./REPL[1]:2; myfunc()
```

REPL refers to the fact that we defined myfunc in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. The [1] shows that the function myfunc was the first expression evaluated in this REPL session. Line 2 of myfunc() contains the call to rand, and there were 52 (out of 80) backtraces that occurred at this line. Below that, you can see a call to dsfmt\_fill\_array\_close\_open! inside dSFMT.jl.

A little further down, you see:

```
| 28 ./REPL[1]:3; myfunc()
```

Line 3 of myfunc contains the call to maximum, and there were 28 (out of 80) backtraces taken here. Below that, you can see the specific places in base/reduce.jl that carry out the time-consuming operations in the maximum function for this type of input data.

Overall, we can tentatively conclude that generating the random numbers is approximately twice as expensive as finding the maximum element. We could increase our confidence in this result by collecting more samples:

```
| julia> @profile (for i = 1:100; myfunc(); end)
```

```
[....]  
3821 ./REPL[1]:2; myfunc()  
3511 ./random.jl:431; rand!(::MersenneTwister,  
→ ::Array{Float64,3}, ::Int64, ::Type...  
3511 ./dSFMT.jl:84;  
→ dsfmt_fill_array_close_open! (::Base.dSFMT.DSFMT_state,  
→ ::Ptr...  
310 ./random.jl:278; rand  
[....]  
2893 ./REPL[1]:3; myfunc()  
2893 ./reduce.jl:270; _mapreduce(::Base.#identity,  
→ ::Base.#scalarmax, ::IndexLinea...  
[....]
```

In general, if you have  $N$  samples collected at a line, you can expect an uncertainty on the order of  $\sqrt{N}$  (barring other sources of noise, like how busy the computer is with other tasks). The major exception to this rule is garbage collection, which runs infrequently but tends to be quite expensive. (Since Julia's garbage collector is written in C, such events can be detected using the `C=true` output mode described below, or by using [ProfileView.jl](#).)

This illustrates the default "tree" dump; an alternative is the "flat" dump, which accumulates counts independent of their nesting:

```
julia> Profile.print(format=:flat)  
Count File Line Function  
6714 ./<missing> -1 anonymous  
6714 ./REPL.jl 66 eval_user_input(::Any,  
→ ::Base.REPL.REPLBackend)  
6714 ./REPL.jl 97 macro expansion  
3821 ./REPL[1] 2 myfunc()
```

636 2893 ./REPL[1]            3 myfunc()	CHAPTER 37. PROFILING
6714 ./REPL[7]            1 macro expansion	
6714 ./boot.jl            235 eval(::Module, ::Any)	
3511 ./dSFMT.jl            84	
↳ dsfmt_fill_array_close_open! (::Base.dSFMT.DSFMT_s...)	
6714 ./event.jl            73	
↳ (::Base.REPL.##1#2{Base.REPL.REPLBackend})()	
6714 ./profile.jl            23 macro expansion	
3511 ./random.jl            431 rand! (::MersenneTwister,	
↳ ::Array{Float64,3}, ::In...	
310 ./random.jl            277 rand	
310 ./random.jl            278 rand	
310 ./random.jl            366 rand	
310 ./random.jl            369 rand	
2893 ./reduce.jl            270 _mapreduce (::Base.#identity,	
↳ ::Base.#scalarmax, :...	
5 ./reduce.jl            420 mapreduce_impl (::Base.#identity,	
↳ ::Base.#scalarma...	
253 ./reduce.jl            426 mapreduce_impl (::Base.#identity,	
↳ ::Base.#scalarma...	
2592 ./reduce.jl            428 mapreduce_impl (::Base.#identity,	
↳ ::Base.#scalarma...	
43 ./reduce.jl            429 mapreduce_impl (::Base.#identity,	
↳ ::Base.#scalarma...	

If your code has recursion, one potentially-confusing point is that a line in a "child" function can accumulate more counts than there are total backtraces. Consider the following function definitions:

```
dumbsum(n::Integer) = n == 1 ? 1 : 1 + dumbsum(n-1)
dumbsum3() = dumbsum(3)
```

If you ACCUMULATION AND CLEARING a backtrace was taken while it was executing `dumbsum(1)`, the backtrace would look like this:

```
dumbsum3
  dumbsum(3)
    dumbsum(2)
      dumbsum(1)
```

Consequently, this child function gets 3 counts, even though the parent only gets one. The "tree" representation makes this much clearer, and for this reason (among others) is probably the most useful way to view the results.

## 37.2 Accumulation and clearing

Results from `@profile` accumulate in a buffer; if you run multiple pieces of code under `@profile`, then `Profile.print()` will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with `Profile.clear()`.

## 37.3 Options for controlling the display of profile results

`Profile.print` has more options than we've described so far. Let's see the full declaration:

```
function print(io::IO = STDOUT, data = fetch(); kwargs...)
```

Let's first discuss the two positional arguments, and later the keyword arguments:

`io` – Allows you to save the results to a buffer, e.g. a file, but the default is to print to `STDOUT` (the console).

638 **data** – Contains the data you want to analyze. CHAPTER 27 PROFILING  
tained from `Profile.fetch()`, which pulls out the backtraces from a  
pre-allocated buffer. For example, if you want to profile the profiler, you  
could say:

```
data = copy(Profile.fetch())
Profile.clear()
@profile Profile.print(STDOUT, data) # Prints the previous
    ↳ results
Profile.print()                      # Prints results from
    ↳ Profile.print()
```

The keyword arguments can be any combination of:

**format** – Introduced above, determines whether backtraces are printed  
with (default, `:tree`) or without (`:flat`) indentation indicating tree struc-  
ture.

**C** – If `true`, backtraces from C and Fortran code are shown (normally they  
are excluded). Try running the introductory example with `Profile.print(C  
= true)`. This can be extremely helpful in deciding whether it's Julia code  
or C code that is causing a bottleneck; setting `C = true` also improves  
the interpretability of the nesting, at the cost of longer profile dumps.

**combine** – Some lines of code contain multiple operations; for example,  
`s += A[i]` contains both an array reference (`A[i]`) and a sum opera-  
tion. These correspond to different lines in the generated machine code,  
and hence there may be two or more different addresses captured dur-  
ing backtraces on this line. `combine = true` lumps them together, and  
is probably what you typically want, but you can generate an output sep-  
arately for each unique instruction pointer with `combine = false`.

**maxdepth** – Limits frames at a depth higher than `maxdepth` in the `:tree`  
format.

37.4 Configuration

The order in :flat format. :filefuncline (639 fault) sorts by the source line, whereas :count sorts in order of number of collected samples.

**noisefloor** – Limits frames that are below the heuristic noise floor of the sample (only applies to format :tree). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which  $n \leq \text{noisefloor} * \sqrt{N}$ , where  $n$  is the number of samples on this line, and  $N$  is the number of samples for the callee.

**mincount** – Limits frames with less than **mincount** occurrences.

File/function names are sometimes truncated (with ...), and indentation is truncated with a +n at the beginning, where n is the number of extra spaces that would have been inserted, had there been room. If you want a complete profile of deeply-nested code, often a good idea is to save to a file using a wide **displaysize** in an [IOContext](#):

```
open("/tmp/prof.txt", "w") do s
    Profile.print(IOContext(s, :displaysize => (24, 500)))
end
```

## 37.4 Configuration

[@profile](#) just accumulates backtraces, and the analysis happens when you call [Profile.print\(\)](#). For a long-running computation, it's entirely possible that the pre-allocated buffer for storing backtraces will be filled. If that happens, the backtraces stop but your computation continues. As a consequence, you may miss some important profiling data (you will get a warning when that happens).

You can obtain and configure the relevant parameters this way:

640

```
Profile.init() # returns the current settings
```

CHAPTER 37. PROFILING

```
Profile.init(n = 10^7, delay = 0.01)
```

`n` is the total number of instruction pointers you can store, with a default value of  $10^6$ . If your typical backtrace is 20 instruction pointers, then you can collect 50000 backtraces, which suggests a statistical uncertainty of less than 1%. This may be good enough for most applications.

Consequently, you are more likely to need to modify `delay`, expressed in seconds, which sets the amount of time that Julia gets between snapshots to perform the requested computations. A very long-running job might not need frequent backtraces. The default setting is `delay = 0.001`. Of course, you can decrease the delay as well as increase it; however, the overhead of profiling grows once the delay becomes similar to the amount of time needed to take a backtrace (~30 microseconds on the author's laptop).

# Chapter 38

## Memory allocation analysis

One of the most common techniques to improve performance is to reduce memory allocation. The total amount of allocation can be measured with `@time` and `@allocated`, and specific lines triggering allocation can often be inferred from profiling via the cost of garbage collection that these lines incur. However, sometimes it is more efficient to directly measure the amount of memory allocated by each line of code.

To measure allocation line-by-line, start Julia with the `--track-allocation=<setting>` command-line option, for which you can choose `none` (the default, do not measure allocation), `user` (measure memory allocation everywhere except Julia's core code), or `all` (measure memory allocation at each line of Julia code). Allocation gets measured for each line of compiled code. When you quit Julia, the cumulative results are written to text files with `.mem` appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The `Coverage` package contains some elementary analysis tools, for example to sort the lines in order of number of bytes allocated.

In interpreting the results, there are a few important details. Under the `user` setting, the first line of any function directly called from the REPL will exhibit allocation due to events that happen in the REPL code itself. More signifi-

642tly, JIT-compilation also CHAPTER 28. MEMORY ALLOCATION AND ANALYSIS compiler is written in Julia (and compilation usually requires memory allocation). The recommended procedure is to force compilation by executing all the commands you want to analyze, then call `Profile.clear_malloc_data()` to reset all allocation counters. Finally, execute the desired commands and quit Julia to trigger the generation of the `.mem` files.

# Chapter 39

## Stack Traces

The `StackTraces` module provides simple stack traces that are both human readable and easy to use programmatically.

### 39.1 Viewing a stack trace

The primary function used to obtain a stack trace is `stacktrace`:

```
julia> stacktrace()
4-element Array{StackFrame,1}:
 eval(::Module, ::Any) at boot.jl:236
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 macro expansion at REPL.jl:97 [inlined]
 (::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
```

Calling `stacktrace()` returns a vector of `StackFrame`s. For ease of use, the alias `StackTrace` can be used in place of `Vector{StackFrame}`. (Examples with `[ ... ]` indicate that output may vary depending on how the code is run.)

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
```

644  
5-element Array{StackFrame,1}:

```
example() at REPL[1]:1
eval(::Module, ::Any) at boot.jl:236
[...]
```

## CHAPTER 39. STACK TRACES

```
julia> @noinline child() = stacktrace()
child (generic function with 1 method)

julia> @noinline parent() = child()
parent (generic function with 1 method)

julia> grandparent() = parent()
grandparent (generic function with 1 method)

julia> grandparent()
7-element Array{StackFrame,1}:
 child() at REPL[3]:1
 parent() at REPL[4]:1
 grandparent() at REPL[5]:1
[...]
```

Note that when calling `stacktrace()` you'll typically see a frame with `eval(...)` at `boot.jl`. When calling `stacktrace()` from the REPL you'll also have a few extra frames in the stack from `REPL.jl`, usually looking something like this:

```
julia> example() = stacktrace()
example (generic function with 1 method)

julia> example()
5-element Array{StackFrame,1}:
 example() at REPL[1]:1
```

```
eval(::Module, ::Any) at boot.jl:236
eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
macro expansion at REPL.jl:97 [inlined]
(:::Base.REPL.##1#2{Base.REPL.REPLBackend}())() at event.jl:73
```

## 39.2 Extracting useful information

Each [StackFrame](#) contains the function name, file name, line number, lambda info, a flag indicating whether the frame has been inlined, a flag indicating whether it is a C function (by default C functions do not appear in the stack trace), and an integer representation of the pointer returned by [backtrace](#):

```
julia> top_frame = stacktrace()[1]
eval(::Module, ::Any) at boot.jl:236

julia> top_frame.func
:eval

julia> top_frame.file
Symbol("./boot.jl")

julia> top_frame.line
236

julia> top_frame.linfo
Nullable{Core.MethodInstance}(MethodInstance for eval(::Module,
→ ::Any))

julia> top_frame.inlined
false
```

```
646  
julia> top_frame.from_c  
false
```

## CHAPTER 39. STACK TRACES

```
julia> top_frame.pointer  
0x00007f390d152a59
```

This makes stack trace information available programmatically for logging, error handling, and more.

### 39.3 Error handling

While having easy access to information about the current state of the call-stack can be helpful in many places, the most obvious application is in error handling and debugging.

```
julia> @noinline bad_function() = undeclared_variable  
bad_function (generic function with 1 method)  
  
julia> @noinline example() = try  
  
        bad_function()  
  
        catch  
  
            stacktrace()  
  
        end  
example (generic function with 1 method)  
  
julia> example()  
5-element Array{StackFrame,1}:  
    example() at REPL[2]:4
```

```
| 39.3 ERROR HANDLING
| eval(::Module, ::Any) at boot.jl:236
| [...]
```

647

You may notice that in the example above the first stack frame points points at line 4, where `stacktrace` is called, rather than line 2, where `bad_function` is called, and `bad_function`'s frame is missing entirely. This is understandable, given that `stacktrace` is called from the context of the catch. While in this example it's fairly easy to find the actual source of the error, in complex cases tracking down the source of the error becomes nontrivial.

This can be remedied by calling `catch_stacktrace` instead of `stacktrace`. Instead of returning callstack information for the current context, `catch_stacktrace` returns stack information for the context of the most recent exception:

```
julia> @noinline bad_function() = undeclared_variable
bad_function (generic function with 1 method)

julia> @noinline example() = try
           bad_function()
       catch
           catch_stacktrace()
       end
example (generic function with 1 method)

julia> example()
6-element Array{StackFrame,1}:
 bad_function() at REPL[1]:1
```

```
|648 example() at REPL[2]:2  
| [...]
```

## CHAPTER 39. STACK TRACES

Notice that the stack trace now indicates the appropriate line number and the missing frame.

```
julia> @noinline child() = error("Whoops!")  
child (generic function with 1 method)  
  
julia> @noinline parent() = child()  
parent (generic function with 1 method)  
  
julia> @noinline function grandparent()  
  
    try  
  
        parent()  
  
    catch err  
  
        println("ERROR: ", err.msg)  
  
        catch_stacktrace()  
  
    end  
  
    end  
grandparent (generic function with 1 method)  
  
julia> grandparent()  
ERROR: Whoops!  
7-element Array{StackFrame,1}:
```

39.4 COMPARISON WITH *BACKTRACE*  
child() at REPL[1]:1  
parent() at REPL[2]:1  
grandparent() at REPL[3]:3  
[ ... ]

649

## 39.4 Comparison with **backtrace**

A call to `backtrace` returns a vector of `Ptr{Void}`, which may then be passed into `stacktrace` for translation:

```
julia> trace = backtrace()  
21-element Array{Ptr{Void},1}:  
 Ptr{Void} @0x00007f10049d5b2f  
 Ptr{Void} @0x00007f0ffeb4d29c  
 Ptr{Void} @0x00007f0ffeb4d2a9  
 Ptr{Void} @0x00007f1004993fe7  
 Ptr{Void} @0x00007f10049a92be  
 Ptr{Void} @0x00007f10049a823a  
 Ptr{Void} @0x00007f10049a9fb0  
 Ptr{Void} @0x00007f10049aa718  
 Ptr{Void} @0x00007f10049c0d5e  
 Ptr{Void} @0x00007f10049a3286  
 Ptr{Void} @0x00007f0ffe9ba3ba  
 Ptr{Void} @0x00007f0ffe9ba3d0  
 Ptr{Void} @0x00007f1004993fe7  
 Ptr{Void} @0x00007f0ded34583d  
 Ptr{Void} @0x00007f0ded345a87  
 Ptr{Void} @0x00007f1004993fe7  
 Ptr{Void} @0x00007f0ded34308f  
 Ptr{Void} @0x00007f0ded343320  
 Ptr{Void} @0x00007f1004993fe7  
 Ptr{Void} @0x00007f10049aeb67
```

```
julia> stacktrace(trace)
5-element Array{StackFrame,1}:
 backtrace() at error.jl:46
 eval(::Module, ::Any) at boot.jl:236
 eval_user_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66
 macro expansion at REPL.jl:97 [inlined]
 (:Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73
```

Notice that the vector returned by `backtrace` had 21 pointers, while the vector returned by `stacktrace` only has 5. This is because, by default, `stacktrace` removes any lower-level C functions from the stack. If you want to include stack frames from C calls, you can do it like this:

```
julia> stacktrace(trace, true)
27-element Array{StackFrame,1}:
 jl_backtrace_from_here at stackwalk.c:103
 backtrace() at error.jl:46
 backtrace() at sys.so:?
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
 do_call at interpreter.c:75
 eval at interpreter.c:215
 eval_body at interpreter.c:519
 jl_interpret_toplevel_thunk at interpreter.c:664
 jl_toplevel_eval_flex at toplevel.c:592
 jl_toplevel_eval_in at builtins.c:614
 eval(::Module, ::Any) at boot.jl:236
 eval(::Module, ::Any) at sys.so:?
 jl_call_method_internal at julia_internal.h:248 [inlined]
 jl_apply_generic at gf.c:2215
```

39.4 COMPARISON WITH *BACKTRACE* 651  
eval\_user\_input(::Any, ::Base.REPL.REPLBackend) at REPL.jl:66  
ip:0x7f1c707f1846  
jl\_call\_method\_internal at julia\_internal.h:248 [inlined]  
jl\_apply\_generic at gf.c:2215  
macro expansion at REPL.jl:97 [inlined]  
(::Base.REPL.##1#2{Base.REPL.REPLBackend})() at event.jl:73  
ip:0x7f1c707ea1ef  
jl\_call\_method\_internal at julia\_internal.h:248 [inlined]  
jl\_apply\_generic at gf.c:2215  
jl\_apply at julia.h:1411 [inlined]  
start\_task at task.c:261  
ip:0xfffffffffffffff

Individual pointers returned by `backtrace` can be translated into [StackFrame](#)s by passing them into `StackTraces.lookup`:

```
julia> pointer = backtrace()[1];  
  
julia> frame = StackTraces.lookup(pointer)  
1-element Array{StackFrame,1}:  
 jl_backtrace_from_here at stackwalk.c:103  
  
julia> println("The top frame is from $(frame[1].func)!")  
The top frame is from jl_backtrace_from_here!
```



# Chapter 40

## Performance Tips

In the following sections, we briefly go through a few techniques that can help make your Julia code run as fast as possible.

### 40.1 Avoid global variables

A global variable might have its value, and therefore its type, change at any point. This makes it difficult for the compiler to optimize code using global variables. Variables should be local, or passed as arguments to functions, whenever possible.

Any code that is performance critical or being benchmarked should be inside a function.

We find that global names are frequently constants, and declaring them as such greatly improves performance:

```
| const DEFAULT_VAL = 0
```

Uses of non-constant globals can be optimized by annotating their types at the point of use:

```
| global x  
| y = f(x::Int + 1)
```

Using functions is better style. It leads to code that is more readable, reusable, and easier to maintain. It also clarifies what steps are being done, and what their inputs and outputs are.

### Note

All code in the REPL is evaluated in global scope, so a variable defined and assigned at toplevel will be a global variable.

In the following REPL session:

```
| julia> x = 1.0
```

is equivalent to:

```
| julia> global x = 1.0
```

so all the performance issues discussed previously apply.

## 40.2 Measure performance with `@time` and pay attention to memory allocation

A useful tool for measuring performance is the `@time` macro. The following example illustrates good working style:

```
| julia> function f(n)
|           s = 0
|           for i = 1:n
|               s += i/2
|           end
```

```
end
f (generic function with 1 method)

julia> @time f(1)
0.012686 seconds (2.09 k allocations: 103.421 KiB)
0.5

julia> @time f(10^6)
0.021061 seconds (3.00 M allocations: 45.777 MiB, 11.69% gc
→ time)
2.5000025e11
```

On the first call (`@time f(1)`), `f` gets compiled. (If you've not yet used `@time` in this session, it will also compile functions needed for timing.) You should not take the results of this run seriously. For the second run, note that in addition to reporting the time, it also indicated that a large amount of memory was allocated.

Unexpected memory allocation is almost always a sign of some problem with your code, usually a problem with type-stability. Consequently, in addition to the allocation itself, it's very likely that the code generated for your function is far from optimal. Take such indications seriously and follow the advice below.

For more serious benchmarking, consider the [BenchmarkTools.jl](#) package which evaluates the function multiple times in order to reduce noise.

As a teaser, an improved version of this function allocates no memory (the allocation reported below is due to running the `@time` macro in global scope) and has an order of magnitude faster execution after the first call:

```
julia> @time f_improved(1)
```

656

0.007008 seconds (1.32 k allocations: 63.640 Kib)

0.5

```
julia> @time f_improved(10^6)
0.002997 seconds (6 allocations: 192 bytes)
2.5000025e11
```

Below you'll learn how to spot the problem with `f` and how to fix it.

In some situations, your function may need to allocate memory as part of its operation, and this can complicate the simple picture above. In such cases, consider using one of the [tools](#) below to diagnose problems, or write a version of your function that separates allocation from its algorithmic aspects (see [Pre-allocating outputs](#)).

### 40.3 Tools

Julia and its package ecosystem includes tools that may help you diagnose problems and improve the performance of your code:

[Profiling](#) allows you to measure the performance of your running code and identify lines that serve as bottlenecks. For complex projects, the [ProfileView](#) package can help you visualize your profiling results.

Unexpectedly-large memory allocations – as reported by [@time](#), [@allocated](#), or the profiler (through calls to the garbage-collection routines) – hint that there might be issues with your code. If you don't see another reason for the allocations, suspect a type problem. You can also start Julia with the `--track-allocation=user` option and examine the resulting `*.mem` files to see information about where those allocations occur. See [Memory allocation analysis](#).

## 40.4 @code\_warn\_type generates deprecation warnings that can help

helpful in finding expressions that result in type uncertainty. See [@code\\_warn\\_type](#) below.

The [Lint](#) package can also warn you of certain types of programming errors.

## 40.4 Avoid containers with abstract type parameters

When working with parameterized types, including arrays, it is best to avoid parameterizing with abstract types where possible.

Consider the following:

```
a = Real[]      # typeof(a) = Array{Real,1}
if (f = rand()) < .8
    push!(a, f)
end
```

Because `a` is an array of abstract type `Real`, it must be able to hold any `Real` value. Since `Real` objects can be of arbitrary size and structure, `a` must be represented as an array of pointers to individually allocated `Real` objects.

Because `f` will always be a `Float64`, we should instead, use:

```
a = Float64[] # typeof(a) = Array{Float64,1}
```

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under [Parametric Types](#).

## 40.5 Type declarations

In many languages with optional type declarations, adding declarations is the principal way to make code run faster. This is not the case in Julia. In

658, the compiler generally knows the types of function arguments, variables, and expressions. However, there are a few specific instances where declarations are helpful.

## Avoid fields with abstract type

Types can be declared without specifying the types of their fields:

```
julia> struct MyAmbiguousType  
|  
|     a  
|  
|  
end
```

This allows `a` to be of any type. This can often be useful, but it does have a downside: for objects of type `MyAmbiguousType`, the compiler will not be able to generate high-performance code. The reason is that the compiler uses the types of objects, not their values, to determine how to build code. Unfortunately, very little can be inferred about an object of type `MyAmbiguousType`:

```
julia> b = MyAmbiguousType("Hello")  
MyAmbiguousType("Hello")  
  
julia> c = MyAmbiguousType(17)  
MyAmbiguousType(17)  
  
julia> typeof(b)  
MyAmbiguousType  
  
julia> typeof(c)  
MyAmbiguousType
```

~~Both types have the same type~~, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field `a`, the fact that the memory representation of a `UInt8` differs from a `Float64` also means that the CPU needs to handle them using two different kinds of instructions. Since the required information is not available in the type, such decisions have to be made at run-time. This slows performance.

You can do better by declaring the type of `a`. Here, we are focused on the case where `a` might be any one of several types, in which case the natural solution is to use parameters. For example:

```
julia> mutable struct MyType{T<:AbstractFloat}

    a::T

end
```

This is a better choice than

```
julia> mutable struct MyStillAmbiguousType

    a::AbstractFloat

end
```

because the first version specifies the type of `a` from the type of the wrapper object. For example:

```
julia> m = MyType(3.2)
MyType{Float64}(3.2)

julia> t = MyStillAmbiguousType(3.2)
MyStillAmbiguousType(3.2)
```

```
julia> typeof(m)
MyType{Float64}

julia> typeof(t)
MyStillAmbiguousType
```

The type of field `a` can be readily determined from the type of `m`, but not from the type of `t`. Indeed, in `t` it's possible to change the type of field `a`:

```
julia> typeof(t.a)
Float64

julia> t.a = 4.5f0
4.5f0

julia> typeof(t.a)
Float32
```

In contrast, once `m` is constructed, the type of `m.a` cannot change:

```
julia> m.a = 4.5f0
4.5f0

julia> typeof(m.a)
Float64
```

The fact that the type of `m.a` is known from `m`'s type – coupled with the fact that its type cannot change mid-function – allows the compiler to generate highly-optimized code for objects like `m` but not for objects like `t`.

Of course, all of this is true only if we construct `m` with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
MyType{AbstractFloat}(3.2)
```

```
julia> typeof(m.a)
```

```
Float64
```

```
julia> m.a = 4.5f0
```

```
4.5f0
```

```
julia> typeof(m.a)
```

```
Float32
```

For all practical purposes, such objects behave identically to those of `MyStillAmbiguousType`.

It's quite instructive to compare the sheer amount code generated for a simple function

```
| func(m::MyType) = m.a+1
```

using

```
| code_llvm(func, Tuple{MyType{Float64}})  
| code_llvm(func, Tuple{MyType{AbstractFloat}})  
| code_llvm(func, Tuple{MyType})
```

For reasons of length the results are not shown here, but you may wish to try this yourself. Because the type is fully-specified in the first case, the compiler doesn't need to generate any code to resolve the type at run-time. This results in shorter and faster code.

Avoid fields with abstract containers

The same best practices also work for container types:

```
julia> mutable struct MySimpleContainer{A<:AbstractVector}
        a::A

    end

julia> mutable struct MyAmbiguousContainer{T}
        a::AbstractVector{T}

    end
```

For example:

```
julia> c = MySimpleContainer(1:3);

julia> typeof(c)
MySimpleContainer{UnitRange{Int64} }

julia> c = MySimpleContainer([1:3;]);

julia> typeof(c)
MySimpleContainer{Array{Int64,1} }

julia> b = MyAmbiguousContainer(1:3);

julia> typeof(b)
MyAmbiguousContainer{Int64}

julia> b = MyAmbiguousContainer([1:3;]);
```

```
MyAmbiguousContainer{Int64}
```

For `MySimpleContainer`, the object is fully-specified by its type and parameters, so the compiler can generate optimized functions. In most instances, this will probably suffice.

While the compiler can now do its job perfectly well, there are cases where you might wish that your code could do different things depending on the element type of `a`. Usually the best way to achieve this is to wrap your specific operation (here, `foo`) in a separate function:

```
julia> function sumfoo(c::MySimpleContainer)
           s = 0
           for x in c.a
               s += foo(x)
           end
           s
       end
sumfoo (generic function with 1 method)

julia> foo(x::Integer) = x
foo (generic function with 1 method)

julia> foo(x::AbstractFloat) = round(x)
foo (generic function with 2 methods)
```

This keeps things simple, while allowing the compiler to generate optimized code in all cases.

However, there are cases where you may need to declare different versions of the outer function for different element types of `a`. You could do it like this:

664 CHAPTER 40. PERFORMANCE TIPS

```
function myfun(c::MySimpleContainer{Vector{T}}) where T<:AbstractFloat
    ...
end

function myfun(c::MySimpleContainer{Vector{T}}) where T<:Integer
    ...
end
```

This works fine for `Vector{T}`, but we'd also have to write explicit versions for `UnitRange{T}` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `MyContainer`:

```
julia> mutable struct MyContainer{T, A<:AbstractVector}
           a::A
       end

julia> MyContainer(v::AbstractVector) = MyContainer{eltype(v),
           →   typeof(v)}(v)
MyContainer

julia> b = MyContainer(1:5);

julia> typeof(b)
MyContainer{Int64, UnitRange{Int64}}
```

Note the somewhat surprising fact that `T` doesn't appear in the declaration of field `a`, a point that we'll return to in a moment. With this approach, one can write functions such as:

```
julia> function myfunc(c::MyContainer{<:Integer, <:AbstractArray})
```

```
    end  
myfunc (generic function with 1 method)  
  
julia> function myfunc(c::MyContainer{<:AbstractFloat})  
  
    return c.a[1]+1  
  
    end  
myfunc (generic function with 2 methods)  
  
julia> function myfunc(c::MyContainer{T,Vector{T}}) where  
→   T<:Integer  
  
    return c.a[1]+2  
  
    end  
myfunc (generic function with 3 methods)
```

### Note

Because we can only define `MyContainer` for `A<:AbstractArray`, and any unspecified parameters are arbitrary, the first function above could have been written more succinctly as `function myfunc(c::MyContainer{<:Integer})`

```
julia> myfunc(MyContainer(1:3))  
2  
  
julia> myfunc(MyContainer(1.0:3))  
3.0
```

```
julia> myfunc(MyContainer([1:3;]))  
4
```

As you can see, with this approach it's possible to specialize on both the element type  $T$  and the array type  $A$ .

However, there's one remaining hole: we haven't enforced that  $A$  has element type  $T$ , so it's perfectly possible to construct an object like this:

```
julia> b = MyContainer{Int64, UnitRange{Float64}}(UnitRange(1.3,  
→ 5.0));  
  
julia> typeof(b)  
MyContainer{Int64,UnitRange{Float64}}
```

To prevent this, we can add an inner constructor:

```
julia> mutable struct MyBetterContainer{T<:Real,  
→ A<:AbstractVector}  
  
    a::A  
  
    MyBetterContainer{T,A}(v::AbstractVector{T}) where  
    {T,A} = new(v)  
  
end  
  
julia> MyBetterContainer(v::AbstractVector) =  
→ MyBetterContainer{eltype(v),typeof(v)}(v)  
MyBetterContainer  
  
julia> b = MyBetterContainer(UnitRange(1.3, 5.0));
```

```
julia> typeof(b)
MyBetterContainer{Float64,UnitRange{Float64}}

julia> b = MyBetterContainer{Int64,
    ↪ UnitRange{Float64}}(UnitRange(1.3, 5.0));
ERROR: MethodError: Cannot `convert` an object of type
    ↪ UnitRange{Float64} to an object of type
    ↪ MyBetterContainer{Int64,UnitRange{Float64}}
This may have arisen from a call to the constructor
    ↪ MyBetterContainer{Int64,UnitRange{Float64}}(...),
since type constructors fall back to convert methods.

Stacktrace:
 [1] MyBetterCon-
    ↪ tainer{Int64,UnitRange{Float64}}(::UnitRange{Float64}) at
    ↪ ./sysimg.jl:114
```

The inner constructor requires that the element type of A be T.

Annotate values taken from untyped locations

It is often convenient to work with data structures that may contain values of any type (arrays of type `Array{Any}`). But, if you're using one of these structures and happen to know the type of an element, it helps to share this knowledge with the compiler:

```
function foo(a::Array{Any,1})
    x = a[1]::Int32
    b = x+1
    ...
end
```

668, we happened to know that the first element of a [CHAPTER 4: PERFORMANCE TIPS](#) Making an annotation like this has the added benefit that it will raise a run-time error if the value is not of the expected type, potentially catching certain bugs earlier.

In the case that the type of `a[1]` is not known precisely, `x` can be declared via `x = convert(Int32, a[1]):Int32`. The use of the `convert` function allows `a[1]` to be any object convertible to an `Int32` (such as `UInt8`), thus increasing the genericity of the code by loosening the type requirement. Notice that `convert` itself needs a type annotation in this context in order to achieve type stability. This is because the compiler cannot deduce the type of the return value of a function, even `convert`, unless the types of all the function's arguments are known.

Type annotation will not enhance (and can actually hinder) performance if the type is constructed at run-time. This is because the compiler cannot use the annotation to specialize the subsequent code, and the type-check itself takes time. For example, in the code:

```
function nr(a, prec)
    ctype = prec == 32 ? Float32 : Float64
    b = Complex{ctype}(a)
    c = (b + 1.0f0)::Complex{ctype}
    abs(c)
end
```

the annotation of `c` harms performance. To write performant code involving types constructed at run-time, use the [function-barrier technique](#) discussed below, and ensure that the constructed type appears among the argument types of the kernel function so that the kernel operations are properly specialized by the compiler. For example, in the above snippet, as soon as `b` is constructed, it can be passed to another function `k`, the kernel. If, for exam-

**40.6 BREAK FUNCTIONS INT O**n Multiple Definitions

If a function has a type parameter, then a type annotation appearing in an assignment statement within `k` of the form:

```
| c = (b + 1.0f0)::Complex{T}
```

does not hinder performance (but does not help either) since the compiler can determine the type of `c` at the time `k` is compiled.

Declare types of keyword arguments

Keyword arguments can have declared types:

```
function with_keyword(x; name::Int = 1)
    ...
end
```

Functions are specialized on the types of keyword arguments, so these declarations will not affect performance of code inside the function. However, they will reduce the overhead of calls to the function that include keyword arguments.

Functions with keyword arguments have near-zero overhead for call sites that pass only positional arguments.

Passing dynamic lists of keyword arguments, as in `f(x; keywords...)`, can be slow and should be avoided in performance-sensitive code.

## 40.6 Break functions into multiple definitions

Writing a function as many small definitions allows the compiler to directly call the most applicable code, or even inline it.

Here is an example of a "compound function" that should really be written as multiple definitions:

```
function norm(A)
    if isa(A, Vector)
        return sqrt(real(dot(A,A)))
    elseif isa(A, Matrix)
        return maximum(svd(A)[2])
    else
        error("norm: invalid argument")
    end
end
```

This can be written more concisely and efficiently as:

```
| norm(x::Vector) = sqrt(real(dot(x,x)))
| norm(A::Matrix) = maximum(svd(A)[2])
```

## 40.7 Write “type-stable” functions

When possible, it helps to ensure that a function always returns a value of the same type. Consider the following definition:

```
| pos(x) = x < 0 ? 0 : x
```

Although this seems innocent enough, the problem is that `0` is an integer (of type `Int`) and `x` might be of any type. Thus, depending on the value of `x`, this function might return a value of either of two types. This behavior is allowed, and may be desirable in some cases. But it can easily be fixed as follows:

```
| pos(x) = x < 0 ? zero(x) : x
```

There is also a `one` function, and a more general `oftype(x, y)` function, which returns `y` converted to the type of `x`.

An analogous “type-stability” problem exists for variables used repeatedly within a function:

```
function foo()
    x = 1
    for i = 1:10
        x = x/bar()
    end
    return x
end
```

Local variable `x` starts as an integer, and after one loop iteration becomes a floating-point number (the result of `/` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

Initialize `x` with `x = 1.0`

Declare the type of `x`: `x::Float64 = 1`

Use an explicit conversion: `x = oneunit(T)`

Initialize with the first loop iteration, to `x = 1/bar()`, then loop `for i = 2:10`

## 40.9 Separate kernel functions (aka, function barriers)

Many functions follow a pattern of performing some set-up work, and then running many iterations to perform a core computation. Where possible, it is a good idea to put these core computations in separate functions. For example, the following contrived function returns an array of a randomly-chosen type:

672 julia> **function** strange\_twos(n)

CHAPTER 40. PERFORMANCE TIPS

```
a = Vector{rand(Bool) ? Int64 : Float64}(n)
```

```
for i = 1:n
```

```
    a[i] = 2
```

```
end
```

```
return a
```

```
end
```

```
strange_twos (generic function with 1 method)
```

julia> strange\_twos(3)

```
3-element Array{Float64,1}:
```

```
2.0
```

```
2.0
```

```
2.0
```

This should be written as:

julia> **function** fill\_twos!(a)

```
for i=eachindex(a)
```

```
    a[i] = 2
```

```
end
```

```
end
```

```
julia> function strange_twos(n)

    a = Array{rand(Bool) ? Int64 : Float64}(n)

    fill_twos!(a)

    return a

end

strange_twos (generic function with 1 method)

julia> strange_twos(3)
3-element Array{Float64,1}:
 2.0
 2.0
 2.0
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of `a` during the loop (since it is chosen randomly). Therefore the second version is generally faster since the inner loop can be recompiled as part of `fill_twos!` for different types of `a`.

The second form is also often better style and can lead to more code reuse.

This pattern is used in several places in the standard library. For example, see `hvcat_fill` in `abstractarray.jl`, or the `fill!` function, which we could have used instead of writing our own `fill_twos!`.

Functions like `strange_twos` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers,

## 40.10 Types with values-as-parameters

Let's say you want to create an  $N$ -dimensional array that has size 3 along each axis. Such arrays can be created like this:

```
julia> A = fill(5.0, (3, 3))
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

This approach works very well: the compiler can figure out that `A` is an `Array{Float64, 2}` because it knows the type of the fill value (`5.0::Float64`) and the dimensionality (`(3, 3)::NTuple{2, Int}`). This implies that the compiler can generate very efficient code for any future usage of `A` in the same function.

But now let's say you want to write a function that creates a  $3 \times 3 \times \dots$  array in arbitrary dimensions; you might be tempted to write a function

```
julia> function array3(fillval, N)
           fill(fillval, ntuple(d->3, N))
       end
array3 (generic function with 1 method)

julia> array3(5.0, 2)
3×3 Array{Float64,2}:
 5.0  5.0  5.0
```

```
5.0 5.0 5.0
```

This works, but (as you can verify for yourself using `@code_warntype array3(5.0, 2)`) the problem is that the output type cannot be inferred: the argument `N` is a value of type `Int`, and type-inference does not (and cannot) predict its value in advance. This means that code using the output of this function has to be conservative, checking the type on each access of `A`; such code will be very slow.

Now, one very good way to solve such problems is by using the [function-barrier technique](#). However, in some cases you might want to eliminate the type-instability altogether. In such cases, one approach is to pass the dimensionality as a parameter, for example through `Val{T}()` (see “[Value types](#)”):

```
julia> function array3(fillval, ::Val{N}) where N
           fill(fillval, ntuple(d->3, Val(N)))
end
array3 (generic function with 1 method)

julia> array3(5.0, Val(2))
3×3 Array{Float64,2}:
 5.0  5.0  5.0
 5.0  5.0  5.0
 5.0  5.0  5.0
```

Julia has a specialized version of `ntuple` that accepts a `Val{:,:Int}` instance as the second parameter; by passing `N` as a type-parameter, you make its “value” known to the compiler. Consequently, this version of `array3` allows the compiler to predict the return type.

ample, it would be of no help if you called `array3` from a function like this:

```
function call_array3(fillval, n)
    A = array3(fillval, Val(n))
end
```

Here, you've created the same problem all over again: the compiler can't guess what `n` is, so it doesn't know the type of `Val(n)`. Attempting to use `Val`, but doing so incorrectly, can easily make performance worse in many situations. (Only in situations where you're effectively combining `Val` with the function-barrier trick, to make the kernel function more efficient, should code like the above be used.)

An example of correct usage of `Val` would be:

```
function filter3(A::AbstractArray{T,N}) where {T,N}
    kernel = array3(1, Val(N))
    filter(A, kernel)
end
```

In this example, `N` is passed as a parameter, so its "value" is known to the compiler. Essentially, `Val(T)` works only when `T` is either hard-coded/literal (`Val(3)`) or already specified in the type-domain.

## 40.11 The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)

Once one learns to appreciate multiple dispatch, there's an understandable tendency to go crazy and try to use it for everything. For example, you might imagine using it to store information, e.g.

```
struct Car{Make,Model}
```

end

and then dispatch on objects like `Car{ :Honda, :Accord}(year, args...)`.

This might be worthwhile when the following are true:

You require CPU-intensive processing on each `Car`, and it becomes vastly more efficient if you know the `Make` and `Model` at compile time.

You have homogenous lists of the same type of `Car` to process, so that you can store them all in an `Array{Car{ :Honda, :Accord}, N}`.

When the latter holds, a function processing such a homogenous array can be productively specialized: Julia knows the type of each element in advance (all objects in the container have the same concrete type), so Julia can "look up" the correct method calls when the function is being compiled (obviating the need to check at run-time) and thereby emit efficient code for processing the whole list.

When these do not hold, then it's likely that you'll get no benefit; worse, the resulting "combinatorial explosion of types" will be counterproductive. If `items[i+1]` has a different type than `item[i]`, Julia has to look up the type at run-time, search for the appropriate method in method tables, decide (via type intersection) which one matches, determine whether it has been JIT-compiled yet (and do so if not), and then make the call. In essence, you're asking the full type-system and JIT-compilation machinery to basically execute the equivalent of a switch statement or dictionary lookup in your own code.

Some run-time benchmarks comparing (1) type dispatch, (2) dictionary lookup, and (3) a "switch" statement can be found [on the mailing list](#).

Perhaps even worse than the run-time impact is the compilation overhead. Julia will compile specialized functions for each different `Car{Make, Model}`; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom `get_year` function you might write yourself, to the generic `push!` function in the standard library) will have hundreds or thousands of variants compiled for it. Each of these increases the size of the cache of compiled code, the length of internal lists of methods, etc. Excess enthusiasm for values-as-parameters can easily waste enormous resources.

## 40.12 Access arrays in memory order, along columns

Multidimensional arrays in Julia are stored in column-major order. This means that arrays are stacked one column at a time. This can be verified using the `vec` function or the syntax `[:]` as shown below (notice that the array is ordered `[1 3 2 4]`, not `[1 2 3 4]`):

```
julia> x = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> x[:]
4-element Array{Int64,1}:
 1
 3
 2
 4
```

This convention for ordering arrays is common in many languages like Fortran, Matlab, and R (to name a few). The alternative to column-major ordering is

among other languages. Remembering the ordering of arrays can have significant performance effects when looping over arrays. A rule of thumb to keep in mind is that with column-major arrays, the first index changes most rapidly. Essentially this means that looping will be faster if the inner-most loop index is the first to appear in a slice expression.

Consider the following contrived example. Imagine we wanted to write a function that accepts a `Vector` and returns a square `Matrix` with either the rows or the columns filled with copies of the input vector. Assume that it is not important whether rows or columns are filled with these copies (perhaps the rest of the code can be easily adapted accordingly). We could conceivably do this in at least four ways (in addition to the recommended call to the built-in `repmat`):

```
function copy_cols(x::Vector{T}) where T
    inds = indices(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[:, i] = x
    end
    out
end

function copy_rows(x::Vector{T}) where T
    inds = indices(x, 1)
    out = similar(Array{T}, inds, inds)
    for i = inds
        out[i, :] = x
    end
    out
end
```

```

function copy_col_row(x::Vector{T}) where T
    inds = indices(x, 1)
    out = similar(Array{T}, inds, inds)
    for col = inds, row = inds
        out[row, col] = x[row]
    end
    out
end

function copy_row_col(x::Vector{T}) where T
    inds = indices(x, 1)
    out = similar(Array{T}, inds, inds)
    for row = inds, col = inds
        out[row, col] = x[col]
    end
    out
end

```

Now we will time each of these functions using the same random 10000 by 1 input vector:

```

julia> x = randn(10000);

julia> fmt(f) = println(rpad(string(f)*": ", 14, ' '), @elapsed
  → f(x))

julia> map(fmt, Any[copy_cols, copy_rows, copy_col_row,
  → copy_row_col]);
copy_cols:    0.331706323
copy_rows:   1.799009911

```

```
| copy_col_row: 0.415650047
```

```
| copy_row_col: 1.721531501
```

Notice that `copy_cols` is much faster than `copy_rows`. This is expected because `copy_cols` respects the column-based memory layout of the `Matrix` and fills it one column at a time. Additionally, `copy_col_row` is much faster than `copy_row_col` because it follows our rule of thumb that the first element to appear in a slice expression should be coupled with the inner-most loop.

## 40.13 Pre-allocating outputs

If your function returns an `Array` or some other complex type, it may have to allocate memory. Unfortunately, oftentimes allocation and its converse, garbage collection, are substantial bottlenecks.

Sometimes you can circumvent the need to allocate memory on each function call by preallocating the output. As a trivial example, compare

```
function xinc(x)
    return [x, x+1, x+2]
end

function loopinc()
    y = 0
    for i = 1:10^7
        ret = xinc(i)
        y += ret[2]
    end
    y
end
```

with

```
function xinc!(ret::AbstractVector{T}, x::T) where T
    ret[1] = x
    ret[2] = x+1
    ret[3] = x+2
    nothing
end

function loopinc_prealloc()
    ret = Array{Int}(3)
    y = 0
    for i = 1:10^7
        xinc!(ret, i)
        y += ret[2]
    end
    y
end
```

Timing results:

```
julia> @time loopinc()
0.529894 seconds (40.00 M allocations: 1.490 GiB, 12.14% gc
→ time)
50000015000000

julia> @time loopinc_prealloc()
0.030850 seconds (6 allocations: 288 bytes)
50000015000000
```

Preallocation has other advantages, for example by allowing the caller to control the "output" type from an algorithm. In the example above, we could have passed a `SubArray` rather than an `Array`, had we so desired.

Take more dots, fuse vectorized operations

code uglier, so perform  
mance measurements and some judgment may be required. However, for "vectorized" (element-wise) functions, the convenient syntax `x .= f.(y)` can be used for in-place operations with fused loops and no temporary arrays (see the [dot syntax for vectorizing functions](#)).

## 40.14 More dots: Fuse vectorized operations

Julia has a special [dot syntax](#) that converts any scalar function into a "vectorized" function call, and any operator into a "vectorized" operator, with the special property that nested "dot calls" are fusing: they are combined at the syntax level into a single loop, without allocating temporary arrays. If you use `.=` and similar assignment operators, the result can also be stored in-place in a pre-allocated array (see above).

In a linear-algebra context, this means that even though operations like `vector + vector` and `vector * scalar` are defined, it can be advantageous to instead use `vector .+ vector` and `vector .* scalar` because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

```
f(x) = 3x.^2 + 4x + 7x.^3

fdot(x) = @. 3x^2 + 4x + 7x^3 # equivalent to 3 .* x.^2 .+ 4 .* x
                                .+ 7 .* x.^3
```

Both `f` and `fdot` compute the same thing. However, `fdot` (defined with the help of the `@.` macro) is significantly faster when applied to an array:

```
julia> x = rand(10^6);

julia> @time f(x);
```

```
julia> @time fdot(x);  
0.010986 seconds (18 allocations: 53.400 MiB, 11.45% gc time)  
  
julia> @time f.(x);  
0.003470 seconds (6 allocations: 7.630 MiB)  
  
julia> @time f.(x);  
0.003297 seconds (30 allocations: 7.631 MiB)
```

That is, `fdot(x)` is three times faster and allocates 1/7 the memory of `f(x)`, because each `*` and `+` operation in `f(x)` allocates a new temporary array and executes in a separate loop. (Of course, if you just do `f.(x)` then it is as fast as `fdot(x)` in this example, but in many contexts it is more convenient to just sprinkle some dots in your expressions rather than defining a separate function for each vectorized operation.)

## 40.15 Consider using views for slices

In Julia, an array "slice" expression like `array[1:5, :]` creates a copy of that data (except on the left-hand side of an assignment, where `array[1:5, :]` = ... assigns in-place to that portion of `array`). If you are doing many operations on the slice, this can be good for performance because it is more efficient to work with a smaller contiguous copy than it would be to index into the original array. On the other hand, if you are just doing a few simple operations on the slice, the cost of the allocation and copy operations can be substantial.

An alternative is to create a "view" of the array, which is an array object (a `SubArray`) that actually references the data of the original array in-place, without making a copy. (If you write to a view, it modifies the original array's data as well.) This can be done for individual slices by calling `view`, or more simply for a whole expression or block of code by putting `@views` in front of

```
julia> fcopy(x) = sum(x[2:end-1])  
  
julia> @views fview(x) = sum(x[2:end-1])  
  
julia> x = rand(10^6);  
  
julia> @time fcopy(x);  
0.003051 seconds (7 allocations: 7.630 MB)  
  
julia> @time fview(x);  
0.001020 seconds (6 allocations: 224 bytes)
```

Notice both the  $3\times$  speedup and the decreased memory allocation of the `fview` version of the function.

## 40.16 Copying data is not always bad

Arrays are stored contiguously in memory, lending themselves to CPU vectorization and fewer memory accesses due to caching. These are the same reasons that it is recommended to access arrays in column-major order (see above). Irregular access patterns and non-contiguous views can drastically slow down computations on arrays because of non-sequential memory access.

Copying irregularly-accessed data into a contiguous array before operating on it can result in a large speedup, such as in the example below. Here, a matrix and a vector are being accessed at 800,000 of their randomly-shuffled indices before being multiplied. Copying the views into plain arrays speeds the multiplication by more than a factor of 2 even with the cost of the copying operation.

```
julia> x = randn(1_000_000);  
  
julia> inds = shuffle(1:1_000_000)[1:800000];  
  
julia> A = randn(50, 1_000_000);  
  
julia> xtmp = zeros(800_000);  
  
julia> Atmp = zeros(50, 800_000);  
  
julia> @time sum(view(A, :, inds) * view(x, inds))  
0.640320 seconds (41 allocations: 1.391 KiB)  
7253.242699002263  
  
julia> @time begin  
    copy!(xtmp, view(x, inds))  
    copy!(Atmp, view(A, :, inds))  
    sum(Atmp * xtmp)  
end  
0.261294 seconds (41 allocations: 1.391 KiB)  
7253.242699002323
```

Provided there is enough memory for the copies, the cost of copying the view to an array is far outweighed by the speed boost from doing the matrix multiplication on a contiguous array.

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
|println(file, "$a $b")
```

use:

```
|println(file, a, " ", b)
```

The first version of the code forms a string, then writes it to the file, while the second version writes values directly to the file. Also notice that in some cases string interpolation can be harder to read. Consider:

```
|println(file, "$(f(a))$(f(b))")
```

versus:

```
|println(file, f(a), f(b))
```

## 40.18 Optimize network I/O during parallel execution

When executing a remote function in parallel:

```
responses = Vector{Any}(nworkers())
@sync begin
    for (idx, pid) in enumerate(workers())
        @async responses[idx] = remotecall_fetch(pid, foo,
            ↳ args...)
    end
end
```

is faster than:

```
688 refs = Vector{Any}(nworkers())
for (idx, pid) in enumerate(workers())
    refs[idx] = @spawnat pid foo(args...)
end
responses = [fetch(r) for r in refs]
```

## CHAPTER 40. PERFORMANCE TIPS

The former results in a single network round-trip to every worker, while the latter results in two network calls – first by the `@spawnat` and the second due to the `fetch` (or even a `wait`). The `fetch/wait` is also being executed serially resulting in an overall poorer performance.

### 40.19 Fix deprecation warnings

A deprecated function internally performs a lookup in order to print a relevant warning only once. This extra lookup can cause a significant slowdown, so all uses of deprecated functions should be modified as suggested by the warnings.

### 40.20 Tweaks

These are some minor points that might help in tight inner loops.

Avoid unnecessary arrays. For example, instead of `sum([x,y,z])` use `x+y+z`.

Use `abs2(z)` instead of `abs(z)^2` for complex `z`. In general, try to rewrite code to use `abs2` instead of `abs` for complex arguments.

Use `div(x,y)` for truncating division of integers instead of `trunc(x/y)`, `fld(x,y)` instead of `floor(x/y)`, and `cld(x,y)` instead of `ceil(x/y)`.

Sometimes you can enable better optimization by promising certain program properties.

Use `@inbounds` to eliminate array bounds checking within expressions.

Be certain before doing this. If the subscripts are ever out of bounds, you may suffer crashes or silent corruption.

Use `@fastmath` to allow floating point optimizations that are correct for real numbers, but lead to differences for IEEE numbers. Be careful when doing this, as this may change numerical results. This corresponds to the `-ffast-math` option of clang.

Write `@simd` in front of `for` loops that are amenable to vectorization. This feature is experimental and could change or disappear in future versions of Julia.

The common idiom of using `1:n` to index into an `AbstractArray` is not safe if the Array uses unconventional indexing, and may cause a segmentation fault if bounds checking is turned off. Use `linearindices(x)` or `eachindex(x)` instead (see also [offset-arrays](#)).

Note: While `@simd` needs to be placed directly in front of a loop, both `@inbounds` and `@fastmath` can be applied to several statements at once, e.g. using `begin ... end`, or even to a whole function.

Here is an example with both `@inbounds` and `@simd` markup:

```
function inner(x, y)
    s = zero(eltype(x))
    for i=eachindex(x)
        @inbounds s += x[i]*y[i]
    end
```

```
690 s
```

## CHAPTER 40. PERFORMANCE TIPS

```
end
```

```
function innersimd(x, y)
    s = zero(eltype(x))
    @simd for i=eachindex(x)
        @inbounds s += x[i]*y[i]
    end
    s
```

```
end
```

```
function timeit(n, reps)
    x = rand(Float32,n)
    y = rand(Float32,n)
    s = zero(Float64)
    time = @elapsed for j in 1:reps
        s+=inner(x,y)
    end
    println("GFlop/sec      = ", 2.0*n*reps/time*1E-9)
    time = @elapsed for j in 1:reps
        s+=innersimd(x,y)
    end
    println("GFlop/sec (SIMD) = ", 2.0*n*reps/time*1E-9)
end

timeit(1000,1000)
```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```
| GFlop/sec      = 1.9467069505224963
| GFlop/sec (SIMD) = 17.578554163920018
```

(GFlop PERFORMANCE ANNOTATION, and larger numbers are better.) The range for a `@simd` `for` loop should be a one-dimensional range. A variable used for accumulating, such as `s` in the example, is called a reduction variable. By using `@simd`, you are asserting several properties of the loop:

It is safe to execute iterations in arbitrary or overlapping order, with special consideration for reduction variables.

Floating-point operations on reduction variables can be reordered, possibly causing different results than without `@simd`.

No iteration ever waits on another iteration to make forward progress.

A loop containing `break`, `continue`, or `@goto` will cause a compile-time error.

Using `@simd` merely gives the compiler license to vectorize. Whether it actually does so depends on the compiler. To actually benefit from the current implementation, your loop should have the following additional properties:

The loop must be an innermost loop.

The loop body must be straight-line code. This is why `@inbounds` is currently needed for all array accesses. The compiler can sometimes turn short `&&`, `||`, and `? :` expressions into straight-line code, if it is safe to evaluate all operands unconditionally. Consider using the `ifelse` function instead of `? :` in the loop if it is safe to do so.

Accesses must have a stride pattern and cannot be "gathers" (random-index reads) or "scatters" (random-index writes).

The stride should be unit stride.

In some simple cases, for example with 2-3 arrays accessed in a loop, the LLVM auto-vectorization may kick in automatically, leading to no further speedup with `@simd`.

602e is an example with all three kinds of marks. It first computes the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

```
function init!(u::Vector)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds @simd for i in 1:n #by asserting that `u` is a `Vector` we can assume it has 1-based indexing
        u[i] = sin(2pi*dx*i)
    end
end

function deriv!(u::Vector, du)
    n = length(u)
    dx = 1.0 / (n-1)
    @fastmath @inbounds du[1] = (u[2] - u[1]) / dx
    @fastmath @inbounds @simd for i in 2:n-1
        du[i] = (u[i+1] - u[i-1]) / (2*dx)
    end
    @fastmath @inbounds du[n] = (u[n] - u[n-1]) / dx
end

function norm(u::Vector)
    n = length(u)
    T = eltype(u)
    s = zero(T)
    @fastmath @inbounds @simd for i in 1:n
        s += u[i]^2
    end
    @fastmath @inbounds return sqrt(s/n)
end
```

```
function main()
    n = 2000
    u = Array{Float64}(n)
    init!(u)
    du = similar(u)

    deriv!(u, du)
    nu = norm(du)

    @time for i in 1:10^6
        deriv!(u, du)
        nu = norm(du)
    end

    println(nu)
end

main()
```

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

```
$ julia wave.jl;
elapsed time: 1.207814709 seconds (0 bytes allocated)

$ julia --math-mode=ieee wave.jl;
elapsed time: 4.487083643 seconds (0 bytes allocated)
```

Here, the option `--math-mode=ieee` disables the `@fastmath` macro, so that we can compare results.

In this case, the speedup due to `@fastmath` is a factor of about 3.7. This is unusually large – in general, the speedup will be smaller. (In this particular

example, the working set of the benchmark fits entirely in the cache of the processor, so that memory access latency does not play a role, and computing time is dominated by CPU usage. In many real world programs this is not the case.) Also, in this case this optimization does not change the result – in general, the result will be slightly different. In some cases, especially for numerically unstable algorithms, the result can be very different.

The annotation `@fastmath` re-arranges floating point expressions, e.g. changing the order of evaluation, or assuming that certain special cases (inf, nan) cannot occur. In this case (and on this particular computer), the main difference is that the expression `1 / (2*dx)` in the function `deriv` is hoisted out of the loop (i.e. calculated outside the loop), as if one had written `idx = 1 / (2*dx)`. In the loop, the expression `... / (2*dx)` then becomes `... * idx`, which is much faster to evaluate. Of course, both the actual optimization that is applied by the compiler as well as the resulting speedup depend very much on the hardware. You can examine the change in generated code by using Julia's `code_native` function.

## 40.22 Treat Subnormal Numbers as Zeros

Subnormal numbers, formerly called `denormal numbers`, are useful in many contexts, but incur a performance penalty on some hardware. A call `set_zero_subnormals(true)` grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call `set_zero_subnormals(false)` enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:

```
function timestep(b::Vector{T}, a::Vector{T}, Δt::T) where T
    @assert length(a)==length(b)
```

40.22 n = length(b)

695

```
b[1] = 1                                # Boundary condition
for i=2:n-1
    b[i] = a[i] + (a[i-1] - T(2)*a[i] + a[i+1]) * Δt
end
b[n] = 0                                # Boundary condition
end

function heatflow(a::Vector{T}, nstep::Integer) where T
    b = similar(a)
    for t=1:div(nstep,2)                  # Assume nstep is even
        timestep(b,a,T(0.1))
        timestep(a,b,T(0.1))
    end
end

heatflow(zeros(Float32,10),2)             # Force compilation
for trial=1:6
    a = zeros(Float32,1000)
    set_zero_subnormals(iseven(trial))   # Odd trials use strict
    → IEEE arithmetic
    @time heatflow(a,1000)
end
```

This example generates many subnormal numbers because the values in `a` become an exponentially decreasing curve, which slowly flattens out over time.

Treating subnormals as zeros should be used with caution, because doing so breaks some identities, such as `x-y == 0` implies `x == y`:

```
julia> x = 3f-38; y = 2f-38;
```

696

**julia>** set\_zero\_subnormals(true); (x <= y, x == y) PERFORMANCE TIPS  
(0.0f0, false)

**julia>** set\_zero\_subnormals(false); (x - y, x == y)  
(1.0000001f-38, false)

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing `a` with zeros, initialize it with:

```
| a = rand(Float32, 1000) * 1.f-9
```

## 40.23 @code\_warntype

The macro `@code_warntype` (or its function variant `code_warntype`) can sometimes be helpful in diagnosing type-related problems. Here's an example:

```
pos(x) = x < 0 ? 0 : x

function f(x)
    y = pos(x)
    sin(y*x+1)
end

julia> @code_warntype f(3.2)
Variables:
#self#::#f
x::Float64
y::Union{FLOAT64, INT64}
fy::Float64
#temp#@_5::Union{FLOAT64, INT64}
```

40.23 @CODE\_WARNTYPE  
  #temp#@\_6::Core.MethodInstance  
  #temp#@\_7::Float64

697

Body:

```
begin
  $(Expr(:inbounds, false))
  # meta: location REPL[1] pos 1
  # meta: location float.jl < 487
  fy::Float64 = (Core.ty-
    → peassert)((Base.sitofp)(Float64, 0)::Float64, Float64)::Float64
  # meta: pop location
unless
  → (Base.or_int)((Base.lt_float)(x::Float64, fy::Float64)::Bool, (Base.
  → goto 9
#temp#@_5::UNION{FLOAT64, INT64} = 0
goto 11
9:
#temp#@_5::UNION{FLOAT64, INT64} = x::Float64
11:
# meta: pop location
$(Expr(:inbounds, :pop))
y::UNION{FLOAT64, INT64} = #temp#@_5::UNION{FLOAT64, INT64} #
  → line 3:
unless (y::UNION{FLOAT64, INT64} isa Int64)::ANY goto 19
#temp#@_6::Core.MethodInstance = MethodInstance for
  → *(::Int64, ::Float64)
goto 28
19:
unless (y::UNION{FLOAT64, INT64} isa Float64)::ANY goto 23
#temp#@_6::Core.MethodInstance = MethodInstance for
  → *(::Float64, ::Float64)
```

```
698  goto 28
      23:
      goto 25
      25:
      #temp#@_7::Float64 = (y::UNION{FLOAT64, INT64} *
      ↳  x::Float64)::Float64
      goto 30
      28:
      #temp#@_7::Float64 = $(Expr(:invoke, :(#temp#@_6), :(Main.*),
      ↳  :(y), :(x)))
      30:
      return $(Expr(:invoke, MethodInstance for sin(::Float64),
      ↳  :(Main.sin),
      ↳  :((Base.add_float)(#temp#@_7, (Base.sitofp)(Float64, 1)::Float64)::Float64
end::Float64
```

## CHAPTER 40. PERFORMANCE TIPS

Interpreting the output of `@code_warntype`, like that of its cousins `@code_lowered`, `@code_typed`, `@code_llvm`, and `@code_native`, takes a little practice. Your code is being presented in form that has been partially digested on its way to generating compiled machine code. Most of the expressions are annotated by a type, indicated by the `::T` (where `T` might be `Float64`, for example). The most important characteristic of `@code_warntype` is that non-concrete types are displayed in red; in the above example, such output is shown in all-caps.

The top part of the output summarizes the type information for the different variables internal to the function. You can see that `y`, one of the variables you created, is a `Union{Int64, Float64}`, due to the type-instability of `pos`. There is another variable, `_var4`, which you can see also has the same type.

The next lines represent the body of `f`. The lines starting with a number followed by a colon (`1:`, `2:`) are labels, and represent targets for jumps (via

~~6.23. in @CODE\_WARN\_TYPE~~ Looking at the body, you can see that `pos` has been inlined into `f` – everything before `2:` comes from code defined in `pos`.

Starting at `2:`, the variable `y` is defined, and again annotated as a Union type. Next, we see that the compiler created the temporary variable `_var1` to hold the result of `y*x`. Because a `Float64` times either an `Int64` or `Float64` yields a `Float64`, all type-instability ends here. The net result is that `f(x::Float64)` will not be type-unstable in its output, even if some of the intermediate computations are type-unstable.

How you use this information is up to you. Obviously, it would be far and away best to fix `pos` to be type-stable: if you did so, all of the variables in `f` would be concrete, and its performance would be optimal. However, there are circumstances where this kind of ephemeral type instability might not matter too much: for example, if `pos` is never used in isolation, the fact that `f`'s output is type-stable (for `Float64` inputs) will shield later code from the propagating effects of type instability. This is particularly relevant in cases where fixing the type instability is difficult or impossible: for example, currently it's not possible to infer the return type of an anonymous function. In such cases, the tips above (e.g., adding type annotations and/or breaking up functions) are your best tools to contain the "damage" from type instability.

The following examples may help you interpret expressions marked as containing non-leaf types:

Function body ending in `end ::Union{T1, T2}`)

- Interpretation: function with unstable return type
- Suggestion: make the return value type-stable, even if you have to annotate it

`f(x::T)::Union{T1, T2}`

- Interpretation: call to a type-unstable function

700 – Suggestion: fix the function, or if necessary annotate the type

CHAPTER 40. PERFORMANCE TIPS

`(top(arrayref))(A::Array{Any, 1}, 1)::Any`

- Interpretation: accessing elements of poorly-typed arrays
- Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses

`(top(getfield))(A::ArrayContainer{Float64}, :data)::Array{Float64, N}`

- Interpretation: getting a field that is of non-leaf type. In this case, `ArrayContainer` had a field `data::Array{T}`. But `Array` needs the dimension `N`, too, to be a concrete type.
- Suggestion: use concrete types like `Array{T, 3}` or `Array{T, N}`, where `N` is now a parameter of `ArrayContainer`

# Chapter 41

## Workflow Tips

Here are some tips for working with Julia efficiently.

### 41.1 REPL-based workflow

As already elaborated in [Interacting With Julia](#), Julia's REPL provides rich functionality that facilitates an efficient interactive workflow. Here are some tips that might further enhance your experience at the command line.

#### A basic editor/REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the `julia` command line. A common pattern includes the following elements:

Put code under development in a temporary module. Create a file, say `Tmp.jl`, and include within it

```
module Tmp  
  
<your definitions here>  
  
end
```

702 Put your test code in another file. Create `CHAPTER11/workflow.jl`, say `WORKFLOW.JL` begins with

```
| import Tmp
```

and includes tests for the contents of `Tmp`. The value of using `import` versus `using` is that you can call `reload("Tmp")` instead of having to restart the REPL when your definitions change. Of course, the cost is the need to prepend `Tmp.` to uses of names defined in your module. (You can lower that cost by keeping your module name short.)

Alternatively, you can wrap the contents of your test file in a module, as

```
module Tst
    using Tmp

    <scratch work>

end
```

The advantage is that you can now do `using Tmp` in your test code and can therefore avoid prepending `Tmp.` everywhere. The disadvantage is that code can no longer be selectively copied to the REPL without some tweaking.

Lather. Rinse. Repeat. Explore ideas at the `julia` command prompt. Save good ideas in `tst.jl`. Occasionally restart the REPL, issuing

```
| reload("Tmp")
| include("tst.jl")
```

Simplify initialization

To simplify restarting the REPL, put project-specific initialization code in a file, say `_init.jl`, which you can run on startup by issuing the command:

If you further add the following to your `.juliarc.jl` file

```
| isfile("_init.jl") && include(joinpath(pwd(), "_init.jl"))
```

then calling `julia` from that directory will run the initialization code without the additional command line argument.

## 41.2 Browser-based workflow

It is also possible to interact with a Julia REPL in the browser via [IJulia](#). See the package home for details.



# Chapter 42

## Style Guide

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

### 42.1 Write functions, not just scripts

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia’s compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like `pi`).

### 42.2 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
|convert(Complex{Float64}, x)
```

```
| complex(float(x))
```

The second version will convert `x` to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type `Int` or `Int32` if it really could be any integer, expressed with the abstract type `Integer`. In fact, in many cases you can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a `MethodError` will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as [duck typing](#).)

For example, consider the following definitions of a function `addone` that returns one plus its argument:

```
addone(x::Int) = x + 1                      # works only for Int
addone(x::Integer) = x + oneunit(x)          # any integer type
addone(x::Number) = x + oneunit(x)          # any numeric type
addone(x) = x + oneunit(x)                   # any type supporting + and
    ↵ oneunit
```

The last definition of `addone` handles any type supporting `oneunit` (which returns 1 in the same type as `x`, which avoids unwanted type promotion) and the `+` function with those arguments. The key thing to realize is that there is no performance penalty to defining only the general `addone(x) = x + oneunit(x)`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `addone(12)`, Julia will automatically compile a specialized `addone` function for `x::Int` arguments, with the call to `oneunit` replaced by its inlined value 1. Therefore, the first three definitions of `addone` above are completely redundant with the fourth definition.

Instead of:

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

This is better style because `foo` does not really accept numbers of all types; it really needs `Int`s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more "space" for future method definitions.

## 42.4 Append `!` to names of functions that modify their arguments

Instead of:

```
function double(a::AbstractArray{<:Number})
    for i = 1:endof(a)
        a[i] *= 2
```

```
708   end  
      return a  
end
```

## CHAPTER 42. STYLE GUIDE

use:

```
function double!(a::AbstractArray{<:Number})  
    for i = 1:endof(a)  
        a[i] *= 2  
    end  
    return a  
end
```

The Julia standard library uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `sort` and `sort!`), and others which are just modifying (e.g., `push!`, `pop!`, `splice!`). It is typical for such functions to also return the modified array for convenience.

### 42.5 Avoid strange type **Unions**

Types such as `Union{Function, AbstractString}` are often a sign that some design could be cleaner.

### 42.6 Avoid type Unions in fields

When creating a type such as:

```
mutable struct MyType  
    ...  
    x::Union{Void,T}  
end
```

ask whether the option for `x` to be `nothing` (of type `Void`) is really necessary. Here are some alternatives to consider:

Introduce another type that lacks `x`

If there are many fields like `x`, store them in a dictionary

Determine whether there is a simple rule for when `x` is `nothing`. For example, often the field will start as `nothing` but get initialized at some well-defined point. In that case, consider leaving it undefined at first.

If `x` really needs to hold no value at some times, define it as `::Nullable{T}` instead, as this guarantees type-stability in the code accessing this field (see [Nullable types](#)).

## 42.7 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
| a = Array{Union{Int,AbstractString,Tuple,Array}}(n)
```

In this case `Array{Any}(n)` is better. It is also more helpful to the compiler to annotate specific uses (e.g. `a[i]::Int`) than to try to pack many alternatives into one type.

## 42.8 Use naming conventions consistent with Julia's `base/`

modules and type names use capitalization and camel case: `module SparseArrays, struct UnitRange`.

functions are lowercase (`maximum`, `convert`) and, when readable, with multiple words squashed together (`isequal`, `haskey`). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (`remotecall_fetch` as a more efficient implementation of `fetch(remotecall(...))`) or as modifiers (`sum_kbn`).

710 conciseness is valued, but avoid abbreviation [CHAPTER 14: STYLE GUIDE](#)  
[dxin](#)) as it becomes difficult to remember whether and how particular words are abbreviated.

If a function name requires multiple words, consider whether it might represent more than one concept and might be better split into pieces.

## 42.9 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

## 42.10 Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
| if a == b
```

instead of:

```
| if (a == b)
```

## 42.11 Don't overuse ...

Splicing function arguments can be addictive. Instead of `[a..., b...]`, use simply `[a; b]`, which already concatenates arrays. `collect(a)` is better than `[a...]`, but since `a` is already iterable it is often even better to leave it alone, and not convert it to an array.

## 42.12 Don't use unnecessary static parameters

A function signature:

```
| foo(x::T) where {T<:Real} = ...
```

```
| foo(x::Real) = ...
```

instead, especially if T is not used in the function body. Even if T is used, it can be replaced with `typeof(x)` if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ [Avoid fields with abstract containers](#) for more information.

#### 42.13 Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
| foo(::Type{MyType}) = ...
| foo(::MyType) = foo(MyType)
```

Decide whether the concept in question will be written as `MyType` or `MyType()`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `Type{MyType}` later if they become necessary to solve some problem.

If a type is effectively an enumeration, it should be defined as a single (ideally immutable struct or primitive) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

Be aware of when a macro could really be a function instead.

Calling `eval` inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

## 42.15 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
end
```

don't write definitions like the following:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

The problem is that users of this type can write `x[i]` without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have `unsafe` somewhere in its name to alert callers.

## 42.16 Don't overload methods of base container types

It is possible to write definitions like the following:

```
show(io::IO, v::Vector{MyType}) = ...
```

This ~~is two AND~~<sup>is</sup> ~~TYPE PIRACY~~<sup>TYPE PLACEMENT</sup> showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like `Vector()` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

## 42.17 Avoid type piracy

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In some cases, you can get away with type piracy with little ill effect. In extreme cases, however, you can even crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a `ccall`). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

As an example, suppose you wanted to define multiplication on symbols in a module:

```
module A
import Base./*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

The problem is that now any other module that uses `Base.*` will also see this definition. Since `Symbol` is defined in Base and is used by other modules, this can change the behavior of unrelated code unexpectedly. There are several alternatives here, including using a different function name, or wrapping the `Symbol`s in another type that you define.

Sometimes, coupled packages may engage in type piracy to separate features from definitions, especially when the packages were designed by collaborating authors, and when the definitions are reusable. For example, one package

CHAPTER 42. ANOTHER PACKAGE

might provide some types useful for working with color spaces. Another example might be a package that acts as a thin wrapper for some C code, which another package might then pirate to implement a higher-level, Julia-friendly API.

## 42.18 Be careful with type equality

You generally want to use `isa` and `<:` for testing types, not `==`. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `T == Float64`), or if you really, really know what you're doing.

## 42.19 Do not write `x->f(x)`

Since higher-order functions are often called with anonymous functions, it is easy to conclude that this is desirable or even necessary. But any function can be passed directly, without being "wrapped" in an anonymous function. Instead of writing `map(x->f(x), a)`, write `map(f, a)`.

## 42.20 Avoid using floats for numeric literals in generic code when possible

If you write generic code which handles numbers, and which can be expected to run with many different numeric type arguments, try using literals of a numeric type that will affect the arguments as little as possible through promotion.

For example,

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)
```

```
julia> f(1//2)
```

```
1.0
```

```
julia> f(1/2)
```

```
1.0
```

```
julia> f(1)
```

```
2.0
```

while

```
julia> g(x) = 2 * x
```

```
g (generic function with 1 method)
```

```
julia> g(1//2)
```

```
1//1
```

```
julia> g(1/2)
```

```
1.0
```

```
julia> g(1)
```

```
2
```

As you can see, the second version, where we used an `Int` literal, preserved the type of the input argument, while the first didn't. This is because e.g. `promote_type(Int, Float64) == Float64`, and promotion happens with the multiplication. Similarly, `Rational` literals are less type disruptive than `Float64` literals, but more disruptive than `Ints`:

```
julia> h(x) = 2//1 * x
```

```
h (generic function with 1 method)
```

```
julia> h(1//2)
```

```
1//1
```

```
julia> h(1/2)
```

```
1.0
```

```
julia> h(1)
```

```
2//1
```

Thus, use `Int` literals when possible, with `Rational{Int}` for literal non-integer numbers, in order to make it easier to use your code.

## Chapter 43

# Frequently Asked Questions

### 43.1 Sessions and the REPL

How do I delete an object in memory?

Julia does not have an analog of MATLAB's `clear` function; once a name is defined in a Julia session (technically, in module `Main`), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if `A` is a gigabyte-sized array that you no longer need, you can free the memory with `A = nothing`. The memory will be released the next time the garbage collector runs; you can force this to happen with `gc()`. Moreover, an attempt to use `A` will likely result in an error, because most methods are not defined on type `Void`.

How can I modify the declaration of a type in my session?

Perhaps you've defined a type and then realize you need to add a new field. If you try this at the REPL, you get the error:

```
|ERROR: invalid redefinition of constant MyType
```

Types in module `Main` cannot be redefined.

While this can be inconvenient, it is a frequent question asked by other users. A better excellent workaround. Modules can be replaced by redefining them, and so if you wrap all your new code inside a module you can redefine types and constants. You can't import the type names into `Main` and then expect to be able to redefine them there, but you can use the module name to resolve the scope. In other words, while developing you might use a workflow something like this:

```
include("mynewcode.jl")          # this defines a module
→ MyModule
obj1 = MyModule.ObjConstructor(a, b)
obj2 = MyModule.somefunction(obj1)
# Got an error. Change something in "mynewcode.jl"
include("mynewcode.jl")          # reload the module
obj1 = MyModule.ObjConstructor(a, b) # old objects are no longer
→ valid, must reconstruct
obj2 = MyModule.somefunction(obj1)  # this time it worked!
obj3 = MyModule.someotherfunction(obj2, c)
...
```

## 43.2 Functions

I passed an argument `x` to a function, modified it inside that function, but on the outside, the variable `x` is still unchanged. Why?

Suppose you call a function like this:

```
julia> x = 10
10

julia> function change_value!(y)
```

```
    end  
change_value! (generic function with 1 method)  
  
julia> change_value!(x)  
17  
  
julia> x # x is unchanged!  
10
```

In Julia, the binding of a variable `x` cannot be changed by passing `x` as an argument to a function. When calling `change_value!(x)` in the above example, `y` is a newly created variable, bound initially to the value of `x`, i.e. 10; then `y` is rebound to the constant 17, while the variable `x` of the outer scope is left untouched.

But here is a thing you should pay attention to: suppose `x` is bound to an object of type `Array` (or any other mutable type). From within the function, you cannot "unbind" `x` from this `Array`, but you can change its content. For example:

```
julia> x = [1,2,3]  
3-element Array{Int64,1}:  
1  
2  
3  
  
julia> function change_array!(A)  
    A[1] = 5
```

720        **end**

## CHAPTER 43. FREQUENTLY ASKED QUESTIONS

change\_array! (generic function with 1 method)

```
julia> change_array!(x)
```

```
5
```

```
julia> x
```

```
3-element Array{Int64,1}:
```

```
5
```

```
2
```

```
3
```

Here we created a function `change_array!`, that assigns `5` to the first element of the passed array (bound to `x` at the call site, and bound to `A` within the function). Notice that, after the function call, `x` is still bound to the same array, but the content of that array changed: the variables `A` and `x` were distinct bindings referring to the same mutable `Array` object.

Can I use **using** or **import** inside a function?

No, you are not allowed to have a `using` or `import` statement inside a function. If you want to import a module but only use its symbols inside a specific function or set of functions, you have two options:

1. Use `import`:

```
import Foo  
function bar(...)  
    # ... refer to Foo symbols via Foo.baz ...  
end
```

This loads the module `Foo` and defines a variable `Foo` that refers to the module, but does not import any of the other symbols from the module into

43.2 the FUNCTIONS namespace. You refer to the Foo symbols by their qualified names `Foo.bar` etc.

2. Wrap your function in a module:

```
module Bar
export bar
using Foo
function bar(...)
    # ... refer to Foo.baz as simply baz ....
end
end
using Bar
```

This imports all the symbols from Foo, but only inside the module Bar.

What does the `...` operator do?

The two uses of the `...` operator: slurping and splatting

Many newcomers to Julia find the use of `...` operator confusing. Part of what makes the `...` operator confusing is that it means two different things depending on context.

`...` combines many arguments into one argument in function definitions

In the context of function definitions, the `...` operator is used to combine many different arguments into a single argument. This use of `...` for combining many different arguments into a single argument is called slurping:

```
julia> function printargs(args...)
@printf("%s\n", typeof(args))
```

```
for (i, arg) in enumerate(args)
    @printf("Arg %d = %s\n", i, arg)
end
end
printargs (generic function with 1 method)

julia> printargs(1, 2, 3)
Tuple{Int64, Int64, Int64}
Arg 1 = 1
Arg 2 = 2
Arg 3 = 3
```

If Julia were a language that made more liberal use of ASCII characters, the slurping operator might have been written as <-... instead of ....

... splits one argument into many different arguments in function calls

In contrast to the use of the ... operator to denote slurping many different arguments into one argument when defining a function, the ... operator is also used to cause a single function argument to be split apart into many different arguments when used in the context of a function call. This use of ... is called splatting:

```
julia> function threeargs(a, b, c)
    @printf("a = %s::%s\n", a, typeof(a))
    @printf("b = %s::%s\n", b, typeof(b))
```

```
@printf("c = %s::%s\n", c, typeof(c))

end
threeargs (generic function with 1 method)

julia> vec = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> threeargs(vec...)
a = 1::Int64
b = 2::Int64
c = 3::Int64
```

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as `...->` instead of `...`.

What is the return value of an assignment?

The operator `=` always returns the right-hand side, therefore:

```
julia> function threeint()

    x::Int = 3.0

    x # returns variable x

end
threeint (generic function with 1 method)
```

```
julia> function threefloat()
           x::Int = 3.0 # returns 3.0

       end
threefloat (generic function with 1 method)
```

```
julia> threeint()
```

```
3
```

```
julia> threefloat()
```

```
3.0
```

and similarly:

```
julia> function threetup()
           x, y = [3, 3]

           x, y # returns a tuple

       end
threetup (generic function with 1 method)
```

```
julia> function threearr()
```

```
           x, y = [3, 3] # returns an array
```

```
       end
threearr (generic function with 1 method)
```

```
julia> threetup()
```

```
julia> threearr()  
2-element Array{Int64,1}:  
3  
3
```

### 43.3 Types, type declarations, and constructors

What does "type-stable" mean?

It means that the type of the output is predictable from the types of the inputs. In particular, it means that the type of the output cannot vary depending on the values of the inputs. The following code is not type-stable:

```
julia> function unstable(flag::Bool)  
  
    if flag  
  
        return 1  
  
    else  
  
        return 1.0  
  
    end  
  
    end  
unstable (generic function with 1 method)
```

It returns either an `Int` or a `Float64` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time,

726 computation that uses it CHAPTER 13. FREQUENTLY ASKED QUESTIONS occurring, making generation of fast machine code difficult.

Why does Julia give a **DomainError** for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

```
julia> sqrt(-2.0)
ERROR: DomainError with -2.0:
sqrt will only return a complex result if called with a complex
→ argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `sqrt`, most users want `sqrt(2.0)` to give a real number, and would be unhappy if it produced the complex number `1.4142135623730951 + 0.0im`. One could write the `sqrt` function to switch to a complex-valued output only when passed a negative number (which is what `sqrt` does in some other languages), but then the result would not be `type-stable` and the `sqrt` function would have poor performance.

In these and other cases, you can get the result you want by choosing an input type that conveys your willingness to accept an output type in which the result can be represented:

```
julia> sqrt(-2.0+0im)
0.0 + 1.4142135623730951im
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Int` values is bounded and wraps around at either end so that

Adding types to type declarations, and constructors or underflow, led to some results that can be unsettling at first:

```
julia> typemax(Int)  
9223372036854775807
```

```
julia> ans+1  
-9223372036854775808
```

```
julia> -ans  
-9223372036854775808
```

```
julia> 2*ans  
0
```

Clearly, this is far from the way mathematical integers behave, and you might think it less than ideal for a high-level programming language to expose this to the user. For numerical work where efficiency and transparency are at a premium, however, the alternatives are worse.

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as [Int128](#) or [BigInt](#) in the case of overflow. Unfortunately, this introduces major overhead on every integer operation (think incrementing a loop counter) – it requires emitting code to perform run-time overflow checks after arithmetic instructions and branches to handle potential overflows. Worse still, this would cause every computation involving integers to be type-unstable. As we mentioned above, [type-stability is crucial](#) for effective generation of efficient code. If you can't count on the results of integer operations being integers, it's impossible to generate fast, simple code the way C and Fortran compilers do.

A variation on this approach, which avoids the appearance of type instability

internally changes representation when a result no longer fits into the size of a machine integer. While this superficially avoids type-instability at the level of Julia code, it just sweeps the problem under the rug by foisting all of the same difficulties onto the C code implementing this hybrid integer type. This approach can be made to work and can even be made quite fast in many cases, but has several drawbacks. One problem is that the in-memory representation of integers and arrays of integers no longer match the natural representation used by C, Fortran and other languages with native machine integers. Thus, to interoperate with those languages, we would ultimately need to introduce native integer types anyway. Any unbounded representation of integers cannot have a fixed number of bits, and thus cannot be stored in-line in an array with fixed-size slots – large integer values will always require separate heap-allocated storage. And of course, no matter how clever a hybrid integer implementation one uses, there are always performance traps – situations where performance degrades unexpectedly. Complex representation, lack of interoperability with C and Fortran, the inability to represent integer arrays without additional heap storage, and unpredictable performance characteristics make even the cleverest hybrid integer implementations a poor choice for high-performance numerical work.

An alternative to using hybrid integers or promoting to BigInts is to use saturating integer arithmetic, where adding to the largest integer value leaves it unchanged and likewise for subtracting from the smallest integer value. This is precisely what Matlab™ does:

```
>> int64(9223372036854775807)

ans =

9223372036854775807
```

```
>> int64(9223372036854775807) + 1  
  
ans =  
  
9223372036854775807  
  
>> int64(-9223372036854775808)  
  
ans =  
  
-9223372036854775808  
  
>> int64(-9223372036854775808) - 1  
  
ans =  
  
-9223372036854775808
```

At first blush, this seems reasonable enough since 9223372036854775807 is much closer to 9223372036854775808 than -9223372036854775808 is and integers are still represented with a fixed size in a natural way that is compatible with C and Fortran. Saturated integer arithmetic, however, is deeply problematic. The first and most obvious issue is that this is not the way machine integer arithmetic works, so implementing saturated operations requires emitting instructions after each machine integer operation to check for underflow or overflow and replace the result with `typemin(Int)` or `typemax(Int)` as appropriate. This alone expands each integer operation from a single, fast instruction into half a dozen instructions, probably including branches. Ouch. But it gets worse – saturating integer arithmetic isn't associative. Consider this Matlab computation:

```
|>30n = int64(2)^62  
4611686018427387904
```

## CHAPTER 43. FREQUENTLY ASKED QUESTIONS

```
|>> n + (n - 1)  
9223372036854775807  
  
>> (n + n) - 1  
9223372036854775806
```

This makes it hard to write many basic integer algorithms since a lot of common techniques depend on the fact that machine addition with overflow is associative. Consider finding the midpoint between integer values `lo` and `hi` in Julia using the expression `(lo + hi) >>> 1`:

```
|julia> n = 2^62  
4611686018427387904  
  
|julia> (n + 2n) >>> 1  
6917529027641081856
```

See? No problem. That's the correct midpoint between  $2^{62}$  and  $2^{63}$ , despite the fact that `n + 2n` is  $-4611686018427387904$ . Now try it in Matlab:

```
|>> (n + 2*n)/2  
  
ans =  
  
4611686018427387904
```

Oops. Adding a `>>>` operator to Matlab wouldn't help, because saturation that occurs when adding `n` and `2n` has already destroyed the information necessary to compute the correct midpoint.

~~All But TYPE DECLARATIONS AND CONSTRUCTORS~~ 721  
it for techniques like this, but it also defeats almost anything compilers might want to do to optimize integer arithmetic. For example, since Julia integers use normal machine integer arithmetic, LLVM is free to aggressively optimize simple little functions like  $f(k) = 5k - 1$ . The machine code for this function is just this:

```
julia> code_native(f, Tuple{Int})  
.text  
Filename: none  
pushq %rbp  
movq %rsp, %rbp  
Source line: 1  
leaq -1(%rdi,%rdi,4), %rax  
popq %rbp  
retq  
nopl (%rax,%rax)
```

The actual body of the function is a single `leaq` instruction, which computes the integer multiply and add at once. This is even more beneficial when `f` gets inlined into another function:

```
julia> function g(k, n)  
    for i = 1:n  
        k = f(k)  
    end  
    return k
```

732        end

## CHAPTER 43. FREQUENTLY ASKED QUESTIONS

g (generic function with 1 methods)

```
julia> code_native(g, Tuple{Int, Int})
```

.text

Filename: none

pushq %rbp

  movq %rsp, %rbp

Source line: 2

  testq %rsi, %rsi

  jle L26

  nopl (%rax)

Source line: 3

L16:

  leaq -1(%rdi,%rdi,4), %rdi

Source line: 2

  decq %rsi

  jne L16

Source line: 5

L26:

  movq %rdi, %rax

  popq %rbp

  retq

  nop

Since the call to `f` gets inlined, the loop body ends up being just a single `leaq` instruction. Next, consider what happens if we make the number of loop iterations fixed:

```
julia> function g(k)
```

```
    for i = 1:10
```

```
    k = f(k)

  end

  return k

end

g (generic function with 2 methods)
```

```
julia> code_native(g,(Int,))
.text
Filename: none
pushq %rbp
movq %rsp, %rbp
Source line: 3
imulq $9765625, %rdi, %rax      # imm = 0x9502F9
addq $-2441406, %rax              # imm = 0xFFDABF42
Source line: 5
popq %rbp
retq
nopw %cs:(%rax,%rax)
```

Because the compiler knows that integer addition and multiplication are associative and that multiplication distributes over addition – neither of which is true of saturating arithmetic – it can optimize the entire loop down to just a multiply and an add. Saturated arithmetic completely defeats this kind of optimization since associativity and distributivity can fail at each loop iteration, causing different outcomes depending on which iteration the failure occurs in. The compiler can unroll the loop, but it cannot algebraically reduce multiple operations into fewer equivalent operations.

THE most reasonable alternative to checked integer arithmetic is to do checked arithmetic everywhere, raising errors when adds, subtracts, and multiplies overflow, producing values that are not value-correct. In this [blog post](#), Dan Luu analyzes this and finds that rather than the trivial cost that this approach should in theory have, it ends up having a substantial cost due to compilers (LLVM and GCC) not gracefully optimizing around the added overflow checks. If this improves in the future, we could consider defaulting to checked integer arithmetic in Julia, but for now, we have to live with the possibility of overflow.

What are the possible causes of an **UndefVarError** during remote execution?

As the error states, an immediate cause of an **UndefVarError** on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

```
julia> module Foo

    foo() = remotecall_fetch(x->x, 2, "Hello")

end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: Foo not defined
Stacktrace:
[...]
```

The closure `x->x` carries a reference to `Foo`, and since `Foo` is unavailable on node 2, an **UndefVarError** is thrown.

~~Globals~~ TYPES, TYPE DECLARATIONS, AND CONSTRUCTORS are sent by value to the remote node. Only a reference is sent. Functions which create global bindings (except under `Main`) may cause an `UndefVarError` to be thrown later.

```
julia> @everywhere module Foo

    function foo()

        global gvar = "Hello"

        remotecall_fetch(()->gvar, 2)

    end

end

julia> Foo.foo()
ERROR: On worker 2:
UndefVarError: gvar not defined
Stacktrace:
[...]
```

In the above example, `@everywhere module Foo` defined `Foo` on all nodes. However the call to `Foo.foo()` created a new global binding `gvar` on the local node, but this was not found on node 2 resulting in an `UndefVarError` error.

Note that this does not apply to globals created under module `Main`. Globals under module `Main` are serialized and new bindings created under `Main` on the remote node.

```
julia> gvar_self = "Node1"
"Node1"
```

```
julia> remotecall_fetch(()->gvar_self, 2)
"Node1"

julia> remotecall_fetch(whos, 2)
      From worker 2:                                Base
→ 41762 KB    Module
      From worker 2:                                Core
→ 27337 KB    Module
      From worker 2:                                Foo
→ 2477 bytes  Module
      From worker 2:                                Main
→ 46191 KB    Module
      From worker 2:                                gvar_self
→ 13 bytes   String
```

This does not apply to `function` or `type` declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

```
julia> bar() = 1
bar (generic function with 1 method)

julia> remotecall_fetch(bar, 2)
ERROR: On worker 2:
UndefVarError: #bar not defined
[...]

julia> anon_bar = ()->1
(::#21) (generic function with 1 method)

julia> remotecall_fetch(anon_bar, 2)
```

What is the difference between "using" and "import"?

There is only one difference, and on the surface (syntax-wise) it may seem very minor. The difference between `using` and `import` is that with `using` you need to say `function Foo.bar( ...` to extend module Foo's function bar with a new method, but with `import Foo.bar`, you only need to say `function bar( ...` and it automatically extends module Foo's function bar.

The reason this is important enough to have been given separate syntax is that you don't want to accidentally extend a function that you didn't know existed, because that could easily cause a bug. This is most likely to happen with a method that takes a common type like a string or integer, because both you and the other module could define a method to handle such a common type. If you use `import`, then you'll replace the other module's implementation of `bar(s::AbstractString)` with your new implementation, which could easily do something completely different (and break all/many future usages of the other functions in module Foo that depend on calling bar).

## 43.5 Nothingness and missing values

How does "null" or "nothingness" work in Julia?

Unlike many languages (for example, C and Java), Julia does not have a "null" value. When a reference (variable, object field, or array element) is uninitialized, accessing it will immediately throw an error. This situation can be detected using the `isdefined` function.

Some functions are used only for their side effects, and do not need to return a value. In these cases, the convention is to return the value `nothing`, which is just a singleton object of type `Void`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that

REPL does not print anything. CHAPTER 43 FREQUENTLY ASKED QUESTIONS  
not otherwise have a value also yield `nothing`, for example `if false; end`.

For situations where a value exists only sometimes (for example, missing statistical data), it is best to use the `Nullable{T}` type, which allows specifying the type of a missing value.

The empty tuple `(( ))` is another form of nothingness. But, it should not really be thought of as nothing but rather a tuple of zero values.

In code written for Julia prior to version 0.4 you may occasionally see `None`, which is quite different. It is the empty (or "bottom") type, a type with no values and no subtypes (except itself). This is now written as `Union{}` (an empty union type). You will generally not need to use this type.

## 43.6 Memory

Why does `x += y` allocate memory when `x` and `y` are arrays?

In Julia, `x += y` gets replaced during parsing by `x = x + y`. For arrays, this has the consequence that, rather than storing the result in the same location in memory as `x`, it allocates a new array to store the result.

While this behavior might surprise some, the choice is deliberate. The main reason is the presence of immutable objects within Julia, which cannot change their value once created. Indeed, a number is an immutable object: the statements `x = 5; x += 1` do not modify the meaning of 5, they modify the value bound to `x`. For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

```
function power_by_squaring(x, n::Int)
    ispow2(n) || error("This implementation only works for powers
        → of 2")
```

```
while n >= 2
    x *= x
    n >>= 1
end
x
end
```

After a call like `x = 5; y = power_by_squaring(x, 4)`, you would get the expected result: `x == 5 && y == 625`. However, now suppose that `*=`, when used with matrices, instead mutated the left hand side. There would be two problems:

For general square matrices, `A = A*B` cannot be implemented without temporary storage: `A[1,1]` gets computed and stored on the left hand side before you're done using it on the right hand side.

Suppose you were willing to allocate a temporary for the computation (which would eliminate most of the point of making `*=` work in-place); if you took advantage of the mutability of `x`, then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable `x`, after the call you'd have (in general) `y != x`, but for mutable `x` you'd have `y == x`.

Because supporting generic programming is deemed more important than potential performance optimizations that can be achieved by other means (e.g., using explicit loops), operators like `+=` and `*=` work by rebinding new values.

Why do concurrent writes to the same stream result in inter-mixed output?

While the streaming I/O API is synchronous, the underlying implementation is fully asynchronous.

Consider the printed output from the following:

```
julia> @sync for i in 1:3  
    @async write(STDOUT, string(i), " Foo ", " Bar ")  
end  
123 Foo Foo Foo Bar Bar Bar
```

This is happening because, while the `write` call is synchronous, the writing of each argument yields to other tasks while waiting for that part of the I/O to complete.

`print` and `println` “lock” the stream during a call. Consequently changing `write` to `println` in the above example results in:

```
julia> @sync for i in 1:3  
    @async println(STDOUT, string(i), " Foo ", " Bar ")  
end  
1 Foo Bar  
2 Foo Bar  
3 Foo Bar
```

You can lock your writes with a `ReentrantLock` like this:

43.8 JULIA RELEASES  
Julia> l = ReentrantLock()

741

```
ReentrantLock(Nullable{Task}(), Condition(Any[]), 0)
```

```
julia> @sync for i in 1:3
```

```
    @async begin
```

```
        lock(l)
```

```
        try
```

```
            write(STDOUT, string(i), " Foo ", " Bar ")
```

```
        finally
```

```
            unlock(l)
```

```
        end
```

```
    end
```

```
end
```

```
1 Foo  Bar 2 Foo  Bar 3 Foo  Bar
```

## 43.8 Julia Releases

Do I want to use a release, beta, or nightly version of Julia?

You may prefer the release version of Julia if you are looking for a stable code base. Releases generally occur every 6 months, giving you a stable platform for writing code.

You may prefer the beta version of Julia if you don't mind being slightly behind

The latest bugfixes and changes CHAPTER 14. THE FREQUENTLY ASKED QUESTIONS make Julia more appealing. Additionally, these binaries are tested before they are published to ensure they are fully functional.

You may prefer the nightly version of Julia if you want to take advantage of the latest updates to the language, and don't mind if the version available today occasionally doesn't actually work.

Finally, you may also consider building Julia from source for yourself. This option is mainly for those individuals who are comfortable at the command line, or interested in learning. If this describes you, you may also be interested in reading our [guidelines for contributing](#).

Links to each of these download types can be found on the download page at <https://julialang.org/downloads/>. Note that not all versions of Julia are available for all platforms.

## When are deprecated functions removed?

Deprecated functions are removed after the subsequent release. For example, functions marked as deprecated in the 0.1 release will not be available starting with the 0.2 release.

## Chapter 44

# Noteworthy Differences from other Languages

### 44.1 Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is not a MATLAB clone. There are major syntactic and functional differences. The following are some noteworthy differences that may trip up Julia users accustomed to MATLAB:

Julia arrays are indexed with square brackets, `A[i, j]`.

Julia arrays are assigned by reference. After `A=B`, changing elements of `B` will modify `A` as well.

Julia values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller.

Julia does not automatically grow arrays in an assignment statement. Whereas in MATLAB `a(4) = 3.2` can create the array `a = [0 0 0 3.2]` and `a(5) = 7` can grow it into `a = [0 0 0 3.2 7]`, the corresponding Julia statement `a[5] = 7` throws an error if the length of `a` is less than 5 or if this statement is the first use of the identifier `a`. Julia has [push!](#)

`a(end+1) = val.`

The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `i` or `j` as in MATLAB.

In Julia, literal numbers without a decimal point (such as `42`) create integers instead of floating point numbers. Arbitrarily large integer literals are supported. As a result, some operations such as `2^-1` will throw a domain error as the result is not an integer (see [the FAQ entry on domain errors](#) for details).

In Julia, multiple values are returned and assigned as tuples, e.g. `(a, b) = (1, 2)` or `a, b = 1, 2`. MATLAB's `nargout`, which is often used in MATLAB to do optional work based on the number of returned values, does not exist in Julia. Instead, users can use optional and keyword arguments to achieve similar capabilities.

Julia has true one-dimensional arrays. Column vectors are of size `N`, not `Nx1`. For example, `rand(N)` makes a 1-dimensional array.

In Julia, `[x, y, z]` will always construct a 3-element array containing `x`, `y` and `z`.

- To concatenate in the first ("vertical") dimension use either `vcat(x, y, z)` or separate with semicolons (`[x; y; z]`).
- To concatenate in the second ("horizontal") dimension use either `hcat(x, y, z)` or separate with spaces (`[x y z]`).
- To construct block matrices (concatenating in the first two dimensions), use either `hvcat` or combine spaces and semicolons (`[a b; c d]`).

In Julia, `a:b` and `a:b:c` construct `AbstractRange` objects. To construct a full vector like in MATLAB, use `collect(a:b)`. Generally, there is no need

Object will act like a normal array in most cases but is more efficient because it lazily computes its values. This pattern of creating specialized objects instead of full arrays is used frequently, and is also seen in functions such as `linspace`, or with iterators such as `enumerate`, and `zip`. The special objects can mostly be used as if they were normal arrays.

Functions in Julia return values from their last expression or the `return` keyword instead of listing the names of variables to return in the function definition (see [The return Keyword](#) for details).

A Julia script may contain any number of functions, and all definitions will be externally visible when the file is loaded. Function definitions can be loaded from files outside the current working directory.

In Julia, reductions such as `sum`, `prod`, and `max` are performed over every element of an array when called with a single argument, as in `sum(A)`, even if A has more than one dimension.

In Julia, functions such as `sort` that operate column-wise by default (`sort(A)` is equivalent to `sort(A, 1)`) do not have special behavior for 1xN arrays; the argument is returned unmodified since it still performs `sort(A, 1)`. To sort a 1xN matrix like a vector, use `sort(A, 2)`.

In Julia, parentheses must be used to call a function with zero arguments, like in `rand()`.

Julia discourages the use of semicolons to end statements. The results of statements are not automatically printed (except at the interactive prompt), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output.

In Julia, if A and B are arrays, logical comparison operations like `A == B` do not return an array of booleans. Instead, use `A .== B`, and similarly

In Julia, the operators `&`, `|`, and `(xor)` perform the bitwise operations equivalent to `and`, `or`, and `xor` respectively in MATLAB, and have precedence similar to Python’s bitwise operators (unlike C). They can operate on scalars or element-wise across arrays and can be used to combine logical arrays, but note the difference in order of operations: parentheses may be required (e.g., to select elements of `A` equal to 1 or 2 use `(A .== 1) .| (A .== 2)`).

In Julia, the elements of a collection can be passed as arguments to a function using the splat operator `...`, as in `xs=[1,2]; f(xs...)`.

Julia’s `svd` returns singular values as a vector instead of as a dense diagonal matrix.

In Julia, `...` is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.

In both Julia and MATLAB, the variable `ans` is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, `ans` is not set when Julia code is run in non-interactive mode.

Julia’s `types` do not support dynamically adding fields at runtime, unlike MATLAB’s `classes`. Instead, use a `Dict`.

In Julia each module has its own global scope/namespace, whereas in MATLAB there is just one global scope.

In MATLAB, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x(x>3)` or in the statement `x(x>3) = []` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding translitera-

44.2 ~~to NOTEWORTHY DIFFERENCES FROM R~~ Using `filter!` reduces the use<sup>747</sup> of temporary arrays.

The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in `vertcat(A{:})` in MATLAB, is written using the splat operator in Julia, e.g. as `vcat(A...)`.

## 44.2 Noteworthy differences from R

One of Julia's goals is to provide an effective language for data analysis and statistical programming. For users coming to Julia from R, these are some noteworthy differences:

Julia's single quotes enclose characters, not strings.

Julia can create substrings by indexing into strings. In R, strings must be converted into character vectors before creating substrings.

In Julia, like Python but unlike R, strings can be created with triple quotes `""" ... """`. This syntax is convenient for constructing strings that contain line breaks.

In Julia, varargs are specified using the splat operator `...`, which always follows the name of a specific variable, unlike R, for which `...` can occur in isolation.

In Julia, modulus is `mod(a, b)`, not `a % b`. `%` in Julia is the remainder operator.

In Julia, not all data structures support logical indexing. Furthermore, logical indexing in Julia is supported only with vectors of length equal to the object being indexed. For example:

- In R, `c(1, 2, 3, 4)[c(TRUE, FALSE)]` is equivalent to `c(1, 3)`.

lent to `c(1, 3)`.

- In Julia, `[1, 2, 3, 4][[true, false]]` throws a [BoundsError](#).
- In Julia, `[1, 2, 3, 4][[true, false, true, false]]` produces `[1, 3]`.

Like many languages, Julia does not always allow operations on vectors of different lengths, unlike R where the vectors only need to share a common index range. For example, `c(1, 2, 3, 4) + c(1, 2)` is valid R but the equivalent `[1, 2, 3, 4] + [1, 2]` will throw an error in Julia.

Julia allows an optional trailing comma when that comma does not change the meaning of code. This can cause confusion among R users when indexing into arrays. For example, `x[1, ]` in R would return the first row of a matrix; in Julia, however, the comma is ignored, so `x[1, ] == x[1]`, and will return the first element. To extract a row, be sure to use `:`, as in `x[1, :]`.

Julia's [map](#) takes the function first, then its arguments, unlike `lapply(<structure>, function, ...)` in R. Similarly Julia's equivalent of `apply(X, MARGIN, FUN, ...)` in R is [mapslices](#) where the function is the first argument.

Multivariate apply in R, e.g. `mapply(choose, 11:13, 1:3)`, can be written as `broadcast(binomial, 11:13, 1:3)` in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions `binomial.(11:13, 1:3)`.

Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/`for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assign-

44.2 ~~NOTEWORTHY DIFFERENCES FROM R~~  
boxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`.

In Julia, `<-`, `<<-` and `->` are not assignment operators.

Julia's `->` creates an anonymous function, like Python.

Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.

Julia's `*` operator can perform matrix multiplication, unlike in R. If A and B are matrices, then `A * B` denotes a matrix multiplication in Julia, equivalent to R's `A %*% B`. In R, this same notation would perform an element-wise (Hadamard) product. To get the element-wise multiplication operation, you need to write `A .* B` in Julia.

Julia performs matrix transposition using the `.'` operator and conjugated transposition using the `'` operator. Julia's `A.'` is therefore equivalent to R's `t(A)`.

Julia does not require parentheses when writing `if` statements or `for/while` loops: use `for i in [1, 2, 3]` instead of `for (i in c(1, 2, 3))` and `if i == 1` instead of `if (i == 1)`.

Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.

Julia does not provide `nrow` and `ncol`. Instead, use `size(M, 1)` for `nrow(M)` and `size(M, 2)` for `ncol(M)`.

Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `c(1)` are the same. In Julia, they cannot be used interchangeably.

Julia's `diag` and `diagm` are not like R's.

an assignment operation: you cannot write `diag(M) = ones(n)`.

Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in [packages](#) under the [JuliaStats organization](#). For example:

- Functions pertaining to probability distributions are provided by the [Distributions package](#).
- The [DataFrames package](#) provides data frames.
- Generalized linear models are provided by the [GLM package](#).

Julia provides tuples and real hash tables, but not R-style lists. When returning multiple items, you should typically use a tuple: instead of `list(a = 1, b = 2)`, use `(1, 2)`.

Julia encourages users to write their own types, which are easier to use than S3 or S4 objects in R. Julia's multiple dispatch system means that `table(x::TypeA)` and `table(x::TypeB)` act like R's `table.TypeA(x)` and `table.TypeB(x)`.

In Julia, values are passed and assigned by reference. If a function modifies an array, the changes will be visible in the caller. This is very different from R and allows new functions to operate on large data structures much more efficiently.

In Julia, vectors and matrices are concatenated using `hcat`, `vcat` and `hvcat`, not `c`, `rbind` and `cbind` like in R.

In Julia, a range like `a:b` is not shorthand for a vector like in R, but is a specialized `AbstractRange` object that is used for iteration without high memory overhead. To convert a range into a vector, use `collect(a:b)`.

## 44.2 Julia's `maximum` and `minimum` are different from R's `pmax` and `pmin` respectively [51]

R, but both arguments need to have the same dimensions. While `maximum` and `minimum` replace `max` and `min` in R, there are important differences.

Julia's `sum`, `prod`, `maximum`, and `minimum` are different from their counterparts in R. They all accept one or two arguments. The first argument is an iterable collection such as an array. If there is a second argument, then this argument indicates the dimensions, over which the operation is carried out. For instance, let `A=[[1 2],[3 4]]` in Julia and `B=rbind(c(1,2),c(3,4))` be the same matrix in R. Then `sum(A)` gives the same result as `sum(B)`, but `sum(A, 1)` is a row vector containing the sum over each column and `sum(A, 2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where `sum(B, 1)=11` and `sum(B, 2)=12`. If the second argument is a vector, then it specifies all the dimensions over which the sum is performed, e.g., `sum(A, [1,2])=10`. It should be noted that there is no error checking regarding the second argument.

Julia has several functions that can mutate their arguments. For example, it has both `sort` and `sort!`.

In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.

Julia is eagerly evaluated and does not support R-style lazy evaluation. For most users, this means that there are very few unquoted expressions or column names.

Julia does not support the `NULL` type.

Julia lacks the equivalent of R's `assign` or `get`.

In Julia, `return` does not require parentheses.

dexing, like in the expression `x[x>3]` or in the statement `x = x[x>3]` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.

### 44.3 Noteworthy differences from Python

Julia requires `end` to end a block. Unlike Python, Julia has no `pass` keyword.

In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.

Julia's slice indexing includes the last element, unlike in Python. `a[2:3]` in Julia is `a[1:3]` in Python.

Julia does not support negative indexes. In particular, the last element of a list or array is indexed with `end` in Julia, not `-1` as in Python.

Julia's `for`, `if`, `while`, etc. blocks are terminated by the `end` keyword. Indentation level is not significant as it is in Python.

Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.

Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of [Performance Tips](#)).

#### 44.4 Julia Noteworthy differences from C/C++<sup>753</sup>

NumPy's are. This means `A = ones(4); B = A; B += 3` doesn't change values in `A`, it rather rebinds the name `B` to the result of the right-hand side `B = B + 3`, which is a new array. For in-place operation, use `B .=+ 3` (see also [dot operators](#)), explicit loops, or `InplaceOps.jl`.

Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand, the function `g(x=[1,2]) = push!(x,3)` returns `[1,2,3]` every time it is called as `g()`.

In Julia `%` is the remainder operator, whereas in Python it is the modulus.

#### 44.4 Noteworthy differences from C/C++

Julia arrays are indexed with square brackets, and can have more than one dimension `A[i, j]`. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the Julia documentation for the syntax for array construction (it has changed between versions).

In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.

Julia arrays are assigned by reference. After `A=B`, changing elements of `B` will modify `A` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `A = A + B` which rebinds the left-hand side to the result of the right-hand side expression.

Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to C/C++ (see relevant section of [Performance Tips](#)).

an array, the changes will be visible in the caller.

In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.

In Julia, literal numbers without a decimal point (such as `42`) create signed integers, of type `Int`, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as `Int64` (if `Int` is `Int32`), `Int128`, or the arbitrarily large `BigInt` type. There are no numeric literal suffixes, such as `L`, `LL`, `U`, `UL`, `ULL` to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with `0x` like C/C++), are unsigned. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the length of the literal, including leading 0s. For example, `0x0` and `0x00` have type `UInt8`, `0x000` and `0x0000` have type `UInt16`, then literals with 5 to 8 hex digits have type `UInt32`, 9 to 16 hex digits type `UInt64` and 17 to 32 hex digits type `UInt128`. This needs to be taken into account when defining hexadecimal masks, for example `~0xf == 0xf0` is very different from `~0x000f == 0xffff0`. 64 bit `Float64` and 32 bit `Float32` bit literals are expressed as `1.0` and `1.0f0` respectively. Floating point literals are rounded (and not promoted to the `BigFloat` type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with `0o`) and binary (prefixed with `0b`) literals are also treated as unsigned.

String literals can be delimited with either `"` or `""", """`. `"""` delimited literals can contain `"` characters without quoting it like `"\""`. String literals can have values of other variables or expressions interpolated into them, indicated by `$variablename` or `$(expression)`, which evaluates the variable name or the expression in the context of the function.

44.4 / ~~NOTE~~ WORTH ~~NATIONAL~~ DIFFERENCES FROM C/C++  
single-line comment (white is # in Julia)

#= indicates the start of a multiline comment, and =# ends it.

Functions in Julia return values from their last expression(s) or the `return` keyword. Multiple values can be returned from functions and assigned as tuples, e.g. `(a, b) = myfunction()` or `a, b = myfunction()`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `a = myfunction(&b)`).

Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output. In the REPL, ; can be used to suppress output. ; also has a different meaning within [ ], something to watch out for. ; can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.

In Julia, the operator (`xor`) performs the bitwise XOR operation, i.e. `^` in C/C++. Also, the bitwise operators do not have the same precedence as C++, so parenthesis may be required.

Julia's `^` is exponentiation (`pow`), not bitwise XOR as in C/C++ (use `,` or `xor`, in Julia)

Julia has two right-shift operators, `>>` and `>>>`. `>>>` performs an arithmetic shift, `>>` always performs a logical shift, unlike C/C++, where the meaning of `>>` depends on the type of the value being shifted.

Julia's `->` creates an anonymous function, it does not access a member via a pointer.

loops: use `for i in [1, 2, 3]` instead of `for (int i=1; i <= 3; i++)` and `if i == 1` instead of `if (i == 1)`.

Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.

Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while/ for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`, because of the operator precedence.

Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.

Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the `@` character, and have both a function-like syntax, `@mymacro(arg1, arg2, arg3)`, and a statement-like syntax, `@mymacro arg1 arg2 arg3`. The forms are interchangeable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the parallel `for` construct: `@parallel for i in 1:n; #= body =#; end`. Where the end of the macro construct may be unclear, use the function-like form.

44.4 JUNEWORTHY DIFFERENCES FROM C++  
The differences from C++ are expressed using the macro `@enum`(`name`, `value1`, `value2`, ...). For example: `@enum(Fruit, banana=1, apple, pear)`

By convention, functions that modify their arguments have a `!` at the end of the name, for example `push!`.

In C++, by default, you have static dispatch, i.e. you need to annotate a function as `virtual`, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only `this`, using the most-specific-declaration rule).



# Chapter 45

## Unicode Input

The following table lists Unicode characters that can be entered via tab completion of LaTeX-like abbreviations in the Julia REPL (and in various other editing environments). You can also get information on how to type a symbol by entering it in the REPL help, i.e. by typing ? and then entering the symbol in the REPL (e.g., by copy-paste from somewhere you saw the symbol).

### Warning

This table may appear to contain missing characters in the second column, or even show characters that are inconsistent with the characters as they are rendered in the Julia REPL. In these cases, users are strongly advised to check their choice of fonts in their browser and REPL environment, as there are known issues with glyphs in many fonts.



## Part IV

# Standard Library



# Chapter 46

## Essentials

### 46.1 Introduction

The Julia standard library contains a range of functions and macros appropriate for performing scientific and numerical computing, but is also as broad as those of many general purpose programming languages. Additional functionality is available from a growing collection of available packages. Functions are grouped by topic below.

Some general notes:

To use module functions, use `import Module` to import the module, and `Module.fn(x)` to use the functions.

Alternatively, `using Module` will import all exported `Module` functions into the current namespace.

By convention, function names ending with an exclamation point (!) modify their arguments. Some functions have both modifying (e.g., `sort!`) and non-modifying (`sort`) versions.

### 46.2 Getting Around

`|Base.exit`

764  
Base.quit  
Base.atexit  
Base.atreplinit  
Base.isinteractive  
Base.whos  
Base.summarysize  
Base.edit(::AbstractString, ::Integer)  
Base.edit(::Any)  
Base.@edit  
Base.less(::AbstractString)  
Base.less(::Any)  
Base.@less  
Base.clipboard(::Any)  
Base.clipboard()  
Base.reload  
Base.require  
Base.compilecache  
Base.\_\_precompile\_\_  
Base.include  
Base.include\_string  
Base.include\_dependency  
Base.Docs.apropos  
Base.which(::Any, ::Any)  
Base.which(::Symbol)  
Base.@which  
Base.methods  
Base.methodswith  
Base.@show  
Base.versioninfo  
Base.workspace  
ans

**module** – Keyword.

```
| module
```

**module** declares a Module, which is a separate global variable workspace. Within a module, you can control which names from other modules are visible (via importing), and specify which of your names are intended to be public (via exporting). Modules allow you to create top-level definitions without worrying about name conflicts when your code is used together with somebody else's. See the [manual section about modules](#) for more details.

Examples

```
module Foo
import Base.show
export MyType, foo

struct MyType
    x
end

bar(x) = 2x
foo(a::MyType) = bar(a.x) + 1
show(io::IO, a::MyType) = print(io, "MyType $(a.x)")
end
```

**source**

**export** – Keyword.

```
| export
```

CHAPTER 46. ESSENTIALS  
766 `export` is used within modules to tell Julia which names are available to the user. For example: `export foo` makes the name `foo` available when `using` the module. See the [manual section about modules](#) for details.

`source`

`import` – Keyword.

| `import`

`import Foo` will load the module or package `Foo`. Names from the imported `Foo` module can be accessed with dot syntax (e.g. `Foo.foo` to access the name `foo`). See the [manual section about modules](#) for details.

`source`

`using` – Keyword.

| `using`

`using Foo` will load the module or package `Foo` and make its `exported` names available for direct use. Names can also be used via dot syntax (e.g. `Foo.foo` to access the name `foo`), whether they are `exported` or not. See the [manual section about modules](#) for details. “”

`source`

`baremodule` – Keyword.

| `baremodule`

`baremodule` declares a module that does not contain `using Base` or a definition of `eval`. It does still import `Core`.

`source`

`function` – Keyword.

Functions are defined with the `function` keyword:

```
function add(a, b)
    return a + b
end
```

Or the short form notation:

```
add(a, b) = a + b
```

The use of the `return` keyword is exactly the same as in other languages, but is often optional. A function without an explicit `return` statement will return the last expression in the function body.

### source

`macro` – Keyword.

```
macro
```

`macro` defines a method to include generated code in the final body of a program. A macro maps a tuple of arguments to a returned expression, and the resulting expression is compiled directly rather than requiring a runtime `eval` call. Macro arguments may include expressions, literal values, and symbols. For example:

Examples

```
julia> macro sayhello(name)

        return :( println("Hello, ", $name, "!") )

end
@sayhello (macro with 1 method)
```

```
julia> @sayhello "Charlie"  
Hello, Charlie!
```

`source`

`return` – Keyword.

```
| return
```

`return` can be used in function bodies to exit early and return a given value, e.g.

```
function compare(a, b)  
    a == b && return "equal to"  
    a < b ? "less than" : "greater than"  
end
```

In general you can place a `return` statement anywhere within a function body, including within deeply nested loops or conditionals, but be careful with `do` blocks. For example:

```
function test1(xs)  
    for x in xs  
        iseven(x) && return 2x  
    end  
end  
  
function test2(xs)  
    map(xs) do x  
        iseven(x) && return 2x  
        x  
    end  
end
```

46.3 In **KEY WORDS**, the return breaks out of its enclosing function as shown as it hits an even number, so `test1([5,6,7])` returns 12.

You might expect the second example to behave the same way, but in fact the `return` there only breaks out of the inner function (inside the `do` block) and gives a value back to `map`. `test2([5,6,7])` then returns `[5,12,7]`.

**source**

`do` – Keyword.

| `do`

The `do` keyword creates an anonymous function. For example:

| `map(1:10) do x`  
|     `2x`  
| `end`

is equivalent to `map(x->2x, 1:10)`.

Use multiple arguments like so:

| `map(1:10, 11:20) do x, y`  
|     `x + y`  
| `end`

**source**

`begin` – Keyword.

| `begin`

`begin...end` denotes a block of code.

| `begin`  
|     `println("Hello, ")`

```
770 |     println("World!")  
| end
```

CHAPTER 46. ESSENTIALS

Usually **begin** will not be necessary, since keywords such as **function** and **let** implicitly begin blocks of code. See also **;**.

**source**

**end** – Keyword.

```
| end
```

**end** marks the conclusion of a block of expressions, for example **module**, **struct**, **mutable struct**, **begin**, **let**, **for** etc. **end** may also be used when indexing into an array to represent the last index of a dimension.

Examples

```
julia> A = [1 2; 3 4]  
2×2 Array{Int64,2}:  
 1 2  
 3 4  
  
julia> A[end, :]  
2-element Array{Int64,1}:  
 3  
 4
```

**source**

**let** – Keyword.

```
| let
```

**let** statements allocate new variable bindings each time they run. Whereas an assignment modifies an existing value location, **let** creates new locations. This difference is only detectable in the case of variables that out-

46.3 ~~KEYWORDS~~<sup>771</sup> via closures. The `let` syntax accepts a comma-separated series of assignments and variable names:

```
let var1 = value1, var2, var3 = value3
    code
end
```

The assignments are evaluated in order, with each right-hand side evaluated in the scope before the new variable on the left-hand side has been introduced. Therefore it makes sense to write something like `let x = x`, since the two `x` variables are distinct and have separate storage.

`source`

`if` – Keyword.

```
| if/elseif/else
```

`if/elseif/else` performs conditional evaluation, which allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `if/elseif/else` conditional syntax:

```
if x < y
    println("x is less than y")
elseif x > y
    println("x is greater than y")
else
    println("x is equal to y")
end
```

If the condition expression `x < y` is true, then the corresponding block is evaluated; otherwise the condition expression `x > y` is evaluated, and if it is true, the corresponding block is evaluated; if neither expression

772 is true, the `else` block is evaluated. The `else` block is optional, and as many `elseif` blocks as desired can be used.

### source

`for` – Keyword.

```
| for
```

`for` loops repeatedly evaluate the body of the loop by iterating over a sequence of values.

Examples

```
julia> for i in [1, 4, 0]
           println(i)
       end
1
4
0
```

### source

`while` – Keyword.

```
| while
```

`while` loops repeatedly evaluate a conditional expression, and continues evaluating the body of the while loop so long as the expression remains `true`. If the condition expression is false when the while loop is first reached, the body is never evaluated.

Examples

```
julia> i = 1
1
```

```
julia> while i < 5  
    println(i)  
    i += 1  
end  
1  
2  
3  
4
```

source

**break** – Keyword.

| **break**

**break** breaks out of a loop immediately.

Examples

```
julia> i = 0  
0  
  
julia> while true  
    i += 1  
    i > 5 && break  
    println(i)
```

1  
2  
3  
4  
5

source

**continue** – Keyword.

| continue

**continue** skips the rest of the current loop, then carries on looping.

Examples

```
julia> for i = 1:6  
  
        iseven(i) && continue  
  
        println(i)  
  
    end
```

1  
3  
5

source

**try** – Keyword.

| try/catch

A **try/catch** statement allows for **Exceptions** to be tested for. For example, a customized square root function can be written to automatically

46.3cd **KEYWORD**s real or complex square root method on demand using **EX5**  
ceptions:

```
f(x) = try
      sqrt(x)
catch
      sqrt(complex(x, 0))
end
```

try/catch statements also allow the **Exception** to be saved in a variable, e.g. **catch** *y*.

The **catch** clause is not strictly necessary; when omitted, the default return value is **nothing**. The power of the try/catch construct lies in the ability to unwind a deeply nested computation immediately to a much higher level in the stack of calling functions.

**source**

**finally** – Keyword.

```
| finally
```

**finally** provides a way to run some code when a given block of code exits, regardless of how it exits. For example, here is how we can guarantee that an opened file is closed:

```
f = open("file")
try
  operate_on_file(f)
finally
  close(f)
end
```

When control leaves the **try** block (for example due to a **return**, or just finishing normally), **close(f)** will be executed. If the **try** block exits due

776 to an exception, the exception will continue propagating. **finally** blocks may be combined with **try** and **finally** as well. In this case the **finally** block will run after **catch** has handled the error.

### source

**quote** – Keyword.

```
| quote
```

**quote** creates multiple expression objects in a block without using the explicit **Expr** constructor. For example:

```
ex = quote
    x = 1
    y = 2
    x + y
end
```

Unlike the other means of quoting, `:(` ... `)`, this form introduces **QuoteNode** elements to the expression tree, which must be considered when directly manipulating the tree. For other purposes, `:(` ... `)` and **quote** ... **end** blocks are treated identically.

### source

**local** – Keyword.

```
| local
```

**local** introduces a new local variable.

Examples

```
julia> function foo(n)
    x = 0
```

```
for i = 1:n  
  
    local x # introduce a loop-local x  
  
    x = i  
  
end  
  
x  
  
end  
foo (generic function with 1 method)  
  
julia> foo(10)  
0  
  
source
```

**global** – Keyword.

```
| global
```

**global** x makes x in the current scope and its inner scopes refer to the global variable of that name.

Examples

```
julia> z = 3  
3  
  
julia> function foo()  
  
    global z = 6 # use the z variable defined outside  
    ↪ foo
```

```
    end  
foo (generic function with 1 method)
```

```
julia> foo()
```

```
6
```

```
julia> z
```

```
6
```

**source**

**const** – Keyword.

```
| const
```

**const** is used to declare global variables which are also constant. In almost all code (and particularly performance sensitive code) global variables should be declared constant in this way.

```
| const x = 5
```

Multiple variables can be declared within a single **const**:

```
| const y, z = 7, 11
```

Note that **const** only applies to one = operation, therefore **const x = y = 1** declares x to be constant but not y. On the other hand, **const x = const y = 1** declares both x and y as constants.

Note that "constant-ness" is not enforced inside containers, so if x is an array or dictionary (for example) you can still add and remove elements.

Technically, you can even redefine **const** variables, although this will generate a warning from the compiler. The only strict requirement is that

46.3th KEYWORD<sup>779</sup> The variable does not change, which is why **const** variables<sup>779</sup> are much faster than regular globals.

**source**

**struct** – Keyword.

| **struct**

The most commonly used kind of type in Julia is a struct, specified as a name and a set of fields.

```
| struct Point  
|   x  
|   y  
end
```

Fields can have type restrictions, which may be parameterized:

```
| struct Point{X}  
|   x::X  
|   y::Float64  
end
```

A struct can also declare an abstract super type via <: syntax:

```
| struct Point <: AbstractPoint  
|   x  
|   y  
end
```

**structs** are immutable by default: an instance of one of these types cannot be modified after construction. Use **mutable struct** instead to declare a type whose instances can be modified.

See the manual section on [Composite Types](#) for more details, such as how to define constructors.

`mutable struct` – Keyword.

| `mutable struct`

`mutable struct` is similar to `struct`, but additionally allows the fields of the type to be set after construction. See the manual section on [Composite Types](#) for more information.

[source](#)

`abstract type` – Keyword.

| `abstract type`

`abstract type` declares a type that cannot be instantiated, and serves only as a node in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. Abstract types form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations. For example:

| `abstract type Number end`  
| `abstract type Real <: Number end`

`Number` has no supertype, whereas `Real` is an abstract subtype of `Number`.

[source](#)

`primitive type` – Keyword.

| `primitive type`

`primitive type` declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some example built-in primitive type declarations:

```
primitive type Char 32 end
```

```
primitive type Bool <: Integer 8 end
```

The number after the name indicates how many bits of storage the type requires. Currently, only sizes that are multiples of 8 bits are supported. The `Bool` declaration shows how a primitive type can be optionally declared to be a subtype of some supertype.

#### source

`...` – Keyword.

```
| ...
```

The “splat” operator, `...`, represents a sequence of arguments. `...` can be used in function definitions, to indicate that the function accepts an arbitrary number of arguments. `...` can also be used to apply a function to a sequence of arguments.

#### Examples

```
julia> add(xs...) = reduce(+, xs)
add (generic function with 1 method)
```

```
julia> add(1, 2, 3, 4, 5)
```

```
15
```

```
julia> add([1, 2, 3]...)
```

```
6
```

```
julia> add(7, 1:100..., 1000:1100...)
```

```
111107
```

#### source

`;` – Keyword.

; has a similar role in Julia as in many C-like languages, and is used to delimit the end of the previous statement. ; is not necessary after new lines, but can be used to separate statements on a single line or to join statements into a single expression. ; is also used to suppress output printing in the REPL and similar interfaces.

### Examples

```
julia> function foo()
           x = "Hello, "; x *= "World!"
           return x
end
foo (generic function with 1 method)

julia> bar() = (x = "Hello, Mars!"; return x)
bar (generic function with 1 method)

julia> foo();
          

julia> bar()
"Hello, Mars!"
```

**source**

## 46.4 Base Modules

```
Base.BLAS
Base.Dates
Base.Distributed
Base.Docs
Base.Iterators
```

```
Base.LibGit2
Base.Libc
Base.Libdl
Base.LinAlg
Base.Markdown
Base.Meta
Base.Pkg
Base.Serializer
Base.SparseArrays
Base.StackTraces
Base.Sys
Base.Threads
```

## 46.5 All Objects

`Core.==` – Function.

```
==(x,y) -> Bool
(x,y) -> Bool
```

Determine whether `x` and `y` are identical, in the sense that no program could distinguish them. First it compares the types of `x` and `y`. If those are identical, it compares mutable objects by address in memory and immutable objects (such as numbers) by contents at the bit level. This function is sometimes called "egal".

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a == b
true
```

784 | **julia>** a === b

false

**julia>** a === a

true

**source**

**Core.isa** – Function.

| **isa(x, type)** -> Bool

Determine whether x is of the given type. Can also be used as an infix operator, e.g. x isa type.

Examples

**julia>** isa(1, Int)

true

**julia>** isa(1, Matrix)

false

**julia>** isa(1, Char)

false

**julia>** isa(1, Number)

true

**julia>** 1 isa Number

true

**source**

**Base.isequal** – Function.

CHAPTER 46. ESSENTIALS

Similar to `==`, except treats all floating-point `NaN` values as equal to each other, and treats `-0.0` as unequal to `0.0`. The default implementation of `isequal` calls `==`, so if you have a type that doesn't have these floating-point subtleties then you probably only need to define `==`.

`isequal` is the comparison function used by hash tables (`Dict`). `isequal(x, y)` must imply that `hash(x) == hash(y)`.

This typically means that if you define your own `==` function then you must define a corresponding `hash` (and vice versa). Collections typically implement `isequal` by calling `isequal` recursively on all contents.

Scalar types generally do not need to implement `isequal` separate from `==`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `isnan`, `signbit`, and `==`).

## Examples

```
julia> isequal([1., NaN], [1., NaN])
true
```

```
julia> [1., NaN] == [1., NaN]
false
```

```
julia> 0.0 == -0.0
true
```

```
julia> isequal(0.0, -0.0)
false
```

[source](#)

786 | `isequal(x::Nullable, y::Nullable)`

CHAPTER 46. ESSENTIALS

If neither `x` nor `y` is null, compare them according to their values (i.e. `isequal(get(x), get(y))`). Else, return `true` if both arguments are null, and `false` if one is null but not the other: nulls are considered equal.

Examples

```
julia> isequal(Nullable(5), Nullable(5))
```

```
true
```

```
julia> isequal(Nullable(5), Nullable(4))
```

```
false
```

```
julia> isequal(Nullable(5), Nullable())
```

```
false
```

```
julia> isequal(Nullable(), Nullable())
```

```
true
```

`source`

`Base.isless` – Function.

```
| isless(x, y)
```

Test whether `x` is less than `y`, according to a canonical total order. Values that are normally unordered, such as `NaN`, are ordered in an arbitrary but consistent fashion. This is the default comparison used by `sort`. Non-numeric types with a canonical total order should implement this function. Numeric types only need to implement it if they have special values such as `NaN`.

`source`

`Base.isless` – Method.

If neither `x` nor `y` is null, compare them according to their values (i.e. `isless(get(x), get(y))`). Else, return `true` if only `y` is null, and `false` otherwise: nulls are always considered greater than non-nulls, but not greater than another null.

Examples

```
julia> isless(Nullable(6), Nullable(5))
false
```

```
julia> isless(Nullable(5), Nullable(6))
true
```

```
julia> isless(Nullable(5), Nullable(4))
false
```

```
julia> isless(Nullable(5), Nullable())
true
```

```
julia> isless(Nullable(), Nullable())
false
```

```
julia> isless(Nullable(), Nullable(5))
false
```

`source`

`Base.ifelse` – Function.

```
| ifelse(condition::Bool, x, y)
```

Return `x` if `condition` is `true`, otherwise return `y`. This differs from `?` or `if` in that it is an ordinary function, so all the arguments are evaluated first.

788 In some cases, using `ifelse` instead of an `if` statement can eliminate the branch in generated code and provide higher performance in tight loops.

Examples

```
| julia> ifelse(1 > 2, 1, 2)
| 2
```

`source`

`Base.lexcmp` – Function.

```
| lexcmp(x, y)
```

Compare `x` and `y` lexicographically and return -1, 0, or 1 depending on whether `x` is less than, equal to, or greater than `y`, respectively. This function should be defined for lexicographically comparable types, and `lexless` will call `lexcmp` by default.

Examples

```
| julia> lexcmp("abc", "abd")
| -1
```

```
| julia> lexcmp("abc", "abc")
| 0
```

`source`

`Base.lexless` – Function.

```
| lexless(x, y)
```

Determine whether `x` is lexicographically less than `y`.

Examples

46.5 | ALL OBJECTS  
| **julia>** lexless("abc", "abd")  
| true

789

source

[Core.typeof](#) – Function.

| **typeof(x)**

Get the concrete type of x.

source

[Core.tuple](#) – Function.

| **tuple(xs...)**

Construct a tuple of the given objects.

Examples

| **julia> tuple(1, 'a', pi)**  
| (1, 'a', π = 3.1415926535897...)

source

[Base.ntuple](#) – Function.

| **ntuple(f::Function, n::Integer)**

Create a tuple of length n, computing each element as f(i), where i is the index of the element.

| **julia> ntuple(i -> 2\*i, 4)**  
| (2, 4, 6, 8)

source

[Base.object\\_id](#) – Function.

Get a hash value for `x` based on object identity. `object_id(x)==object_id(y)` if `x === y`.

`source`

`Base.hash` – Function.

|`hash(x[, h::UInt])`

Compute an integer hash code such that `isequal(x, y)` implies `hash(x)==hash(y)`.

The optional second argument `h` is a hash code to be mixed with the result.

New types should implement the 2-argument form, typically by calling the 2-argument `hash` method recursively in order to mix hashes of the contents with each other (and with `h`). Typically, any type that implements `hash` should also implement its own `==` (hence `isequal`) to guarantee the property mentioned above.

`source`

`Base.finalizer` – Function.

|`finalizer(f, x)`

Register a function `f(x)` to be called when there are no program-accessible references to `x`, and return `x`. The type of `x` must be a `mutable struct`, otherwise the behavior of this function is unpredictable.

`source`

`Base.finalize` – Function.

|`finalize(x)`

Immediately run finalizers registered for object `x`.

`source`

```
| copy(x)
```

Create a shallow copy of `x`: the outer structure is copied, but not all internal values. For example, copying an array produces a new array with identically-same elements as the original.

[source](#)

[Base . deepcopy](#) – Function.

```
| deepcopy(x)
```

Create a deep copy of `x`: everything is copied recursively, resulting in a fully independent object. For example, deep-copying an array produces a new array whose elements are deep copies of the original elements. Calling `deepcopy` on an object should generally have the same effect as serializing and then deserializing it.

As a special case, functions can only be actually deep-copied if they are anonymous, otherwise they are just copied. The difference is only relevant in the case of closures, i.e. functions which may contain hidden internal references.

While it isn't normally necessary, user-defined types can override the default `deepcopy` behavior by defining a specialized version of the function `deepcopy_internal(x::T, dict::ObjectIDDict)` (which shouldn't otherwise be used), where `T` is the type to be specialized for, and `dict` keeps track of objects copied so far within the recursion. Within the definition, `deepcopy_internal` should be used in place of `deepcopy`, and the `dict` variable should be updated as appropriate before returning.

[source](#)

[Core . isdefined](#) – Function.

```
792| isdefined(m::Module, s::Symbol)
    | isdefined(object, s::Symbol)
    | isdefined(object, index::Int)
```

CHAPTER 46. ESSENTIALS

Tests whether an assignable location is defined. The arguments can be a module and a symbol or a composite object and field name (as a symbol) or index.

**source**

[Base.@isdefined](#) – Macro.

```
| @isdefined s -> Bool
```

Tests whether variable `s` is defined in the current scope.

Examples

```
julia> function f()
    println(@isdefined x)
    x = 3
    println(@isdefined x)
end
f (generic function with 1 method)

julia> f()
false
true

source
```

[Base.convert](#) – Function.

Convert x to a value of type T.

If T is an `Integer` type, an `InexactError` will be raised if x is not representable by T, for example if x is not integer-valued, or is outside the range supported by T.

Examples

```
julia> convert(Int, 3.0)
3

julia> convert(Int, 3.5)
ERROR: InexactError: convert(Int64, 3.5)
Stacktrace:
 [1] convert(::Type{Int64}, ::Float64) at ./float.jl:703
```

If T is a `AbstractFloat` or `Rational` type, then it will return the closest value to x representable by T.

```
julia> x = 1/3
0.3333333333333333

julia> convert(Float32, x)
0.33333334f0

julia> convert(Rational{Int32}, x)
1 // 3

julia> convert(Rational{Int64}, x)
6004799503160661 // 18014398509481984
```

If T is a collection type and x a collection, the result of `convert(T, x)` may alias x.

794 | **julia>** x = Int[1,2,3];

CHAPTER 46. ESSENTIALS

**julia>** y = convert(Vector{Int}, x);

**julia>** y === x

true

Similarly, if T is a composite type and x a related instance, the result of convert(T, x) may alias part or all of x.

**julia>** x = sparse(1.0I, 5, 5);

**julia>** typeof(x)

SparseMatrixCSC{Float64, Int64}

**julia>** y = convert(SparseMatrixCSC{Float64, Int64}, x);

**julia>** z = convert(SparseMatrixCSC{Float32, Int64}, y);

**julia>** y === x

true

**julia>** z === x

false

**julia>** z.colptr === x.colptr

true

**source**

[Base.promote](#) – Function.

| promote(xs...)

46.5. `promote` – Converts arguments to a common type, and return them all (as a tuple).<sup>105</sup>

If no arguments can be converted, an error is raised.

Examples

```
| julia> promote(Int8(1), Float16(4.5), Float32(4.1))
| (1.0f0, 4.5f0, 4.1f0)
```

source

`Base.oftype` – Function.

```
| oftype(x, y)
```

Convert `y` to the type of `x` (`convert(typeof(x), y)`).

source

`Base.widen` – Function.

```
| widen(x)
```

If `x` is a type, return a “larger” type (for numeric types, this will be a type with at least as much range and precision as the argument, and usually more). Otherwise `x` is converted to `widen(typeof(x))`.

Examples

```
| julia> widen(Int32)
```

Int64

```
| julia> widen(1.5f0)
```

1.5

source

`Base.identity` – Function.

```
| identity(x)
```

Examples

```
| julia> identity("Well, what did you expect?")
| "Well, what did you expect?"
```

source

## 46.6 Dealing with Types

`Base.supertype` – Function.

```
| supertype(T::DataType)
```

Return the supertype of `DataType T`.

```
| julia> supertype(Int32)
| Signed
```

source

`Core.:(<:` – Function.

```
| <:(T1, T2)
```

Subtype operator: returns `true` if and only if all values of type `T1` are also of type `T2`.

```
| julia> Float64 <: AbstractFloat
| true
```

```
| julia> Vector{Int} <: AbstractArray
| true
```

```
| julia> Matrix{Float64} <: Matrix{AbstractFloat}
| false
```

`Base.>:` – Function.

```
| >:(T1, T2)
```

Supertype operator, equivalent to `T2 <: T1`.

`source`

`Base.subtypes` – Function.

```
| subtypes(T::DataType)
```

Return a list of immediate subtypes of `DataType T`. Note that all currently loaded subtypes are included, including those not visible in the current module.

Examples

```
julia> subtypes(Integer)
3-element Array{Union{DataType, UnionAll},1}:
 Bool
 Signed
 Unsigned
```

`source`

`Base.typemin` – Function.

```
| typemin(T)
```

The lowest value representable by the given (real) numeric `DataType T`.

Examples

```
julia> typemin(Float16)
-Inf16
```

```
798 | julia> typemin(Float32)  
| -Inf32
```

CHAPTER 46. ESSENTIALS

[source](#)

[Base.typemax](#) – Function.

```
| typemax(T)
```

The highest value representable by the given (real) numeric `DataType`.

[source](#)

[Base.realmin](#) – Function.

```
| realmin(T)
```

The smallest in absolute value non-subnormal value representable by the given floating-point `DataType T`.

[source](#)

[Base.realmax](#) – Function.

```
| realmax(T)
```

The highest finite value representable by the given floating-point `DataType T`.

Examples

```
julia> realmax(Float16)  
Float16(6.55e4)
```

```
julia> realmax(Float32)  
3.4028235f38
```

[source](#)

[Base.maxintfloat](#) – Function.

The largest integer losslessly representable by the given floating-point DataType T.

**source**

```
| maxintfloat(T, S)
```

The largest integer losslessly representable by the given floating-point DataType T that also does not exceed the maximum integer representable by the integer DataType S.

**source**

**Base.sizeof** – Method.

```
| sizeof(T)
```

Size, in bytes, of the canonical binary representation of the given DataType T, if any.

Examples

```
julia> sizeof(Float32)
```

```
4
```

```
julia> sizeof(Complex128)
```

```
16
```

If T does not have a specific size, an error is thrown.

```
julia> sizeof(Base.LinAlg.LU)
```

```
ERROR: argument is an abstract type; size is indeterminate
```

```
Stacktrace:
```

```
[...]
```

**source**

```
| eps(::Type{T}) where T<:AbstractFloat  
| eps()
```

Returns the machine epsilon of the floating point type  $T$  ( $T = \text{Float64}$  by default). This is defined as the gap between 1 and the next largest value representable by  $T$ , and is equivalent to `eps(one(T))`.

```
julia> eps()  
2.220446049250313e-16
```

```
julia> eps(Float32)  
1.1920929f-7
```

```
julia> 1.0 + eps()  
1.000000000000002
```

```
julia> 1.0 + eps()/2  
1.0
```

[source](#)

```
| eps(x::AbstractFloat)
```

Returns the unit in last place (ulp) of  $x$ . This is the distance between consecutive representable floating point values at  $x$ . In most cases, if the distance on either side of  $x$  is different, then the larger of the two is taken, that is

```
| eps(x) == max(x-prevfloat(x), nextfloat(x)-x)
```

## 46.6. The `eps` Function Without Types

The `eps` function without types are the smallest and largest finite values (801) `nextfloat(-Inf)` and `prevfloat(Inf)` for `Float64`, which round to the smaller of the values.

The rationale for this behavior is that `eps` bounds the floating point rounding error. Under the default `RoundNearest` rounding mode, if  $y$  is a real number and  $x$  is the nearest floating point number to  $y$ , then

$$|y - x| \leq \text{eps}(x)/2.$$

```
julia> eps(1.0)
```

```
2.220446049250313e-16
```

```
julia> eps(prevfloat(2.0))
```

```
2.220446049250313e-16
```

```
julia> eps(2.0)
```

```
4.440892098500626e-16
```

```
julia> x = prevfloat(Inf)      # largest finite Float64
```

```
1.7976931348623157e308
```

```
julia> x + eps(x)/2          # rounds up
```

```
Inf
```

```
julia> x + prevfloat(eps(x)/2) # rounds down
```

```
1.7976931348623157e308
```

```
source
```

`Base.promote_type` – Function.

```
| promote_type(type1, type2)
```

802 Promotion refers to converting values of mixed types to a common type. `promote_type` represents the default promotion behavior in Julia when operators (usually mathematical) are given arguments of differing types. `promote_type` generally tries to return a type which can at least approximate most values of either input type without excessively widening. Some loss is tolerated; for example, `promote_type(Int64, Float64)` returns `Float64` even though strictly, not all `Int64` values can be represented exactly as `Float64` values.

```
julia> promote_type(Int64, Float64)
```

```
Float64
```

```
julia> promote_type(Int32, Int64)
```

```
Int64
```

```
julia> promote_type(Float32, BigInt)
```

```
BigFloat
```

```
julia> promote_type(Int16, Float16)
```

```
Float16
```

```
julia> promote_type(Int64, Float16)
```

```
Float16
```

```
julia> promote_type(Int8, UInt16)
```

```
UInt16
```

`source`

`Base.promote_rule` – Function.

```
| promote_rule(type1, type2)
```

46.6 **DEFINING WITH TYPES** Should be used by `promote` when given values of types `type1` and `type2`. This function should not be called directly, but should have definitions added to it for new types as appropriate.

`source`

`Core.getfield` – Function.

```
| getfield(value, name::Symbol)
```

Extract a named field from a `value` of composite type. The syntax `a.b` calls `getfield(a, :b)`.

Examples

```
julia> a = 1//2
1//2
```

```
julia> getfield(a, :num)
1
```

```
julia> a.num
1
```

`source`

`Core.setfield!` – Function.

```
| setfield!(value, name::Symbol, x)
```

Assign `x` to a named field in `value` of composite type. The syntax `a.b = c` calls `setfield!(a, :b, c)`. `value` must be mutable.

Examples

```
julia> mutable struct MyMutableStruct
```

```
end
```

```
julia> a = MyMutableStruct(1);
```

```
julia> setfield!(a, :field, 2);
```

```
julia> getfield(a, :field)
```

```
2
```

```
julia> a = 1//2
```

```
1//2
```

```
julia> setfield!(a, :num, 3);
```

```
ERROR: type Rational is immutable
```

```
source
```

`Base.fieldoffset` – Function.

```
| fieldoffset(type, i)
```

The byte offset of field `i` of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

```
julia> structinfo(T) = [(fieldoffset(T,i), fieldname(T,i),
→ fieldtype(T,i)) for i = 1:fieldcount(T)];
```

```
julia> structinfo(Base.Filesystem.StatStruct)
```

```
12-element Array{Tuple{UInt64,Symbol,DataType},1}:
```

```
 (0x0000000000000000, :device, UInt64)
```

```
 (0x0000000000000008, :inode, UInt64)
```

46.6 DEALING WITH TYPES  
(0x0000000000000010, :mode, UInt64)  
(0x0000000000000018, :nlink, Int64)  
(0x0000000000000020, :uid, UInt64)  
(0x0000000000000028, :gid, UInt64)  
(0x0000000000000030, :rdev, UInt64)  
(0x0000000000000038, :size, Int64)  
(0x0000000000000040, :blksize, Int64)  
(0x0000000000000048, :blocks, Int64)  
(0x0000000000000050, :mtime, Float64)  
(0x0000000000000058, :ctime, Float64)

805

source

[Core.fieldtype](#) – Function.

| `fieldtype(T, name::Symbol | index::Int)`

Determine the declared type of a field (specified by name or index) in a composite DataType T.

Examples

```
julia> struct Foo

        x::Int64

        y::String

    end

julia> fieldtype(Foo, :x)
Int64

julia> fieldtype(Foo, 2)
String
```

[Base.isimmutable](#) – Function.

```
| isimmutable(v)
```

Return `true` iff value `v` is immutable. See [Mutable Composite Types](#) for a discussion of immutability. Note that this function works on values, so if you give it a type, it will tell you that a value of `DataType` is mutable.

Examples

```
julia> isimmutable(1)
true

julia> isimmutable([1,2])
false
```

[source](#)

[Base.isbits](#) – Function.

```
| isbits(T)
```

Return `true` if `T` is a “plain data” type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as [UInt8](#), [Float64](#), and [Complex{Float64}](#).

Examples

```
julia> isbits(Complex{Float64})
true

julia> isbits(Complex)
false
```

[source](#)

```
| isconcrete(T)
```

Determine whether T is a concrete type, meaning it can have direct instances (values x such that `typeof(x) === T`).

Examples

```
julia> isconcrete(Complex)
false

julia> isconcrete(Complex{Float32})
true

julia> isconcrete(Vector{Complex})
true

julia> isconcrete(Vector{Complex{Float32}})
true

julia> isconcrete(Union{})
false

julia> isconcrete(Union{Int, String})
false
```

source

`Base.typejoin` – Function.

```
| typejoin(T, S)
```

Compute a type that contains both T and S.

source

`Base.typeintersect` – Function.

CHAPTER 46. ESSENTIALS

```
| typeintersect(T, S)
```

Compute a type that contains the intersection of T and S. Usually this will be the smallest such type or one close to it.

`source`

`Base.instances` – Function.

```
| instances(T::Type)
```

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `@enum`).

Example

```
julia> @enum Color red blue green  
  
julia> instances(Color)  
(red::Color = 0, blue::Color = 1, green::Color = 2)
```

`source`

## 46.7 Special Types

`Core.Void` – Type.

```
| Void
```

A type with no fields that is the type `nothing`.

`source`

`Core.Any` – Type.

```
| Any::DataType
```

46.7 Any is the **SPECIAL TYPES** of all types. It has the defining property `isa(x, Any)`<sup>809</sup> `true` for any `x`. `Any` therefore describes the entire universe of possible values. For example `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other integer types.

`source`

`Base.Enums.@enum` – Macro.

```
| @enum EnumName[::BaseType] value1[=x] value2[=y]
```

Create an `Enum{BaseType}` subtype with name `EnumName` and enum member values of `value1` and `value2` with optional assigned values of `x` and `y`, respectively. `EnumName` can be used just like other types and enum member values as regular values, such as

Examples

```
| julia> @enum Fruit apple=1 orange=2 kiwi=3
```

```
| julia> f(x::Fruit) = "I'm a Fruit with value: $(Int(x))"
| f (generic function with 1 method)
```

```
| julia> f(apple)
| "I'm a Fruit with value: 1"
```

`BaseType`, which defaults to `Int32`, must be a primitive subtype of `Integer`. Member values can be converted between the enum type and `BaseType`. `read` and `write` perform these conversions automatically.

`source`

`Core.Union` – Type.

```
| Union{Types...}
```

810 A type union is an abstract type which includes argument types. The empty union `Union{}` is the bottom type of Julia.

Examples

```
julia> IntOrString = Union{Int,AbstractString}
Union{Int64, AbstractString}

julia> 1 :: IntOrString
1

julia> "Hello!" :: IntOrString
"Hello!"

julia> 1.0 :: IntOrString
ERROR: TypeError: in typeassert, expected Union{Int64,
→ AbstractString}, got Float64
```

`source`

`Union{}` – Keyword.

`Union{}`

`Union{}`, the empty `Union` of types, is the type that has no values. That is, it has the defining property `isa(x, Union{}) == false` for any `x`. `Base.Bottom` is defined as its alias and the type of `Union{}` is `Core.TypeofBottom`.

Examples

```
julia> isa(nothing, Union{})
false
```

`source`

`Core.UnionAll` – Type.

A union of types over all values of a type parameter. UnionAll is used to describe parametric types where the values of some parameters are not known.

Examples

```
julia> typeof(Vector)
UnionAll

julia> typeof(Vector{Int})
DataType

source
```

[Core.Tuple](#) – Type.

| Tuple{Types...}

Tuples are an abstraction of the arguments of a function – without the function itself. The salient aspects of a function's arguments are their order and their types. Therefore a tuple type is similar to a parameterized immutable type where each parameter is the type of one field. Tuple types may have any number of parameters.

Tuple types are covariant in their parameters: `Tuple{Int}` is a subtype of `Tuple{Any}`. Therefore `Tuple{Any}` is considered an abstract type, and tuple types are only concrete if their parameters are. Tuples do not have field names; fields are only accessed by index.

See the manual section on [Tuple Types](#).

source

[Base.Val](#) – Type.

Return `Val{c}()`, which contains no run-time data. Types like this can be used to pass the information between functions through the value `c`, which must be an `isbits` value. The intent of this construct is to be able to dispatch on constants directly (at compile time) without having to test the value of the constant at run time.

### Examples

```
julia> f(::Val{true}) = "Good"
f (generic function with 1 method)

julia> f(::Val{false}) = "Bad"
f (generic function with 2 methods)

julia> f(Val(true))
"Good"

source
```

[Core.Vararg](#) – Type.

### Vararg{T,N}

The last parameter of a tuple type `Tuple` can be the special type `Vararg`, which denotes any number of trailing elements. The type `Vararg{T,N}` corresponds to exactly `N` elements of type `T`. `Vararg{T}` corresponds to zero or more elements of type `T`. `Vararg` tuple types are used to represent the arguments accepted by varargs methods (see the section on [Varargs Functions](#) in the manual.)

### Examples

```
julia> mytupletype = Tuple{AbstractString,Vararg{Int}}
Tuple{AbstractString,Vararg{Int64,N} where N}
```

```
julia> isa(("1",), mytupletype)
true

julia> isa(("1",1), mytupletype)
true

julia> isa(("1",1,2), mytupletype)
true

julia> isa(("1",1,2,3.0), mytupletype)
false
```

source

## 46.8 Generic Functions

[Core.Function](#) – Type.

| Function

Abstract type of all functions.

```
julia> isa(+, Function)
true

julia> typeof(sin)
typeof(sin)

julia> ans <: Function
true
```

source

[Base.method\\_exists](#) – Function.

Determine whether the given generic function has a method matching the given `Tuple` of argument types with the upper bound of world age given by `world`.

Examples

```
julia> method_exists(length, Tuple{Array})
```

```
true
```

```
source
```

`Core.applicable` – Function.

```
| applicable(f, args...) -> Bool
```

Determine whether the given generic function has a method applicable to the given arguments.

Examples

```
julia> function f(x, y)
```

```
    x + y
```

```
end;
```

```
julia> applicable(f, 1)
```

```
false
```

```
julia> applicable(f, 1, 2)
```

```
true
```

```
source
```

`Core.invoke` – Function.

Invoke a method for the given generic function `f` matching the specified types `argtypes` on the specified arguments `args` and passing the keyword arguments `kwargs`. The arguments `args` must conform with the specified types in `argtypes`, i.e. conversion is not automatically performed. This method allows invoking a method other than the most specific matching method, which is useful when the behavior of a more general definition is explicitly needed (often as part of the implementation of a more specific method of the same function).

Examples

```
julia> f(x::Real) = x^2;  
  
julia> f(x::Integer) = 1 + invoke(f, Tuple{Real}, x);  
  
julia> f(2)  
5
```

`source`

`Base.invokelatest` – Function.

```
| invokelatest(f, args...; kwargs...)
```

Calls `f(args...; kwargs...)`, but guarantees that the most recent method of `f` will be executed. This is useful in specialized circumstances, e.g. long-running event loops or callback functions that may call obsolete versions of a function `f`. (The drawback is that `invokelatest` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

`source`

```
| new
```

Special function available to inner constructors which created a new object of the type. See the manual section on [Inner Constructor Methods](#) for more information.

`source`

`Base. : |>` – Function.

```
| |>(x, f)
```

Applies a function to the preceding argument. This allows for easy function chaining.

Examples

```
julia> [1:5;] |> x->x.^2 |> sum |> inv
0.01818181818181818
```

`source`

`Base. :` – Function.

```
| f ∘ g
```

Compose functions: i.e. `(f ∘ g)(args...)` means `f(g(args...))`. The symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ``<tab>`.

Examples

```
julia> map(uppercasehex, 250:255)
6-element Array{String,1}:
 "FA"
 "FB"
```

```
"FD"  
"FE"  
"FF"
```

**source**

[Base.equalto](#) – Type.

```
| equalto(x)
```

Create a function that compares its argument to `x` using [isequal](#); i.e. returns `y->isequal(x,y)`.

The returned function is of type `Base.EqualTo`. This allows dispatching to specialized methods by using e.g. `f::Base.EqualTo` in a method signature.

**source**

## 46.9 Syntax

[Core.eval](#) – Function.

```
| eval([m::Module], expr::Expr)
```

Evaluate an expression in the given module and return the result. Every `Module` (except those defined with `baremodule`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

**source**

[Base.@eval](#) – Macro.

```
| @eval [mod,] ex
```

Evaluate an expression with values interpolated into it using `eval`. If two arguments are provided, the first is the module to evaluate in.

**Base.evalfile** – Function.

```
| evalfile(path::AbstractString, args::Vector{String}=String[])
```

Load the file using [include](#), evaluate all expressions, and return the value of the last one.

**source**

**Base.esc** – Function.

```
| esc(e)
```

Only valid in the context of an [Expr](#) returned from a macro. Prevents the macro hygiene pass from turning embedded variables into gensym variables. See the [Macros](#) section of the Metaprogramming chapter of the manual for more details and examples.

**source**

**Base.@inbounds** – Macro.

```
| @inbounds(blk)
```

Eliminates array bounds checking within expressions.

In the example below the in-range check for referencing element **i** of array **A** is skipped to improve performance.

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

Using `@inbounds` may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually. Only use `@inbounds` when it is certain from the information locally available that all accesses are in bounds.

### source

`Base.@boundscheck` – Macro.

```
|@boundscheck(blk)
```

Annotates the expression `blk` as a bounds checking block, allowing it to be elided by `@inbounds`.

Note that the function in which `@boundscheck` is written must be inlined into its caller with `@inline` in order for `@inbounds` to have effect.

```
julia> @inline function g(A, i)

    @boundscheck checkbounds(A, i)

    return "accessing ($A)[$i]"

end

f1() = return g(1:2, -1)

f2() = @inbounds return g(1:2, -1)
f2 (generic function with 1 method)

julia> f1()
ERROR: BoundsError: attempt to access 2-element
→ UnitRange{Int64} at index [-1]
```

```
[1] throw_boundserror(::UnitRange{Int64}, ::Tuple{Int64}) at  
→ ./abstractarray.jl:435  
[2] checkbounds at ./abstractarray.jl:399 [inlined]  
[3] g at ./none:2 [inlined]  
[4] f1() at ./none:1
```

```
julia> f2()
```

```
"accessing (1:2)[-1]"
```

### Warning

The `@boundscheck` annotation allows you, as a library writer, to opt-in to allowing other code to remove your bounds checks with `@inbounds`. As noted there, the caller must verify—using information they can access—that their accesses are valid before using `@inbounds`. For indexing into your `AbstractArray` subclasses, for example, this involves checking the indices against its `size`. Therefore, `@boundscheck` annotations should only be added to a `getindex` or `setindex!` implementation after you are certain its behavior is correct.

### source

`Base.@inline` – Macro.

```
|@inline
```

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the `@inline` annotation, as the compiler does it automatically. By using `@inline` on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

46.9 @SYNTAX **function** bigfunction(x)

```
#=  
    Function Definition  
#=  
end
```

source

Base.**@noinline** – Macro.

```
|@noinline
```

Prevents the compiler from inlining a function.

Small functions are typically inlined automatically. By using **@noinline** on small functions, auto-inlining can be prevented. This is shown in the following example:

```
@noinline function smallfunction(x)  
#=  
    Function Definition  
#=  
end
```

source

Base.**@nospecialize** – Macro.

```
|@nospecialize
```

Applied to a function argument name, hints to the compiler that the method should not be specialized for different types of that argument. This is only a hint for avoiding excess code generation. Can be applied to an argument within a formal argument list, or in the function body. When applied to an argument, the macro must wrap the entire argument expression.

822 When used in a function body, the macro must occur **statements** and before any code.

```
function example_function(@nospecialize x)
  ...
end

function example_function(@nospecialize(x = 1), y)
  ...
end

function example_function(x, y, z)
  @nospecialize x y
  ...
end
```

**source**

**Base.gensym** – Function.

| **gensym**([tag])

Generates a symbol which will not conflict with other variable names.

**source**

**Base.@gensym** – Macro.

| **@gensym**

Generates a gensym symbol for a variable. For example, **@gensym** x y is transformed into **x = gensym("x"); y = gensym("y")**.

**source**

**Base.@goto** – Macro.

| **@goto** name

46.1 @~~goto~~<sup>NDI API</sup> `@label name` Unconditionally jumps to the statement at the location [@823](#)

`bel name.`

`@label` and `@goto` cannot create jumps to different top-level statements.

Attempts cause an error. To still use `@goto`, enclose the `@label` and `@goto` in a block.

`source`

`Base.@label` – Macro.

`|@label name`

Labels a statement with the symbolic label `name`. The label marks the end-point of an unconditional jump with `@goto name`.

`source`

`Base.@polly` – Macro.

`|@polly`

Tells the compiler to apply the polyhedral optimizer Polly to a function.

`source`

## 46.10 Nullables

`Base.Nullable` – Type.

`|Nullable(x, hasvalue::Bool=true)`

Wrap value `x` in an object of type `Nullable`, which indicates whether a value is present. `Nullable(x)` yields a non-empty wrapper and `Nullable{T}()` yields an empty instance of a wrapper that might contain a value of type `T`.

`Nullable(x, false)` yields `Nullable{typeof(x)}()` with `x` stored in the result's `value` field.

```
| julia> Nullable(1)
| Nullable{Int64}(1)

| julia> Nullable{Int64}()
| Nullable{Int64}()

| julia> Nullable(1, false)
| Nullable{Int64}()

| julia> dump(Nullable(1, false))
| Nullable{Int64}
|   hasvalue: Bool false
|   value: Int64 1
```

### source

[Base.get](#) – Method.

```
| get(x::Nullable[, y])
```

Attempt to access the value of x. Returns the value if it is present; otherwise, returns y if provided, or throws a `NullException` if not.

### Examples

```
| julia> get(Nullable(5))
| 5

| julia> get(Nullable())
| ERROR: NullException()
| Stacktrace:
| [ ... ]
```

### source

```
| isnull(x)
```

Return whether or not `x` is null for `Nullable` `x`; return `false` for all other `x`.

Examples

```
julia> x = Nullable(1, false)
```

```
Nullable{Int64}()
```

```
julia> isnull(x)
```

```
true
```

```
julia> x = Nullable(1, true)
```

```
Nullable{Int64}(1)
```

```
julia> isnull(x)
```

```
false
```

```
julia> x = 1
```

```
1
```

```
julia> isnull(x)
```

```
false
```

```
source
```

`Base.unsafe_get` – Function.

```
| unsafe_get(x)
```

Return the value of `x` for `Nullable` `x`; return `x` for all other `x`.

This method does not check whether or not `x` is null before attempting to access the value of `x` for `x::Nullable` (hence "unsafe").

```
julia> x = Nullable(1)
Nullable{Int64}(1)

julia> unsafe_get(x)
1

julia> x = Nullable{String}()
Nullable{String}()

julia> unsafe_get(x)
ERROR: UndefRefError: access to undefined reference
Stacktrace:
[...]

julia> x = 1
1

julia> unsafe_get(x)
1

source
```

## 46.11 System

[Base.run](#) – Function.

```
| run(command, args...)
```

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

[source](#)

```
| spawn(command)
```

Run a command object asynchronously, returning the resulting Process object.

**source**

[Base.DevNull](#) – Constant.

```
| DevNull
```

Used in a stream redirect to discard all data written to it. Essentially equivalent to /dev/null on Unix or NUL on Windows. Usage:

```
| run(pipeline(`cat test.txt`, DevNull))
```

**source**

[Base.success](#) – Function.

```
| success(command)
```

Run a command object, constructed with backticks, and tell whether it was successful (exited with a code of 0). An exception is raised if the process cannot be started.

**source**

[Base.process\\_running](#) – Function.

```
| process_running(p::Process)
```

Determine whether a process is currently running.

**source**

[Base.process\\_exited](#) – Function.

```
| process_exited(p::Process)
```

**source**

**Base.kill** – Method.

| `kill(p::Process, signum=SIGTERM)`

Send a signal to a process. The default is to terminate the process. Returns successfully if the process has already exited, but throws an error if killing the process failed for other reasons (e.g. insufficient permissions).

**source**

**Base.Sys.set\_process\_title** – Function.

| `Sys.set_process_title(title::AbstractString)`

Set the process title. No-op on some operating systems.

**source**

**Base.Sys.get\_process\_title** – Function.

| `Sys.get_process_title()`

Get the process title. On some systems, will always return an empty string.

**source**

**Base.readandwrite** – Function.

| `readandwrite(command)`

Starts running a command asynchronously, and returns a tuple (stdout,stdin,process) of the output stream and input stream of the process, and the process object itself.

**source**

**Base.ignorestatus** – Function.

Mark a command object so that running it will not throw an error if the result code is non-zero.

**source**

[Base.detach](#) – Function.

| detach(command)

Mark a command object so that it will be run in a new process group, allowing it to outlive the julia process, and not have Ctrl-C interrupts passed to it.

**source**

[Base.Cmd](#) – Type.

| Cmd(cmd::Cmd; ignorestatus, detach, windows\_verbatim,  
| windows\_hide, env, dir)

Construct a new `Cmd` object, representing an external program and arguments, from `cmd`, while changing the settings of the optional keyword arguments:

`ignorestatus::Bool`: If `true` (defaults to `false`), then the `Cmd` will not throw an error if the return code is nonzero.

`detach::Bool`: If `true` (defaults to `false`), then the `Cmd` will be run in a new process group, allowing it to outlive the `julia` process and not have Ctrl-C passed to it.

`windows_verbatim::Bool`: If `true` (defaults to `false`), then on Windows the `Cmd` will send a command-line string to the process with no quoting or escaping of arguments, even arguments containing

spaces. (On Windows, arguments are sent as a single "command-line" string, and programs are responsible for parsing it into arguments. By default, empty arguments and arguments with spaces or tabs are quoted with double quotes " in the command line, and \ or " are preceded by backslashes. `windows_verbatim=true` is useful for launching programs that parse their command line in non-standard ways.) Has no effect on non-Windows systems.

`windows_hide::Bool`: If `true` (defaults to `false`), then on Windows no new console window is displayed when the `Cmd` is executed. This has no effect if a console is already open or on non-Windows systems.

`env`: Set environment variables to use when running the `Cmd`. `env` is either a dictionary mapping strings to strings, an array of strings of the form `"var=val"`, an array or tuple of `"var"=>val` pairs, or nothing. In order to modify (rather than replace) the existing environment, create `env` by `copy(ENV)` and then set `env["var"] = val` as desired.

`dir::AbstractString`: Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from `cmd` are used. Normally, to create a `Cmd` object in the first place, one uses backticks, e.g.

```
| Cmd(`echo "Hello world"`, ignorestatus=true, detach=false)
```

`source`

[Base.setenv](#) – Function.

```
| setenv(command::Cmd, env; dir="")
```

Set environment variables to use when running the given `command`. `env` is either a dictionary mapping strings to strings, an array of strings of

46.1 the `setenv("var"=val)`, or zero or more "var"=>val pair arguments. In order to modify (rather than replace) the existing environment, create env by `copy(ENV)` and then setting `env["var"] = val` as desired, or use `withenv`.

The `dir` keyword argument can be used to specify a working directory for the command.

`source`

`Base.withenv` – Function.

```
| withenv(f::Function, kv::Pair...)
```

Execute `f` in an environment that is temporarily modified (not replaced as in `setenv`) by zero or more "var"=>val arguments `kv`. `withenv` is generally used via the `withenv(kv...) do ... end` syntax. A value of `nothing` can be used to temporarily unset an environment variable (if it is set). When `withenv` returns, the original environment has been restored.

`source`

`Base.pipeline` – Method.

```
| pipeline(from, to, ...)
```

Create a pipeline from a data source to a destination. The source and destination can be commands, I/O streams, strings, or results of other `pipeline` calls. At least one argument must be a command. Strings refer to filenames. When called with more than two arguments, they are chained together from left to right. For example `pipeline(a, b, c)` is equivalent to `pipeline(pipeline(a, b), c)`. This provides a more concise way to specify multi-stage pipelines.

Examples:

```
832 | run(pipeline(`ls`, `grep xyz`))  
| run(pipeline(`ls`, "out.txt"))  
| run(pipeline("out.txt", `grep xyz`))
```

CHAPTER 46. ESSENTIALS

source

`Base.pipeline` – Method.

```
| pipeline(command; stdin, stdout, stderr, append=false)
```

Redirect I/O to or from the given `command`. Keyword arguments specify which of the command's streams should be redirected. `append` controls whether file output appends to the file. This is a more general version of the 2-argument `pipeline` function. `pipeline(from, to)` is equivalent to `pipeline(from, stdout=to)` when `from` is a command, and to `pipeline(to, stdin=from)` when `from` is another kind of data source.

Examples:

```
| run(pipeline(`dothings`, stdout="out.txt", stderr="errs.txt"))  
| run(pipeline(`update`, stdout="log.txt", append=true))
```

source

`Base.Libc.gethostname` – Function.

```
| gethostname() -> AbstractString
```

Get the local machine's host name.

source

`Base.getipaddr` – Function.

```
| getipaddr() -> IPAddr
```

Get the IP address of the local machine.

source

```
| getpid() -> Int32
```

Get Julia's process ID.

[source](#)

Base.Libc.time – Method.

```
| time()
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

[source](#)

Base.time\_ns – Function.

```
| time_ns()
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

[source](#)

Base.@time – Macro.

```
| @time
```

A macro to execute an expression, printing the time it took to execute, the number of allocations, and the total number of bytes its execution caused to be allocated, before returning the value of the expression.

See also [@timev](#), [@timed](#), [@elapsed](#), and [@allocated](#).

```
julia> @time rand(10^6);
```

```
0.001525 seconds (7 allocations: 7.630 MiB)
```

```
julia> @time begin
```

```
    sleep(0.3)

    1+1

end
0.301395 seconds (8 allocations: 336 bytes)
```

2

`source`

`Base.@timev` – Macro.

`@timev`

This is a verbose version of the `@time` macro. It first prints the same information as `@time`, then any non-zero memory allocation counters, and then returns the value of the expression.

See also `@time`, `@timed`, `@elapsed`, and `@allocated`.

```
julia> @timev rand(10^6);
0.001006 seconds (7 allocations: 7.630 MiB)
elapsed time (ns): 1005567
bytes allocated: 8000256
pool allocs: 6
malloc() calls: 1
```

`source`

`Base.@timed` – Macro.

`@timed`

A macro to execute an expression, and return the value of the expression, elapsed time, total bytes allocated, garbage collection time, and an object with various memory allocation counters.

```
julia> val, t, bytes, gctime, memallocs = @timed rand(10^6);

julia> t
0.006634834

julia> bytes
8000256

julia> gctime
0.0055765

julia> fieldnames(typeof(memallocs))
9-element Array{Symbol,1}:
 :allocd
 :malloc
 :realloc
 :poolalloc
 :bigalloc
 :freecall
 :total_time
 :pause
 :full_sweep

julia> memallocs.total_time
5576500
```

source

Base.@elapsed – Macro.

| @elapsed

836 A macro to evaluate an expression, discarding the resulting value, instead returning the number of seconds it took to execute as a floating-point number.

See also [@time](#), [@timev](#), [@timed](#), and [@allocated](#).

```
julia> @elapsed sleep(0.3)
0.301391426
```

[source](#)

[Base.@allocated](#) – Macro.

```
| @allocated
```

A macro to evaluate an expression, discarding the resulting value, instead returning the total number of bytes allocated during evaluation of the expression. Note: the expression is evaluated inside a local function, instead of the current context, in order to eliminate the effects of compilation, however, there still may be some allocations due to JIT compilation. This also makes the results inconsistent with the [@time](#) macros, which do not try to adjust for the effects of compilation.

See also [@time](#), [@timev](#), [@timed](#), and [@elapsed](#).

```
julia> @allocated rand(10^6)
8000080
```

[source](#)

[Base.EnvDict](#) – Type.

```
| EnvDict() -> EnvDict
```

A singleton of this type provides a hash table interface to environment variables.

[source](#)

| ENV

Reference to the singleton EnvDict, providing a dictionary interface to system environment variables.

source

Base.Sys.isunix – Function.

| Sys.isunix([os])

Predicate for testing if the OS provides a Unix-like interface. See documentation in [Handling Operating System Variation](#).

source

Base.Sys.isapple – Function.

| Sys.isapple([os])

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in [Handling Operating System Variation](#).

source

Base.Sys.islinux – Function.

| Sys.islinux([os])

Predicate for testing if the OS is a derivative of Linux. See documentation in [Handling Operating System Variation](#).

source

Base.Sys.isbsd – Function.

| Sys.isbsd([os])

838 Predicate for testing if the OS is a derivative of Microsoft Windows NT.  
in [Handling Operating System Variation](#).

**source**

[Base.Sys.iswindows](#) – Function.

| `Sys.iswindows([os])`

Predicate for testing if the OS is a derivative of Microsoft Windows NT.

See documentation in [Handling Operating System Variation](#).

**source**

[Base.Sys.windows\\_version](#) – Function.

| `Sys.windows_version()`

Returns the version number for the Windows NT Kernel as a `VersionNumber`, i.e. `v"major.minor.build"`, or `v"0.0.0"` if this is not running on Windows.

**source**

[Base.@static](#) – Macro.

| `@static`

Partially evaluate an expression at parse time.

For example, `@static Sys.iswindows() ? foo : bar` will evaluate `Sys.iswindows()` and insert either `foo` or `bar` into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a `ccall` to a non-existent function. `@static if Sys.isapple() foo end` and `@static foo <&&, ||> bar` are also valid syntax.

**source**

`Base.error` – Function.

```
| error(message::AbstractString)
```

Raise an `ErrorException` with the given message.

`source`

```
| error(msg...)
```

Raise an `ErrorException` with the given message.

See also [logging](#).

`source`

`Core.throw` – Function.

```
| throw(e)
```

Throw an object as an exception.

`source`

`Base.rethrow` – Function.

```
| rethrow([e])
```

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a `catch` block).

`source`

`Base.backtrace` – Function.

```
| backtrace()
```

Get a backtrace object for the current program point.

`source`

`Base.catch_backtrace` – Function.

CHAPTER 46. ESSENTIALS

```
| catch_backtrace()
```

Get the backtrace of the current exception, for use within `catch` blocks.

`source`

`Base.assert` – Function.

```
| assert(cond)
```

Throw an `AssertionError` if `cond` is `false`. Also available as the macro `@assert`.

`source`

`Base.@assert` – Macro.

```
| @assert cond [text]
```

Throw an `AssertionError` if `cond` is `false`. Preferred syntax for writing assertions. Message `text` is optionally displayed upon assertion failure.

Examples

```
julia> @assert iseven(3) "3 is an odd number!"
```

```
ERROR: AssertionError: 3 is an odd number!
```

```
julia> @assert isodd(3) "What even are numbers?"
```

`source`

`Core.ArgumentError` – Type.

```
| ArgumentError(msg)
```

The parameters to a function call do not match a valid signature. Argument `msg` is a descriptive error string.

`source`

```
| AssertionError([msg])
```

The asserted condition did not evaluate to `true`. Optional argument `msg` is a descriptive error string.

`source`

`Core.BoundsError` – Type.

```
| BoundsError([a],[i])
```

An indexing operation into an array, `a`, tried to access an out-of-bounds element at index `i`.

Examples

```
julia> A = ones(7);
```

```
julia> A[8]
```

```
ERROR: BoundsError: attempt to access 7-element  
→ Array{Float64,1} at index [8]
```

Stacktrace:

```
[1] getindex(::Array{Float64,1}, ::Int64) at ./array.jl:758
```

```
julia> B = ones(2, 3);
```

```
julia> B[2, 4]
```

```
ERROR: BoundsError: attempt to access 2×3 Array{Float64,2} at  
→ index [2, 4]
```

Stacktrace:

```
[1] getindex(::Array{Float64,2}, ::Int64, ::Int64) at  
→ ./array.jl:759
```

```
julia> B[9]
```

842 | ERROR: BoundsError: attempt to access 2×8 Array{Float64,2} at  
|   ↑ index [9]  
Stacktrace:  
[1] getindex(::Array{Float64,2}, ::Int64) at ./array.jl:758

**source**

[Base.DimensionMismatch](#) – Type.

| DimensionMismatch([msg])

The objects called do not have matching dimensionality. Optional argument `msg` is a descriptive error string.

**source**

[Core.DivideError](#) – Type.

| DivideError()

Integer division was attempted with a denominator value of 0.

Examples

```
julia> 2/0
Inf

julia> div(2, 0)
ERROR: DivideError: integer division error
Stacktrace:
[...]
```

**source**

[Core.DomainError](#) – Type.

| DomainError(val)
| DomainError(val, msg)

## 46.1 The Errors Argument

843

Examples

```
julia> sqrt(-1)
ERROR: DomainError with -1.0:
sqrt will only return a complex result if called with a
→ complex argument. Try sqrt(Complex(x)).
Stacktrace:
[...]
```

source

[Base.EOFError](#) – Type.

```
| EOFError()
```

No more data was available to read from a file or stream.

source

[Core.ErrorException](#) – Type.

```
| ErrorException(msg)
```

Generic error type. The error message, in the `.msg` field, may provide more specific details.

source

[Core.InexactError](#) – Type.

```
| InexactError(name::Symbol, T, val)
```

Cannot exactly convert `val` to type `T` in a method of function `name`.

Examples

```
julia> convert(Float64, 1+2im)
ERROR: InexactError: convert(Float64, 1 + 2im)
```

844 | Stacktrace:

CHAPTER 46. ESSENTIALS

```
| [1] convert(::Type{Float64}, ::Complex{Int64}) at  
|   ↳ ./complex.jl:37
```

source

[Core.InterruptException](#) – Type.

```
| InterruptException()
```

The process was stopped by a terminal interrupt (CTRL+C).

source

[Base.KeyError](#) – Type.

```
| KeyError(key)
```

An indexing operation into an [Associative \(Dict\)](#) or [Set](#) like object tried to access or delete a non-existent element.

source

[Core.LoadError](#) – Type.

```
| LoadError(file::AbstractString, line::Int, error)
```

An error occurred while `includeing`, `requireing`, or `using` a file. The error specifics should be available in the `.error` field.

source

[Core.MethodError](#) – Type.

```
| MethodError(f, args)
```

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

source

```
| NullException()
```

An attempted access to a **Nullable** with no defined value.

Examples

```
| julia> a = Nullable{Int}()
```

```
Nullable{Int64}()
```

```
| julia> get(a)
```

```
ERROR: NullException()
```

```
Stacktrace:
```

```
[...]
```

source

```
| OutOfMemoryError()
```

An operation allocated too much memory for either the system or the garbage collector to handle properly.

source

```
| ReadOnlyMemoryError()
```

An operation tried to write to memory that is read-only.

source

```
| OverflowError(msg)
```

846 The result of an expression is too large for the system to handle.  
CHAPTER 16. ESSENTIAL JULIA TYPES

cause a wraparound.

`source`

`Base.ParseError` – Type.

`| ParseError(msg)`

The expression passed to the `parse` function could not be interpreted as a valid Julia expression.

`source`

`Base.Distributed.ProcessExitedException` – Type.

`| ProcessExitedException()`

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

`source`

`Core.StackOverflowError` – Type.

`| StackOverflowError()`

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

`source`

`Base.SystemError` – Type.

`| SystemError(prefix::AbstractString, [errno::Int32])`

A system call failed with an error code (in the `errno` global variable).

`source`

`Core.TypeError` – Type.

46.12 ERRORS  
| typeError(func::Symbol, context::AbstractString, expected::Type<sup>847</sup>  
| , got)

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

source

[Core.UndefRefError](#) – Type.

| UndefRefError()

The item or field is not defined for the given object.

source

[Core.UndefVarError](#) – Type.

| UndefVarError(var::Symbol)

A symbol in the current scope is not defined.

source

[Core.InitError](#) – Type.

| InitError(mod::Symbol, error)

An error occurred when running a module's `__init__` function. The actual error thrown is available in the `.error` field.

source

[Base.retry](#) – Function.

| retry(f::Function; delays=ExponentialBackOff(), check=nothing)  
| –> Function

Returns an anonymous function that calls function `f`. If an exception arises, `f` is repeatedly called again, each time `check` returns `true`, after waiting

848 the number of seconds specified in `delays`. CHAPTER 46. ESSENTIALS  
current state and the Exception.

Examples

```
retry(f, delays=fill(5.0, 3))
retry(f, delays=rand(5:10, 2))
retry(f, delays=Base.ExponentialBackOff(n=3, first_delay=5,
    ↴ max_delay=1000))
retry(http_get, check=(s,e)->e.status == "503")(url)
retry(read, check=(s,e)->isa(e, UVError))(io, 128; all=false)
```

`source`

[Base.ExponentialBackOff](#) – Type.

```
ExponentialBackOff(; n=1, first_delay=0.05, max_delay=10.0,
    ↴ factor=5.0, jitter=0.1)
```

A `Float64` iterator of length `n` whose elements exponentially increase at a rate in the interval `factor * (1 ± jitter)`. The first element is `first_delay` and all elements are clamped to `max_delay`.

`source`

## 46.13 Events

[Base.Timer](#) – Method.

```
Timer(callback::Function, delay, repeat=0)
```

Create a timer that wakes up tasks waiting for it (by calling `wait` on the timer object) and calls the function `callback`.

Waiting tasks are woken and the function `callback` is called after an initial delay of `delay` seconds, and then repeating with the given `repeat` interval in seconds. If `repeat` is equal to `0`, the timer is only triggered

46.13 ~~onEvent~~<sup>849</sup> function `callback` is called with a single argument, the timer itself. When the timer is closed (by `close`) waiting tasks are woken with an error. Use `isOpen` to check whether a timer is still active.

### Examples

Here the first number is printed after a delay of two seconds, then the following numbers are printed quickly.

```
julia> begin  
  
    i = 0  
  
    cb(timer) = (global i += 1; println(i))  
  
    t = Timer(cb, 2, 0.2)  
  
    wait(t)  
  
    sleep(0.5)  
  
    close(t)  
  
end  
1  
2  
3
```

source

`Base.Timer` – Type.

```
| Timer(delay, repeat=0)
```

850 Create a timer that wakes up tasks waiting for `CHARTERING_WAITABLES` (by calling `wait` on the timer object).

Waiting tasks are woken after an initial delay of `delay` seconds, and then repeating with the given `repeat` interval in seconds. If `repeat` is equal to 0, the timer is only triggered once. When the timer is closed (by `close`) waiting tasks are woken with an error. Use `isopen` to check whether a timer is still active.

`source`

`Base.AsyncCondition` – Type.

`| AsyncCondition()`

Create a async condition that wakes up tasks waiting for it (by calling `wait` on the object) when notified from C by a call to `uv_async_send`. Waiting tasks are woken with an error when the object is closed (by `close`). Use `isopen` to check whether it is still active.

`source`

`Base.AsyncCondition` – Method.

`| AsyncCondition(callback::Function)`

Create a async condition that calls the given `callback` function. The `callback` is passed one argument, the async condition object itself.

`source`

## 46.14 Reflection

`Base.module_name` – Function.

`| module_name(m::Module) -> Symbol`

Get the name of a `Module` as a `Symbol`.

```
| julia> module_name(Base.LinAlg)  
| :LinAlg
```

**source**

**Base.module\_parent** – Function.

```
| module_parent(m::Module) -> Module
```

Get a module's enclosing **Module**. **Main** is its own parent, as is **LastMain** after **workspace()**.

Examples

```
| julia> module_parent(Main)
```

Main

```
| julia> module_parent(Base.LinAlg.BLAS)
```

Base.LinAlg

**source**

**Base. @\_MODULE\_** – Macro.

```
| @_MODULE_ -> Module
```

Get the **Module** of the toplevel eval, which is the **Module** code is currently being read from.

**source**

**Base.fullname** – Function.

```
| fullname(m::Module)
```

Get the fully-qualified name of a module as a tuple of symbols. For example,

Examples

852 | **julia>** fullname(Base.Pkg)  
(:Base, :Pkg)

CHAPTER 46. ESSENTIALS

**julia>** fullname(Main)  
(:Main, )

**source**

**Base.names** – Function.

| names(x::Module, all::Bool=false, imported::Bool=false)

Get an array of the names exported by a **Module**, excluding deprecated names. If **all** is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If **imported** is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in **Main** are considered "exported", since it is not idiomatic to explicitly export names from **Main**.

**source**

**Core.nfields** – Function.

| nfields(x) -> Int

Get the number of fields in the given object.

**source**

**Base.fieldnames** – Function.

| fieldnames(x::DataType)

Get an array of the fields of a **DataType**.

Examples

```
2-element Array{Symbol,1}:
 :data
 :uplo
```

**source**

**Base.fieldname** – Function.

```
| fieldname(x::DataType, i::Integer)
```

Get the name of field **i** of a **DataType**.

Examples

```
julia> fieldname(SparseMatrixCSC, 1)
:m
```

```
julia> fieldname(SparseMatrixCSC, 5)
:nzval
```

**source**

**Base.fieldcount** – Function.

```
| fieldcount(t::Type)
```

Get the number of fields that an instance of the given type would have.

An error is thrown if the type is too abstract to determine this.

**source**

**Base.datatype\_module** – Function.

```
| Base.datatype_module(t::DataType) -> Module
```

Determine the module containing the definition of a (potentially UnionAll-wrapped) **DataType**.

Examples

```
julia> module Foo

    struct Int end

end
Foo

julia> Base.datatype_module(Int)
Core

julia> Base.datatype_module(Foo.Int)
Foo

source
```

`Base.datatype_name` – Function.

```
| Base.datatype_name(t) -> Symbol
```

Get the name of a (potentially UnionAll-wrapped) `DataType` (without its parent module) as a symbol.

Examples

```
julia> module Foo

    struct S{T}
end

end
Foo

julia> Base.datatype_name(Foo.S{T} where T)
:S
```

**Base.isconst** – Function.

```
| isconst(m::Module, s::Symbol) -> Bool
```

Determine whether a global is declared `const` in a given `Module`.

**source**

**Base.function\_name** – Function.

```
| Base.function_name(f::Function) -> Symbol
```

Get the name of a generic `Function` as a symbol, or `:anonymous`.

**source**

**Base.function\_module** – Method.

```
| Base.function_module(f::Function) -> Module
```

Determine the module containing the (first) definition of a generic function.

**source**

**Base.function\_module** – Method.

```
| Base.function_module(f::Function, types) -> Module
```

Determine the module containing a given definition of a generic function.

**source**

**Base.functionloc** – Method.

```
| functionloc(f::Function, types)
```

Returns a tuple (`filename, line`) giving the location of a generic `Function` definition.

**source**

`Base.functionloc` – Method.

CHAPTER 46. ESSENTIALS

```
| functionloc(m::Method)
```

Returns a tuple (`filename, line`) giving the location of a `Method` definition.

`source`

`Base.@functionloc` – Macro.

```
| @functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple (`filename, line`) giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

`source`

## 46.15 Internals

`Base.gc` – Function.

```
| gc()
```

Perform garbage collection. This should not generally be used.

`source`

`Base.gc_enable` – Function.

```
| gc_enable(on::Bool)
```

Control whether garbage collection is enabled using a boolean argument (`true` for enabled, `false` for disabled). Returns previous GC state. Disabling garbage collection should be used only with extreme caution, as it can cause memory use to grow without bound.

`source`

```
| lower(m, x)
```

Takes the expression `x` and returns an equivalent expression in lowered form for executing in module `m`. See also `code_lowered`.

`source`

`Base.Meta.@lower` – Macro.

```
| @lower [m] x
```

Return lowered form of the expression `x` in module `m`. By default `m` is the module in which the macro is called. See also `lower`.

`source`

`Base.Meta.parse` – Method.

```
| parse(str, start; greedy=true, raise=true)
```

Parse the expression string and return an expression (which could later be passed to `eval` for execution). `start` is the index of the first character to start parsing. If `greedy` is `true` (default), `parse` will try to consume as much input as it can; otherwise, it will stop as soon as it has parsed a valid expression. Incomplete but otherwise syntactically valid expressions will return `Expr(:incomplete, "(error message)")`. If `raise` is `true` (default), syntax errors other than incomplete expressions will raise an error. If `raise` is `false`, `parse` will return an expression that will raise an error upon evaluation.

```
julia> Meta.parse("x = 3, y = 5", 7)
(:y = 5, 13)
```

```
julia> Meta.parse("x = 3, y = 5", 5)
(:((3, y) = 5), 13)
```

[Base.Meta.parse](#) – Method.

```
| parse(str; raise=true)
```

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `raise` is `true` (default), syntax errors will raise an error; otherwise, `parse` will return an expression that will raise an error upon evaluation.

```
julia> Meta.parse("x = 3")
:(x = 3)
```

```
julia> Meta.parse("x = ")
:($(Expr(:incomplete, "incomplete: premature end of input")))
```

```
julia> Meta.parse("1.0.2")
ERROR: ParseError("invalid numeric constant \"1.0.\\"")
Stacktrace:
[...]
```

```
julia> Meta.parse("1.0.2"; raise = false)
:($(Expr(:error, "invalid numeric constant \"1.0.\\"")))
```

[source](#)

[Base.macroexpand](#) – Function.

```
| macroexpand(m::Module, x; recursive=true)
```

Takes the expression `x` and returns an equivalent expression with all macros removed (expanded) for executing in module `m`. The `recursive` keyword controls whether deeper levels of nested macros are also expanded. This is demonstrated in the example below:

```
macro m1()

    42

end

macro m2()

    :(@m1())

end

end

M

julia> macroexpand(M, :(@m2()), recursive=true)
42

julia> macroexpand(M, :(@m2()), recursive=false)
:(#= REPL[16]:6 =# M.@m1)

source
```

Base.@macroexpand – Macro.

| @macroexpand

Return equivalent expression with all macros removed (expanded).

There are differences between `@macroexpand` and `macroexpand`.

While `macroexpand` takes a keyword argument `recursive`, `@macroexpand`

860 is always recursive. For a non recursive macro see [macroindex](#)

While `macroexpand` has an explicit `module` argument, `@macroexpand` always

expands with respect to the module in which it is called. This is best seen in the following example:

```
julia> module M

    macro m()
        1

    end

    function f()

        (@macroexpand(@m),
         macroexpand(M, :(@m)),
         macroexpand(Main, :(@m))

    )

    end

end

M

julia> macro m()
```

```
2
```

```
end
```

```
@m (macro with 1 method)
```

```
julia> M.f()
```

```
(1, 1, 2)
```

With `@macroexpand` the expression expands where `@macroexpand` appears in the code (module `M` in the example). With `macroexpand` the expression expands in the module given as the first argument.

`source`

`Base.@macroexpand1` – Macro.

```
@macroexpand1
```

Non recursive version of `@macroexpand`.

`source`

`Base.code_lowered` – Function.

```
code_lowered(f, types, expand_generated = true)
```

Return an array of the lowered forms (IR) for the methods matching the given generic function and type signature.

If `expand_generated` is `false`, the returned `CodeInfo` instances will correspond to fallback implementations. An error is thrown if no fallback implementation exists. If `expand_generated` is `true`, these `CodeInfo` instances will correspond to the method bodies yielded by expanding the generators.

862 Note that an error will be thrown if `types` are CHAPTER 46. ESSENTIALS  
passed to `code_warntype` and `check_type` when `check_type`  
`pand_generated` is `true` and the corresponding method is a `@generated` method.

`source`

`Base.@code_lowered` – Macro.

`| @code_lowered`

Evaluates the arguments to the function or macro call, determines their types, and calls `code_lowered` on the resulting expression.

`source`

`Base.code_typed` – Function.

`| code_typed(f, types; optimize=true)`

Returns an array of type-inferred lowered form (IR) for the methods matching the given generic function and type signature. The keyword argument `optimize` controls whether additional optimizations, such as inlining, are also applied.

`source`

`Base.@code_typed` – Macro.

`| @code_typed`

Evaluates the arguments to the function or macro call, determines their types, and calls `code_typed` on the resulting expression.

`source`

`Base.code_warntype` – Function.

`| code_warntype([io::IO], f, types)`

46.1 Prints INTERNALS and type-inferred ASTs for the methods matching the given generic function and type signature to `io` which defaults to STDOUT. The ASTs are annotated in such a way as to cause "non-leaf" types to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability. Not all non-leaf types are particularly problematic for performance, so the results need to be used judiciously. See [@code\\_warntype](#) for more information.

`source`

`Base.@code_warntype` – Macro.

| `@code_warntype`

Evaluates the arguments to the function or macro call, determines their types, and calls [code\\_warntype](#) on the resulting expression.

`source`

`Base.code_llvm` – Function.

| `code_llvm([io=STDOUT,], f, types)`

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to `io`.

All metadata and `dbg.*` calls are removed from the printed bitcode. Use `code_llvm_raw` for the full IR.

`source`

`Base.@code_llvm` – Macro.

| `@code_llvm`

Evaluates the arguments to the function or macro call, determines their types, and calls [code\\_llvm](#) on the resulting expression.

`source`

`Base.code_native` – Function.

CHAPTER 46. ESSENTIALS

```
| code_native([io=STDOUT,], f, types, syntax=:att)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to `io`. Switch assembly syntax using `syntax` symbol parameter set to `:att` for AT&T syntax or `:intel` for Intel syntax.

`source`

`Base.@code_native` – Macro.

```
| @code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_native` on the resulting expression.

`source`

`Base.compile` – Function.

```
| compile(f, args::Tuple{Vararg{Any}})
```

Compile the given function `f` for the argument tuple (of types) `args`, but do not execute it.

`source`

## Chapter 47

# Collections and Data Structures

### 47.1 Iteration

Sequential iteration is implemented by the methods `start`, `done`, and `next`.

The general `for` loop:

```
for i = I    # or  "for i in I"  
    # body  
end
```

is translated into:

```
state = start(I)  
while !done(I, state)  
    (i, state) = next(I, state)  
    # body  
end
```

The `state` object may be anything, and should be chosen appropriately for each iterable type. See the [manual section on the iteration interface](#) for more details about defining a custom iterable type.

`Base.start` – Function.

866| **start**(iter) -> state CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

Get initial iteration state for an iterable object.

Examples

```
julia> start(1:5)
```

```
1
```

```
julia> start([1;2;3])
```

```
1
```

```
julia> start([4;2;3])
```

```
1
```

source

**Base.done** – Function.

```
| done(iter, state) -> Bool
```

Test whether we are done iterating.

Examples

```
julia> done(1:5, 3)
```

```
false
```

```
julia> done(1:5, 5)
```

```
false
```

```
julia> done(1:5, 6)
```

```
true
```

source

**Base.next** – Function.

For a given iterable object and iteration state, return the current item and the next iteration state.

Examples

```
julia> next(1:5, 3)
(3, 4)
```

```
julia> next(1:5, 5)
(5, 6)
```

`source`

`Base.iteratorsize` – Function.

```
| iteratorsize(itertype::Type) -> IteratorSize
```

Given the type of an iterator, returns one of the following values:

`SizeUnknown()` if the length (number of elements) cannot be determined in advance.

`HasLength()` if there is a fixed, finite length.

`HasShape()` if there is a known length plus a notion of multidimensional shape (as for an array). In this case the `size` function is valid for the iterator.

`IsInfinite()` if the iterator yields values forever.

The default value (for iterators that do not define this function) is `HasLength()`.

This means that most iterators are assumed to implement `length`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

```
julia> Base.iteratorsize(1:5)
```

```
Base.HasShape()
```

```
julia> Base.iteratorsize((2,3))
```

```
Base.HasLength()
```

[source](#)

`Base.iteratoreltype` – Function.

```
| iteratoreltype(itertype::Type) -> IteratorEltype
```

Given the type of an iterator, returns one of the following values:

`EltypeUnknown()` if the type of elements yielded by the iterator is not known in advance.

`HasEltype()` if the element type is known, and `eltype` would return a meaningful value.

`HasEltype()` is the default, since iterators are assumed to implement `eltype`.

This trait is generally used to select between algorithms that pre-allocate a specific type of result, and algorithms that pick a result type based on the types of yielded values.

```
julia> Base.iteratoreltype(1:5)
```

```
Base.HasEltype()
```

[source](#)

Fully implemented by:

`AbstractRange`

`UnitRange`

Number

AbstractArray

BitSet

ObjectIdDict

Dict

WeakKeyDict

EachLine

AbstractString

Set

Pair

## 47.2 General Collections

Baseisempty – Function.

```
| isempty(collection) -> Bool
```

Determine whether a collection is empty (has no elements).

Examples

```
julia> isempty([])
```

```
true
```

```
julia> isempty([1 2 3])
```

```
false
```

source

Baseempty! – Function.

870 | empty!(collection) → collection  
CHAPTER 47 COLLECTIONS AND DATA STRUCTURES

Remove all elements from a collection.

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

```
julia> empty!(A);
```

```
julia> A
Dict{String,Int64} with 0 entries
```

source

[Base.length](#) – Function.

```
| length(collection) -> Integer
```

Return the number of elements in the collection.

Use [endof](#) to get the last valid index of an indexable collection.

Examples

```
julia> length(1:5)
5
```

```
julia> length([1, 2, 3, 4])
4
```

```
julia> length([1 2; 3 4])
4
```

source

The number of characters in string s.

Examples

```
julia> length("julia")
5
```

source

Fully implemented by:

AbstractRange

UnitRange

Tuple

Number

AbstractArray

BitSet

ObjectIdDict

Dict

WeakKeyDict

AbstractString

Set

## 47.3 Iterable Collections

[Base.in](#) – Function.

872 | `in(item, collection)` → CHAPTER 47. COLLECTIONS AND DATA STRUCTURES `Bool`

```
(item, collection) -> Bool  
(collection, item) -> Bool  
(item, collection) -> Bool  
(collection, item) -> Bool
```

Determine whether an item is in the given collection, in the sense that it is `==` to one of the values generated by iterating over the collection. Some collections need a slightly different definition; for example [Sets](#) check whether the item [isequal](#) to one of the elements. [Dicts](#) look for `(key, value)` pairs, and the key is compared using [isequal](#). To test for the presence of a key in a dictionary, use [haskey](#) or `k in keys(dict)`.

```
julia> a = 1:3:20
```

```
1:3:19
```

```
julia> 4 in a
```

```
true
```

```
julia> 5 in a
```

```
false
```

source

[Base.eltype](#) – Function.

```
| eltype(type)
```

Determine the type of the elements generated by iterating a collection of the given `type`. For associative collection types, this will be a `Pair{KeyType, ValType}`. The definition `eltype(x) = eltype(typeof(x))` is provided for convenience so that instances can be passed instead of types. However the form that accepts a type argument should be defined for new types.

```
julia> eltype(ones(Float32,2,2))  
Float32
```

```
julia> eltype(ones(Int8,2,2))  
Int8
```

source

`Base.indexin` – Function.

```
| indexin(a, b)
```

Returns a vector containing the highest index in `b` for each value in `a` that is a member of `b`. The output vector contains 0 wherever `a` is not a member of `b`.

Examples

```
julia> a = ['a', 'b', 'c', 'b', 'd', 'a'];
```

```
julia> b = ['a', 'b', 'c'];
```

```
julia> indexin(a,b)  
6-element Array{Int64,1}:
```

```
1  
2  
3  
2  
0  
1
```

```
julia> indexin(b,a)  
3-element Array{Int64,1}:
```

```
| 6  
| 4  
| 3
```

`source`

`Base.findin` – Function.

```
| findin(a, b)
```

Returns the indices of elements in collection `a` that appear in collection `b`.

Examples

```
julia> a = collect(1:3:15)
```

```
5-element Array{Int64,1}:
```

```
 1  
 4  
 7  
10  
13
```

```
julia> b = collect(2:4:10)
```

```
3-element Array{Int64,1}:
```

```
 2  
 6  
10
```

```
julia> findin(a,b) # 10 is the only common element
```

```
1-element Array{Int64,1}:
```

```
 4
```

`source`

`Base.unique` – Function.

Return an array containing only the unique elements of collection `itr`, as determined by `isequal`, in the order that the first of each set of equivalent elements originally appears. The element type of the input is preserved.

Examples

```
julia> unique([1, 2, 6, 2])
3-element Array{Int64,1}:
 1
 2
 6
```

```
julia> unique(Real[1, 1.0, 2])
2-element Array{Real,1}:
 1
 2
```

source

```
| unique(f, itr)
```

Returns an array containing one value from `itr` for each unique value produced by `f` applied to elements of `itr`.

Examples

```
julia> unique(x -> x^2, [1, -1, 3, -3, 4])
3-element Array{Int64,1}:
 1
 3
 4
```

source

876| unique(A::AbstractArray, dim::Int) CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

Return unique regions of A along dimension dim.

Examples

```
julia> A = map(isodd, reshape(collect(1:8), (2,2,2)))
2×2×2 Array{Bool,3}:
[:, :, 1] =
  true  true
 false  false

[:, :, 2] =
  true  true
 false  false

julia> unique(A)
2-element Array{Bool,1}:
 true
 false

julia> unique(A, 2)
2×1×2 Array{Bool,3}:
[:, :, 1] =
  true
 false

[:, :, 2] =
  true
 false

julia> unique(A, 3)
2×2×1 Array{Bool,3}:
```

true	true
false	false

**source**

[Base.unique!](#) – Function.

```
| unique!(A::AbstractVector)
```

Remove duplicate items as determined by [isequal](#), then return the modified A. `unique!` will return the elements of A in the order that they occur. If you do not care about the order of the returned data, then calling `(sort!(A); unique!(A))` will be much more efficient as long as the elements of A can be sorted.

### Examples

```
julia> unique!([1, 1, 1])
```

```
1-element Array{Int64,1}:
```

```
1
```

```
julia> A = [7, 3, 2, 3, 7, 5];
```

```
julia> unique!(A)
```

```
4-element Array{Int64,1}:
```

```
7
```

```
3
```

```
2
```

```
5
```

```
julia> B = [7, 6, 42, 6, 7, 42];
```

878 | **julia>** sort!(B); #`unique!` is able to process sorted data much  
| → more efficiently.

```
| julia> unique!(B)  
3-element Array{Int64,1}:  
 6  
 7  
 42
```

source

[Base.allunique](#) – Function.

```
| allunique(itr) -> Bool
```

Return `true` if all values from `itr` are distinct when compared with `isequal`.

Examples

```
| julia> a = [1; 2; 3]  
3-element Array{Int64,1}:  
 1  
 2  
 3  
  
| julia> allunique([a, a])  
false
```

source

[Base.reduce](#) – Method.

```
| reduce(op, v0, itr)
```

## 47.3 Reducible Collections

`reduce(op, itr)` with the given binary operator `op`. `v0` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `v0` is used for non-empty collections.

Reductions for certain commonly-used operators may have special implementations, and should be used instead: `maximum(itr)`, `minimum(itr)`, `sum(itr)`, `prod(itr)`, `any(itr)`, `all(itr)`.

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `-` because it is undefined whether `reduce(-,[1,2,3])` should be evaluated as  $(1-2)-3$  or  $1-(2-3)$ . Use `foldl` or `foldr` instead for guaranteed left or right associativity.

Some operations accumulate error. Parallelism will be easier if the reduction can be executed in groups. Future versions of Julia might change the algorithm. Note that the elements are not reordered if you use an ordered collection.

Examples

```
julia> reduce(*, 1, [2; 3; 4])
```

```
24
```

`source`

`Base.reduce` – Method.

```
| reduce(op, itr)
```

Like `reduce(op, v0, itr)`. This cannot be used with empty collections, except for some special cases (e.g. when `op` is one of `+`, `*`, `max`, `min`, `&`, `|`) when Julia can determine the neutral element of `op`.

```
julia> reduce(*, [2; 3; 4])
```

```
24
```

`Base.foldl` – Method.

```
| foldl(op, v0, itr)
```

Like `reduce`, but with guaranteed left associativity. `v0` will be used exactly once.

```
| julia> foldl(-, 1, 2:5)
| -13
```

`source`

`Base.foldl` – Method.

```
| foldl(op, itr)
```

Like `foldl(op, v0, itr)`, but using the first element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

```
| julia> foldl(-, 2:5)
| -10
```

`source`

`Base.foldr` – Method.

```
| foldr(op, v0, itr)
```

Like `reduce`, but with guaranteed right associativity. `v0` will be used exactly once.

```
| julia> foldr(-, 1, 2:5)
| -1
```

`source`

```
| foldr(op, itr)
```

Like `foldr(op, v0, itr)`, but using the last element of `itr` as `v0`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

```
| julia> foldr(-, 2:5)
```

```
-2
```

`source`

`Base.maximum` – Method.

```
| maximum(itr)
```

Returns the largest element in a collection.

```
| julia> maximum(-20.5:10)
```

```
9.5
```

```
| julia> maximum([1,2,3])
```

```
3
```

`source`

`Base.maximum` – Method.

```
| maximum(A, dims)
```

Compute the maximum value of an array over the given dimensions. See also the `max(a,b)` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `max.(a,b)`.

```
| julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
1 2  
3 4
```

```
julia> maximum(A, 1)
```

```
1×2 Array{Int64,2}:  
3 4
```

```
julia> maximum(A, 2)
```

```
2×1 Array{Int64,2}:  
2  
4
```

source

`Base.maximum!` – Function.

```
|maximum!(r, A)
```

Compute the maximum value of `A` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:  
1 2  
3 4
```

```
julia> maximum!([1; 1], A)
```

```
2-element Array{Int64,1}:  
2  
4
```

```
julia> maximum!([1 1], A)
```

```
| 1×2 Array{Int64,2}:
```

```
|   3  4
```

**source**

[Base.minimum](#) – Method.

```
| minimum(itr)
```

Returns the smallest element in a collection.

```
| julia> minimum(-20.5:10)
```

```
| -20.5
```

```
| julia> minimum([1,2,3])
```

```
| 1
```

**source**

[Base.minimum](#) – Method.

```
| minimum(A, dims)
```

Compute the minimum value of an array over the given dimensions. See also the [min\(a, b\)](#) function to take the minimum of two or more arguments, which can be applied elementwise to arrays via [min.\(a, b\)](#).

Examples

```
| julia> A = [1 2; 3 4]
```

```
| 2×2 Array{Int64,2}:
```

```
|   1  2
```

```
|   3  4
```

```
| julia> minimum(A, 1)
```

```
| 1×2 Array{Int64,2}:
```

```
julia> minimum(A, 2)
2×1 Array{Int64,2}:
 1
 3
```

source

[Base.minimum!](#) – Function.

```
|minimum!(r, A)
```

Compute the minimum value of  $A$  over the singleton dimensions of  $r$ , and write results to  $r$ .

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> minimum!([1; 1], A)
2-element Array{Int64,1}:
 1
 3
```

```
julia> minimum!([1 1], A)
1×2 Array{Int64,2}:
 1  2
```

source

[Base.extrema](#) – Method.

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

```
julia> extrema(2:10)
(2, 10)

julia> extrema([9, pi, 4.5])
(3.141592653589793, 9.0)
```

[source](#)

[Base.extrema](#) – Method.

```
| extrema(A, dims) -> Array{Tuple{}}
```

Compute the minimum and maximum elements of an array over the given dimensions.

Examples

```
julia> A = reshape(collect(1:2:16), (2,2,2))
2×2×2 Array{Int64,3}:
[:, :, 1] =
 1  5
 3  7

[:, :, 2] =
 9  13
11  15

julia> extrema(A, (1,2))
1×1×2 Array{Tuple{Int64, Int64},3}:
[:, :, 1] =
```

886 | (1, 7)

CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

| [ :, :, 2] =  
| (9, 15)

**source**

**Base.indmax** – Function.

| **indmax(itr) -> Integer**

Returns the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned.

The collection must not be empty.

Examples

| **julia> indmax([8,0.1,-9,pi])**

| 1

| **julia> indmax([1,7,7,6])**

| 2

| **julia> indmax([1,7,7,NaN])**

| 4

**source**

**Base.indmin** – Function.

| **indmin(itr) -> Integer**

Returns the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned.

The collection must not be empty.

Examples

```
3
```

```
julia> indmin([7,1,1,6])
```

```
2
```

```
julia> indmin([7,1,1,NaN])
```

```
4
```

[source](#)

`Base.findmax` – Method.

```
| findmax(itr) -> (x, index)
```

Returns the maximum element of the collection `itr` and its index. If there are multiple maximal elements, then the first one will be returned. If any data element is `NaN`, this element is returned. The result is in line with `max`.

The collection must not be empty.

Examples

```
julia> findmax([8,0.1,-9,pi])
```

```
(8.0, 1)
```

```
julia> findmax([1,7,7,6])
```

```
(7, 2)
```

```
julia> findmax([1,7,7,NaN])
```

```
(NaN, 4)
```

[source](#)

`Base.findmax` – Method.

```
| findmax(A, region) -> (maxval, index)
```

888 For an array input, CHAPTER 47 VALIDATIONS AND DATA STRUCTURES given region. **NaN** is treated as greater than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0

julia> findmax(A,1)
([3.0 4.0], CartesianIndex{2}[CartesianIndex(2, 1)
    ↳ CartesianIndex(2, 2)])

julia> findmax(A,2)
([2.0; 4.0], CartesianIndex{2}[CartesianIndex(1, 2);
    ↳ CartesianIndex(2, 2)])
```

**source**

[Base.findmin](#) – Method.

```
| findmin(itr) -> (x, index)
```

Returns the minimum element of the collection **itr** and its index. If there are multiple minimal elements, then the first one will be returned. If any data element is **NaN**, this element is returned. The result is in line with **min**.

The collection must not be empty.

Examples

```
julia> findmin([8,0.1,-9,pi])
(-9.0, 3)

julia> findmin([7,1,1,6])
```

```
julia> findmin([7,1,1,NaN])
(NaN, 4)
```

source

Base.**findmin** – Method.

```
| findmin(A, region) -> (minval, index)
```

For an array input, returns the value and index of the minimum over the given region. **NaN** is treated as less than all other values.

Examples

```
julia> A = [1.0 2; 3 4]
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0
```

```
julia> findmin(A, 1)
([1.0 2.0], CartesianIndex{2}[CartesianIndex(1, 1)
 →  CartesianIndex(1, 2)])
```

```
julia> findmin(A, 2)
([1.0; 3.0], CartesianIndex{2}[CartesianIndex(1, 1);
 →  CartesianIndex(2, 1)])
```

source

Base.**findmax!** – Function.

```
| findmax!(rval, rind, A, [init=true]) -> (maxval, index)
```

890 Find the maximum of **CHAPTER 47 COMPUTATIONS AND DATA STRUCTURES** dimensions of **rval** and **rind**, and store the results in **rval** and **rind**. **Nan** is treated as greater than all other values.

**source**

**Base.findmin!** – Function.

```
| findmin!(rval, rind, A, [init=true]) -> (minval, index)
```

Find the minimum of **A** and the corresponding linear index along singleton dimensions of **rval** and **rind**, and store the results in **rval** and **rind**. **Nan** is treated as less than all other values.

**source**

**Base.sum** – Function.

```
| sum(f, itr)
```

Sum the results of calling function **f** on each element of **itr**.

The return type is **Int** for signed integers of less than system word size, and **UInt** for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

```
| julia> sum(abs2, [2; 3; 4])
```

29

Note the important difference between **sum(A)** and **reduce(+, A)** for arrays with small integer eltype:

```
| julia> sum(Int8[100, 28])
```

128

```
| julia> reduce(+, Int8[100, 28])
```

-128

47.3 In **ITERABLE COLLECTIONS**, integers are widened to system word size ~~891~~ therefore the result is 128. In the latter case, no such widening happens and integer overflow results in -128.

**source**

```
| sum(itr)
```

Returns the sum of all elements in a collection.

The return type is **Int** for signed integers of less than system word size, and **UInt** for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

```
| julia> sum(1:20)
```

```
210
```

**source**

```
| sum(A, dims)
```

Sum elements of an array over the given dimensions.

Examples

```
| julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
 1  2
```

```
 3  4
```

```
| julia> sum(A, 1)
```

```
1×2 Array{Int64,2}:
```

```
 4  6
```

```
| julia> sum(A, 2)
```

```
2×1 Array{Int64,2}:
```

**source****Base.sum!** – Function.**sum!(r, A)**

Sum elements of **A** over the singleton dimensions of **r**, and write results to **r**.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> sum!([1; 1], A)
2-element Array{Int64,1}:
 3
 7

julia> sum!([1 1], A)
1×2 Array{Int64,2}:
 4  6
```

**source****Base.prod** – Function.**prod(f, itr)**

Returns the product of **f** applied to each element of **itr**.

## 47.3. THE TABLE COLLECTIONS

signed integers of less than system word size<sup>323</sup> and UInt for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

```
julia> prod(abs2, [2; 3; 4])
```

```
576
```

[source](#)

```
| prod(itr)
```

Returns the product of all elements of a collection.

The return type is Int for signed integers of less than system word size, and UInt for unsigned integers of less than system word size. For all other arguments, a common return type is found to which all arguments are promoted.

```
julia> prod(1:20)
```

```
2432902008176640000
```

[source](#)

```
| prod(A, dims)
```

Multiply elements of an array over the given dimensions.

Examples

```
julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
 1  2
```

```
 3  4
```

```
julia> prod(A, 1)
```

```
1×2 Array{Int64,2}:
```

```
julia> prod(A, 2)
2×1 Array{Int64,2}:
 2
 12
```

source

[Base.prod!](#) – Function.

```
| prod!(r, A)
```

Multiply elements of  $A$  over the singleton dimensions of  $r$ , and write results to  $r$ .

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> prod!([1; 1], A)
2-element Array{Int64,1}:
 2
 12

julia> prod!([1 1], A)
1×2 Array{Int64,2}:
 3  8
```

source

[Base.any](#) – Method.

Test whether any elements of a boolean collection are `true`, returning `true` as soon as the first `true` value in `itr` is encountered (short-circuiting).

```
julia> a = [true, false, false, true]
4-element Array{Bool,1}:
  true
  false
  false
  true

julia> any(a)
true

julia> any((println(i); v) for (i, v) in enumerate(a))
1
true
```

source

[Base.any](#) – Method.

| `any(A, dims)`

Test whether any values along the given dimensions of an array are `true`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
  true  false
  true  false
```

896 | **julia>** any(A, 1) CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
1×2 Array{Bool,2}:
 true  false
```

```
julia> any(A, 2)
```

```
2×1 Array{Bool,2}:
 true
 true
```

**source**

[Base.any!](#) – Function.

```
| any!(r, A)
```

Test whether any values in A along the singleton dimensions of r are `true`, and write results to r.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false
```

```
julia> any!([1; 1], A)
```

```
2-element Array{Int64,1}:
 1
 1
```

```
julia> any!([1 1], A)
```

```
1×2 Array{Int64,2}:
 1  0
```

**source**

```
| all(itr) -> Bool
```

Test whether all elements of a boolean collection are `true`, returning `false` as soon as the first `false` value in `itr` is encountered (short-circuiting).

```
julia> a = [true, false, false, true]
```

```
4-element Array{Bool,1}:
  true
  false
  false
  true
```

```
julia> all(a)
```

```
false
```

```
julia> all((println(i); v) for (i, v) in enumerate(a))
```

```
1
2
false
```

`source`

[Base.all](#) – Method.

```
| all(A, dims)
```

Test whether all values along the given dimensions of an array are `true`.

Examples

```
julia> A = [true false; true true]
```

```
2×2 Array{Bool,2}:
  true  false
```

898 true true CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
julia> all(A, 1)
1×2 Array{Bool,2}:
 true  false

julia> all(A, 2)
2×1 Array{Bool,2}:
 false
 true
```

source

Base.all! – Function.

```
| all!(r, A)
```

Test whether all values in A along the singleton dimensions of r are true, and write results to r.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false

julia> all!([1; 1], A)
2-element Array{Int64,1}:
 0
 0

julia> all!([1 1], A)
1×2 Array{Int64,2}:
 1  0
```

`Base.count` – Function.

```
| count(p, itr) -> Integer  
| count(itr) -> Integer
```

Count the number of elements in `itr` for which predicate `p` returns `true`. If `p` is omitted, counts the number of `true` elements in `itr` (which should be a collection of boolean values).

```
julia> count(i->(4<=i<=6), [2,3,4,5,6])
```

```
3
```

```
julia> count([true, false, true, true])
```

```
3
```

`source`

```
LibGit2.count(f::Function, walker::GitRevWalker; oid::GitHash=  
    GitHash(), by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` `walker` to "walk" over every commit in the repository's history, find the number of commits which return `true` when `f` is applied to them. The keyword arguments are:

- \* `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors.
- \* `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPLOGICAL`), to sort forwards in time (`LibGit2.Consts.most_ancient_first`) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first).
- \* `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

900 | cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker  
CHAPTER 47. COLLECTIONS AND DATA STRUCTURES  
| count((oid, repo)->(oid == commit\_oid1), walker,  
| → oid=commit\_oid1, by=LibGit2.Consts.SORT\_TIME)  
| end

`count` finds the number of commits along the walk with a certain `GitHash` `commit_oid1`, starting the walk from that commit and moving forwards in time from it. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`source`

`Base.any` – Method.

| `any(p, itr) -> Bool`

Determine whether predicate `p` returns `true` for any elements of `itr`, returning `true` as soon as the first item in `itr` for which `p` returns `true` is encountered (short-circuiting).

**julia>** `any(i->(4<=i<=6), [3,5,7])`

`true`

**julia>** `any(i -> (println(i); i > 3), 1:10)`

`1`

`2`

`3`

`4`

`true`

`source`

`Base.all` – Method.

| `all(p, itr) -> Bool`

47.3 ITERABLE COLLECTIONS

Sate `p` returns `true` for all elements of `itr`, 901 turning `false` as soon as the first item in `itr` for which `p` returns `false` is encountered (short-circuiting).

```
julia> all(i->(4<=i<=6), [4,5,6])
true

julia> all(i -> (println(i); i < 3), 1:10)
1
2
3
false
```

`source`

`Base.foreach` – Function.

```
| foreach(f, c...) -> Void
```

Call function `f` on each element of iterable `c`. For multiple iterable arguments, `f` is called elementwise. `foreach` should be used instead of `map` when the results of `f` are not needed, for example in `foreach(println, array)`.

Examples

```
julia> a = 1:3:7;

julia> foreach(x -> println(x^2), a)
1
16
49
```

`source`

`Base.map` – Function.

902 | `map(f, c...)` -> CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

Transform collection `c` by applying `f` to each element. For multiple collection arguments, apply `f` elementwise.

See also: [mapslices](#)

Examples

```
julia> map(x -> x * 2, [1, 2, 3])
```

```
3-element Array{Int64,1}:
```

```
2
```

```
4
```

```
6
```

```
julia> map(+, [1, 2, 3], [10, 20, 30])
```

```
3-element Array{Int64,1}:
```

```
11
```

```
22
```

```
33
```

[source](#)

```
| map(f, x::Nullable)
```

Return `f` applied to the value of `x` if it has one, as a `Nullable`. If `x` is null, then return a null value of type `Nullable{S}`. `S` is guaranteed to be either `Union{}` or a concrete type. Whichever of these is chosen is an implementation detail, but typically the choice that maximizes performance would be used. If `x` has a value, then the return type is guaranteed to be of type `Nullable{typeof(f(x))}`.

Examples

```
julia> map(isodd, Nullable(1))
```

```
Nullable{Bool}(true)
```

```
julia> map(isodd, Nullable(2))
Nullable{Bool}(false)

julia> map(isodd, Nullable{Int}())
Nullable{Bool}()
```

#### source

```
LibGit2.map(f::Function, walker::GitRevWalker; oid::GitHash=
    GitHash(), range::AbstractString="", by::Cint=Consts.
    SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` walker to "walk" over every commit in the repository's history, apply `f` to each commit in the walk. The keyword arguments are:

- \* `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors.
- \* `range`: A range of `GitHashes` in the format `oid1..oid2`. `f` will be applied to all commits between the two.
- \* `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first).
- \* `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

#### Examples

```
oids = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.map((oid, repo)->string(oid), walker,
        → by=LibGit2.Consts.SORT_TIME)
end
```

Here, `map` visits each commit using the `GitRevWalker` and finds its `GitHash`.

`Base.map!` – Function.

```
| map!(function, destination, collection...)
```

Like `map`, but stores the result in `destination` rather than a new collection. `destination` must be at least as large as the first collection.

Examples

```
| julia> x = zeros(3);
```

```
| julia> map!(x -> x * 2, x, [1, 2, 3]);
```

```
| julia> x
```

```
3-element Array{Float64,1}:
```

```
 2.0
```

```
 4.0
```

```
 6.0
```

`source`

`Base.mapreduce` – Method.

```
| mapreduce(f, op, v0, itr)
```

Apply function `f` to each element in `itr`, and then reduce the result using the binary function `op`. `v0` must be a neutral element for `op` that will be returned for empty collections. It is unspecified whether `v0` is used for non-empty collections.

`mapreduce` is functionally equivalent to calling `reduce(op, v0, map(f, itr))`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `reduce` and `map`.

```
| julia> mapreduce(x->x^2, +, [1:3;]) # == 1 + 4 + 9
```

This reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `itr`. Use `mapfoldl` or `mapfoldr` instead for guaranteed left or right associativity and invocation of `f` for every value.

[source](#)

`Base.mapreduce` – Method.

```
| mapreduce(f, op, itr)
```

Like `mapreduce(f, op, v0, itr)`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

[source](#)

`Base.mapfoldl` – Method.

```
| mapfoldl(f, op, v0, itr)
```

Like `mapreduce`, but with guaranteed left associativity, as in `foldl`. `v0` will be used exactly once.

[source](#)

`Base.mapfoldl` – Method.

```
| mapfoldl(f, op, itr)
```

Like `mapfoldl(f, op, v0, itr)`, but using the first element of `itr` to generate `v0`. Specifically, `mapfoldl(f, op, itr)` produces the same result as `mapfoldl(f, op, f(first(itr)), drop(itr, 1))`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

[source](#)

`Base.mapfoldr` – Method.

906| `mapfoldr(f, op, v0, itr)` CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

Like `mapreduce`, but with guaranteed right associativity, as in `foldr`. `v0` will be used exactly once.

`source`

`Base.mapfoldr` – Method.

| `mapfoldr(f, op, itr)`

Like `mapfoldr(f, op, v0, itr)`, but using the first element of `itr` to generate `v0`. Specifically, `mapfoldr(f, op, itr)` produces the same result as `mapfoldr(f, op, f(last(itr)), take(itr, length(itr)-1))`. In general, this cannot be used with empty collections (see `reduce(op, itr)`).

`source`

`Base.first` – Function.

| `first(coll)`

Get the first element of an iterable collection. Returns the start point of an `AbstractRange` even if it is empty.

Examples

| `julia> first(2:2:10)`

| 2

| `julia> first([1; 2; 3; 4])`

| 1

`source`

| `first(str::AbstractString, nchar::Integer)`

```
julia> first("≠0: >0", 0)
" "
julia> first("≠0: >0", 1)
"≠"
julia> first("≠0: >0", 3)
"≠"
```

source

`Base.last` – Function.

```
| last(coll)
```

Get the last element of an ordered collection, if it can be computed in O(1) time. This is accomplished by calling `endof` to get the last index. Returns the end point of an `AbstractRange` even if it is empty.

Examples

```
julia> last(1:2:10)
9
julia> last([1; 2; 3; 4])
4
```

source

```
| last(str::AbstractString, nchar::Integer)
```

Get a string consisting of the last `nchar` characters of `str`.

```
julia> last("≠0: >0", 0)
" "
```

```
julia> last("≠0: ^>0", 1)  
"0"
```

```
julia> last("≠0: ^>0", 3)  
"^>0"
```

source

[Base.step](#) – Function.

```
| step(r)
```

Get the step size of an `AbstractRange` object.

```
julia> step(1:10)  
1
```

```
julia> step(1:2:10)  
2
```

```
julia> step(2.5:0.3:10.9)  
0.3
```

```
julia> step(linspace(2.5,10.9,85))  
0.1
```

source

[Base.collect](#) – Method.

```
| collect(collection)
```

Return an `Array` of all items in a collection or iterator. For associative collections, returns `Pair{KeyType, ValType}`. If the argument is array-

47.3 like ITERABLE COLLECTIONS HasShape( ) trait, the result will have the same shape and number of dimensions as the argument.

Examples

```
julia> collect(1:2:13)
7-element Array{Int64,1}:
 1
 3
 5
 7
 9
 11
 13
```

source

[Base.collect](#) – Method.

```
| collect(element_type, collection)
```

Return an **Array** with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as **collection**.

Examples

```
julia> collect(Float64, 1:2:5)
3-element Array{Float64,1}:
 1.0
 3.0
 5.0
```

source

[Base.issubset](#) – Method.

```
910| issubset(a, b)    CHAPTER 47. COLLECTIONS AND DATA STRUCTURES
| (a,b) -> Bool
| (a,b) -> Bool
| (a,b) -> Bool
```

Determine whether every element of **a** is also in **b**, using [in](#).

Examples

```
| julia> issubset([1, 2], [1, 2, 3])
true
```

```
| julia> issubset([1, 2, 3], [1, 2])
false
```

[source](#)

[Base.filter](#) – Function.

```
| filter(f, a::AbstractArray)
```

Return a copy of **a**, removing elements for which **f** is **false**. The function **f** is passed one argument.

Examples

```
| julia> a = 1:10
1:10
```

```
| julia> filter(isodd, a)
5-element Array{Int64,1}:
 1
 3
 5
 7
 9
```

```
| filter(f, d::Associative)
```

Return a copy of `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Examples

```
julia> d = Dict(1=>"a", 2=>"b")
Dict{Int64,String} with 2 entries:
  2 => "b"
  1 => "a"
```

```
julia> filter(p->isodd(p.first), d)
Dict{Int64,String} with 1 entry:
  1 => "a"
```

source

```
| filter(p, x::Nullable)
```

Return `null` if either `x` is `null` or `p(get(x))` is `false`, and `x` otherwise.

Examples

```
julia> filter(isodd, Nullable(5))
Nullable{Int64}(5)
```

```
julia> filter(isodd, Nullable(4))
Nullable{Int64}()
```

```
julia> filter(isodd, Nullable{Int}())
Nullable{Int64}()
```

source

## Base.filter! – Function

### CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
| filter!(f, a::AbstractVector)
```

Update `a`, removing elements for which `f` is `false`. The function `f` is passed one argument.

Examples

```
| julia> filter!(isodd, collect(1:10))
```

```
| 5-element Array{Int64,1}:
```

```
|   1
```

```
|   3
```

```
|   5
```

```
|   7
```

```
|   9
```

source

```
| filter!(f, d::Associative)
```

Update `d`, removing elements for which `f` is `false`. The function `f` is passed `key=>value` pairs.

Example

```
| julia> d = Dict(1=>"a", 2=>"b", 3=>"c")
```

```
| Dict{Int64,String} with 3 entries:
```

```
|   2 => "b"
```

```
|   3 => "c"
```

```
|   1 => "a"
```

```
| julia> filter!(p->isodd(p.first), d)
```

```
| Dict{Int64,String} with 2 entries:
```

```
|   3 => "c"
```

```
|   1 => "a"
```

## 47.4 Indexable Collections

`Base.getindex` – Function.

```
| getindex(collection, key...)
```

Retrieve the value(s) stored at the given key or index within a collection.

The syntax `a[i, j, ...]` is converted by the compiler to `getindex(a, i, j, ...)`.

Examples

```
julia> A = Dict("a" => 1, "b" => 2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> getindex(A, "a")
1
```

`source`

`Base.setindex!` – Function.

```
| setindex!(collection, value, key...)
```

Store the given value at the given key or index within a collection. The syntax `a[i, j, ...] = x` is converted by the compiler to `(setindex!(a, x, i, j, ...); x)`.

`source`

`Base.endof` – Function.

```
| endof(collection) -> Integer
```

Examples

```
julia> endof([1,2,4])  
3
```

source

Fully implemented by:

[Array](#)

[BitArray](#)

[AbstractArray](#)

[SubArray](#)

Partially implemented by:

[AbstractRange](#)

[UnitRange](#)

[Tuple](#)

[AbstractString](#)

[Dict](#)

[ObjectIdDict](#)

[WeakKeyDict](#)

## 47.5 Associative Collections

[Dict](#) is the standard associative collection. Its implementation uses [hash](#) as the hashing function for the key, and [isequal](#) to determine equality. Define

These associative collections allow you to override how they are stored in a hash table.

`ObjectIdDict` is a special hash table where the keys are always object identities.

`WeakKeyDict` is a hash table implementation where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

`Dict`s can be created by passing pair objects constructed with `=>` to a `Dict` constructor: `Dict("A"=>1, "B"=>2)`. This call will attempt to infer type information from the keys and values (i.e. this example creates a `Dict{String, Int64}`). To explicitly specify types use the syntax `Dict{KeyType, ValueType}(...)`. For example, `Dict{String, Int32}("A"=>1, "B"=>2)`.

Associative collections may also be created with generators. For example, `Dict(i => f(i) for i = 1:10)`.

Given a dictionary `D`, the syntax `D[x]` returns the value of key `x` (if it exists) or throws an error, and `D[x] = y` stores the key-value pair `x => y` in `D` (replacing any existing value for the key `x`). Multiple arguments to `D[...]` are converted to tuples; for example, the syntax `D[x, y]` is equivalent to `D[(x, y)]`, i.e. it refers to the value keyed by the tuple `(x, y)`.

`Base.Dict` – Type.

```
| Dict([itr])
```

`Dict{K,V}()` constructs a hash table with keys of type `K` and values of type `V`.

Given a single iterable argument, constructs a `Dict` whose key-value pairs are taken from 2-tuples `(key, value)` generated by the argument.

```
| julia> Dict([( "A", 1), ( "B", 2)])  
Dict{String,Int64} with 2 entries:
```

916 | "B" => 2  
| "A" => 1

## CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

Alternatively, a sequence of pair arguments may be passed.

```
| julia> Dict("A"=>1, "B"=>2)  
Dict{String,Int64} with 2 entries:  
  "B" => 2  
  "A" => 1
```

[source](#)

[Base.ObjectDict](#) – Type.

```
| ObjectDict([itr])
```

`ObjectDict()` constructs a hash table where the keys are (always) object identities. Unlike `Dict` it is not parameterized on its key and value type and thus its `eltype` is always `Pair{Any, Any}`.

See [Dict](#) for further help.

[source](#)

[Base.WeakKeyDict](#) – Type.

```
| WeakKeyDict([itr])
```

`WeakKeyDict()` constructs a hash table where the keys are weak references to objects, and thus may be garbage collected even when referenced in a hash table.

See [Dict](#) for further help.

[source](#)

[Base.haskey](#) – Function.

```
| haskey(collection, key) -> Bool
```

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> haskey(a, 'a')
true

julia> haskey(a, 'c')
false
```

source

`Base.get` – Method.

```
| get(collection, key, default)
```

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);

julia> get(d, "a", 3)
1

julia> get(d, "c", 3)
3
```

source

`Base.get` – Function.

```
| get(collection, key, default)
```

918 Return the value stored for the given key, or if no mapping for the key is present.

Examples

```
julia> d = Dict("a"=>1, "b"=>2);
```

```
julia> get(d, "a", 3)
```

```
1
```

```
julia> get(d, "c", 3)
```

```
3
```

source

```
| get(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, return `f()`. Use `get!` to also store the default value in the dictionary.

This is intended to be called using `do` block syntax

```
| get(dict, key) do
|     # default value calculated here
|     time()
| end
```

source

```
| get(x::Nullable[, y])
```

Attempt to access the value of `x`. Returns the value if it is present; otherwise, returns `y` if provided, or throws a `NullException` if not.

Examples

```
julia> get(Nullable(5))  
5  
  
julia> get(Nullable())  
ERROR: NullException()  
Stacktrace:  
[ ... ]
```

**source**

`Base.get!` – Method.

```
| get!(collection, key, default)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => default`, and return `default`.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);
```

```
julia> get!(d, "a", 5)
```

```
1
```

```
julia> get!(d, "d", 4)
```

```
4
```

```
julia> d
```

Dict{String,Int64} with 4 entries:

```
  "c" => 3  
  "b" => 2  
  "a" => 1  
  "d" => 4
```

**source**

`Base.get!` – Method.CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
| get!(f::Function, collection, key)
```

Return the value stored for the given key, or if no mapping for the key is present, store `key => f()`, and return `f()`.

This is intended to be called using `do` block syntax:

```
| get!(dict, key) do
    # default value calculated here
    time()
end
```

`source`

`Base.getkey` – Function.

```
| getkey(collection, key, default)
```

Return the key matching argument `key` if one exists in `collection`, otherwise return `default`.

```
julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2

julia> getkey(a, 'a', 1)
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)

julia> getkey(a, 'd', 'a')
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

`source`

`Base.delete!` – Function.

Delete the mapping for the given key in a collection, and return the collection.

Examples

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1
```

```
julia> delete!(d, "b")
Dict{String,Int64} with 1 entry:
  "a" => 1
```

source

`Base.pop!` – Method.

```
| pop!(collection, key[, default])
```

Delete and return the mapping for `key` if it exists in `collection`, otherwise return `default`, or throw an error if `default` is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);
```

```
julia> pop!(d, "a")
```

```
1
```

```
julia> pop!(d, "d")
```

```
ERROR: KeyError: key "d" not found
```

```
Stacktrace:
```

```
[...]
```

```
| julia> pop!(d, "e", 4)
| 4
```

**source**

**Base.keys** – Function.

```
| keys(iterator)
```

For an iterator or collection that has keys and values (e.g. arrays and dictionaries), return an iterator over the keys.

**source**

**Base.values** – Function.

```
| values(iterator)
```

For an iterator or collection that has keys and values, return an iterator over the values. This function simply returns its argument by default, since the elements of a general iterator are normally considered its “values”.

**source**

```
| values(a::Associative)
```

Return an iterator over all values in a collection. `collect(values(a))` returns an array of values. Since the values are stored internally in a hash table, the order in which they are returned may vary. But `keys(a)` and `values(a)` both iterate `a` and return the elements in the same order.

Examples

```
| julia> a = Dict('a'=>2, 'b'=>3)
Dict{Char,Int64} with 2 entries:
  'b' => 3
  'a' => 2
```

```
julia> collect(values(a))  
2-element Array{Int64,1}:  
3  
2
```

**source**

[Base.pairs](#) – Function.

```
| pairs(collection)
```

Return an iterator over `key => value` pairs for any collection that maps a set of keys to a set of values. This includes arrays, where the keys are the array indices.

**source**

```
| pairs(IndexLinear(), A)  
| pairs(IndexCartesian(), A)  
| pairs(IndexStyle(A), A)
```

An iterator that accesses each element of the array `A`, returning `i => x`, where `i` is the index for the element and `x = A[i]`. Identical to `pairs(A)`, except that the style of index can be selected. Also similar to `enumerate(A)`, except `i` will be a valid index for `A`, while `enumerate` always counts from 1 regardless of the indices of `A`.

Specifying `IndexLinear()` ensures that `i` will be an integer; specifying `IndexCartesian()` ensures that `i` will be a `CartesianIndex`; specifying `IndexStyle(A)` chooses whichever has been defined as the native indexing style for array `A`.

Examples

```
julia> A = ["a" "d"; "b" "e"; "c" "f"];
```

924

CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
julia> for (index, value) in pairs(IndexStyle(A), A)
           println("$index $value")

       end
1 a
2 b
3 c
4 d
5 e
6 f

julia> S = view(A, 1:2, :);

julia> for (index, value) in pairs(IndexStyle(S), S)
           println("$index $value")

       end
CartesianIndex(1, 1) a
CartesianIndex(2, 1) b
CartesianIndex(1, 2) d
CartesianIndex(2, 2) e
```

See also: [IndexStyle](#), [indices](#).

[source](#)

[Base.merge](#) – Function.

```
| merge(d::Associative, others::Associative...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate

If the same key is present in another collection, the value for that key will be the value it has in the last collection listed.

### Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)
```

```
Dict{String,Float64} with 2 entries:
```

```
    "bar" => 42.0  
    "foo" => 0.0
```

```
julia> b = Dict("baz" => 17, "bar" => 4711)
```

```
Dict{String,Int64} with 2 entries:
```

```
    "bar" => 4711  
    "baz" => 17
```

```
julia> merge(a, b)
```

```
Dict{String,Float64} with 3 entries:
```

```
    "bar" => 4711.0  
    "baz" => 17.0  
    "foo" => 0.0
```

```
julia> merge(b, a)
```

```
Dict{String,Float64} with 3 entries:
```

```
    "bar" => 42.0  
    "baz" => 17.0  
    "foo" => 0.0
```

### source

```
| merge(combine, d::Associative, others::Associative...)
```

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate

926 the types of the merged collections will be the same as the structures  
CHAPTER 17. COLLECTIONS AND DATA STRUCTURES  
combined using the combiner function.

### Examples

```
julia> a = Dict("foo" => 0.0, "bar" => 42.0)  
Dict{String,Float64} with 2 entries:  
    "bar" => 42.0  
    "foo" => 0.0
```

```
julia> b = Dict("baz" => 17, "bar" => 4711)  
Dict{String,Int64} with 2 entries:  
    "bar" => 4711  
    "baz" => 17
```

```
julia> merge(+, a, b)  
Dict{String,Float64} with 3 entries:  
    "bar" => 4753.0  
    "baz" => 17.0  
    "foo" => 0.0
```

### source

```
| merge(a::NamedTuple, b::NamedTuple)
```

Construct a new named tuple by merging two existing ones. The order of fields in **a** is preserved, but values are taken from matching fields in **b**. Fields present only in **b** are appended at the end.

```
julia> merge((a=1, b=2, c=3), (b=4, d=5))  
(a = 1, b = 4, c = 3, d = 5)
```

### source

```
| merge(a::NamedTuple, iterable)
```

47.5 **ASSOCIATIVE COLLECTIONS** Value pairs as a named tuple, and perform a merge.

```
julia> merge((a=1, b=2, c=3), [ :b=>4, :d=>5])
(a = 1, b = 4, c = 3, d = 5)
```

source

[Base.merge!](#) – Method.

```
merge!(d::Associative, others::Associative...)
```

Update collection with pairs from the other collections. See also [merge](#).

Examples

```
julia> d1 = Dict(1 => 2, 3 => 4);
```

```
julia> d2 = Dict(1 => 4, 4 => 5);
```

```
julia> merge!(d1, d2);
```

```
julia> d1
```

```
Dict{Int64,Int64} with 3 entries:
```

```
 4 => 5
```

```
 3 => 4
```

```
 1 => 4
```

source

[Base.merge!](#) – Method.

```
merge!(combine, d::Associative, others::Associative...)
```

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

Examples

928 | **julia>** d1 = Dict(1 => 2, 3 => 4); CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

**julia>** d2 = Dict(1 => 4, 4 => 5);

**julia>** merge!(+, d1, d2);

**julia>** d1

Dict{Int64, Int64} with 3 entries:

4 => 5

3 => 4

1 => 6

**julia>** merge!(-, d1, d1);

**julia>** d1

Dict{Int64, Int64} with 3 entries:

4 => 0

3 => 0

1 => 0

source

**Base.sizehint!** – Function.

| sizehint!(s, n)

Suggest that collection **s** reserve capacity for at least **n** elements. This can improve performance.

source

**Base.keytype** – Function.

| keytype(type)

47.5. ASSOCIATIVE COLLECTIONS  
Associative collection type. Behaves similarly to [eltype](#).

Examples

```
| julia> keytype(Dict(Int32(1) => "foo"))  
| Int32
```

source

[Base.valtype](#) – Function.

```
| valtype(type)
```

Get the value type of an associative collection type. Behaves similarly to [eltype](#).

Examples

```
| julia> valtype(Dict(Int32(1) => "foo"))  
| String
```

source

Fully implemented by:

[ObjectIdDict](#)

[Dict](#)

[WeakKeyDict](#)

Partially implemented by:

[BitSet](#)

[Set](#)

[EnvDict](#)

[BitArray](#)

## 47.6 Set-Like Collections

[Base.Set](#) – Type.

```
| Set([itr])
```

Construct a [Set](#) of the values generated by the given iterable object, or an empty set. Should be used instead of [BitSet](#) for sparse integer sets, or for sets of arbitrary objects.

[source](#)

[Base.BitSet](#) – Type.

```
| BitSet([itr])
```

Construct a sorted set of positive [Ints](#) generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only [Ints](#) greater than 0 can be stored. If the set will be sparse (for example holding a few very large integers), use [Set](#) instead.

[source](#)

[Base.union](#) – Function.

```
| union(s1, s2...)
| (s1, s2...)
```

Construct the union of two or more sets. Maintains order with arrays.

Examples

```
| julia> union([1, 2], [3, 4])
| 4-element Array{Int64,1}:
```

```
2  
3  
4
```

```
julia> union([1, 2], [2, 4])
```

```
3-element Array{Int64,1}:
```

```
1  
2  
4
```

```
julia> union([4, 2], [1, 2])
```

```
3-element Array{Int64,1}:
```

```
4  
2  
1
```

**source**

**Base.union!** – Function.

```
|union!(s, iterable)
```

Union each element of `iterable` into set `s` in-place.

Examples

```
julia> a = Set([1, 3, 4, 5]);
```

```
julia> union!(a, 1:2:8);
```

```
julia> a
```

```
Set([7, 4, 3, 5, 1])
```

**source**

## Base.intersect – Function

CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

```
| intersect(s1, s2...)
| (s1, s2)
```

Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

Examples

```
julia> intersect([1, 2, 3], [3, 4, 5])
1-element Array{Int64,1}:
 3

julia> intersect([1, 4, 4, 5, 6], [4, 6, 6, 7, 8])
3-element Array{Int64,1}:
 4
 4
 6
```

source

## Base.setdiff – Function.

```
| setdiff(a, b)
```

Construct the set of elements in **a** but not **b**. Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, **setdiff(set, element)** where **element** is a potential member of **set**, will not work in general.

Examples

```
julia> setdiff([1,2,3],[3,4,5])
2-element Array{Int64,1}:
 1
 2
```

`Base.setdiff!` – Function.

```
| setdiff!(s, iterable)
```

Remove each element of `iterable` from set `s` in-place.

Examples

```
| julia> a = Set([1, 3, 4, 5]);
```

```
| julia> setdiff!(a, 1:2:6);
```

```
| julia> a
```

```
Set([4])
```

`source`

`Base.symdiff` – Function.

```
| symdiff(a, b, rest...)
```

Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

Examples

```
| julia> symdiff([1,2,3],[3,4,5],[4,5,6])
```

```
3-element Array{Int64,1}:
```

```
1
```

```
2
```

```
6
```

`source`

`Base.symdiff!` – Method.

```
| symdiff!(s, n)
```

`source`

`Base.symdiff!` – Method.

```
| symdiff!(s, itr)
```

For each element in `itr`, destructively toggle its inclusion in set `s`.

`source`

`Base.symdiff!` – Method.

```
| symdiff!(s, itr)
```

For each element in `itr`, destructively toggle its inclusion in set `s`.

`source`

`Base.intersect!` – Function.

```
| intersect!(s1::BitSet, s2::BitSet)
```

Intersects sets `s1` and `s2` and overwrites the set `s1` with the result. If needed, `s1` will be expanded to the size of `s2`.

`source`

`Base.issubset` – Function.

```
| issubset(a, b)
  (a,b) -> Bool
  (a,b) -> Bool
  (a,b) -> Bool
```

Determine whether every element of `a` is also in `b`, using `in`.

Examples

```
| julia> issubset([1, 2], [1, 2, 3])
| true
```

```
julia> issubset([1, 2, 3], [1, 2])
false
```

[source](#)

Fully implemented by:

[BitSet](#)

[Set](#)

Partially implemented by:

[Array](#)

## 47.7 Dequeues

[Base.push!](#) – Function.

```
| push!(collection, items...) -> collection
```

Insert one or more `items` at the end of `collection`.

Examples

```
julia> push!([1, 2, 3], 4, 5, 6)
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

936 Use `append!` to add the elements of one collection to another.

The result of the preceding example is equivalent to `append!([1, 2, 3], [4, 5, 6])`.

`source`

`Base.pop!` – Function.

`| pop!(collection) -> item`

Remove an item in `collection` and return it. If `collection` is an ordered container, the last item is returned.

Examples

```
julia> A=[1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> pop!(A)
3
```

```
julia> A
2-element Array{Int64,1}:
 1
 2
```

```
julia> S = Set([1, 2])
Set([2, 1])
```

```
julia> pop!(S)
2
```

```
julia> S
```

```
Set([1])
```

```
julia> pop!(Dict(1=>2))
```

```
1 => 2
```

```
source
```

```
| pop!(collection, key[, default])
```

Delete and return the mapping for **key** if it exists in **collection**, otherwise return **default**, or throw an error if **default** is not specified.

Examples

```
julia> d = Dict("a"=>1, "b"=>2, "c"=>3);
```

```
julia> pop!(d, "a")
```

```
1
```

```
julia> pop!(d, "d")
```

```
ERROR: KeyError: key "d" not found
```

```
Stacktrace:
```

```
[...]
```

```
julia> pop!(d, "e", 4)
```

```
4
```

```
source
```

[Base.unshift!](#) – Function.

```
| unshift!(collection, items...) -> collection
```

Insert one or more **items** at the beginning of **collection**.

Examples

938 | **julia>** unshift!([1, 2, 3, 4], 5, 6) CHAPTER 47, COLLECTIONS AND DATA STRUCTURES

```
6-element Array{Int64,1}:
 5
 6
 1
 2
 3
 4
```

source

[Base.shift!](#) – Function.

```
| shift!(collection) -> item
```

Remove the first `item` from `collection`.

Examples

```
julia> A = [1, 2, 3, 4, 5, 6]
```

```
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

```
julia> shift!(A)
```

```
1
```

```
julia> A
```

```
5-element Array{Int64,1}:
 2
```

```
source
```

[Base.insert!](#) – Function.

```
insert!(a::Vector, index::Integer, item)
```

Insert an `item` into `a` at the given `index`. `index` is the index of `item` in the resulting `a`.

Examples

```
julia> insert!([6, 5, 4, 2, 1], 4, 3)
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

```
source
```

[Base.deleteat!](#) – Function.

```
deleteat!(a::Vector, i::Integer)
```

Remove the item at the given `i` and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 2)
5-element Array{Int64,1}:
```

```
6  
4  
3  
2  
1
```

### source

```
| deleteat!(a::Vector, inds)
```

Remove the items at the indices given by `inds`, and return the modified `a`. Subsequent items are shifted to fill the resulting gap.

`inds` can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as `a` with `true` indicating entries to delete.

### Examples

```
julia> deleteat!([6, 5, 4, 3, 2, 1], 1:2:5)
```

```
3-element Array{Int64,1}:
```

```
5  
3  
1
```

```
julia> deleteat!([6, 5, 4, 3, 2, 1], [true, false, true,  
    ↪ false, true, false])
```

```
3-element Array{Int64,1}:
```

```
5  
3  
1
```

```
julia> deleteat!([6, 5, 4, 3, 2, 1], (2, 2))
```

```
ERROR: ArgumentError: indices must be unique and sorted
```

source

[Base.splice!](#) – Function.

```
splice!(a::Vector, index::Integer, [replacement]) -> item
```

Remove the item at the given index, and return the removed item. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed item.

Examples

```
julia> A = [6, 5, 4, 3, 2, 1]; splice!(A, 5)
```

```
2
```

```
julia> A
```

```
5-element Array{Int64,1}:
```

```
6  
5  
4  
3  
1
```

```
julia> splice!(A, 5, -1)
```

```
1
```

```
julia> A
```

```
5-element Array{Int64,1}:
```

```
6  
5
```

```
4  
3  
-1
```

```
julia> splice!(A, 1, [-1, -2, -3])
```

```
6
```

```
julia> A
```

```
7-element Array{Int64,1}:
```

```
-1  
-2  
-3  
5  
4  
3  
-1
```

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

#### source

```
| splice!(a::Vector, range, [replacement]) -> items
```

Remove items in the specified index range, and return a collection containing the removed items. Subsequent items are shifted left to fill the resulting gap. If specified, replacement values from an ordered collection will be spliced in place of the removed items.

To insert `replacement` before an index `n` without removing any items, use `splice!(collection, n:n-1, replacement)`.

#### Examples

```
| julia> splice!(A, 4:3, 2)
```

```
julia> A
8-element Array{Int64,1}:
-1
-2
-3
2
5
4
3
-1
```

[source](#)

[Base.resize!](#) – Function.

```
| resize!(a::Vector, n::Integer) -> Vector
```

Resize `a` to contain `n` elements. If `n` is smaller than the current collection length, the first `n` elements will be retained. If `n` is larger, the new elements are not guaranteed to be initialized.

Examples

```
julia> resize!([6, 5, 4, 3, 2, 1], 3)
3-element Array{Int64,1}:
6
5
4

julia> a = resize!([6, 5, 4, 3, 2, 1], 8);

julia> length(a)
```

```
julia> a[1:6]
6-element Array{Int64,1}:
 6
 5
 4
 3
 2
 1
```

### source

`Base.append!` – Function.

```
| append!(collection, collection2) -> collection.
```

Add the elements of `collection2` to the end of `collection`.

### Examples

```
julia> append!([1],[2,3])
3-element Array{Int64,1}:
 1
 2
 3
```

```
julia> append!([1, 2, 3], [4, 5, 6])
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

## 47.8 Use [Utility Collections](#)

Push individual items to `collection` which are not already in themselves in another collection. The result is of the preceding example is equivalent to `push!([1, 2, 3], 4, 5, 6)`.

[source](#)

[Base.prepend!](#) – Function.

```
prepend!(a::Vector, items) -> collection
```

Insert the elements of `items` to the beginning of `a`.

Examples

```
julia> prepend!([3],[1,2])
```

```
3-element Array{Int64,1}:
```

```
1
```

```
2
```

```
3
```

[source](#)

Fully implemented by:

`Vector` (a.k.a. 1-dimensional [Array](#))

`BitVector` (a.k.a. 1-dimensional [BitArray](#))

## 47.8 Utility Collections

[Base.Pair](#) – Type.

```
Pair(x, y)  
x => y
```

Construct a `Pair` object with type `Pair{typeof(x), typeof(y)}`. The elements are stored in the fields `first` and `second`. They can also be accessed via iteration.

946 See also: [Dict](#) CHAPTER 47. COLLECTIONS AND DATA STRUCTURES

## Examples

```
julia> p = "foo" => 7  
"foo" => 7
```

```
julia> typeof(p)  
Pair{String,Int64}
```

```
julia> p.first  
"foo"
```

```
julia> for x in p  
    println(x)  
end  
foo  
7
```

source

# Chapter 48

## Mathematics

### 48.1 Mathematical Operators

Base.`:-` – Method.

```
-(x)
```

Unary minus operator.

Examples

```
julia> -1
-1

julia> -(2)
-2

julia> -[1 2; 3 4]
2×2 Array{Int64,2}:
 -1   -2
 -3   -4
```

`source`

Base.`:+` – Function.

948|+(x, y...)

CHAPTER 48. MATHEMATICS

Addition operator. `x+y+z+...` calls this function with all arguments, i.e. `+(x, y, z, ...)`.

Examples

| **julia>** 1 + 20 + 4

| 25

| **julia>** +(1, 20, 4)

| 25

| **source**

| `dt::Date + t::Time -> DateTime`

The addition of a `Date` with a `Time` produces a `DateTime`. The hour, minute, second, and millisecond parts of the `Time` are used along with the year, month, and day of the `Date` to create the new `DateTime`. Non-zero microseconds or nanoseconds in the `Time` type will result in an `InexactError` being thrown.

| **source**

`Base.:-` – Method.

| `-(x, y)`

Subtraction operator.

Examples

| **julia>** 2 - 3

| -1

| **julia>** -(2, 4.5)

| -2.5

Base.`:*` – Method.

```
| *(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

Examples

```
julia> 2 * 7 * 8
```

```
112
```

```
julia> *(2, 7, 8)
```

```
112
```

```
source
```

Base.`:`/`/` – Function.

```
| /(x, y)
```

Right division operator: multiplication of `x` by the inverse of `y` on the right. Gives floating-point results for integer arguments.

Examples

```
julia> 1/2
```

```
0.5
```

```
julia> 4/2
```

```
2.0
```

```
julia> 4.5/2
```

```
2.25
```

```
source
```

`Base.\` – Method.

CHAPTER 48. MATHEMATICS

`\(x, y)`

Left division operator: multiplication of  $y$  by the inverse of  $x$  on the left.  
Gives floating-point results for integer arguments.

Examples

`julia> 3 \ 6`

`2.0`

`julia> inv(3) * 6`

`2.0`

`julia> A = [1 2; 3 4]; x = [5, 6];`

`julia> A \ x`

`2-element Array{Float64,1}:`

`-4.0`

`4.5`

`julia> inv(A) * x`

`2-element Array{Float64,1}:`

`-4.0`

`4.5`

`source`

`Base.^` – Method.

`^(x, y)`

Exponentiation operator. If  $x$  is a matrix, computes matrix exponentiation.

If  $y$  is an `Int` literal (e.g. `2` in  $x^2$  or `-3` in  $x^{-3}$ ), the Julia code  $x^y$  is transformed by the compiler to `Base.literal_pow(^, x, Val(y))`.

48.1 to [MATHEMATICAL OPERATORS](#) on the value of the exponent. (A [§51](#) default fallback we have `Base.literal_pow(^, x, Val(y)) = ^(x,y)`, where usually `^ == Base.^` unless `^` has been defined in the calling namespace.)

```
julia> 3^5
```

```
243
```

```
julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
1 2
```

```
3 4
```

```
julia> A^3
```

```
2×2 Array{Int64,2}:
```

```
37 54
```

```
81 118
```

[source](#)

[Base.fma](#) – Function.

```
| fma(x, y, z)
```

Computes  $x*y+z$  without rounding the intermediate result  $x*y$ . On some systems this is significantly more expensive than  $x*y+z$ . `fma` is used to improve accuracy in certain algorithms. See [muladd](#).

[source](#)

[Base.muladd](#) – Function.

```
| muladd(x, y, z)
```

Combined multiply-add, computes  $x*y+z$  allowing the add and multiply to be contracted with each other or ones from other `muladd` and `@fastmath`

952 to form `fma` if the transformation can improve performance. The results may be different on different machines and can also be different on the same machine due to constant propagation or other optimizations. See [fma](#).

### Examples

```
julia> muladd(3, 2, 1)
```

```
7
```

```
julia> 3 * 2 + 1
```

```
7
```

`source`

`Base.inv` – Method.

```
| inv(x)
```

Return the multiplicative inverse of `x`, such that `x*inv(x)` or `inv(x)*x` yields `one(x)` (the multiplicative identity) up to roundoff errors.

If `x` is a number, this is essentially the same as `one(x)/x`, but for some types `inv(x)` may be slightly more efficient.

### Examples

```
julia> inv(2)
```

```
0.5
```

```
julia> inv(1 + 2im)
```

```
0.2 - 0.4im
```

```
julia> inv(1 + 2im) * (1 + 2im)
```

```
1.0 + 0.0im
```

```
julia> inv(2//3)
```

```
3//2
```

**Base.div** – Function.

```
| div(x, y)  
| ÷(x, y)
```

The quotient from Euclidean division. Computes  $x/y$ , truncated to an integer.

Examples

```
| julia> 9 ÷ 4  
| 2
```

```
| julia> -5 ÷ 3  
| -1
```

**source**

**Base.fld** – Function.

```
| fld(x, y)
```

Largest integer less than or equal to  $x/y$ .

Examples

```
| julia> fld(7.3, 5.5)  
| 1.0
```

**source**

**Base.cld** – Function.

```
| cld(x, y)
```

Smallest integer larger than or equal to  $x/y$ .

Examples

```
954 | julia> cld(5.5,2.2)
      | 3.0
```

## CHAPTER 48. MATHEMATICS

[source](#)

[Base.mod](#) – Function.

```
| mod(x, y)
| rem(x, y, RoundDown)
```

The reduction of  $x$  modulo  $y$ , or equivalently, the remainder of  $x$  after floored division by  $y$ , i.e.

```
| x - y*fld(x,y)
```

if computed without intermediate rounding.

The result will have the same sign as  $y$ , and magnitude less than  $\text{abs}(y)$  (with some exceptions, see note below).

### Note

When used with floating point values, the exact result may not be representable by the type, and so rounding error may occur. In particular, if the exact result is very close to  $y$ , then it may be rounded to  $y$ .

```
julia> mod(8, 3)
```

2

```
julia> mod(9, 3)
```

0

```
julia> mod(8.9, 3)
```

2.900000000000004

```
julia> mod(eps(), 5)
```

```
2.220446049250313e-16
```

```
julia> mod(-eps(), 3)
```

```
3.0
```

source

```
rem(x::Integer, T::Type{<:Integer}) -> T  
mod(x::Integer, T::Type{<:Integer}) -> T  
%(x::Integer, T::Type{<:Integer}) -> T
```

Find  $y::T$  such that  $x \equiv y \pmod{n}$ , where  $n$  is the number of integers representable in  $T$ , and  $y$  is an integer in  $[typemin(T), typemax(T)]$ . If  $T$  can represent any integer (e.g.  $T == BigInt$ ), then this operation corresponds to a conversion to  $T$ .

```
julia> 129 % Int8
```

```
-127
```

source

Base.rem – Function.

```
rem(x, y)
```

```
%(x, y)
```

Remainder from Euclidean division, returning a value of the same sign as  $x$ , and smaller in magnitude than  $y$ . This value is always exact.

Examples

```
julia> x = 15; y = 4;
```

```
julia> x % y
```

```
3
```

```
julia> x == div(x, y) * y + rem(x, y)
true
```

[source](#)

[Base.Math.rem2pi](#) – Function.

```
rem2pi(x, r::RoundingMode)
```

Compute the remainder of  $x$  after integer division by  $2\pi$ , with the quotient rounded according to the rounding mode  $r$ . In other words, the quantity

```
x - pi2*round(xpi/(2), r)
```

without any intermediate rounding. This internally uses a high precision approximation of  $2\pi$ , and so will give a more accurate result than  $\text{rem}(x, 2\pi, r)$

if  $r == \text{RoundNearest}$ , then the result is in the interval  $[-,]$ . This will generally be the most accurate result.

if  $r == \text{RoundToZero}$ , then the result is in the interval  $[0, 2]$  if  $x$  is positive,. or  $[-2, 0]$  otherwise.

if  $r == \text{RoundDown}$ , then the result is in the interval  $[0, 2]$ .

if  $r == \text{RoundUp}$ , then the result is in the interval  $[-2, 0]$ .

Examples

```
julia> rem2pi(7pi/4, RoundNearest)
-0.7853981633974485
```

```
julia> rem2pi(7pi/4, RoundDown)
5.497787143782138
```

[source](#)

```
| mod2pi(x)
```

Modulus after division by  $2\pi$ , returning in the range  $[0, 2]$ .

This function computes a floating point representation of the modulus after division by numerically exact  $2\pi$ , and is therefore not exactly the same as `mod(x, 2π)`, which would compute the modulus of  $x$  relative to division by the floating-point number  $2\pi$ .

Examples

```
| julia> mod2pi(9*pi/4)
```

```
0.7853981633974481
```

```
source
```

Base.`divrem` – Function.

```
| divrem(x, y)
```

The quotient and remainder from Euclidean division. Equivalent to `(div(x, y), rem(x, y))` or `(x÷y, x%y)`.

```
| julia> divrem(3, 7)
```

```
(0, 3)
```

```
| julia> divrem(7, 3)
```

```
(2, 1)
```

```
source
```

Base.`fldmod` – Function.

```
| fldmod(x, y)
```

The floored quotient and modulus after division. Equivalent to `(fld(x, y), mod(x, y))`.

[Base.fld1](#) – Function.

```
| fld1(x, y)
```

Flooring division, returning a value consistent with `mod1(x, y)`

See also: [mod1](#).

Examples

```
julia> x = 15; y = 4;
```

```
julia> fld1(x, y)
```

```
4
```

```
julia> x == fld(x, y) * y + mod(x, y)
```

```
true
```

```
julia> x == (fld1(x, y) - 1) * y + mod1(x, y)
```

```
true
```

[source](#)

[Base.mod1](#) – Function.

```
| mod1(x, y)
```

Modulus after flooring division, returning a value `r` such that `mod(r, y) == mod(x, y)` in the range  $(0, y]$  for positive `y` and in the range  $[y, 0)$  for negative `y`.

Examples

```
julia> mod1(4, 2)
```

```
2
```

```
| 1
```

**source**

**Base.fldmod1** – Function.

```
| fldmod1(x, y)
```

Return  $(\text{fld1}(x, y), \text{mod1}(x, y))$ .

See also: [fld1](#), [mod1](#).

**source**

**Base.://** – Function.

```
| //(num, den)
```

Divide two integers or rational numbers, giving a [Rational](#) result.

```
| julia> 3 // 5
```

```
| 3//5
```

```
| julia> (3 // 5) // (2 // 1)
```

```
| 3//10
```

**source**

**Base.rationalize** – Function.

```
| rationalize([T<:Integer=Int, ] x; tol::Real=eps(x))
```

Approximate floating point number  $x$  as a [Rational](#) number with components of the given integer type. The result will differ from  $x$  by no more than  $\text{tol}$ .

```
| julia> rationalize(5.6)
```

```
| 28//5
```

```
julia> a = rationalize(BigInt, 10.3)
103//10
```

```
julia> typeof(numerator(a))
BigInt
```

source

[Base.numerator](#) – Function.

```
| numerator(x)
```

Numerator of the rational representation of  $x$ .

```
julia> numerator(2//3)
2
```

```
julia> numerator(4)
4
```

source

[Base.denominator](#) – Function.

```
| denominator(x)
```

Denominator of the rational representation of  $x$ .

```
julia> denominator(2//3)
3
```

```
julia> denominator(4)
1
```

source

```
| <<(x, n)
```

Left bit shift operator,  $x \ll n$ . For  $n \geq 0$ , the result is  $x$  shifted left by  $n$  bits, filling with 0s. This is equivalent to  $x * 2^n$ . For  $n < 0$ , this is equivalent to  $x \gg -n$ .

Examples

```
julia> Int8(3) << 2
```

```
12
```

```
julia> bitstring(Int8(3))
```

```
"00000011"
```

```
julia> bitstring(Int8(12))
```

```
"00001100"
```

See also [>>](#), [>>>](#).

[source](#)

```
| <<(B::BitVector, n) -> BitVector
```

Left bit shift operator,  $B \ll n$ . For  $n \geq 0$ , the result is  $B$  with elements shifted  $n$  positions backwards, filling with `false` values. If  $n < 0$ , elements are shifted forwards. Equivalent to  $B \gg -n$ .

Examples

```
julia> B = BitVector([true, false, true, false, false])
```

```
5-element BitArray{1}:
```

```
 true
```

```
 false
```

```
 true
```

```
 false
```

```
julia> B << 1
5-element BitArray{1}:
false
true
false
false
false
```

```
julia> B << -1
5-element BitArray{1}:
false
true
false
true
false
```

### source

[Base. :>>](#) – Function.

```
|>>(x, n)
```

Right bit shift operator,  $x \gg n$ . For  $n \geq 0$ , the result is  $x$  shifted right by  $n$  bits, where  $n \geq 0$ , filling with 0s if  $x \geq 0$ , 1s if  $x < 0$ , preserving the sign of  $x$ . This is equivalent to  $\text{fld}(x, 2^n)$ . For  $n < 0$ , this is equivalent to  $x \ll -n$ .

### Examples

```
julia> Int8(13) >> 2
```

```
3
```

```
julia> bitstring(Int8(13))
"00001101"

julia> bitstring(Int8(3))
"00000011"

julia> Int8(-14) >> 2
-4

julia> bitstring(Int8(-14))
"11110010"

julia> bitstring(Int8(-4))
"11111100"
```

See also `>>>`, `<<`.

### source

```
|>>(B::BitVector, n) -> BitVector
```

Right bit shift operator, `B >> n`. For `n >= 0`, the result is `B` with elements shifted `n` positions forward, filling with `false` values. If `n < 0`, elements are shifted backwards. Equivalent to `B << -n`.

### Examples

```
julia> B = BitVector([true, false, true, false, false])
5-element BitArray{1}:
  true
  false
  true
  false
  false
```

```
julia> B >> 1
5-element BitArray{1}:
 false
 true
 false
 true
 false
```

```
julia> B >> -1
5-element BitArray{1}:
 false
 true
 false
 false
 false
```

### source

`Base.:``>>>` – Function.

```
|>>>(x, n)
```

Unsigned right bit shift operator, `x >>> n`. For `n >= 0`, the result is `x` shifted right by `n` bits, where `n >= 0`, filling with 0s. For `n < 0`, this is equivalent to `x << -n`.

For `Unsigned` integer types, this is equivalent to `>>`. For `Signed` integer types, this is equivalent to `signed(unsigned(x) >> n)`.

Examples

```
julia> Int8(-14) >>> 2
60

julia> bitstring(Int8(-14))
```

```
julia> bitstring(Int8(60))  
"00111100"
```

`BigInt`s are treated as if having infinite size, so no filling is required and this is equivalent to `>>`.

See also `>>`, `<<`.

**source**

```
|>>>(B::BitVector, n) -> BitVector
```

Unsigned right bitshift operator, `B >>> n`. Equivalent to `B >> n`. See `>>` for details and examples.

**source**

`Base.colon` – Function.

```
|colon(start, [step], stop)
```

Called by `:` syntax for constructing ranges.

```
julia> colon(1, 2, 5)  
1:2:5
```

**source**

```
|:(start, [step], stop)
```

Range operator. `a:b` constructs a range from `a` to `b` with a step size of 1, and `a:s:b` is similar but uses a step size of `s`. These syntaxes call the function `colon`. The colon is also used in indexing to select whole dimensions.

**source**

`Base.range` – Function.

CHAPTER 48. MATHEMATICS

| `range(start, [step], length)`

Construct a range by length, given a starting value and optional step (defaults to 1).

`source`

`Base.OneTo` – Type.

| `Base.OneTo(n)`

Define an `AbstractUnitRange` that behaves like `1:n`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

`source`

`Base.StepRangeLen` – Type.

| `StepRangeLen{T,R,S}(ref::R, step::S, len, [offset=1]) where {T,  
R,S}`

| `StepRangeLen( ref::R, step::S, len, [offset=1]) where {  
R,S}`

A range `r` where `r[i]` produces values of type `T` (in the second form, `T` is deduced automatically), parameterized by a `reference` value, a `step`, and the `length`. By default `ref` is the starting value `r[1]`, but alternatively you can supply it as the value of `r[offset]` for some other index `1 <= offset <= len`. In conjunction with `TwicePrecision` this can be used to implement ranges that are free of roundoff error.

`source`

`Base.==` – Function.

| `==(x, y)`

## 48.1. MATHEMATICAL OPERATORS

Returning a single `Bool` result. Falls back to `==`.

Should be implemented for all types with a notion of equality, based on the abstract value that an instance represents. For example, all numeric types are compared by numeric value, ignoring type. Strings are compared as sequences of characters, ignoring encoding.

Follows IEEE semantics for floating-point numbers.

Collections should generally implement `==` by calling `==` recursively on all contents.

New numeric types should implement this function for two arguments of the new type, and handle comparison to other types via promotion rules where possible.

`source`

```
| ==(a::AbstractString, b::AbstractString)
```

Test whether two strings are equal character by character.

Examples

```
julia> "abc" == "abc"
```

```
true
```

```
julia> "abc" == "αβγ"
```

```
false
```

`source`

`Base.!=` – Function.

```
!=(x, y)≠
```

```
(x, y)
```

968 Not-equals comparison operator. Always gives the opposite result of `==`. New types should generally not implement this, and rely on the fallback definition `!=(x,y) = !(x==y)` instead.

Examples

```
julia> 3 != 2
true

julia> "foo" ≠ "foo"
false
```

source

`Base.!=` – Function.

```
| !==(x, y)
| (x, y)
```

Equivalent to `!(x === y)`.

Examples

```
julia> a = [1, 2]; b = [1, 2];

julia> a != b
true

julia> a == a
false
```

source

`Base.<` – Function.

```
| <(x, y)
```

This function for two arguments of the new type. Because of the behavior of floating-point NaN values, `<` implements a partial order. Types with a canonical partial order should implement `<`, and types with a canonical total order should implement `isless`.

Examples

```
julia> 'a' < 'b'
```

```
true
```

```
julia> "abc" < "abd"
```

```
true
```

```
julia> 5 < 3
```

```
false
```

`source`

`Base.:=` – Function.

```
|<=(x, y)≤  
|(x,y)
```

Less-than-or-equals comparison operator.

Examples

```
julia> 'a' <= 'b'
```

```
true
```

```
julia> 7 ≤ 7 ≤ 9
```

```
true
```

```
julia> "abc" ≤ "abc"
```

970 | true

CHAPTER 48. MATHEMATICS

```
| julia> 5 <= 3  
| false
```

source

Base.`:>` – Function.

```
| >(x, y)
```

Greater-than comparison operator. Generally, new types should implement `<` instead of this function, and rely on the fallback definition  $\text{>}(x, y) = y < x$ .

Examples

```
| julia> 'a' > 'b'
```

false

```
| julia> 7 > 3 > 1
```

true

```
| julia> "abc" > "abd"
```

false

```
| julia> 5 > 3
```

true

source

Base.`:>=` – Function.

```
| >=(x, y)≥  
| (x, y)
```

Examples

```
julia> 'a' >= 'b'
```

```
false
```

```
julia> 7 ≥ 7 ≥ 3
```

```
true
```

```
julia> "abc" ≥ "abc"
```

```
true
```

```
julia> 5 >= 3
```

```
true
```

`source`

`Base.cmp` – Function.

```
| cmp(x,y)
```

Return -1, 0, or 1 depending on whether  $x$  is less than, equal to, or greater than  $y$ , respectively. Uses the total order implemented by `isless`. For floating-point numbers, uses `<` but throws an error for unordered arguments.

Examples

```
julia> cmp(1, 2)
```

```
-1
```

```
julia> cmp(2, 1)
```

```
1
```

972

```
julia> cmp(2+im, 3-im)
ERROR: MethodError: no method matching
         isless(::Complex{Int64}, ::Complex{Int64})
Stacktrace:
[...]
```

CHAPTER 48. MATHEMATICS

### source

```
| cmp(a::AbstractString, b::AbstractString)
```

Compare two strings for equality.

Return 0 if both strings have the same length and the character at each index is the same in both strings. Return -1 if a is a substring of b, or if a comes before b in alphabetical order. Return 1 if b is a substring of a, or if b comes before a in alphabetical order.

### Examples

```
julia> cmp("abc", "abc")
```

```
0
```

```
julia> cmp("ab", "abc")
```

```
-1
```

```
julia> cmp("abc", "ab")
```

```
1
```

```
julia> cmp("ab", "ac")
```

```
-1
```

```
julia> cmp("ac", "ab")
```

```
1
```

```
julia> cmp("a", "a")  
1  
  
julia> cmp("b", "β")  
-1
```

source

Base.:~ – Function.

```
| ~(x)
```

Bitwise not.

Examples

```
julia> ~4  
-5  
  
julia> ~10  
-11  
  
julia> ~true  
false
```

source

Base.:& – Function.

```
| &(x, y)
```

Bitwise and.

Examples

```
julia> 4 & 10  
0
```

```
julia> 4 & 12
```

```
4
```

```
source
```

Base.| – Function.

```
| |(x, y)
```

Bitwise or.

Examples

```
julia> 4 | 10
```

```
14
```

```
julia> 4 | 1
```

```
5
```

```
source
```

Base.xor – Function.

```
xor(x, y)
```

```
(x, y)
```

Bitwise exclusive or of  $x$  and  $y$ . The infix operation  $a \oplus b$  is a synonym for `xor(a, b)`, and can be typed by tab-completing `\xor` or `\veebar` in the Julia REPL.

Examples

```
julia> [true; true; false] . [true; false; false]
```

```
3-element BitArray{1}:
```

```
  false
```

```
   true
```

```
  false
```

`Base.!:!` – Function.

```
| !(x)
```

Boolean not.

Examples

```
julia> !(true)
```

```
false
```

```
julia> !(false)
```

```
true
```

```
julia> .![true false true]
```

```
1×3 BitArray{2}:
```

```
 false  true  false
```

```
source
```

```
| !f::Function
```

Predicate function negation: when the argument of `!` is a function, it returns a function which computes the boolean negation of `f`.

Examples

```
julia> str = " ε > 0, δ > 0: |x-y| < δ |f(x)-f(y)| < ε"
" ε > 0, δ > 0: |x-y| < δ |f(x)-f(y)| < ε"
```

```
julia> filter(isalpha, str)
```

```
"εδxyδfxfyε"
```

```
julia> filter(!isalpha, str)
```

```
"  > 0,   > 0: |-| < |()-()| < "
```

`&&` – Keyword.

```
| x && y
```

Short-circuiting boolean AND.

[source](#)

`||` – Keyword.

```
| x || y
```

Short-circuiting boolean OR.

[source](#)

## 48.2 Mathematical Functions

`Base.isapprox` – Function.

```
| isapprox(x, y; rtol::Real=atol>0 ? √eps : 0, atol::Real=0, nans
      ::Bool=false, norm::Function)
```

Inexact equality comparison: `true` if `norm(x-y) <= max(atol, rtol*max(norm(x), norm(y)))`. The default `atol` is zero and the default `rtol` depends on the types of `x` and `y`. The keyword argument `nans` determines whether or not NaN values are considered equal (defaults to false).

For real or complex floating-point values, if an `atol > 0` is not specified, `rtol` defaults to the square root of `eps` of the type of `x` or `y`, whichever is bigger (least precise). This corresponds to requiring equality of about half of the significand digits. Otherwise, e.g. for integer arguments or if an `atol > 0` is supplied, `rtol` defaults to zero.

`x` and `y` may also be arrays of numbers, in which case `norm` defaults to `vecnorm` but may be changed by passing a `norm::Function` keyword

48.2. **MATHEMATICAL FUNCTIONS** (is the same thing as `abs()`.) When `x` and `y` are arrays, if `norm(x-y)` is not finite (i.e. `±Inf` or `NaN`), the comparison falls back to checking whether all elements of `x` and `y` are approximately equal component-wise.

The binary operator `≈` is equivalent to `isapprox` with the default arguments, and `x ≈ y` is equivalent to `!isapprox(x,y)`.

Examples

```
julia> 0.1 ≈ (0.1 - 1e-10)
true

julia> isapprox(10, 11; atol = 2)
true

julia> isapprox([10.0^9, 1.0], [10.0^9, 2.0])
true
```

source

`Base.sin` – Method.

```
| sin(x)
```

Compute sine of `x`, where `x` is in radians.

source

`Base.cos` – Method.

```
| cos(x)
```

Compute cosine of `x`, where `x` is in radians.

source

`Base.Math.sincos` – Method.

978|**sincos(x)**

CHAPTER 48. MATHEMATICS

Simultaneously compute the sine and cosine of  $x$ , where the  $x$  is in radians.

**source**

**Base.tan** – Method.

|**tan(x)**

Compute tangent of  $x$ , where  $x$  is in radians.

**source**

**Base.Math.sind** – Function.

|**sind(x)**

Compute sine of  $x$ , where  $x$  is in degrees.

**source**

**Base.Math.cosd** – Function.

|**cosd(x)**

Compute cosine of  $x$ , where  $x$  is in degrees.

**source**

**Base.Math.tand** – Function.

|**tand(x)**

Compute tangent of  $x$ , where  $x$  is in degrees.

**source**

**Base.Math.sinpi** – Function.

|**sinpi(x)**

## 48.2. MATHEMATICAL FUNCTIONS

[source](#)

**Base.Math.cospi** – Function.

| `cospi(x)`

Compute  $\cos(\pi x)$  more accurately than  $\cos(\text{pi} \times x)$ , especially for large  $x$ .

[source](#)

**Base.sinh** – Method.

| `sinh(x)`

Compute hyperbolic sine of  $x$ .

[source](#)

**Base.cosh** – Method.

| `cosh(x)`

Compute hyperbolic cosine of  $x$ .

[source](#)

**Base.tanh** – Method.

| `tanh(x)`

Compute hyperbolic tangent of  $x$ .

[source](#)

**Base.asin** – Method.

| `asin(x)`

Compute the inverse sine of  $x$ , where the output is in radians.

[source](#)

**Base.acos** – Method.

CHAPTER 48. MATHEMATICS

| `acos(x)`

Compute the inverse cosine of  $x$ , where the output is in radians

[source](#)

**Base.atan** – Method.

| `atan(x)`

Compute the inverse tangent of  $x$ , where the output is in radians.

[source](#)

**Base.Math.atan2** – Function.

| `atan2(y, x)`

Compute the inverse tangent of  $y/x$ , using the signs of both  $x$  and  $y$  to determine the quadrant of the return value.

[source](#)

**Base.Math.asind** – Function.

| `asind(x)`

Compute the inverse sine of  $x$ , where the output is in degrees.

[source](#)

**Base.Math.acosd** – Function.

| `acosd(x)`

Compute the inverse cosine of  $x$ , where the output is in degrees.

[source](#)

**Base.Math.atand** – Function.

Compute the inverse tangent of  $x$ , where the output is in degrees.

[source](#)

`Base.Math.sec` – Method.

`| sec(x)`

Compute the secant of  $x$ , where  $x$  is in radians.

[source](#)

`Base.Math.csc` – Method.

`| csc(x)`

Compute the cosecant of  $x$ , where  $x$  is in radians.

[source](#)

`Base.Math.cot` – Method.

`| cot(x)`

Compute the cotangent of  $x$ , where  $x$  is in radians.

[source](#)

`Base.Math.secd` – Function.

`| secd(x)`

Compute the secant of  $x$ , where  $x$  is in degrees.

[source](#)

`Base.Math.cscd` – Function.

`| cscd(x)`

982 Compute the cosecant of  $x$ , where  $x$  is in degrees.

[source](#)

**Base.Math.cotd** – Function.

| cotd( $x$ )

Compute the cotangent of  $x$ , where  $x$  is in degrees.

[source](#)

**Base.Math.asec** – Method.

| asec( $x$ )

Compute the inverse secant of  $x$ , where the output is in radians.

[source](#)

**Base.Math.acsc** – Method.

| acsc( $x$ )

Compute the inverse cosecant of  $x$ , where the output is in radians.

[source](#)

**Base.Math.acot** – Method.

| acot( $x$ )

Compute the inverse cotangent of  $x$ , where the output is in radians.

[source](#)

**Base.Math.asecd** – Function.

| asecd( $x$ )

Compute the inverse secant of  $x$ , where the output is in degrees.

[source](#)

```
| acscd(x)
```

Compute the inverse cosecant of  $x$ , where the output is in degrees.

[source](#)

**Base.Math.acotd** – Function.

```
| acotd(x)
```

Compute the inverse cotangent of  $x$ , where the output is in degrees.

[source](#)

**Base.Math.sech** – Method.

```
| sech(x)
```

Compute the hyperbolic secant of  $x$ .

[source](#)

**Base.Math.csch** – Method.

```
| csch(x)
```

Compute the hyperbolic cosecant of  $x$ .

[source](#)

**Base.Math.coth** – Method.

```
| coth(x)
```

Compute the hyperbolic cotangent of  $x$ .

[source](#)

**Base.asinh** – Method.

```
| asinh(x)
```

984 Compute the inverse hyperbolic sine of x. CHAPTER 48. MATHEMATICS

[source](#)

[Base.acosh](#) – Method.

|  $\text{acosh}(x)$

Compute the inverse hyperbolic cosine of x.

[source](#)

[Base.atanh](#) – Method.

|  $\text{atanh}(x)$

Compute the inverse hyperbolic tangent of x.

[source](#)

[Base.Math.asech](#) – Method.

|  $\text{asech}(x)$

Compute the inverse hyperbolic secant of x.

[source](#)

[Base.Math.acsch](#) – Method.

|  $\text{acsch}(x)$

Compute the inverse hyperbolic cosecant of x.

[source](#)

[Base.Math.acoth](#) – Method.

|  $\text{acoth}(x)$

Compute the inverse hyperbolic cotangent of x.

[source](#)

```
| sinc(x)
```

Compute  $\sin(\pi x)/(\pi x)$  if  $x \neq 0$ , and 1 if  $x = 0$ .

[source](#)

`Base.Math.cosc` – Function.

```
| cosc(x)
```

Compute  $\cos(\pi x)/x - \sin(\pi x)/(\pi x^2)$  if  $x \neq 0$ , and 0 if  $x = 0$ . This is the derivative of `sinc(x)`.

[source](#)

`Base.Math.deg2rad` – Function.

```
| deg2rad(x)
```

Convert  $x$  from degrees to radians.

```
| julia> deg2rad(90)
| 1.5707963267948966
```

[source](#)

`Base.Math.rad2deg` – Function.

```
| rad2deg(x)
```

Convert  $x$  from radians to degrees.

```
| julia> rad2deg(pi)
| 180.0
```

[source](#)

`Base.Math.hypot` – Function.

986 | `hypot(x, y)`

CHAPTER 48. MATHEMATICS

Compute the hypotenuse  $\sqrt{x^2 + y^2}$  avoiding overflow and underflow.

Examples

```
| julia> a = 10^10;
```

```
| julia> hypot(a, a)
```

```
1.4142135623730951e10
```

```
| julia> √(a^2 + a^2) # a^2 overflows
```

```
ERROR: DomainError with -2.914184810805068e18:
```

```
sqrt will only return a complex result if called with a  
→ complex argument. Try sqrt(Complex(x)).
```

```
Stacktrace:
```

```
[...]
```

`source`

```
| hypot(x...)
```

Compute the hypotenuse  $\sqrt{\sum x_i^2}$  avoiding overflow and underflow.

`source`

`Base.log` – Method.

```
| log(x)
```

Compute the natural logarithm of `x`. Throws `DomainError` for negative `Real` arguments. Use complex negative arguments to obtain complex results.

There is an experimental variant in the `Base.Math.JuliaLibm` module, which is typically faster and more accurate.

`source`

```
| log(b, x)
```

Compute the base **b** logarithm of **x**. Throws [DomainError](#) for negative [Real](#) arguments.

```
julia> log(4,8)
```

```
1.5
```

```
julia> log(4,2)
```

```
0.5
```

Note

If **b** is a power of 2 or 10, [log2](#) or [log10](#) should be used, as these will typically be faster and more accurate. For example,

```
julia> log(100,1000000)
```

```
2.999999999999996
```

```
julia> log10(1000000)/2
```

```
3.0
```

[source](#)

[Base.log2](#) – Function.

```
| log2(x)
```

Compute the logarithm of **x** to base 2. Throws [DomainError](#) for negative [Real](#) arguments.

Examples

```
julia> log2(4)
```

```
2.0
```

```
988 | julia> log2(10)  
| 3.321928094887362
```

## CHAPTER 48. MATHEMATICS

`source`

`Base.log10` – Function.

```
| log10(x)
```

Compute the logarithm of  $x$  to base 10. Throws `DomainError` for negative `Real` arguments.

Examples

```
julia> log10(100)  
2.0
```

```
julia> log10(2)  
0.3010299956639812
```

`source`

`Base.log1p` – Function.

```
| log1p(x)
```

Accurate natural logarithm of  $1+x$ . Throws `DomainError` for `Real` arguments less than -1.

There is an experimental variant in the `Base.Math.JuliaLibm` module, which is typically faster and more accurate.

Examples

```
julia> log1p(-0.5)  
-0.6931471805599453
```

```
julia> log1p(0)  
0.0
```

**Base.Math.frexp** – Function.

```
| frexp(val)
```

Return  $(x, \exp)$  such that  $x$  has a magnitude in the interval  $[1/2, 1)$  or 0, and **val** is equal to  $x \times 2^{\exp}$ .

**source**

**Base.exp** – Method.

```
| exp(x)
```

Compute the natural base exponential of  $x$ , in other words  $e^x$ .

```
julia> exp(1.0)
2.718281828459045
```

**source**

**Base.exp2** – Function.

```
| exp2(x)
```

Compute the base 2 exponential of  $x$ , in other words  $2^x$ .

Examples

```
julia> exp2(5)
32.0
```

**source**

**Base.exp10** – Function.

```
| exp10(x)
```

Compute the base 10 exponential of  $x$ , in other words  $10^x$ .

Examples

```
990 | julia> exp10(2)  
| 100.0
```

## CHAPTER 48. MATHEMATICS

`source`

```
| exp10(x)
```

Compute  $10^x$ .

Examples

```
| julia> exp10(2)
```

```
| 100.0
```

```
| julia> exp10(0.2)
```

```
| 1.5848931924611136
```

`source`

[Base.Math.ldexp](#) – Function.

```
| ldexp(x, n)
```

Compute  $x \times 2^n$ .

Examples

```
| julia> ldexp(5., 2)
```

```
| 20.0
```

`source`

[Base.Math.modf](#) – Function.

```
| modf(x)
```

Return a tuple (fpart,ipart) of the fractional and integral parts of a number.

Both parts have the same sign as the argument.

Examples

```
julia> modf(3.5)
```

```
(0.5, 3.0)
```

**source**

[Base.expm1](#) – Function.

```
| expm1(x)
```

Accurately compute  $e^x - 1$ .

**source**

[Base.round](#) – Method.

```
| round([T,] x, [digits, [base]], [r::RoundingMode])
```

Rounds  $x$  to an integer value according to the provided [RoundingMode](#), returning a value of the same type as  $x$ . When not specifying a rounding mode the global mode will be used (see [rounding](#)), which by default is round to the nearest integer ([RoundNearest](#) mode), with ties (fractional values of 0.5) being rounded to the nearest even integer.

Examples

```
julia> round(1.7)
```

```
2.0
```

```
julia> round(1.5)
```

```
2.0
```

```
julia> round(2.5)
```

```
2.0
```

The optional [RoundingMode](#) argument will change how the number gets rounded.

992 `round(T, x, [r::RoundingMode])` converts the result to type `T`, throw-  
ing an `InexactError` if the value is not representable.

`round(x, digits)` rounds to the specified number of digits after the decimal place (or before if negative). `round(x, digits, base)` rounds using a base other than 10.

Examples

```
julia> round(pi, 2)
```

```
3.14
```

```
julia> round(pi, 3, 2)
```

```
3.125
```

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the `Float64` value represented by 1.15 is actually less than 1.15, yet will be rounded to 1.2.

# Chapter 49

## Examples

```
julia> x = 1.15
1.15

julia> @sprintf "%.*f" x
"1.1499999999999991118"

julia> x < 115//100
true

julia> round(x, 1)
1.2
```

See also [signif](#) for rounding to significant digits.

[source](#)

[Base.Rounding.RoundingMode](#) – Type.

| RoundingMode

A type used for controlling the rounding mode of floating point operations (via [rounding/setrounding](#) functions), or as optional arguments for rounding to the nearest integer (via the [round](#) function).

[RoundNearest](#) (default)  
[RoundNearestTiesAway](#)  
[RoundNearestTiesUp](#)  
[RoundToZero](#)  
[RoundFromZero](#) ([BigFloat](#) only)  
[RoundUp](#)  
[RoundDown](#)

[source](#)

[Base.Rounding.RoundNearest](#) – Constant.

| [RoundNearest](#)

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

[source](#)

[Base.Rounding.RoundNearestTiesAway](#) – Constant.

| [RoundNearestTiesAway](#)

Rounds to nearest integer, with ties rounded away from zero (C/C++ [round](#) behaviour).

[source](#)

[Base.Rounding.RoundNearestTiesUp](#) – Constant.

| [RoundNearestTiesUp](#)

Rounds to nearest integer, with ties rounded toward positive infinity (Java/- JavaScript [round](#) behaviour).

[source](#)

[Base.Rounding.RoundToZero](#) – Constant.

995

| RoundToZero

[round](#) using this rounding mode is an alias for [trunc](#).

[source](#)

[Base.Rounding.RoundUp](#) – Constant.

| RoundUp

[round](#) using this rounding mode is an alias for [ceil](#).

[source](#)

[Base.Rounding.RoundDown](#) – Constant.

| RoundDown

[round](#) using this rounding mode is an alias for [floor](#).

[source](#)

[Base.round](#) – Method.

| `round(z, RoundingModeReal, RoundingModeImaginary)`

Returns the nearest integral value of the same type as the complex-valued `z` to `z`, breaking ties using the specified [RoundingModes](#). The first [RoundingMode](#) is used for rounding the real components while the second is used for rounding the imaginary components.

Example

| `julia> round(3.14 + 4.5im)`

| `3.0 + 4.0im`

[source](#)

[Base.ceil](#) – Function.

```
996|ceil([T,] x, [digits, [base]])
```

CHAPTER 49. EXAMPLES

`ceil(x)` returns the nearest integral value of the same type as `x` that is greater than or equal to `x`.

`ceil(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

`source`

`Base.floor` – Function.

```
|floor([T,] x, [digits, [base]])
```

`floor(x)` returns the nearest integral value of the same type as `x` that is less than or equal to `x`.

`floor(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

`source`

`Base.trunc` – Function.

```
|trunc([T,] x, [digits, [base]])
```

`trunc(x)` returns the nearest integral value of the same type as `x` whose absolute value is less than or equal to `x`.

`trunc(T, x)` converts the result to type `T`, throwing an `InexactError` if the value is not representable.

`digits` and `base` work as for `round`.

`source`

`Base.unsafe_trunc` – Function.

```
| unsafe_trunc(T, x)
```

997

`unsafe_trunc(T, x)` returns the nearest integral value of type `T` whose absolute value is less than or equal to `x`. If the value is not representable by `T`, an arbitrary value will be returned.

`source`

`Base.signif` – Function.

```
| signif(x, digits, [base])
```

Rounds (in the sense of `round`) `x` so that there are `digits` significant digits, under a base `base` representation, default 10.

Examples

```
julia> signif(123.456, 2)
```

```
120.0
```

```
julia> signif(357.913, 4, 2)
```

```
352.0
```

`source`

`Base.min` – Function.

```
| min(x, y, ...)
```

Return the minimum of the arguments. See also the `minimum` function to take the minimum element from a collection.

Examples

```
julia> min(2, 5, 1)
```

```
1
```

`source`

`Base.max` – Function.

CHAPTER 49. EXAMPLES

```
| max(x, y, ...)
```

Return the maximum of the arguments. See also the `maximum` function to take the maximum element from a collection.

Examples

```
| julia> max(2, 5, 1)
| 5
```

`source`

`Base.minmax` – Function.

```
| minmax(x, y)
```

Return  $(\min(x, y), \max(x, y))$ . See also: `extrema` that returns  $(\min(x), \max(x))$ .

Examples

```
| julia> minmax('c', 'b')
| ('b', 'c')
```

`source`

`Base.Math.clamp` – Function.

```
| clamp(x, lo, hi)
```

Return  $x$  if  $lo \leq x \leq hi$ . If  $x < lo$ , return  $lo$ . If  $x > hi$ , return  $hi$ .

Arguments are promoted to a common type.

```
| julia> clamp.([pi, 1.0, big(10.)], 2., 9.)
3-element Array{BigFloat,1}:
```

```
|   → 3.1415926535897932384626433832795028841971693993751058209749445923078
```

```
| 2.0  
| 9.0
```

999

[source](#)

[Base.Math.clamp!](#) – Function.

```
| clamp!(array::AbstractArray, lo, hi)
```

Restrict values in `array` to the specified range, in-place. See also [clamp](#).

[source](#)

[Base.abs](#) – Function.

```
| abs(x)
```

The absolute value of `x`.

When `abs` is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when `abs` is applied to the minimum representable value of a signed integer. That is, when `x == typemin(typeof(x))`, `abs(x) == x < 0`, not `-x` as might be expected.

```
julia> abs(-3)
```

3

```
julia> abs(1 + im)
```

1.4142135623730951

```
julia> abs(typemin(Int64))
```

-9223372036854775808

[source](#)

[Base.Checked.checked\\_abs](#) – Function.

1000 | `Base.checked_abs(x)`

## CHAPTER 49. EXAMPLES

Calculates `abs(x)`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `abs(typemin(Int))`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

`source`

[Base.Checked.checked\\_neg](#) – Function.

| `Base.checked_neg(x)`

Calculates `-x`, checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g. `Int`) cannot represent `-typemin(Int)`, thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

`source`

[Base.Checked.checked\\_add](#) – Function.

| `Base.checked_add(x, y)`

Calculates `x+y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`source`

[Base.Checked.checked\\_sub](#) – Function.

| `Base.checked_sub(x, y)`

Calculates `x-y`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

`source`

[Base.Checked.checked\\_mul](#) – Function.

1001

| `Base.checked_mul(x, y)`

Calculates  $x * y$ , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

[source](#)

[Base.Checked.checked\\_div](#) – Function.

| `Base.checked_div(x, y)`

Calculates  $\text{div}(x, y)$ , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

[source](#)

[Base.Checked.checked\\_rem](#) – Function.

| `Base.checked_rem(x, y)`

Calculates  $x \% y$ , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

[source](#)

[Base.Checked.checked\\_fld](#) – Function.

| `Base.checked_fld(x, y)`

Calculates  $\text{fld}(x, y)$ , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

[source](#)

[Base.Checked.checked\\_mod](#) – Function.

| `Base.checked_mod(x, y)`

## 100 Calculates `mod(x,y)`, checking for overflow errors

## CHAPTER 49 EXAMPLES

The overflow protection may impose a perceptible performance penalty.

[source](#)

`Base.Checked.checked_cld` – Function.

| `Base.checked_cld(x, y)`

Calculates `cld(x,y)`, checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

[source](#)

`Base.Checked.add_with_overflow` – Function.

| `Base.add_with_overflow(x, y) -> (r, f)`

Calculates `r = x+y`, with the flag `f` indicating whether overflow has occurred.

[source](#)

`Base.Checked.sub_with_overflow` – Function.

| `Base.sub_with_overflow(x, y) -> (r, f)`

Calculates `r = x-y`, with the flag `f` indicating whether overflow has occurred.

[source](#)

`Base.Checked.mul_with_overflow` – Function.

| `Base.mul_with_overflow(x, y) -> (r, f)`

Calculates `r = x*y`, with the flag `f` indicating whether overflow has occurred.

[source](#)

`Base.abs2` – Function.

1003

```
| abs2(x)
```

Squared absolute value of  $x$ .

```
| julia> abs2(-3)
```

```
| 9
```

`source`

`Base.copysign` – Function.

```
| copysign(x, y) -> z
```

Return  $z$  which has the magnitude of  $x$  and the same sign as  $y$ .

Examples

```
| julia> copysign(1, -2)
```

```
| -1
```

```
| julia> copysign(-1, 2)
```

```
| 1
```

`source`

`Base.sign` – Function.

```
| sign(x)
```

Return zero if  $x==0$  and  $x/|x|$  otherwise (i.e.,  $\pm 1$  for real  $x$ ).

`source`

`Base.signbit` – Function.

```
| signbit(x)
```

Returns `true` if the value of the sign of  $x$  is negative, otherwise `false`.

Examples

```
1004 julia> signbit(-4)
true
```

## CHAPTER 49. EXAMPLES

```
julia> signbit(5)
```

```
false
```

```
julia> signbit(5.5)
```

```
false
```

```
julia> signbit(-4.1)
```

```
true
```

`source`

`Base.flipsign` – Function.

```
| flipsign(x, y)
```

Return `x` with its sign flipped if `y` is negative. For example `abs(x) = flipsign(x, x)`.

```
julia> flipsign(5, 3)
```

```
5
```

```
julia> flipsign(5, -3)
```

```
-5
```

`source`

`Base.sqrt` – Method.

```
| sqrt(x)
```

Return  $\sqrt{x}$ . Throws `DomainError` for negative `Real` arguments. Use complex negative arguments instead. The prefix operator  `$\sqrt{}$`  is equivalent to `sqrt`.

source

1005

[Base.sqrt](#) – Function.

```
| isqrt(n::Integer)
```

Integer square root: the largest integer  $m$  such that  $m*m \leq n$ .

```
| julia> isqrt(5)
```

```
| 2
```

source

[Base.Math.cbrt](#) – Function.

```
| cbrt(x::Real)
```

Return the cube root of  $x$ , i.e.  $x^{1/3}$ . Negative values are accepted (returning the negative real root when  $x < 0$ ).

The prefix operator `cbrt` is equivalent to `cbrt`.

```
| julia> cbrt(big(27))
```

```
| 3.0
```

source

[Base.real](#) – Method.

```
| real(z)
```

Return the real part of the complex number  $z$ .

Examples

```
| julia> real(1 + 3im)
```

```
| 1
```

source

`Base.imag` – Function.

CHAPTER 49. EXAMPLES

```
| imag(z)
```

Return the imaginary part of the complex number `z`.

Examples

```
| julia> imag(1 + 3im)
```

```
| 3
```

```
source
```

`Base.reim` – Function.

```
| reim(z)
```

Return both the real and imaginary parts of the complex number `z`.

Examples

```
| julia> reim(1 + 3im)
```

```
| (1, 3)
```

```
source
```

`Base.conj` – Function.

```
| conj(v::RowVector)
```

Returns a `ConjArray` lazy view of the input, where each element is conjugated.

Examples

```
| julia> v = [1+im, 1-im].'
```

```
| 1×2 RowVector{Complex{Int64}, Array{Complex{Int64}, 1}}:
```

```
| 1+1im 1-1im
```

```
julia> conj(v)
1×2 RowVector{Complex{Int64}}
→ [1-1im 1+1im]
```

**source**

```
|conj(z)
```

Compute the complex conjugate of a complex number  $z$ .

Examples

```
julia> conj(1 + 3im)
1 - 3im
```

**source**

[Base.angle](#) – Function.

```
|angle(z)
```

Compute the phase angle in radians of a complex number  $z$ .

Examples

```
julia> rad2deg(angle(1 + im))
45.0
```

```
julia> rad2deg(angle(1 - im))
-45.0
```

```
julia> rad2deg(angle(-1 - im))
-135.0
```

**source**

[Base.cis](#) – Function.

1008 | `cis(z)`

## CHAPTER 49. EXAMPLES

Return  $\exp(iz)$ .

Examples

```
| julia> cis(pi) ≈ -1  
| true
```

`source`

`Base.binomial` – Function.

```
| binomial(n, k)
```

Number of ways to choose  $k$  out of  $n$  items.

Examples

```
| julia> binomial(5, 3)  
| 10  
  
| julia> factorial(5) ÷ (factorial(5-3) * factorial(3))  
| 10
```

`source`

`Base.factorial` – Function.

```
| factorial(n)
```

Factorial of  $n$ . If  $n$  is an `Integer`, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if  $n$  is not small, but you can use `factorial(big(n))` to compute the result exactly in arbitrary precision. If  $n$  is not an `Integer`, `factorial(n)` is equivalent to `gamma(n+1)`.

```
julia> factorial(6)          1009
720

julia> factorial(21)
ERROR: OverflowError: 21 is too large to look up in the table
Stacktrace:
[...]

julia> factorial(21.0)
5.109094217170944e19

julia> factorial(big(21))
51090942171709440000
```

`source`

`Base.gcd` – Function.

```
| gcd(x,y)
```

Greatest common (positive) divisor (or zero if  $x$  and  $y$  are both zero).

Examples

```
julia> gcd(6,9)
```

```
3
```

```
julia> gcd(6,-9)
```

```
3
```

`source`

`Base.lcm` – Function.

```
| lcm(x,y)
```

Examples

```
julia> lcm(2,3)
```

```
6
```

```
julia> lcm(-2,3)
```

```
6
```

`source`

`Base.gcdx` – Function.

```
| gcdx(x,y)
```

Computes the greatest common (positive) divisor of  $x$  and  $y$  and their Bzout coefficients, i.e. the integer coefficients  $u$  and  $v$  that satisfy  $ux + vy = d = \gcd(x, y)$ . `gcdx(x, y)` returns  $(d, u, v)$ .

Examples

```
julia> gcdx(12, 42)
```

```
(6, -3, 1)
```

```
julia> gcdx(240, 46)
```

```
(2, -9, 47)
```

Note

Bzout coefficients are not uniquely defined. `gcdx` returns the minimal Bzout coefficients that are computed by the extended Euclidean algorithm. (Ref: D. Knuth, TAoCP, 2/e, p. 325, Algorithm X.) For signed integers, these coefficients  $u$  and  $v$  are minimal in the sense that  $|u| < |y/d|$  and  $|v| < |x/d|$ . Furthermore, the signs of  $u$  and  $v$  are chosen so that  $d$  is positive. For unsigned integers, the

coefficients  $u$  and  $v$  might be near their `typemax`, and the identity<sup>1011</sup> then holds only via the unsigned integers' modulo arithmetic.

`source`

`Base.ispow2` – Function.

```
| ispow2(n::Integer) -> Bool
```

Test whether  $n$  is a power of two.

Examples

```
| julia> ispow2(4)
```

```
true
```

```
| julia> ispow2(5)
```

```
false
```

`source`

`Base.nextpow2` – Function.

```
| nextpow2(n::Integer)
```

The smallest power of two not less than  $n$ . Returns 0 for  $n==0$ , and returns  $-nextpow2(-n)$  for negative arguments.

Examples

```
| julia> nextpow2(16)
```

```
16
```

```
| julia> nextpow2(17)
```

```
32
```

`source`

```
| prevpow2(n::Integer)
```

The largest power of two not greater than n. Returns 0 for n==0, and returns -prevpow2(-n) for negative arguments.

Examples

```
| julia> prevpow2(5)
```

```
| 4
```

```
| julia> prevpow2(0)
```

```
| 0
```

[source](#)

```
| nextpow(a, x)
```

The smallest  $a^n$  not less than x, where n is a non-negative integer. a must be greater than 1, and x must be greater than 0.

Examples

```
| julia> nextpow(2, 7)
```

```
| 8
```

```
| julia> nextpow(2, 9)
```

```
| 16
```

```
| julia> nextpow(5, 20)
```

```
| 25
```

```
| julia> nextpow(4, 16)
```

```
| 16
```

See also [prevpow](#).

1013

[source](#)

[Base.prevpow](#) – Function.

```
| prevpow(a, x)
```

The largest  $a^n$  not greater than  $x$ , where  $n$  is a non-negative integer.  $a$  must be greater than 1, and  $x$  must not be less than 1.

Examples

```
julia> prevpow(2, 7)
```

```
4
```

```
julia> prevpow(2, 9)
```

```
8
```

```
julia> prevpow(5, 20)
```

```
5
```

```
julia> prevpow(4, 16)
```

```
16
```

See also [nextpow](#).

[source](#)

[Base.nextprod](#) – Function.

```
| nextprod([k_1, k_2, ...], n)
```

Next integer greater than or equal to  $n$  that can be written as  $\prod k_i^{p_i}$  for integers  $p_1, p_2$ , etc.

Examples

```
1014 julia> nextprod([2, 3], 105)
```

```
108
```

```
julia> 2^2 * 3^3
```

```
108
```

```
source
```

Base.`invmod` – Function.

```
| invmod(x,m)
```

Take the inverse of  $x$  modulo  $m$ :  $y$  such that  $xy = 1 \pmod{m}$ , with  $\text{div}(x,y) = 0$ .

This is undefined for  $m = 0$ , or if  $\text{gcd}(x,m) \neq 1$ .

Examples

```
julia> invmod(2,5)
```

```
3
```

```
julia> invmod(2,3)
```

```
2
```

```
julia> invmod(5,6)
```

```
5
```

```
source
```

Base.`powermod` – Function.

```
| powermod(x::Integer, p::Integer, m)
```

Compute  $x^p \pmod{m}$ .

Examples

```
julia> powermod(2, 6, 5)
```

```
4
```

```

julia> mod(2^6, 5)
4

julia> powermod(5, 2, 20)
5

julia> powermod(5, 2, 19)
6

julia> powermod(5, 3, 19)
11

```

`source`

`Base.Math.gamma` – Function.

```
| gamma(x)
```

Compute the gamma function of  $x$ .

`source`

`Base.Math.lgamma` – Function.

```
| lgamma(x)
```

Compute the logarithm of the absolute value of `gamma` for `Real`  $x$ , while for `Complex`  $x$  compute the principal branch cut of the logarithm of `gamma(x)` (defined for negative `real(x)` by analytic continuation from positive `real(x)`).

`source`

`Base.Math.lfact` – Function.

```
| lfact(x)
```

1016 Compute the logarithmic factorial of a nonnegative integer  $x$ .  
CHAPTER 49x. EXAMPLES  
to [lgamma](#) of  $x + 1$ , but [lgamma](#) extends this function to non-integer  $x$ .

[source](#)

[Base.Math.beta](#) – Function.

| [beta\(x, y\)](#)

Euler integral of the first kind  $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x+y)$ .

[source](#)

[Base.Math.lbeta](#) – Function.

| [lbeta\(x, y\)](#)

Natural logarithm of the absolute value of the [beta](#) function  $\log(|B(x, y)|)$ .

[source](#)

[Base.ndigits](#) – Function.

| [ndigits\(n::Integer, b::Integer=10\)](#)

Compute the number of digits in integer  $n$  written in base  $b$ . The base  $b$  must not be in  $[-1, 0, 1]$ .

Examples

**julia>** [ndigits\(12345\)](#)

5

**julia>** [ndigits\(1022, 16\)](#)

3

**julia>** [base\(16, 1022\)](#)

"3fe"

[source](#)

```
| widemul(x, y)
```

Multiply  $x$  and  $y$ , giving the result as a larger type.

```
| julia> widemul(Float32(3.), 4.)  
| 1.2e+01
```

`source`

```
| @evalpoly(z, c...)
```

Evaluate the polynomial  $\sum_k c[k]z^{k-1}$  for the coefficients  $c[1], c[2], \dots$ ; that is, the coefficients are given in ascending order by power of  $z$ . This macro expands to efficient inline code that uses either Horner's method or, for complex  $z$ , a more efficient Goertzel-like algorithm.

```
| julia> @evalpoly(3, 1, 0, 1)
```

```
| 10
```

```
| julia> @evalpoly(2, 1, 0, 1)
```

```
| 5
```

```
| julia> @evalpoly(2, 1, 1, 1)
```

```
| 7
```

`source`

```
| @fastmath expr
```

Execute a transformed version of the expression, which calls functions that may violate strict IEEE semantics. This allows the fastest possible

1018 operation, but results are undefined – be careful! CHAPTER 19 EXAMPLES  
may change numerical results.

This sets the [LLVM Fast-Math flags](#), and corresponds to the `-ffast-math` option in clang. See [the notes on performance annotations](#) for more details.

Examples

```
julia> @fastmath 1+2  
3  
  
julia> @fastmath(sin(3))  
0.1411200080598672  
  
source
```

## 49.1 Statistics

[Base.mean](#) – Function.

```
| mean(f::Function, v)
```

Apply the function `f` to each element of `v` and take the mean.

```
julia> mean(v, [1, 2, 3])  
1.3820881233139908  
  
julia> mean([√1, √2, √3])  
1.3820881233139908
```

```
source
```

```
| mean(v[, region])
```

Compute the mean of whole array `v`, or optionally along the dimensions in `region`.

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

`source`

`Base.mean!` – Function.

```
| mean!(r, v)
```

Compute the mean of `v` over the singleton dimensions of `r`, and write results to `r`.

Examples

```
julia> v = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> mean!([1., 1.], v)
2-element Array{Float64,1}:
 1.5
 3.5

julia> mean!([1. 1.], v)
1×2 Array{Float64,2}:
 2.0  3.0
```

`source`

`Base.std` – Function.

```
| std(v[, region]; corrected::Bool=true, mean=nothing)
```

1020 Compute the sample standard deviation of a vector `v` along dimensions in `region`. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sqrt(sum((v - mean(v)).^2) / (length(v) - 1))`. A pre-computed `mean` may be provided. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

#### Note

Julia does not ignore `Nan` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

#### source

[Base.`stdm`](#) – Function.

```
| stdm(v, m; corrected::Bool=true)
```

Compute the sample standard deviation of a vector `v` with known mean `m`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

#### Note

Julia does not ignore `Nan` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

#### source

[Base.`var`](#) – Function.

```
| var(v[, region]; corrected::Bool=true, mean=nothing)
```

**49.1. `var`** Compute the sample variance of a vector or array `v`, optionally along dimensions in `region`. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of `v` is an IID drawn from that generative distribution. This computation is equivalent to calculating `sum(abs2, v - mean(v)) / (length(v) - 1)`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`. The mean `mean` over the region may be provided.

#### Note

Julia does not ignore `Nan` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

#### source

[Base.varm](#) – Function.

```
| varm(v, m[, region]; corrected::Bool=true)
```

Compute the sample variance of a collection `v` with known mean(s) `m`, optionally over `region`. `m` may contain means for each dimension of `v`. If `corrected` is `true`, then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

#### Note

Julia does not ignore `Nan` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

#### source

[Base.middle](#) – Function.

1022 `middle(x)`

## CHAPTER 49. EXAMPLES

Compute the middle of a scalar value, which is equivalent to `x` itself, but of the type of `middle(x, x)` for consistency.

`source`

```
| middle(x, y)
```

Compute the middle of two reals `x` and `y`, which is equivalent in both value and type to computing their mean  $((x + y) / 2)$ .

`source`

```
| middle(range)
```

Compute the middle of a range, which consists of computing the mean of its extrema. Since a range is sorted, the mean is performed with the first and last element.

```
| julia> middle(1:10)
```

```
| 5.5
```

`source`

```
| middle(a)
```

Compute the middle of an array `a`, which consists of finding its extrema and then computing their mean.

```
| julia> a = [1,2,3.6,10.9]
```

```
| 4-element Array{Float64,1}:
```

```
|   1.0
```

```
|   2.0
```

```
|   3.6
```

```
| 10.9
```

49.1| STATISTICS  
julia> middle(a)  
5.95

1023

**source**

[Base.median](#) – Function.

| median(v[, region])

Compute the median of an entire array `v`, or, optionally, along the dimensions in `region`. For an even number of elements no exact median element exists, so the result is equivalent to calculating mean of two median elements.

**Note**

Julia does not ignore `Nan` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended.

**source**

[Base.median!](#) – Function.

| median!(v)

Like `median`, but may overwrite the input vector.

**source**

[Base.quantile](#) – Function.

| quantile(v, p; sorted=false)

Compute the quantile(s) of a vector `v` at a specified probability or vector or tuple of probabilities `p`. The keyword argument `sorted` indicates whether `v` can be assumed to be sorted.

1024 The `p` should be on the interval [0,1], and `v` should not have any `NaN` values.

Quantiles are computed via linear interpolation between the points  $((k-1)/(n-1), v[k])$ , for  $k = 1:n$  where  $n = \text{length}(v)$ . This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

### Note

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended. `quantile` will throw an `ArgumentError` in the presence of `NaN` values in the data array.

Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361–365

### source

`Base.quantile!` – Function.

```
| quantile!([q, ] v, p; sorted=false)
```

Compute the quantile(s) of a vector `v` at the probability or probabilities `p`, which can be given as a single value, a vector, or a tuple. If `p` is a vector, an optional output array `q` may also be specified. (If not provided, a new output array is created.) The keyword argument `sorted` indicates whether `v` can be assumed to be sorted; if `false` (the default), then the elements of `v` may be partially sorted.

The elements of `p` should be on the interval [0,1], and `v` should not have any `NaN` values.

Quantiles are computed via linear interpolation between the points  $((k-1)/(n-1), v[k])$ , for  $k = 1:n$  where  $n = \text{length}(v)$ . This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `DataArrays.jl` package is recommended. `quantile!` will throw an `ArgumentError` in the presence of `NaN` values in the data array.

Hyndman, R.J and Fan, Y. (1996) "Sample Quantiles in Statistical Packages", *The American Statistician*, Vol. 50, No. 4, pp. 361–365

### `source`

`Base.cov` – Function.

```
| cov(x::AbstractVector; corrected::Bool=true)
```

Compute the variance of the vector `x`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = length(x)`.

### `source`

```
| cov(X::AbstractMatrix[, vardim::Int=1]; corrected::Bool=true)
```

Compute the covariance matrix of the matrix `X` along the dimension `vardim`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, vardim)`.

### `source`

```
| cov(x::AbstractVector, y::AbstractVector; corrected::Bool=true)
```

Compute the covariance between the vectors `x` and `y`. If `corrected` is `true` (the default), computes  $\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})^*$  where `*` denotes the complex conjugate and `n = length(x) = length(y)`. If `corrected` is `false`, computes  $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})^*$ .

1026

**source**

CHAPTER 49. EXAMPLES

```
| cov(X::AbstractVecOrMat, Y::AbstractVecOrMat[, vardim::Int=1];  
|   corrected::Bool=true)
```

Compute the covariance between the vectors or matrices X and Y along the dimension `vardim`. If `corrected` is `true` (the default) then the sum is scaled with `n-1`, whereas the sum is scaled with `n` if `corrected` is `false` where `n = size(X, vardim) = size(Y, vardim)`.

**source**

`Base.cor` – Function.

```
| cor(x::AbstractVector)
```

Return the number one.

**source**

```
| cor(X::AbstractMatrix[, vardim::Int=1])
```

Compute the Pearson correlation matrix of the matrix X along the dimension `vardim`.

**source**

```
| cor(x::AbstractVector, y::AbstractVector)
```

Compute the Pearson correlation between the vectors x and y.

**source**

```
| cor(X::AbstractVecOrMat, Y::AbstractVecOrMat[, vardim=1])
```

Compute the Pearson correlation between the vectors or matrices X and Y along the dimension `vardim`.

**source**

# Chapter 50

## Numbers

### 50.1 Standard Numeric Types

Abstract number types

`Core.Number` – Type.

| `Number`

Abstract supertype for all number types.

`source`

`Core.Real` – Type.

| `Real <: Number`

Abstract supertype for all real numbers.

`source`

`Core.AbstractFloat` – Type.

| `AbstractFloat <: Real`

Abstract supertype for all floating point numbers.

`source`

[Core.Integer](#) – Type.

CHAPTER 50. NUMBERS

| Integer <: Real

Abstract supertype for all integers.

[source](#)

[Core.Signed](#) – Type.

| Signed <: Integer

Abstract supertype for all signed integers.

[source](#)

[Core.Unsigned](#) – Type.

| Unsigned <: Integer

Abstract supertype for all unsigned integers.

[source](#)

[Base.AbstractIrrational](#) – Type.

| AbstractIrrational <: Real

Number type representing an exact irrational value.

[source](#)

Concrete number types

[Core.Float16](#) – Type.

| Float16 <: AbstractFloat

16-bit floating point number type.

[source](#)

[Core.Float32](#) – Type.

32-bit floating point number type.

`source`

[Core.Float64](#) – Type.

| `Float64 <: AbstractFloat`

64-bit floating point number type.

`source`

[Base.MPFR.BigFloat](#) – Type.

| `BigFloat <: AbstractFloat`

Arbitrary precision floating point number type.

`source`

[Core.Bool](#) – Type.

| `Bool <: Integer`

Boolean type.

`source`

[Core.Int8](#) – Type.

| `Int8 <: Signed`

8-bit signed integer type.

`source`

[Core.UInt8](#) – Type.

| `UInt8 <: Unsigned`

1038-bit unsigned integer type.

CHAPTER 50. NUMBERS

[source](#)

[Core.Int16](#) – Type.

| [Int16 <: Signed](#)

16-bit signed integer type.

[source](#)

[Core.UInt16](#) – Type.

| [UInt16 <: Unsigned](#)

16-bit unsigned integer type.

[source](#)

[Core.Int32](#) – Type.

| [Int32 <: Signed](#)

32-bit signed integer type.

[source](#)

[Core.UInt32](#) – Type.

| [UInt32 <: Unsigned](#)

32-bit unsigned integer type.

[source](#)

[Core.Int64](#) – Type.

| [Int64 <: Signed](#)

64-bit signed integer type.

[source](#)

| UInt64 <: Unsigned

64-bit unsigned integer type.

**source**

[Core.Int128](#) – Type.

| Int128 <: Signed

128-bit signed integer type.

**source**

[Core.UInt128](#) – Type.

| UInt128 <: Unsigned

128-bit unsigned integer type.

**source**

[Base.GMP.BigInt](#) – Type.

| BigInt <: Signed

Arbitrary precision integer type.

**source**

[Base.Complex](#) – Type.

| Complex{T<:Real} <: Number

Complex number type with real and imaginary part of type T.

`Complex32`, `Complex64` and `Complex128` are aliases for `Complex{Float16}`, `Complex{Float32}` and `Complex{Float64}` respectively.

**source**

`Base.Rational` – Type.

CHAPTER 50. NUMBERS

```
| Rational{T<:Integer} <: Real
```

Rational number type, with numerator and denominator of type T.

`source`

`Base.Irrational` – Type.

```
| Irrational{sym} <: AbstractIrrational
```

Number type representing an exact irrational value denoted by the symbol sym.

`source`

## 50.2 Data Formats

`Base.bin` – Function.

```
| bin(n, pad::Int=1)
```

Convert an integer to a binary string, optionally specifying a number of digits to pad to.

```
julia> bin(10,2)
```

```
"1010"
```

```
julia> bin(10,8)
```

```
"00001010"
```

`source`

`Base.hex` – Function.

```
| hex(n, pad::Int=1)
```

Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

```
julia> hex(20)  
"14"  
  
julia> hex(20, 3)  
"014"
```

**source**

**Base.dec** – Function.

```
| dec(n, pad::Int=1)
```

Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

Examples

```
julia> dec(20)  
"20"  
  
julia> dec(20, 3)  
"020"
```

**source**

**Base.oct** – Function.

```
| oct(n, pad::Int=1)
```

Convert an integer to an octal string, optionally specifying a number of digits to pad to.

```
julia> oct(20)  
"24"  
  
julia> oct(20, 3)  
"024"
```

`Base.base` – Function.

```
| base(base::Integer, n::Integer, pad::Integer=1)
```

Convert an integer `n` to a string in the given `base`, optionally specifying a number of digits to pad to.

```
| julia> base(13, 5, 4)
"0005"
```

```
| julia> base(5, 13, 4)
"0023"
```

`source`

`Base.digits` – Function.

```
| digits([T<:Integer], n::Integer, base::T=10, pad::Integer=1)
```

Returns an array with element type `T` (default `Int`) of the digits of `n` in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that `n == sum([digits[k]*base^(k-1) for k=1:length(digits)])`.

Examples

```
| julia> digits(10, 10)
2-element Array{Int64,1}:
 0
 1
```

```
| julia> digits(10, 2)
4-element Array{Int64,1}:
 0
```

```
0  
1
```

```
julia> digits(10, 2, 6)  
6-element Array{Int64,1}:  
0  
1  
0  
1  
0  
0
```

[source](#)

[Base.digits!](#) – Function.

```
| digits!(array, n::Integer, base::Integer=10)
```

Fills an array of the digits of `n` in the given base. More significant digits are at higher indexes. If the array length is insufficient, the least significant digits are filled up to the array length. If the array length is excessive, the excess portion is filled with zeros.

Examples

```
julia> digits!([2,2,2,2], 10, 2)  
4-element Array{Int64,1}:  
0  
1  
0  
1
```

```
julia> digits!([2,2,2,2,2,2], 10, 2)
```

1036 6-element Array{Int64,1}:

## CHAPTER 50. NUMBERS

0  
1  
0  
1  
0  
0

## source

`Base.bitstring` – Function.

`bitstring(n)`

A string giving the literal bit representation of a number.

## Examples

```
julia> bitstring(4)
```

```
julia> bitstring(2.2)
```

## source

## Base.parse – Method.

`parse(type, str, [base])`

Parse a string as a number. If the type is an integer type, then a base can be specified (the default is 10). If the type is a floating point type, the string is parsed as a decimal floating point number. If the string does not contain a valid number, an error is raised.

```
julia> parse(Int, "1234")
```

1234

```
julia> parse(Int, "1234", 5)  
194
```

```
julia> parse(Int, "afc", 16)  
2812
```

```
julia> parse(Float64, "1.2e-3")  
0.0012
```

[source](#)

[Base.tryparse](#) – Function.

```
| tryparse(type, str, [base])
```

Like [parse](#), but returns a [Nullable](#) of the requested type. The result will be null if the string does not contain a valid number.

[source](#)

[Base.big](#) – Function.

```
| big(x)
```

Convert a number to a maximum precision representation (typically [Big-Int](#) or [BigFloat](#)). See [BigFloat](#) for information about some pitfalls with floating-point numbers.

[source](#)

[Base.signed](#) – Function.

```
| signed(x)
```

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

[source](#)

[Base.unsigned](#) – Function.

CHAPTER 50. NUMBERS

```
| unsigned(x) -> Unsigned
```

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

```
julia> unsigned(-2)
```

```
0xffffffffffffffe
```

```
julia> unsigned(2)
```

```
0x0000000000000002
```

```
julia> signed(unsigned(-2))
```

```
-2
```

[source](#)

[Base.float](#) – Method.

```
| float(x)
```

Convert a number or array to a floating point data type. When passed a string, this function is equivalent to `parse(Float64, x)`.

[source](#)

[Base.Math.significand](#) – Function.

```
| significand(x)
```

Extract the `significand(s)` (a.k.a. mantissa), in binary representation, of a floating-point number. If `x` is a non-zero finite number, then the result will be a number of the same type on the interval  $[1, 2)$ . Otherwise `x` is returned.

Examples

50.2| DATA FORMATS  
julia> significand(15.2)/15.2

0.125

julia> significand(15.2)\*8

15.2

source

Base.Math.exponent – Function.

| exponent(x) -> Int

Get the exponent of a normalized floating-point number.

source

Base.complex – Method.

| complex(r, [i])

Convert real numbers or arrays to complex. *i* defaults to zero.

Examples

julia> complex(7)

7 + 0im

julia> complex([1, 2, 3])

3-element Array{Complex{Int64},1}:

1 + 0im

2 + 0im

3 + 0im

source

Base.bswap – Function.

| bswap(n)

## Examples

```
julia> a = bswap(4)
```

288230376151711744

```
julia> bswap(a)
```

4

```
julia> bin(1)
```

"1"

```
julia> bin(bswap(1))
```

## source

`Base.hex2bytes` – Function.

```
hex2bytes(s::Union{AbstractString, AbstractVector{UInt8}})
```

Given a string or array `s` of ASCII codes for a sequence of hexadecimal digits, returns a `Vector<UInt8>` of bytes corresponding to the binary representation: each successive pair of hexadecimal digits in `s` gives the value of one byte in the return vector.

The length of `s` must be even, and the returned array has half of the length of `s`. See also [hex2bytes!](#) for an in-place version, and [bytes2hex](#) for the inverse.

## Examples

```
julia> s = hex(12345)
```

"3039"

```
julia> hex2bytes(s)  
2-element Array{UInt8,1}:  
 0x30  
 0x39
```

```
julia> a = b"01abEF"  
6-element Array{UInt8,1}:  
 0x30  
 0x31  
 0x61  
 0x62  
 0x45  
 0x46
```

```
julia> hex2bytes(a)  
3-element Array{UInt8,1}:  
 0x01  
 0xab  
 0xef
```

**source**

[Base.hex2bytes!](#) – Function.

```
| hex2bytes!(d::AbstractVector{UInt8}, s::AbstractVector{UInt8})
```

Convert an array `s` of bytes representing a hexadecimal string to its binary representation, similar to [hex2bytes](#) except that the output is written in-place in `d`. The length of `s` must be exactly twice the length of `d`.

**source**

[Base.bytes2hex](#) – Function.

```
| bytes2hex(bin_arr::Array{UInt8, 1}) -> String
```

104 Convert an array of bytes to its hexadecimal representation. The characters are in lower-case.

Examples

```
julia> a = hex(12345)
"3039"

julia> b = hex2bytes(a)
2-element Array{UInt8,1}:
 0x30
 0x39

julia> bytes2hex(b)
"3039"
```

[source](#)

### 50.3 General Number Functions and Constants

[Base.one](#) – Function.

```
one(x)
one(T::type)
```

Return a multiplicative identity for  $x$ : a value such that  $\text{one}(x) * x == x * \text{one}(x) == x$ . Alternatively  $\text{one}(T)$  can take a type  $T$ , in which case  $\text{one}$  returns a multiplicative identity for any  $x$  of type  $T$ .

If possible,  $\text{one}(x)$  returns a value of the same type as  $x$ , and  $\text{one}(T)$  returns a value of type  $T$ . However, this may not be the case for types representing dimensionful quantities (e.g. time in days), since the multiplicative identity must be dimensionless. In that case,  $\text{one}(x)$  should return an identity value of the same precision (and shape, for matrices) as  $x$ .

`x` is dimensionful, use `oneunit` instead.

```
julia> one(3.7)
```

```
1.0
```

```
julia> one(Int)
```

```
1
```

```
julia> import Dates; one(Dates.Day(1))
```

```
1
```

`source`

`Base.oneunit` – Function.

```
| oneunit(x::T)
```

```
| oneunit(T::Type)
```

Returns `T(one(x))`, where `T` is either the type of the argument or (if a type is passed) the argument. This differs from `one` for dimensionful quantities: `one` is dimensionless (a multiplicative identity) while `oneunit` is dimensionful (of the same type as `x`, or of type `T`).

```
julia> oneunit(3.7)
```

```
1.0
```

```
julia> import Dates; oneunit(Dates.Day)
```

```
1 day
```

`source`

`Base.zero` – Function.

```
| zero(x)
```

1044 Get the additive identity element for the type of CHAPTER FIFTY NUMBERS  
type itself).

```
julia> zero(1)
0

julia> zero(big"2.0")
0.0

julia> zero(rand(2,2))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0
```

source

Base.`im` – Constant.

| `im`

The imaginary unit.

Examples

```
julia> im * im
-1 + 0im
```

source

Base.MathConstants.`pi` – Constant.

|  $\pi$ 
| `pi`

The constant pi.

```
julia> pi
```

```
π = 3.1415926535897...
```

source

Base.MathConstants. – Constant.

```
e
```

The constant .

```
julia>
```

```
= 2.7182818284590...
```

source

Base.MathConstants.catalan – Constant.

```
catalan
```

Catalan's constant.

```
julia> Base.MathConstants.catalan
```

```
catalan = 0.9159655941772...
```

source

Base.MathConstants.eulergamma – Constant.

```
γ
```

```
eulergamma
```

Euler's constant.

```
julia> Base.MathConstants.eulergamma
```

```
γ = 0.5772156649015...
```

1046 [source](#)

CHAPTER 50. NUMBERS

[Base.MathConstants.golden](#) – Constant.

```
| ϕ  
|  
golden
```

The golden ratio.

```
| julia> Base.MathConstants.golden  
| ϕ = 1.6180339887498...
```

[source](#)

[Base.Inf](#) – Constant.

```
| Inf
```

Positive infinity of type [Float64](#).

[source](#)

[Base.Inf32](#) – Constant.

```
| Inf32
```

Positive infinity of type [Float32](#).

[source](#)

[Base.Inf16](#) – Constant.

```
| Inf16
```

Positive infinity of type [Float16](#).

[source](#)

[Base.NaN](#) – Constant.

```
| NaN
```

`source`

`Base.NaN32` – Constant.

`NaN32`

A not-a-number value of type `Float32`.

`source`

`Base.NaN16` – Constant.

`NaN16`

A not-a-number value of type `Float16`.

`source`

`Base.issubnormal` – Function.

`issubnormal(f) -> Bool`

Test whether a floating point number is subnormal.

`source`

`Base.isfinite` – Function.

`isfinite(f) -> Bool`

Test whether a number is finite.

`julia> isfinite(5)`

`true`

`julia> isfinite(NaN32)`

`false`

`source`

`Base.isinf` – Function.

CHAPTER 50. NUMBERS

```
| isinf(f) -> Bool
```

Test whether a number is infinite.

`source`

`Base.isnan` – Function.

```
| isnan(f) -> Bool
```

Test whether a floating point number is not a number (NaN).

`source`

`Base.iszero` – Function.

```
| iszero(x)
```

Return `true` if `x == zero(x)`; if `x` is an array, this checks whether all of the elements of `x` are zero.

```
| julia> iszero(0.0)
| true
```

`source`

`Base.isone` – Function.

```
| isone(x)
```

Return `true` if `x == one(x)`; if `x` is an array, this checks whether `x` is an identity matrix.

```
| julia> isone(1.0)
| true
```

`source`

`Base.nextfloat` – Function.

The result of  $n$  iterative applications of `nextfloat` to  $x$  if  $n \geq 0$ , or  $-n$  applications of `prevfloat` if  $n < 0$ .

**source**

```
| nextfloat(x::AbstractFloat)
```

Returns the smallest floating point number  $y$  of the same type as  $x$  such  $x < y$ . If no such  $y$  exists (e.g. if  $x$  is `Inf` or `Nan`), then returns  $x$ .

**source**

`Base.prevfloat` – Function.

```
| prevfloat(x::AbstractFloat)
```

Returns the largest floating point number  $y$  of the same type as  $x$  such  $y < x$ . If no such  $y$  exists (e.g. if  $x$  is `-Inf` or `Nan`), then returns  $x$ .

**source**

`Base.isinteger` – Function.

```
| isinteger(x) -> Bool
```

Test whether  $x$  is numerically equal to some integer.

```
| julia> isinteger(4.0)
```

```
| true
```

**source**

`Base.isreal` – Function.

```
| isreal(x) -> Bool
```

Test whether  $x$  or all its elements are numerically equal to some real number.

Examples

```
1050 julia> isreal(5.)
```

CHAPTER 50. NUMBERS

```
true
```

```
julia> isreal([4.; complex(0,1)])
```

```
false
```

[source](#)

[Core.Float32](#) – Method.

```
| Float32(x [, mode::RoundingMode])
```

Create a `Float32` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float32(1/3, RoundDown)
```

```
0.3333333f0
```

```
julia> Float32(1/3, RoundUp)
```

```
0.3333334f0
```

See [RoundingMode](#) for available rounding modes.

[source](#)

[Core.Float64](#) – Method.

```
| Float64(x [, mode::RoundingMode])
```

Create a `Float64` from `x`. If `x` is not exactly representable then `mode` determines how `x` is rounded.

Examples

```
julia> Float64(pi, RoundDown)
```

```
3.141592653589793
```

```
julia> Float64(pi, RoundUp)  
3.1415926535897936
```

See [RoundingMode](#) for available rounding modes.

[source](#)

[Base.GMP.BigInt](#) – Method.

```
BigInt(x)
```

Create an arbitrary precision integer. `x` may be an `Int` (or anything that can be converted to an `Int`). The usual mathematical operators are defined for this type, and results are promoted to a [BigInt](#).

Instances can be constructed from strings via [parse](#), or using the `big` string literal.

```
julia> parse(BigInt, "42")  
42
```

```
julia> big"313"  
313
```

[source](#)

[Base.MPFR.BigFloat](#) – Method.

```
BigFloat(x)
```

Create an arbitrary precision floating point number. `x` may be an [Integer](#), a [Float64](#) or a [BigInt](#). The usual mathematical operators are defined for this type, and results are promoted to a [BigFloat](#).

Note that because decimal literals are converted to floating point numbers when parsed, `BigFloat(2.1)` may not yield what you expect. You

1052 may instead prefer to  
the **big** string literal.

```
julia> BigFloat(2.1)
```

2.10000000000000088817841970012523233890533447265625

```
julia> big"2.1"
```

## source

`Base.Rounding.rounding` – Function.

rounding(T)

Get the current floating point rounding mode for type T, controlling the rounding of basic arithmetic functions (+, -, \*, / and sqrt) and type conversion.

See [RoundingMode](#) for available modes.

## source

`Base.Rounding.setrounding` – Method.

setrounding(T, mode)

Set the rounding mode of floating point type T, controlling the rounding of basic arithmetic functions (+, -, \*, / and `sqrt`) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default RoundNearest.

Note that this may affect other types, for instance changing the rounding mode of `Float64` will change the rounding mode of `Float32`. See [RoundingMode](#) for available modes.

## Warning

50.3. GENERAL MEMBER FUNCTIONS AND CONSTRAINTS 1053

unexpected or incorrect values.

### source

[Base.Rounding.setrounding](#) – Method.

```
| setrounding(f::Function, T, mode)
```

Change the rounding mode of floating point type  $T$  for the duration of  $f$ .

It is logically equivalent to:

```
| old = rounding(T)
| setrounding(T, mode)
| f()
| setrounding(T, old)
```

See [RoundingMode](#) for available rounding modes.

### Warning

This feature is still experimental, and may give unexpected or incorrect values. A known problem is the interaction with compiler optimisations, e.g.

```
| julia> setrounding(Float64, RoundDown) do
|           1.1 + 0.1
|       end
| 1.2000000000000002
```

Here the compiler is constant folding, that is evaluating a known constant expression at compile time, however the rounding mode is only changed at runtime, so this is not reflected in the function result. This can be avoided by moving constants outside the expression, e.g.

```
| julia> x = 1.1; y = 0.1;
```

```
julia> setrounding(Float64, RoundDown) do
           x + y
       end
1.2
```

[source](#)

[Base.Rounding.get\\_zero\\_subnormals](#) – Function.

```
| get_zero_subnormals() -> Bool
```

Returns **false** if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and **true** if they might be converted to zeros.

[source](#)

[Base.Rounding.set\\_zero\\_subnormals](#) – Function.

```
| set_zero_subnormals(yes::Bool) -> Bool
```

If **yes** is **false**, subsequent floating-point operations follow rules for IEEE arithmetic on subnormal values ("denormals"). Otherwise, floating-point operations are permitted (but not required) to convert subnormal inputs or outputs to zero. Returns **true** unless **yes==true** but the hardware does not support zeroing of subnormal numbers.

**set\_zero\_subnormals(true)** can speed up some computations on some hardware. However, it can break identities such as  $(x-y==0) == (x==y)$ .

[source](#)

Integers

[Base.count\\_ones](#) – Function.

Number of ones in the binary representation of x.

```
| julia> count_ones(7)  
| 3
```

**source**

[Base.count\\_zeros](#) – Function.

```
| count_zeros(x::Integer) -> Integer
```

Number of zeros in the binary representation of x.

```
| julia> count_zeros(Int32(2 ^ 16 - 1))  
| 16
```

**source**

[Base.leading\\_zeros](#) – Function.

```
| leading_zeros(x::Integer) -> Integer
```

Number of zeros leading the binary representation of x.

```
| julia> leading_zeros(Int32(1))  
| 31
```

**source**

[Base.leading\\_ones](#) – Function.

```
| leading_ones(x::Integer) -> Integer
```

Number of ones leading the binary representation of x.

```
| julia> leading_ones(UInt32(2 ^ 32 - 2))  
| 31
```

`Base.trailing_zeros` – Function.

```
| trailing_zeros(x::Integer) -> Integer
```

Number of zeros trailing the binary representation of `x`.

```
| julia> trailing_zeros(2)
```

```
| 1
```

`source`

`Base.trailing_ones` – Function.

```
| trailing_ones(x::Integer) -> Integer
```

Number of ones trailing the binary representation of `x`.

```
| julia> trailing_ones(3)
```

```
| 2
```

`source`

`Base.isodd` – Function.

```
| isodd(x::Integer) -> Bool
```

Returns `true` if `x` is odd (that is, not divisible by 2), and `false` otherwise.

```
| julia> isodd(9)
```

```
| true
```

```
| julia> isodd(10)
```

```
| false
```

`source`

`Base.iseven` – Function.

Returns `true` if `x` is even (that is, divisible by 2), and `false` otherwise.

```
julia> iseven(9)
```

```
false
```

```
julia> iseven(10)
```

```
true
```

[source](#)

## 50.4 BigFloats

The `BigFloat` type implements arbitrary-precision floating-point arithmetic using the [GNU MPFR library](#).

`Base.precision` – Function.

```
precision(num::AbstractFloat)
```

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

[source](#)

`Base.precision` – Method.

```
precision(BigFloat)
```

Get the precision (in bits) currently used for `BigFloat` arithmetic.

[source](#)

`Base_MPFR.setprecision` – Function.

```
setprecision([T=BigFloat,] precision::Int)
```

source

```
| setprecision(f::Function, [T=BigFloat,] precision::Integer)
```

Change the T arithmetic precision (in bits) for the duration of f. It is logically equivalent to:

```
| old = precision(BigFloat)
| setprecision(BigFloat, precision)
| f()
| setprecision(BigFloat, old)
```

Often used as `setprecision(T, precision) do ... end`

source

[Base.MPFR.BigFloat](#) – Method.

```
| BigFloat(x, prec::Int)
```

Create a representation of x as a [BigFloat](#) with precision prec.

source

[Base.MPFR.BigFloat](#) – Method.

```
| BigFloat(x, rounding::RoundingMode)
```

Create a representation of x as a [BigFloat](#) with the current global precision and rounding mode rounding.

source

[Base.MPFR.BigFloat](#) – Method.

```
| BigFloat(x, prec::Int, rounding::RoundingMode)
```

Create a representation of x as a [BigFloat](#) with precision prec and rounding mode rounding.

[Base.MPFR.BigFloat](#) – Method.

```
| BigFloat(x::String)
```

Create a representation of the string `x` as a [BigFloat](#).

[source](#)

## 50.5 Random Numbers

Random number generation in Julia uses the [Mersenne Twister library](#) via `MersenneTwister` objects. Julia has a global RNG, which is used by default. Other RNG types can be plugged in by inheriting the `AbstractRNG` type; they can then be used to have multiple streams of random numbers. Besides `MersenneTwister`, Julia also provides the `RandomDevice` RNG type, which is a wrapper over the OS provided entropy.

Most functions related to random generation accept an optional `AbstractRNG` as the first argument, `rng`, which defaults to the global one if not provided. Moreover, some of them accept optionally dimension specifications `dims...` (which can be given as a tuple) to generate arrays of random values.

A `MersenneTwister` or `RandomDevice` RNG can generate random numbers of the following types: [Float16](#), [Float32](#), [Float64](#), [BigFloat](#), [Bool](#), [Int8](#), [UInt8](#), [Int16](#), [UInt16](#), [Int32](#), [UInt32](#), [Int64](#), [UInt64](#), [Int128](#), [UInt128](#), [BigInt](#) (or complex numbers of those types). Random floating point numbers are generated uniformly in  $[0, 1]$ . As `BigInt` represents unbounded integers, the interval must be specified (e.g. `rand(big(1:6))`).

[Base.Random.srand](#) – Function.

```
| srand([rng=GLOBAL_RNG], seed) -> rng
| srand([rng=GLOBAL_RNG]) -> rng
```

106 Reseed the random number generator: `rng` will generate the same sequence of numbers if and only if a `seed` is provided. Some RNGs don't accept a seed, like `RandomDevice`. After the call to `srand`, `rng` is equivalent to a newly created object initialized with the same seed.

## Examples

```
julia> srand(1234);
```

```
julia> x1 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797
```

```
julia> srand(1234);
```

```
julia> x2 = rand(2)
2-element Array{Float64,1}:
 0.590845
 0.766797
```

```
julia> x1 == x2
```

```
true
```

```
julia> rng = MersenneTwister(1234); rand(rng, 2) == x1
true
```

```
julia> MersenneTwister(1) == srand(rng, 1)
true
```

```
julia> rand(srand(rng), Bool) # not reproducible
true
```

```
julia> rand(srand(rng), Bool)
false

julia> rand(MersenneTwister(), Bool) # not reproducible either
true
```

source

[Base.Random.MersenneTwister](#) – Type.

```
MersenneTwister(seed)
MersenneTwister()
```

Create a **MersenneTwister** RNG object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers. The **seed** may be a non-negative integer or a vector of **UInt32** integers. If no seed is provided, a randomly generated one is created (using entropy from the system). See the **srand** function for reseeding an already existing **MersenneTwister** object.

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> x1 = rand(rng, 2)
2-element Array{Float64,1}:
 0.5908446386657102
 0.7667970365022592
```

```
julia> rng = MersenneTwister(1234);
```

```
julia> x2 = rand(rng, 2)
2-element Array{Float64,1}:
```

```
1062 0.5908446386657102  
      0.7667970365022592
```

## CHAPTER 50. NUMBERS

```
julia> x1 == x2  
true
```

**source**

[Base.Random.RandomDevice](#) – Type.

```
RandomDevice()
```

Create a `RandomDevice` RNG object. Two such objects will always generate different streams of random numbers. The entropy is obtained from the operating system.

**source**

[Base.Random.rand](#) – Function.

```
rand([rng=GLOBAL_RNG], [S], [dims...])
```

Pick a random element or array of random elements from the set of values specified by `S`; `S` can be

- an indexable collection (for example `1:n` or `['x', 'y', 'z']`),
- an `Associative` or `AbstractSet` object,
- a string (considered as a collection of characters), or
- a type: the set of values to pick from is then equivalent to `type-min(S):type-max(S)` for integers (this is not applicable to `BigInt`), and to `[0,1]` for floating point numbers;

`S` defaults to [Float64](#).

Examples

```
rand(Int, 2)  
2-element Array{Int64,1}:  
1339893410598768192  
1575814717733606317
```

```
julia> rand(MersenneTwister(0), Dict(1=>2, 3=>4))  
1=>2
```

### Note

The complexity of `rand(rng, s)` (`s` is a union of `Associative` and `AbstractSet`) is linear in the length of `s`, unless an optimized method with constant complexity is available, which is the case for `Dict`, `Set` and `BitSet`. For more than a few calls, use `rand(rng, collect(s))` instead, or either `rand(rng, Dict(s))` or `rand(rng, Set(s))` as appropriate.

### source

[Base.Random.rand!](#) – Function.

```
rand!([rng=GLOBAL_RNG], A, [S=eltype(A)])
```

Populate the array `A` with random values. If `S` is specified (`S` can be a type or a collection, cf. [rand](#) for details), the values are picked randomly from `S`. This is equivalent to `copy!(A, rand(rng, S, size(A)))` but without allocating a new array.

### Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> rand!(rng, zeros(5))  
5-element Array{Float64,1}:  
0.5908446386657102
```

```
1064 0.7667970365022592  
      0.5662374165061859  
      0.4600853424625171  
      0.7940257103317943
```

## CHAPTER 50. NUMBERS

`source`

[Base.Random.bitrand](#) – Function.

```
| bitrand([rng=GLOBAL_RNG], [dims...])
```

Generate a `BitArray` of random boolean values.

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> bitrand(rng, 10)
```

```
10-element BitArray{1}:
```

```
  true
```

```
  true
```

```
  true
```

```
 false
```

```
  true
```

```
 false
```

```
  false
```

```
  true
```

```
  false
```

```
  true
```

`source`

[Base.Random.randn](#) – Function.

```
| randn([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

50.5. RANDOM NUMBERS distributed random number of type T with mean<sup>10.65</sup> and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default), and their `Complex` counterparts. When the type argument is complex, the values are drawn from the circularly symmetric complex normal distribution.

Examples

```
julia> rng = MersenneTwister(1234);

julia> randn(rng, Complex128)
0.6133070881429037 - 0.6376291670853887im

julia> randn(rng, Complex64, (2, 3))
2×3 Array{Complex{Float32},2}:
 -0.349649-0.638457im  0.376756-0.192146im
 ↵ -0.396334-0.0136413im
  0.611224+1.56403im   0.355204-0.365563im
 ↵  0.0905552+1.31012im
```

[source](#)

`Base.Random.randn!` – Function.

```
| randn!([rng=GLOBAL_RNG], A::AbstractArray) -> A
```

Fill the array A with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the `rand` function.

Examples

```
julia> rng = MersenneTwister(1234);
```

```
1066 julia> randn!(rng, zeros(5))
```

CHAPTER 50. NUMBERS

```
5-element Array{Float64,1}:
 0.8673472019512456
 -0.9017438158568171
 -0.4944787535042339
 -0.9029142938652416
 0.8644013132535154
```

[source](#)

[Base.Random.randexp](#) – Function.

```
| randexp([rng=GLOBAL_RNG], [T=Float64], [dims...])
```

Generate a random number of type T according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The `Base` module currently provides an implementation for the types `Float16`, `Float32`, and `Float64` (the default).

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> randexp(rng, Float32)
```

```
2.4835055f0
```

```
julia> randexp(rng, 3, 3)
```

```
3x3 Array{Float64,2}:
```

```
1.5167    1.30652   0.344435
0.604436  2.78029   0.418516
0.695867  0.693292  0.643644
```

[source](#)

[Base.Random.randexp!](#) – Function.

Fill the array A with random numbers following the exponential distribution (with scale 1).

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> randexp!(rng, zeros(5))
```

```
5-element Array{Float64,1}:
```

```
2.4835053723904896
```

```
1.516703605376473
```

```
0.6044364871025417
```

```
0.6958665886385867
```

```
1.3065196315496677
```

source

[Base.Random.randjump](#) – Function.

```
randjump(r::MersenneTwister, jumps::Integer,  
         [jumppoly::AbstractString=dSFMT.JP0LY1e21]) -> Vector{  
    MersenneTwister}
```

Create an array of the size `jumps` of initialized `MersenneTwister` RNG objects. The first RNG object given as a parameter and following `MersenneTwister` RNGs in the array are initialized such that a state of the RNG object in the array would be moved forward (without generating numbers) from a previous RNG object array element on a particular number of steps encoded by the jump polynomial `jumppoly`.

Default jump polynomial moves forward `MersenneTwister` RNG state by  $10^{20}$  steps.

source



# Chapter 51

## Strings

`Base.length` – Method.

```
| length(s::AbstractString)
```

The number of characters in string `s`.

Examples

```
| julia> length("julia")
| 5
```

`source`

`Base.sizeof` – Method.

```
| sizeof(s::AbstractString)
```

The number of bytes in string `s`.

Examples

```
| julia> sizeof("")
| 3
```

`source`

`Base.:+*` – Method.

```
1070 * (s::Union{AbstractString, Char}, t::Union{AbstractString, Char}
      | ...)
```

Concatenate strings and/or characters, producing a [String](#). This is equivalent to calling the [string](#) function on the arguments.

Examples

```
julia> "Hello " * "world"
```

```
"Hello world"
```

```
julia> 'j' * "ulia"
```

```
"julia"
```

[source](#)

[Base.^](#) – Method.

```
| ^(s::Union{AbstractString,Char}, n::Integer)
```

Repeat a string or character  $n$  times. The [repeat](#) function is an alias to this operator.

Examples

```
julia> "Test " ^ 3
```

```
"Test Test Test "
```

[source](#)

[Base.string](#) – Function.

```
| string(xs...)
```

Create a string from any values using the [print](#) function.

Examples

```
| julia> string("a", 1, true)
| "a1true"
```

`source`

`Base.repeat` – Method.

```
| repeat(s::AbstractString, r::Integer)
```

Repeat a string `r` times. This can equivalently be accomplished by calling `s^r`.

Examples

```
| julia> repeat("ha", 3)
| "hahaha"
```

`source`

`Base.repeat` – Method.

```
| repeat(c::Char, r::Integer) -> String
```

Repeat a character `r` times. This can equivalently be accomplished by calling `c^r`.

Examples

```
| julia> repeat('A', 3)
| "AAA"
```

`source`

`Base.repr` – Function.

```
| repr(x)
```

Create a string from any value using the `show` function.

Examples

```
1072 julia> repr(1)
```

```
"1"
```

```
julia> repr(zeros(3))
```

```
"[0.0, 0.0, 0.0]"
```

`source`

[Core.String](#) – Method.

```
| String(s::AbstractString)
```

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

`source`

[Base.SubString](#) – Type.

```
| SubString(s::AbstractString, i::Integer, j::Integer=endof(s))
| SubString(s::AbstractString, r::UnitRange{<:Integer})
```

Like [getindex](#), but returns a view into the parent string `s` within range `i:j` or `r` respectively instead of making a copy.

Examples

```
julia> SubString("abc", 1, 2)
```

```
"ab"
```

```
julia> SubString("abc", 1:2)
```

```
"ab"
```

```
julia> SubString("abc", 2)
```

```
"bc"
```

[Base.transcode](#) – Function.

| `transcode(T, src)`

Convert string data between Unicode encodings. `src` is either a `String` or a `Vector{UIntXX}` of UTF-XX code units, where XX is 8, 16, or 32. `T` indicates the encoding of the return value: `String` to return a (UTF-8 encoded) `String` or `UIntXX` to return a `Vector{UIntXX}` of UTF-XX data. (The alias `Cwchar_t` can also be used as the integer type, for converting `wchar_t*` strings used by external C libraries.)

The `transcode` function succeeds as long as the input data can be reasonably represented in the target encoding; it always succeeds for conversions between UTF-XX encodings, even for invalid Unicode data.

Only conversion to/from UTF-8 is currently supported.

[source](#)

[Base.unsafe\\_string](#) – Function.

| `unsafe_string(p::Ptr{UInt8}, [length::Integer])`

Copy a string from the address of a C-style (NUL-terminated) string encoded as UTF-8. (The pointer can be safely freed afterwards.) If `length` is specified (the length of the data in bytes), the string does not have to be NUL-terminated.

This function is labelled "unsafe" because it will crash if `p` is not a valid memory address to data of the requested length.

[source](#)

[Base.codeunit](#) – Method.

| `codeunit(s::AbstractString, i::Integer)`

1074 Get the *i*th code unit of an encoded string. For example, return the *i*<sup>th</sup> byte of the representation of a UTF-8 string.

Examples

```
julia> s = "δ=γ"; [codeunit(s, i) for i in 1:sizeof(s)]  
5-element Array{UInt8,1}:  
 0xce  
 0xb4  
 0x3d  
 0xce  
 0xb3
```

source

[Base.ascii](#) – Function.

```
ascii(s::AbstractString)
```

Convert a string to **String** type and check that it contains only ASCII data, otherwise throwing an **ArgumentError** indicating the position of the first non-ASCII byte.

Examples

```
julia> ascii("abcdeyfg")  
ERROR: ArgumentError: invalid ASCII at index 6 in "abcdeyfg"  
Stacktrace:  
[...]  
  
julia> ascii("abcdefg")  
"abcdefg"
```

source

[Base.@r\\_str](#) – Macro.

Construct a regex, such as `r"^[a-z]*$"`. The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

`i` enables case-insensitive matching

`m` treats the `^` and `$` tokens as matching the start and end of individual lines, as opposed to the whole string.

`s` allows the `.` modifier to match newlines.

`x` enables "comment mode": whitespace is enabled except when escaped with `\`, and `#` is treated as starting a comment.

For example, this regex has all three flags enabled:

```
| julia> match(r"a+.*b+.*?d$"ism, "Goodbye, \nOh, angry, \nBad  
|   → world\n")  
| RegexMatch("angry, \nBad world")
```

`source`

`Base.@raw_str` – Macro.

```
| @raw_str -> String
```

Create a raw string without interpolation and unescaping. The exception is that quotation marks still must be escaped. Backslashes escape both quotation marks and other backslashes, but only when a sequence of backslashes precedes a quote character. Thus,  $2n$  backslashes followed by a quote encodes  $n$  backslashes and the end of the literal while  $2n+1$  backslashes followed by a quote encodes  $n$  backslashes followed by a quote character.

Examples

```
1076 julia> println(raw"\ $x")  
\\ $x
```

## CHAPTER 51. STRINGS

```
julia> println(raw"\")  
"  
  
julia> println(raw"\\\"")  
\"  
  
julia> println(raw"\\"x \\\")  
\\x \"
```

**source**

`Base.Docs.@html_str` – Macro.

```
| @html_str -> Docs.HTML
```

Create an `HTML` object from a literal string.

**source**

`Base.Docs.@text_str` – Macro.

```
| @text_str -> Docs.Text
```

Create a `Text` object from a literal string.

**source**

`Base.UTF8proc.normalize_string` – Function.

```
| normalize_string(s::AbstractString, normalform::Symbol)
```

Normalize the string `s` according to one of the four “normal forms” of the Unicode standard: `normalform` can be `:NFC`, `:NFD`, `:NFKC`, or `:NFKD`. Normal forms C (canonical composition) and D (canonical decomposition)

convert different visually identical representations of the same abstract string into a single canonical form, with form C being more compact. Normal forms KC and KD additionally canonicalize “compatibility equivalents”: they convert characters that are abstractly similar but visually distinct into a single canonical choice (e.g. they expand ligatures into the individual characters), with form KC being more compact.

Alternatively, finer control and additional transformations may be obtained by calling `normalize_string(s; keywords...)`, where any number of the following boolean keywords options (which all default to `false` except for `compose`) are specified:

`compose=false`: do not perform canonical composition

`decompose=true`: do canonical decomposition instead of canonical composition (`compose=true` is ignored if present)

`compat=true`: compatibility equivalents are canonicalized

`casefold=true`: perform Unicode case folding, e.g. for case-insensitive string comparison

`newline2lf=true`, `newline2ls=true`, or `newline2ps=true`: convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively

`stripmark=true`: strip diacritical marks (e.g. accents)

`stripignore=true`: strip Unicode’s “default ignorable” characters (e.g. the soft hyphen or the left-to-right marker)

`stripcc=true`: strip control characters; horizontal tabs and form feeds are converted to spaces; newlines are also converted to spaces unless a newline-conversion flag was specified

`rejectna=true`: throw an error if unassigned code points are found

For example, NFKC corresponds to the options `compose=true`, `compat=true`, `stable=true`.

Examples

```
julia> "μ" == normalize_string("μ", compat=true) #LHS: Unicode
           ↳ U+03bc, RHS: Unicode U+00b5
true

julia> normalize_string("JuLiA", casefold=true)
"julia"

julia> normalize_string("JúLiA", stripmark=true)
"JuLiA"
```

`source`

[Base.UTF8proc.graphemes](#) – Function.

```
| graphemes(s::AbstractString) -> GraphemeIterator
```

Returns an iterator over substrings of `s` that correspond to the extended graphemes in the string, as defined by Unicode UAX #29. (Roughly, these are what users would perceive as single characters, even though they may contain more than one codepoint; for example a letter combined with an accent mark is a single grapheme.)

`source`

[Base.isvalid](#) – Method.

```
| isvalid(value) -> Bool
```

Returns `true` if the given value is valid for its type, which currently can be either `Char` or `String`.

```
| julia> isvalid(Char(0xd800))  
| false
```

```
| julia> isvalid(Char(0xd799))  
| true
```

source

[Base.isvalid](#) – Method.

```
| isvalid(T, value) -> Bool
```

Returns `true` if the given value is valid for that type. Types currently can be either `Char` or `String`. Values for `Char` can be of type `Char` or `UInt32`. Values for `String` can be of that type, or `Vector{UInt8}`.

Examples

```
| julia> isvalid(Char, 0xd800)  
| false
```

```
| julia> isvalid(Char, 0xd799)  
| true
```

source

[Base.isvalid](#) – Method.

```
| isvalid(str::AbstractString, i::Integer)
```

Tell whether index `i` is valid for the given string.

Examples

```
| julia> str = "αβγδεƒ";
```

```
1080 julia> isvalid(str, 1)
```

```
true
```

```
julia> str[1]
```

```
'a': Unicode U+03b1 (category Ll: Letter, lowercase)
```

```
julia> isvalid(str, 2)
```

```
false
```

```
julia> str[2]
```

```
ERROR: UnicodeError: invalid character index
```

```
Stacktrace:
```

```
[...]
```

`source`

[Base.UTF8proc.is\\_assigned\\_char](#) – Function.

```
| is_assigned_char(c) -> Bool
```

Returns `true` if the given char or integer is an assigned Unicode code point.

Examples

```
julia> is_assigned_char(101)
```

```
true
```

```
julia> is_assigned_char('\'x01')
```

```
true
```

`source`

[Baseismatch](#) – Function.

```
| ismatch(r::Regex, s::AbstractString) -> Bool
```

Test whether a string contains a match of the given regular expression.<sup>1081</sup>

Examples

```
julia> rx = r"a.a"  
r"a.a"  
  
julia> ismatch(rx, "aba")  
true  
  
julia> ismatch(rx, "abba")  
false  
  
julia> rx("aba")  
true
```

source

`Base.match` – Function.

```
| match(r::Regex, s::AbstractString[, idx::Integer[, addopts]])
```

Search for the first match of the regular expression `r` in `s` and return a `RegexMatch` object containing the match, or nothing if the match failed. The matching substring can be retrieved by accessing `m.match` and the captured sequences can be retrieved by accessing `m.captures`. The optional `idx` argument specifies an index at which to start the search.

Examples

```
julia> rx = r"a(.)a"  
r"a(.)a"  
  
julia> m = match(rx, "cabac")  
RegexMatch("aba", 1="b")
```

```
julia> m.captures
1-element Array{Union{Void, SubString{String}},1}:
"b"

julia> m.match
"aba"

julia> match(rx, "cabac", 3) == nothing
true
```

**source**

[Base.eachmatch](#) – Function.

```
| eachmatch(r::Regex, s::AbstractString[, overlap::Bool=false])
```

Search for all matches of a the regular expression `r` in `s` and return a iterator over the matches. If `overlap` is `true`, the matching sequences are allowed to overlap indices in the original string, otherwise they must be from distinct character ranges.

**Examples**

```
julia> rx = r"a.a"
r"a.a"

julia> m = eachmatch(rx, "a1a2a3a")
Base.RegexMatchIterator(r"a.a", "a1a2a3a", false)

julia> collect(m)
2-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a3a")
```

```
julia> collect(eachmatch(rx, "a1a2a3a", true))
3-element Array{RegexMatch,1}:
 RegexMatch("a1a")
 RegexMatch("a2a")
 RegexMatch("a3a")
```

**source**

[Base.matchall](#) – Function.

```
matchall(r::Regex, s::AbstractString[, overlap::Bool=false]) ->
    Vector{AbstractString}
```

Return a vector of the matching substrings from [eachmatch](#).

Examples

```
julia> rx = r"a.a"
r"a.a"
```

```
julia> matchall(rx, "a1a2a3a")
2-element Array{SubString{String},1}:
 "a1a"
 "a3a"
```

```
julia> matchall(rx, "a1a2a3a", true)
3-element Array{SubString{String},1}:
 "a1a"
 "a2a"
 "a3a"
```

**source**

[Base.isless](#) – Method.

1084 `isless(a::AbstractString, b::AbstractString)` CHAPTER 51. STRINGS

Test whether string `a` comes before string `b` in alphabetical order.

Examples

```
julia> isless("a", "b")
```

```
true
```

```
julia> isless("β", "α")
```

```
false
```

```
julia> isless("a", "a")
```

```
false
```

source

`Base.==` – Method.

```
==(a::AbstractString, b::AbstractString)
```

Test whether two strings are equal character by character.

Examples

```
julia> "abc" == "abc"
```

```
true
```

```
julia> "abc" == "αβγ"
```

```
false
```

source

`Base.cmp` – Method.

```
cmp(a::AbstractString, b::AbstractString)
```

Compare two strings for equality.

1085

Return 0 if both strings have the same length and the character at each index is the same in both strings. Return -1 if a is a substring of b, or if a comes before b in alphabetical order. Return 1 if b is a substring of a, or if b comes before a in alphabetical order.

Examples

```
julia> cmp("abc", "abc")
```

```
0
```

```
julia> cmp("ab", "abc")
```

```
-1
```

```
julia> cmp("abc", "ab")
```

```
1
```

```
julia> cmp("ab", "ac")
```

```
-1
```

```
julia> cmp("ac", "ab")
```

```
1
```

```
julia> cmp("a", "a")
```

```
1
```

```
julia> cmp("b", "β")
```

```
-1
```

source

[Base.lpad](#) – Function.

```
| lpad(s, n::Integer, p::AbstractString=" ")
```

1080 Make a string at least n columns wide when printed by padding s on the left with copies of p.

Examples

```
julia> lpad("March", 10)  
"      March"
```

source

[Base.rpad](#) – Function.

```
rpad(s, n::Integer, p::AbstractString=" ")
```

Make a string at least n columns wide when printed by padding s on the right with copies of p.

Examples

```
julia> rpad("March", 20)  
"March"           "
```

source

[Base.search](#) – Function.

```
search(string::AbstractString, chars::Chars, [start::Integer])
```

Search for the first occurrence of the given characters within the given string. The second argument may be a single character, a vector or a set of characters, a string, or a regular expression (though regular expressions are only allowed on contiguous strings, such as ASCII or UTF-8 strings).

The third argument optionally specifies a starting index. The return value is a range of indexes where the matching sequence is found, such that `s[search(s, x)] == x`:

`search(string, "substring") = start:end` such that `string[start:end] == "substring"`, or `0:-1` if unmatched.

`search(string, 'c')` = index such that `string[index] == 'c'`  
0 if unmatched.

Examples

```
julia> search("Hello to the world", "z")
0:-1
```

```
julia> search("JuliaLang", "Julia")
1:5
```

source

`Base.rsearch` – Function.

```
rsearch(s::AbstractString, chars::Chars, [start::Integer])
```

Similar to `search`, but returning the last occurrence of the given characters within the given string, searching in reverse from `start`.

Examples

```
julia> rsearch("aaabbb", "b")
6:6
```

source

`Base.searchindex` – Function.

```
searchindex(s::AbstractString, substring, [start::Integer])
```

Similar to `search`, but return only the start index at which the substring is found, or 0 if it is not.

Examples

```
julia> searchindex("Hello to the world", "z")
0
```

```
julia> searchindex("JuliaLang", "Julia")
```

```
1
```

```
julia> searchindex("JuliaLang", "Lang")
```

```
6
```

source

[Base.rsearchindex](#) – Function.

```
| rsearchindex(s::AbstractString, substring, [start::Integer])
```

Similar to [rsearch](#), but return only the start index at which the substring is found, or `0` if it is not.

Examples

```
julia> rsearchindex("aaabbb", "b")
```

```
6
```

```
julia> rsearchindex("aaabbb", "a")
```

```
3
```

source

[Base.contains](#) – Method.

```
| contains(haystack::AbstractString, needle::Union{AbstractString  
          ,Char})
```

Determine whether the second argument is a substring of the first.

Examples

```
julia> contains("JuliaLang is pretty cool!", "Julia")
```

```
true
```

[Base.reverse](#) – Method.

```
| reverse(s::AbstractString) -> AbstractString
```

Reverses a string.

Technically, this function reverses the codepoints in a string, and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also [reverseind](#) to convert indices in `s` to indices in `reverse(s)` and vice-versa, and [graphemes](#) to operate on user-visible "characters" (graphemes) rather than codepoints. See also [Iterators.reverse](#) for reverse-order iteration without making a copy.

Examples

```
julia> reverse("JuliaLang")
"naLailuJ"
```

```
julia> reverse("axe") # combining characters can lead to
    → surprising results
"exā"
```

```
julia> join(reverse(collect(graphemes("axe")))) # reverses
    → graphemes
"exā"
```

source

[Base.replace](#) – Function.

```
| replace(s::AbstractString, pat, r, [count::Integer])
```

Search for the given pattern `pat` in `s`, and replace each occurrence with `r`. If `count` is provided, replace at most `count` occurrences. As with

1090 **search**, the second argument may be a single character, a set of characters, a string, or a regular expression. If **r** is a function, each occurrence is replaced with **r(s)** where **s** is the matched substring. If **pat** is a regular expression and **r** is a **SubstitutionString**, then capture group references in **r** are replaced with the corresponding matched text. To remove instances of **pat** from **string**, set **r** to the empty **String ("")**.

Examples

```
julia> replace("Python is a programming language.", "Python",
    ↪   "Julia")
"Julia is a programming language."
```

```
julia> replace("The quick foxes run quickly.", "quick",
    ↪   "slow", 1)
"The slow foxes run quickly."
```

```
julia> replace("The quick foxes run quickly.", "quick", "", 1)
"The foxes run quickly."
```

source

**Base.split** – Function.

```
split(s::AbstractString, [chars]; limit::Integer=0, keep::Bool=
    true)
```

Return an array of substrings by splitting the given string on occurrences of the given character delimiters, which may be specified in any of the formats allowed by **search**'s second argument (i.e. a single character, collection of characters, string, or regular expression). If **chars** is omitted, it defaults to the set of all space characters, and **keep** is taken to be **false**. The two keyword arguments are optional: they are a maximum

size for the result and a flag determining whether empty fields should be kept in the result.

Examples

```
julia> a = "Ma.rch"
"Ma.rch"

julia> split(a, ".")
2-element Array{SubString{String},1}:
 "Ma"
 "rch"
```

source

```
split(ce::LibGit2.ConfigEntry) -> Tuple{String, String, String,
String}
```

Break the `ConfigEntry` up to the following pieces: section, subsection, name, and value.

Examples

Given the git configuration file containing:

```
[credential "https://example.com"]
username = me
```

The `ConfigEntry` would look like the following:

```
julia> entry
ConfigEntry("credential.https://example.com.username", "me")

julia> split(entry)
("credential", "https://example.com", "username", "me")
```

Refer to the [git config syntax documentation](#) for more details.

Base.`rsplit` – Function.

```
rsplit(s::AbstractString, [chars]; limit::Integer=0, keep::Bool  
      =true)
```

Similar to `split`, but starting from the end of the string.

Examples

```
julia> a = "M.a.r.c.h"  
"M.a.r.c.h"  
  
julia> rsplit(a, ".")  
5-element Array{SubString{String},1}:  
"M"  
"a"  
"r"  
"c"  
"h"  
  
julia> rsplit(a, "."; limit=1)  
1-element Array{SubString{String},1}:  
"M.a.r.c.h"  
  
julia> rsplit(a, "."; limit=2)  
2-element Array{SubString{String},1}:  
"M.a.r.c"  
"h"
```

source

Base.`strip` – Function.

```
strip(s::AbstractString, [chars::Chars])
```

Return `s` with any leading and trailing whitespace removed. If `chars` (a character, or vector or set of characters) is provided, instead remove characters contained in it.

Examples

```
| julia> strip("{3, 5}\n", ['{', '}', '\n'])  
"3, 5"
```

`source`

[Base.`lstrip`](#) – Function.

```
| lstrip(s::AbstractString[, chars::Chars])
```

Return `s` with any leading whitespace and delimiters removed. The default delimiters to remove are ' ', \t, \n, \v, \f, and \r. If `chars` (a character, or vector or set of characters) is provided, instead remove characters contained in it.

Examples

```
| julia> a = lpad("March", 20)  
"           March"
```

```
| julia> lstrip(a)  
"March"
```

`source`

[Base.`rstrip`](#) – Function.

```
| rstrip(s::AbstractString[, chars::Chars])
```

Return `s` with any trailing whitespace and delimiters removed. The default delimiters to remove are ' ', \t, \n, \v, \f, and \r. If `chars` (a character,

1094 or vector or set of characters) is provided, instead of the characters contained in it.

Examples

```
julia> a = rpad("March", 20)
"March"           "
```

```
julia> rstrip(a)
"March"
```

[source](#)

[Base.startswith](#) – Function.

```
startswith(s::AbstractString, prefix::AbstractString)
```

Returns `true` if `s` starts with `prefix`. If `prefix` is a vector or set of characters, tests whether the first character of `s` belongs to that set.

See also [endswith](#).

Examples

```
julia> startswith("JuliaLang", "Julia")
true
```

[source](#)

[Base.endswith](#) – Function.

```
endswith(s::AbstractString, suffix::AbstractString)
```

Returns `true` if `s` ends with `suffix`. If `suffix` is a vector or set of characters, tests whether the last character of `s` belongs to that set.

See also [startswith](#).

Examples

```
| julia> endswith("Sunday", "day")
| true
```

**source**

**Base.first** – Method.

```
| first(str::AbstractString, nchar::Integer)
```

Get a string consisting of the first `nchar` characters of `str`.

```
| julia> first("≠0: ²>0", 0)
```

""

```
| julia> first("≠0: ²>0", 1)
```

""

```
| julia> first("≠0: ²>0", 3)
```

"≠"

**source**

**Base.last** – Method.

```
| last(str::AbstractString, nchar::Integer)
```

Get a string consisting of the last `nchar` characters of `str`.

```
| julia> last("≠0: ²>0", 0)
```

""

```
| julia> last("≠0: ²>0", 1)
```

"0"

```
| julia> last("≠0: ²>0", 3)
```

"²>0"

`Base.uppercase` – Function.

```
| uppercase(s::AbstractString)
```

Return `s` with all characters converted to uppercase.

Examples

```
| julia> uppercase("Julia")
| "JULIA"
```

`source`

`Base.lowercase` – Function.

```
| lowercase(s::AbstractString)
```

Return `s` with all characters converted to lowercase.

Examples

```
| julia> lowercase("STRINGS AND THINGS")
| "strings and things"
```

`source`

`Base.titlecase` – Function.

```
| titlecase(s::AbstractString)
```

Capitalize the first character of each word in `s`. See also `ucfirst` to capitalize only the first character in `s`.

Examples

```
| julia> titlecase("the julia programming language")
| "The Julia Programming Language"
```

`source`

```
| ucfirst(s::AbstractString)
```

Return `string` with the first character converted to uppercase (technically "title case" for Unicode). See also [titlecase](#) to capitalize the first character of every word in `s`.

Examples

```
| julia> ucfirst("python")
| "Python"
```

[source](#)

```
| lcfirst(s::AbstractString)
```

Return `string` with the first character converted to lowercase.

Examples

```
| julia> lcfirst("Julia")
| "julia"
```

[source](#)

```
| join(io::IO, strings, delim, [last])
```

Join an array of `strings` into a single string, inserting the given delimiter between adjacent strings. If `last` is given, it will be used instead of `delim` between the last two strings. For example,

Examples

1098 **julia>** join(["apples", "bananas", "pineapples"],  
| ↵ ")  
| "apples, bananas and pineapples"

`strings` can be any iterable over elements `x` which are convertible to strings via `print(io::IOBuffer, x)`. `strings` will be printed to `io`.

**source**

`Base.chop` – Function.

| `chop(s::AbstractString, head::Integer=0, tail::Integer=1)`

Remove the first `head` and the last `tail` characters from `s`. The call `chop(s)` removes the last character from `s`. If it is requested to remove more characters than `length(s)` then an empty string is returned.

Examples

**julia>** `a = "March"`

"March"

**julia>** `chop(a)`

"Marc"

**julia>** `chop(a, 1, 2)`

"ar"

**julia>** `chop(a, 5, 5)`

""

**source**

`Base.chomp` – Function.

| `chomp(s::AbstractString)`

Remove a single trailing newline from a string.

1099

Examples

```
| julia> chomp("Hello\n")
| "Hello"
```

source

[Base.ind2chr](#) – Function.

```
| ind2chr(s::AbstractString, i::Integer)
```

Convert a byte index **i** to a character index with respect to string **s**.

See also [chr2ind](#).

Examples

```
| julia> str = "αβγδεֆ";
```

```
| julia> ind2chr(str, 3)
```

```
| 2
```

```
| julia> chr2ind(str, 2)
```

```
| 3
```

source

[Base.chr2ind](#) – Function.

```
| chr2ind(s::AbstractString, i::Integer)
```

Convert a character index **i** to a byte index.

See also [ind2chr](#).

Examples

```
1100 julia> str = "αβγδεֆ";
```

CHAPTER 51. STRINGS

```
julia> chr2ind(str, 2)
```

```
3
```

```
julia> ind2chr(str, 3)
```

```
2
```

source

`Base.nextind` – Function.

```
| nextind(str::AbstractString, i::Integer, nchar::Integer=1)
```

Get the next valid string index after `i`. Returns a value greater than `endof(str)` at or after the end of the string. If the `nchar` argument is given the function goes forward `nchar` characters.

Examples

```
julia> str = "αβγδεֆ";
```

```
julia> nextind(str, 1)
```

```
3
```

```
julia> nextind(str, 1, 2)
```

```
5
```

```
julia> endof(str)
```

```
9
```

```
julia> nextind(str, 9)
```

```
10
```

source

```
| prevind(str::AbstractString, i::Integer, nchar::Integer=1)
```

Get the previous valid string index before `i`. Returns a value less than 1 at the beginning of the string. If the `nchar` argument is given the function goes back `nchar` characters.

Examples

```
| julia> prevind("aβγdef", 3)
```

```
| 1
```

```
| julia> prevind("aβγdef", 1)
```

```
| 0
```

```
| julia> prevind("aβγdef", 3, 2)
```

```
| 0
```

source

```
| randstring([rng=GLOBAL_RNG], [chars], [len=8])
```

Create a random string of length `len`, consisting of characters from `chars`, which defaults to the set of upper- and lower-case letters and the digits 0-9. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Examples

```
| julia> srand(0); randstring()
```

```
| "c03rgKi1"
```

```
| julia> randstring(MersenneTwister(0), 'a':'z', 6)
```

```
julia> randstring("ACGT")
"TATCGGTC"
```

Note

`chars` can be any collection of characters, of type `Char` or `UInt8` (more efficient), provided `rand` can randomly pick characters from it.

`source`

`Base.UTF8proc.textwidth` – Function.

```
| textwidth(c)
```

Give the number of columns needed to print a character.

Examples

```
julia> textwidth('a')
```

```
1
```

```
julia> textwidth(' ')
```

```
2
```

`source`

```
| textwidth(s::AbstractString)
```

Give the number of columns needed to print a string.

Examples

```
julia> textwidth("March")
```

```
5
```

`source`

```
| isalnum(c::Char) -> Bool
```

Tests whether a character is alphanumeric. A character is classified as alphabetic if it belongs to the Unicode general category Letter or Number, i.e. a character whose category code begins with 'L' or 'N'.

Examples

```
julia> isalnum('')
```

```
false
```

```
julia> isalnum('9')
```

```
true
```

```
julia> isalnum('a')
```

```
true
```

source

```
| isalpha(c::Char) -> Bool
```

Tests whether a character is alphabetic. A character is classified as alphabetic if it belongs to the Unicode general category Letter, i.e. a character whose category code begins with 'L'.

Examples

```
julia> isalpha('')
```

```
false
```

```
julia> isalpha('a')
```

```
true
```

```
julia> isalpha('9')
false
```

[source](#)

[Base.isascii](#) – Function.

```
| isascii(c::Union{Char,AbstractString}) -> Bool
```

Test whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

Examples

```
julia> isascii('a')
true
```

```
julia> isascii('a')
false
```

```
julia> isascii("abc")
true
```

```
julia> isascii("αβγ")
false
```

[source](#)

[Base.UTF8proc.iscntrl](#) – Function.

```
| iscntrl(c::Char) -> Bool
```

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

Examples

```
julia> iscntrl('\x01')
```

true

```
julia> iscntrl('a')
```

false

### source

[Base.UTF8proc.isdigit](#) – Function.

```
| isdigit(c::Char) -> Bool
```

Tests whether a character is a numeric digit (0–9).

### Examples

```
julia> isdigit('')
```

false

```
julia> isdigit('9')
```

true

```
julia> isdigit('a')
```

false

### source

[Base.UTF8proc.isgraph](#) – Function.

```
| isgraph(c::Char) -> Bool
```

Tests whether a character is printable, and not a space. Any character that would cause a printer to use ink should be classified with `isgraph(c)==true`.

### Examples

```
1106 julia> isgraph('\x01')
```

```
false
```

```
julia> isgraph('A')
```

```
true
```

[source](#)

`Base.UTF8proc.islower` – Function.

```
| islower(c::Char) -> Bool
```

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category LI, Letter: Lowercase.

Examples

```
julia> islower('a')
```

```
true
```

```
julia> islower('Γ')
```

```
false
```

```
julia> islower('')
```

```
false
```

[source](#)

`Base.UTF8proc.isnumber` – Function.

```
| isnumber(c::Char) -> Bool
```

Tests whether a character is numeric. A character is classified as numeric if it belongs to the Unicode general category Number, i.e. a character whose category code begins with 'N'.

Examples

```
julia> isnumber('9')
```

true

```
julia> isnumber('a')
```

false

```
julia> isnumber(' ')
```

false

### source

[Base.UTF8proc.isprint](#) – Function.

```
| isprint(c::Char) -> Bool
```

Tests whether a character is printable, including spaces, but not a control character.

### Examples

```
julia> isprint('\x01')
```

false

```
julia> isprint('A')
```

true

### source

[Base.UTF8proc.ispunct](#) – Function.

```
| ispunct(c::Char) -> Bool
```

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

### Examples

```
1108 julia> ispunct('a')
```

```
false
```

```
julia> ispunct('/')
```

```
true
```

```
julia> ispunct(';')
```

```
true
```

source

[Base.UTF8proc.isspace](#) – Function.

```
| isspace(c::Char) -> Bool
```

Tests whether a character is any whitespace character. Includes ASCII characters 'Wt', 'Wn', 'Wv', 'Wf', 'Wr', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

Examples

```
julia> isspace('\n')
```

```
true
```

```
julia> isspace('\r')
```

```
true
```

```
julia> isspace(' ')
```

```
true
```

```
julia> isspace('\x20')
```

```
true
```

source

```
| isupper(c::Char) -> Bool
```

Tests whether a character is an uppercase letter. A character is classified as uppercase if it belongs to Unicode category Lu, Letter: Uppercase, or Lt, Letter: Titlecase.

Examples

```
julia> isupper('y')
```

```
false
```

```
julia> isupper('Γ')
```

```
true
```

```
julia> isupper('')
```

```
false
```

source

```
| isxdigit(c::Char) -> Bool
```

Test whether a character is a valid hexadecimal digit. Note that this does not include x (as in the standard 0x prefix).

Examples

```
julia> isxdigit('a')
```

```
true
```

```
julia> isxdigit('x')
```

```
false
```

source

## `Core.Symbol` – Type.

## CHAPTER 51. STRINGS

```
| Symbol(x...) -> Symbol
```

Create a `Symbol` by concatenating the string representations of the arguments together.

Examples

```
julia> Symbol("my", "name")
```

```
:myname
```

```
julia> Symbol("day", 4)
```

```
:day4
```

`source`

## `Base.escape_string` – Function.

```
| escape_string([io,] str::AbstractString[, esc::AbstractString])  
|     -> AbstractString
```

General escaping of traditional C and Unicode escape sequences. Any characters in `esc` are also escaped (with a backslash). See also [unescape\\_string](#).

`source`

## `Base.unescape_string` – Function.

```
| unescape_string([io,] s::AbstractString) -> AbstractString
```

General unescaping of traditional C and Unicode escape sequences. Reverse of [escape\\_string](#).

`source`

# Chapter 52

## Arrays

### 52.1 Constructors and Types

[Core.AbstractArray](#) – Type.

| `AbstractArray{T,N}`

Supertype for  $N$ -dimensional arrays (or array-like types) with elements of type  $T$ . [Array](#) and other types are subtypes of this. See the manual section on the [AbstractArray interface](#).

[source](#)

[Base.AbstractVector](#) – Type.

| `AbstractVector{T}`

Supertype for one-dimensional arrays (or array-like types) with elements of type  $T$ . Alias for [AbstractArray{T, 1}](#).

[source](#)

[Base.AbstractMatrix](#) – Type.

| `AbstractMatrix{T}`

Supertype for two-dimensional arrays (or array-like types) with elements of type  $T$ . Alias for [AbstractArray{T, 2}](#).

`Core.Array` – Type.

```
| Array{T,N} <: AbstractArray{T,N}
```

$N$ -dimensional dense array with elements of type  $T$ .

`source`

`Core.Array` – Method.

```
| Array{T}(dims)
| Array{T,N}(dims)
```

Construct an uninitialized  $N$ -dimensional `Array` containing elements of type  $T$ .  $N$  can either be supplied explicitly, as in `Array{T,N}(dims)`, or be determined by the length or number of `dims`. `dims` may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank  $N$  is supplied explicitly, then it must match the length or number of `dims`.

Examples

```
julia> A = Array{Float64,2}(2, 3) # N given explicitly
```

```
2×3 Array{Float64,2}:
 6.90198e-310 6.90198e-310 6.90198e-310
 6.90198e-310 6.90198e-310 0.0
```

```
julia> B = Array{Float64}(2) # N determined by the input
```

```
2-element Array{Float64,1}:
 1.87103e-320
 0.0
```

`source`

`Core.Uninitialized` – Type.

Singleton type used in array initialization, indicating the array-constructor-caller would like an uninitialized array. See also [uninitialized](#), an alias for `Uninitialized()`.

Examples

```
julia> Array{Float64,1}(Uninitialized(), 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

[source](#)

[Core.uninitialized](#) – Constant.

```
| uninitialized
```

Alias for `Uninitialized()`, which constructs an instance of the singleton type [Uninitialized](#), used in array initialization to indicate the array-constructor-caller would like an uninitialized array.

Examples

```
julia> Array{Float64,1}(uninitialized, 3)
3-element Array{Float64,1}:
 2.2752528595e-314
 2.202942107e-314
 2.275252907e-314
```

[source](#)

[Base.Vector](#) – Type.

```
| Vector{T} <: AbstractVector{T}
```

1114 One-dimensional dense array with elements of type `T`.  
CHAPTER 52 ARRAYS  
represent a mathematical vector. Alias for `Array{T, 1}`.

`source`

`Base.Vector` – Method.

`| Vector{T}(n)`

Construct an uninitialized `Vector{T}` of length `n`.

Examples

```
julia> Vector{Float64}(3)
3-element Array{Float64, 1}:
 6.90966e-310
 6.90966e-310
 6.90966e-310
```

`source`

`Base.Matrix` – Type.

`| Matrix{T} <: AbstractMatrix{T}`

Two-dimensional dense array with elements of type `T`, often used to represent a mathematical matrix. Alias for `Array{T, 2}`.

`source`

`Base.Matrix` – Method.

`| Matrix{T}(m, n)`

Construct an uninitialized `Matrix{T}` of size  $m \times n$ .

Examples

```
julia> Matrix{Float64}(2, 3)
2×3 Array{Float64, 2}:
```

52.1 CONSTRUCTORS AND TYPES  
6.93517e-310 6.93517e-310 6.93517e-310  
6.93517e-310 6.93517e-310 1.29396e-320

1115

**source**

[Base.getindex](#) – Method.

```
| getindex(type[, elements...])
```

Construct a 1-d array of the specified type. This is usually called with the syntax `Type[]`. Element values can be specified using `Type[a, b, c, ...]`.

Examples

```
julia> Int8[1, 2, 3]
3-element Array{Int8,1}:
 1
 2
 3
```

```
julia> getindex(Int8, 1, 2, 3)
3-element Array{Int8,1}:
 1
 2
 3
```

**source**

[Base.zeros](#) – Function.

```
zeros([A::AbstractArray,] [T=eltype(A)::Type,] [dims=size(A)::Tuple])
```

Create an array of all zeros with the same layout as `A`, element type `T` and size `dims`. The `A` argument can be skipped, which behaves like `Array{Float64,0}()` was passed. For convenience `dims` may also be passed in variadic form.

```
julia> zeros(1)  
1-element Array{Float64,1}:  
 0.0
```

```
julia> zeros(Int8, 2, 3)  
2×3 Array{Int8,2}:  
 0 0 0  
 0 0 0
```

```
julia> A = [1 2; 3 4]  
2×2 Array{Int64,2}:  
 1 2  
 3 4
```

```
julia> zeros(A)  
2×2 Array{Int64,2}:  
 0 0  
 0 0
```

```
julia> zeros(A, Float64)  
2×2 Array{Float64,2}:  
 0.0 0.0  
 0.0 0.0
```

```
julia> zeros(A, Bool, (3,))  
3-element Array{Bool,1}:  
 false  
 false  
 false
```

See also [ones](#), [similar](#).

`Base.ones` – Function.

```
ones([A::AbstractArray,] [T=eltype(A)::Type,] [dims=size(A)::  
Tuple])
```

Create an array of all ones with the same layout as A, element type T and size dims. The A argument can be skipped, which behaves like `Array{Float64,0}()` was passed. For convenience dims may also be passed in variadic form.

Examples

```
julia> ones(Complex128, 2, 3)  
2×3 Array{Complex{Float64},2}:  
 1.0+0.0im 1.0+0.0im 1.0+0.0im  
 1.0+0.0im 1.0+0.0im 1.0+0.0im
```

```
julia> ones(1,2)  
1×2 Array{Float64,2}:  
 1.0 1.0
```

```
julia> A = [1 2; 3 4]  
2×2 Array{Int64,2}:  
 1 2  
 3 4
```

```
julia> ones(A)  
2×2 Array{Int64,2}:  
 1 1  
 1 1
```

```
julia> ones(A, Float64)
```

```
1118 2×2 Array{Float64, 2}:
```

```
 1.0  1.0  
 1.0  1.0
```

## CHAPTER 52. ARRAYS

```
julia> ones(A, Bool, (3,))
```

```
3-element Array{Bool, 1}:
```

```
true  
true  
true
```

See also [zeros](#), [similar](#).

[source](#)

[Base.BitArray](#) – Type.

```
| BitArray{N} <: DenseArray{Bool, N}
```

Space-efficient N-dimensional boolean array, which stores one bit per boolean value.

[source](#)

[Base.BitArray](#) – Method.

```
| BitArray(dims::Integer...)  
| BitArray{N}(dims::NTuple{N, Int})
```

Construct an uninitialized [BitArray](#) with the given dimensions. Behaves identically to the [Array](#) constructor.

Examples

```
julia> BitArray(2, 2)
```

```
2×2 BitArray{2}:
```

```
false  false  
false  true
```

```
julia> BitArray((3, 1))  
3×1 BitArray{2}:  
 false  
 true  
 false
```

source

[Base.BitArray](#) – Method.

```
| BitArray(itr)
```

Construct a [BitArray](#) generated by the given iterable object. The shape is inferred from the `itr` object.

Examples

```
julia> BitArray([1 0; 0 1])  
2×2 BitArray{2}:  
 true  false  
 false  true  
  
julia> BitArray(x+y == 3 for x = 1:2, y = 1:3)  
2×3 BitArray{2}:  
 false  true  false  
 true  false  false  
  
julia> BitArray(x+y == 3 for x = 1:2 for y = 1:3)  
6-element BitArray{1}:  
 false  
 true  
 false  
 true
```

```
1120 false  
      false
```

## CHAPTER 52. ARRAYS

**source**

**Base.trues** – Function.

```
| trues(dims)
```

Create a **BitArray** with all values set to **true**.

Examples

```
julia> trues(2,3)  
2×3 BitArray{2}:  
  true  true  true  
  true  true  true
```

**source**

```
| trues(A)
```

Create a **BitArray** with all values set to **true** of the same shape as A.

Examples

```
julia> A = [1 2; 3 4]  
2×2 Array{Int64,2}:  
 1  2  
 3  4
```

```
julia> trues(A)  
2×2 BitArray{2}:  
  true  true  
  true  true
```

**source**

```
| falses(dims)
```

Create a `BitArray` with all values set to `false`.

Examples

```
| julia> falses(2,3)
2×3 BitArray{2}:
 false  false  false
 false  false  false
```

source

```
| falses(A)
```

Create a `BitArray` with all values set to `false` of the same shape as A.

Examples

```
| julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
| julia> falses(A)
2×2 BitArray{2}:
 false  false
 false  false
```

source

[Base.fill](#) – Function.

```
| fill(x, dims)
```

Create an array filled with the value x. For example, `fill(1.0, (5,5))` returns a  $5 \times 5$  array of floats, with each element initialized to 1.0.

```
julia> fill(1.0, (5,5))  
5×5 Array{Float64,2}:  
1.0 1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0 1.0
```

If `x` is an object reference, all elements will refer to the same object. `fill(Foo(), dims)` will return an array filled with the result of evaluating `Foo()` once.

#### source

[Base.fill!](#) – Function.

```
| fill!(A, x)
```

Fill array `A` with the value `x`. If `x` is an object reference, all elements will refer to the same object. `fill!(A, Foo())` will return `A` filled with the result of evaluating `Foo()` once.

#### Examples

```
julia> A = zeros(2,3)  
2×3 Array{Float64,2}:  
0.0 0.0 0.0  
0.0 0.0 0.0  
  
julia> fill!(A, 2.)  
2×3 Array{Float64,2}:  
2.0 2.0 2.0  
2.0 2.0 2.0
```

```
julia> a = [1, 1, 1]; A = fill!(Vector{Vector{Int}}(3), a);
         ↵ a[1] = 2; A
3-element Array{Array{Int64,1},1}:
 [2, 1, 1]
 [2, 1, 1]
 [2, 1, 1]

julia> x = 0; f() = (global x += 1; x); fill!(Vector{Int}(3),
         ↵ f())
3-element Array{Int64,1}:
 1
 1
 1
```

### source

`Base.similar` – Method.

```
similar(array, [element_type=eltype(array)], [dims=size(array)
      ])
```

Create an uninitialized mutable array with the given element type and size, based upon the given source array. The second and third arguments are both optional, defaulting to the given array's `eltype` and `size`. The dimensions may be specified either as a single tuple argument or as a series of integer arguments.

Custom `AbstractArray` subtypes may choose which specific array type is best-suited to return for the given element type and dimensionality. If they do not specialize this method, the default is an `Array{element_type}(dims...)`.

112 For example, `similar(1:10, 1, 4)` returns an uninitialized `Array{Int64,2}` since ranges are neither mutable nor support 2 dimensions:

```
julia> similar(1:10, 1, 4)
1×4 Array{Int64,2}:
 4419743872 4374413872 4419743888 0
```

Conversely, `similar(trues(10,10), 2)` returns an uninitialized `BitVector` with two elements since `BitArrays` are both mutable and can support 1-dimensional arrays:

```
julia> similar(trues(10,10), 2)
2-element BitArray{1}:
 false
 false
```

Since `BitArrays` can only store elements of type `Bool`, however, if you request a different element type it will create a regular `Array` instead:

```
julia> similar(falses(10), Float64, 2, 4)
2×4 Array{Float64,2}:
 2.18425e-314 2.18425e-314 2.18425e-314 2.18425e-314
 2.18425e-314 2.18425e-314 2.18425e-314 2.18425e-314
```

`source`

`Base.similar` – Method.

```
similar(storagetype, indices)
```

Create an uninitialized mutable array analogous to that specified by `storagetype`, but with `indices` specified by the last argument. `storagetype` might be a type or a function.

Examples:

creates an array that "acts like" an `Array{Int}` (and might indeed be backed by one), but which is indexed identically to `A`. If `A` has conventional indexing, this will be identical to `Array{Int}(size(A))`, but if `A` has unconventional indexing then the indices of the result will match `A`.

```
| similar(BitArray, (indices(A, 2),))
```

would create a 1-dimensional logical array whose indices match those of the columns of `A`.

```
| similar(dims->zeros(Int, dims), indices(A))
```

would create an array of `Int`, initialized to zero, matching the indices of `A`.

**source**

`Base.linspace` – Function.

```
| linspace(start, stop, n=50)
```

Construct a range of `n` linearly spaced elements from `start` to `stop`.

```
| julia> linspace(1.3, 2.9, 9)
```

```
| 1.3:0.2:2.9
```

**source**

`Base.logspace` – Function.

```
| logspace(start::Real, stop::Real, n::Integer=50; base=10)
```

Construct a vector of `n` logarithmically spaced numbers from `base^start` to `base^stop`.

```
| julia> logspace(1., 10., 5)
```

```
| 5-element Array{Float64,1}:
```

```
1778.2794100389228
316227.7660168379
```

[5.623413251903491e7](#)

[1.0e10](#)

```
julia> logspace(1., 10., 5, base=2)
5-element Array{Float64,1}:
 2.0
 9.513656920021768
 45.254833995939045
 215.2694823049509
 1024.0
```

[source](#)

[Base.Random.randsubseq](#) – Function.

| [randsubseq\(A, p\)](#) → Vector

Return a vector consisting of a random subsequence of the given array A, where each element of A is included (in order) with independent probability p. (Complexity is linear in  $p * \text{length}(A)$ , so this function is efficient even if p is small and A is large.) Technically, this process is known as “Bernoulli sampling” of A.

[source](#)

[Base.Random.randsubseq!](#) – Function.

| [randsubseq!\(S, A, p\)](#)

Like [randsubseq](#), but the results are stored in S (which is resized as needed).

## 52.2 Basic functions

`Base.ndims` – Function.

```
| ndims(A::AbstractArray) -> Integer
```

Returns the number of dimensions of A.

Examples

```
| julia> A = ones(3,4,5);
```

```
| julia> ndims(A)
```

```
| 3
```

`source`

`Base.size` – Function.

```
| size(A::AbstractArray, [dim...])
```

Returns a tuple containing the dimensions of A. Optionally you can specify the dimension(s) you want the length of, and get the length of that dimension, or a tuple of the lengths of dimensions you asked for.

Examples

```
| julia> A = ones(2,3,4);
```

```
| julia> size(A, 2)
```

```
| 3
```

```
| julia> size(A,3,2)
```

```
| (4, 3)
```

`source`

## `Base.indices` – Method.

## CHAPTER 52. ARRAYS

```
| indices(A)
```

Returns the tuple of valid indices for array A.

Examples

```
| julia> A = ones(5,6,7);  
  
| julia> indices(A)  
(Base.OneTo(5), Base.OneTo(6), Base.OneTo(7))
```

`source`

## `Base.indices` – Method.

```
| indices(A, d)
```

Returns the valid range of indices for array A along dimension d.

Examples

```
| julia> A = ones(5,6,7);  
  
| julia> indices(A,2)  
Base.OneTo(6)
```

`source`

## `Base.length` – Method.

```
| length(collection) -> Integer
```

Return the number of elements in the collection.

Use `endof` to get the last valid index of an indexable collection.

Examples

```
| julia> length(5)
```

```
| 5
```

```
| julia> length([1, 2, 3, 4])
```

```
| 4
```

```
| julia> length([1 2; 3 4])
```

```
| 4
```

source

[Base.eachindex](#) – Function.

```
| eachindex(A...)
```

Creates an iterable object for visiting each index of an AbstractArray A in an efficient manner. For array types that have opted into fast linear indexing (like `Array`), this is simply the range `1:length(A)`. For other array types, this returns a specialized Cartesian range to efficiently index into the array with indices specified for every dimension. For other iterables, including strings and dictionaries, this returns an iterator object supporting arbitrary index types (e.g. unevenly spaced or non-integer indices).

Example for a sparse 2-d array:

```
| julia> A = sparse([1, 1, 2], [1, 3, 1], [1, 2, -5])
```

```
| 2×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
```

```
| [1, 1] = 1
```

```
| [2, 1] = -5
```

```
| [1, 3] = 2
```

```
| julia> for iter in eachindex(A)
```

```
    @show A[iter]

end
(iter.I[1], iter.I[2]) = (1, 1)
A[iter] = 1
(iter.I[1], iter.I[2]) = (2, 1)
A[iter] = -5
(iter.I[1], iter.I[2]) = (1, 2)
A[iter] = 0
(iter.I[1], iter.I[2]) = (2, 2)
A[iter] = 0
(iter.I[1], iter.I[2]) = (1, 3)
A[iter] = 2
(iter.I[1], iter.I[2]) = (2, 3)
A[iter] = 0
```

If you supply more than one `AbstractArray` argument, `eachindex` will create an iterable object that is fast for all arguments (a `UnitRange` if all inputs have fast linear indexing, a `CartesianRange` otherwise). If the arrays have different sizes and/or dimensionalities, `eachindex` returns an iterable that spans the largest range along each dimension.

`source`

`Base.linearindices` – Function.

```
| linearindices(A)
```

Returns a `UnitRange` specifying the valid range of indices for `A[i]` where `i` is an `Int`. For arrays with conventional indexing (indices start at 1), or any multidimensional array, this is `1:length(A)`; however, for one-dimensional arrays with unconventional indices, this is `indices(A, 1)`.

52.2. **BASIC FUNCTIONS** Is the “safe” way to write algorithms that exploit linear indexing.

Examples

```
julia> A = ones(5,6,7);  
  
julia> b = linearindices(A);  
  
julia> extrema(b)  
(1, 210)
```

`source`

`Base.IndexStyle` – Type.

```
IndexStyle(A)  
IndexStyle(typeof(A))
```

`IndexStyle` specifies the “native indexing style” for array A. When you define a new `AbstractArray` type, you can choose to implement either linear indexing or cartesian indexing. If you decide to implement linear indexing, then you must set this trait for your array type:

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

The default is `IndexCartesian()`.

Julia’s internal indexing machinery will automatically (and invisibly) convert all indexing operations into the preferred style using `sub2ind` or `ind2sub`. This allows users to access elements of your array using any indexing style, even when explicit methods have not been provided.

If you define both styles of indexing for your `AbstractArray`, this trait can be used to select the most performant indexing style. Some methods check this trait on their inputs, and dispatch to different algorithms de-

1132 pending on the most efficient access pattern. In [CHAPTER 52 ARRAYS](#)  
creates an iterator whose type depends on the setting of this trait.

`source`

[Base.conj!](#) – Function.

```
| conj!(A)
```

Transform an array to its complex conjugate in-place.

See also [conj](#).

Examples

```
julia> A = [1+im 2-im; 2+2im 3+im]
2×2 Array{Complex{Int64},2}:
 1+1im  2-1im
 2+2im  3+1im
```

```
julia> conj!(A);
```

```
julia> A
2×2 Array{Complex{Int64},2}:
 1-1im  2+1im
 2-2im  3-1im
```

`source`

[Base.stride](#) – Function.

```
| stride(A, k::Integer)
```

Returns the distance in memory (in number of elements) between adjacent elements in dimension  $k$ .

Examples

**julia>** stride(A, 2)

3

**julia>** stride(A, 3)

12

**source**

Base.strides – Function.

**| strides(A)**

Returns a tuple of the memory strides in each dimension.

Examples

**julia>** A = ones(3, 4, 5);**julia>** strides(A)

(1, 3, 12)

**source**

Base.ind2sub – Function.

**| ind2sub(a, index) -> subscripts**

Returns a tuple of subscripts into array a corresponding to the linear index index.

Examples

**julia>** A = ones(5, 6, 7);**julia>** ind2sub(A, 35)

1134 (5, 1, 2)

CHAPTER 52. ARRAYS

```
julia> ind2sub(A, 70)
(5, 2, 3)
```

source

```
| ind2sub(dims, index) -> subscripts
```

Returns a tuple of subscripts into an array with dimensions `dims`, corresponding to the linear index `index`.

Examples

```
julia> ind2sub((3,4),2)
(2, 1)
```

```
julia> ind2sub((3,4),3)
(3, 1)
```

```
julia> ind2sub((3,4),4)
(1, 2)
```

source

`Base.sub2ind` – Function.

```
| sub2ind(dims, i, j, k...) -> index
```

The inverse of `ind2sub`, returns the linear index corresponding to the provided subscripts.

Examples

```
julia> sub2ind((5,6,7),1,2,3)
```

[source](#)

[Base.LinAlg.checksquare](#) – Function.

```
| LinAlg.checksquare(A)
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Examples

```
| julia> A = ones(4,4); B = zeros(5,5);
```

```
| julia> LinAlg.checksquare(A, B)
```

```
2-element Array{Int64,1}:
```

```
4
```

```
5
```

[source](#)

## 52.3 Broadcast and vectorization

See also the [dot syntax for vectorizing functions](#); for example, `f.(args...)` implicitly calls `broadcast(f, args...)`. Rather than relying on "vectorized" methods of functions like `sin` to operate on arrays, you should use `sin.(a)` to vectorize via `broadcast`.

[Base.broadcast](#) – Function.

```
| broadcast(f, As...)
```

Broadcasts the arrays, tuples, `Refs`, nullables, and/or scalars `As` to a container of the appropriate type and dimensions. In this context, anything

113 that is not a subtype of `AbstractArray`, `Ref` (except for `Ref{S}`s) or `Nullable` is considered a scalar. The resulting container is established by the following rules:

- If all the arguments are scalars, it returns a scalar.
- If the arguments are tuples and zero or more scalars, it returns a tuple.
- If the arguments contain at least one array or `Ref`, it returns an array (expanding singleton dimensions), and treats `Refs` as 0-dimensional arrays, and tuples as 1-dimensional arrays.

The following additional rule applies to `Nullable` arguments: If there is at least one `Nullable`, and all the arguments are scalars or `Nullable`, it returns a `Nullable` treating `Nullables` as "containers".

A special syntax exists for broadcasting: `f.(args...)` is equivalent to `broadcast(f, args...)`, and nested `f.(g.(args...))` calls are fused into a single broadcast loop.

## Examples

```
julia> A = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
5×2 Array{Int64,2}:
 1   2
 3   4
 5   6
```

```
9 10
```

```
julia> broadcast(+, A, B)
```

```
5×2 Array{Int64,2}:
```

2	3
5	6
8	9
11	12
14	15

```
julia> parse.(Int, ["1", "2"])
```

```
2-element Array{Int64,1}:
```

1
2

```
julia> abs.((1, -2))
```

```
(1, 2)
```

```
julia> broadcast(+, 1.0, (0, -2.0))
```

```
(1.0, -1.0)
```

```
julia> broadcast(+, 1.0, (0, -2.0), Ref(1))
```

```
2-element Array{Float64,1}:
```

2.0
0.0

```
julia> (+).([[0,2], [1,3]], Ref{Vector{Int}}([1,-1]))
```

```
2-element Array{Array{Int64,1},1}:
```

[1, 1]
[2, 2]

```
julia> string.(("one", "two", "three", "four"), ":" , 1:4)
4-element Array{String,1}:
"one: 1"
"two: 2"
"three: 3"
"four: 4"

julia> Nullable("X") .* "Y"
Nullable{String}("XY")

julia> broadcast(/, 1.0, Nullable(2.0))
Nullable{Float64}(0.5)

julia> (1 + im) ./ Nullable{Int}()
Nullable{Complex{Float64}}()
```

**source**

[Base.broadcast!](#) – Function.

**broadcast!(f, dest, As...)**

Like [broadcast](#), but store the result of `broadcast(f, As...)` in the `dest` array. Note that `dest` is only used to store the result, and does not supply arguments to `f` unless it is also listed in the `As`, as in `broadcast!(f, A, A, B)` to perform `A[:] = broadcast(f, A, B)`.

**source**

[Base.Broadcast.@\\_\\_dot\\_\\_](#) – Macro.

**@. expr**

52.3. **BROADCAST AND VECTORIZATION** Convert every function call or in `expr` into a "dot call" (e.g. `f(x)` becomes `f.(x)`), and convert every assignment in `expr` to a "dot assignment" (e.g. convert `+=` to `.+=`).

If you want to avoid adding dots for selected function calls in `expr`, splice those function calls in with `$.` For example, `@. sqrt(abs($sort(x)))` is equivalent to `sqrt.(abs.(sort(x)))` (no dot for `sort`).

(`@.` is equivalent to a call to `@__dot__`)

Examples

```
julia> x = 1.0:3.0; y = similar(x);
```

```
julia> @. y = x + 3 * sin(x)
```

```
3-element Array{Float64,1}:
```

```
 3.5244129544236893
```

```
 4.727892280477045
```

```
 3.4233600241796016
```

```
source
```

[Base.Broadcast.broadcast\\_getindex](#) – Function.

```
broadcast_getindex(A, inds...)
```

Broadcasts the `inds` arrays to a common size like `broadcast` and returns an array of the results `A[ks...]`, where `ks` goes over the positions in the broadcast result `A`.

Examples

```
julia> A = [1, 2, 3, 4, 5]
```

```
5-element Array{Int64,1}:
```

```
 1
```

```
 2
```

```
1140 3  
4  
5
```

## CHAPTER 52. ARRAYS

```
julia> B = [1 2; 3 4; 5 6; 7 8; 9 10]
```

```
5×2 Array{Int64,2}:
```

```
1 2  
3 4  
5 6  
7 8  
9 10
```

```
julia> C = broadcast(+, A, B)
```

```
5×2 Array{Int64,2}:
```

```
2 3  
5 6  
8 9  
11 12  
14 15
```

```
julia> broadcast_getindex(C, [1, 2, 10])
```

```
3-element Array{Int64,1}:
```

```
2  
5  
15
```

source

[Base.Broadcast.broadcast\\_setindex!](#) – Function.

```
| broadcast_setindex!(A, X, inds...)
```

Broadcasts the `X` and `inds` arrays to a common size and stores the value

52.4 from INDEXING AND ASSIGNMENT Indices in A given by the same position 141  
inds.

[source](#)

## 52.4 Indexing and assignment

[Base.getindex](#) – Method.

```
| getindex(A, inds...)
```

Return a subset of array A as specified by `inds`, where each `ind` may be an `Int`, an `AbstractRange`, or a [Vector](#). See the manual section on [array indexing](#) for details.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> getindex(A, 1)
1

julia> getindex(A, [2, 1])
2-element Array{Int64,1}:
 3
 1

julia> getindex(A, 2:4)
3-element Array{Int64,1}:
 3
 2
 4
```

`Base.setindex!` – Method.

```
| setindex!(A, X, inds...)
```

Store values from array `X` within some subset of `A` as specified by `inds`.

`source`

`Base.copy!` – Method.

```
| copy!(dest, Rdest::CartesianRange, src, Rsrc::CartesianRange)
  -> dest
```

Copy the block of `src` in the range of `Rsrc` to the block of `dest` in the range of `Rdest`. The sizes of the two regions must match.

`source`

`Base.isassigned` – Function.

```
| isassigned(array, i) -> Bool
```

Tests whether the given array has a value associated with index `i`. Returns `false` if the index is out of bounds, or has an undefined reference.

```
julia> isassigned(rand(3, 3), 5)
```

```
true
```

```
julia> isassigned(rand(3, 3), 3 * 3 + 1)
```

```
false
```

```
julia> mutable struct Foo end
```

```
julia> v = similar(rand(3), Foo)
```

```
3-element Array{Foo,1}:
```

```
#undef
```

```
#undef  
#undef  
  
julia> isassigned(v, 1)  
false
```

**source**

[Base.Colon](#) – Type.

```
|Colon()
```

Colons (:) are used to signify indexing entire objects or dimensions at once.

Very few operations are defined on Colons directly; instead they are converted by [to\\_indices](#) to an internal vector type ([Base.Slice](#)) to represent the collection of indices they span before being used.

**source**

[Base.IteratorsMD.CartesianIndex](#) – Type.

```
|CartesianIndex(i, j, k...)    -> I  
|CartesianIndex((i, j, k...)) -> I
```

Create a multidimensional index **I**, which can be used for indexing a multidimensional array **A**. In particular, **A[I]** is equivalent to **A[i, j, k...]**. One can freely mix integer and **CartesianIndex** indices; for example, **A[I<sub>pre</sub>, i, I<sub>post</sub>]** (where **I<sub>pre</sub>** and **I<sub>post</sub>** are **CartesianIndex** indices and **i** is an **Int**) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A **CartesianIndex** is sometimes produced by [eachindex](#), and always when iterating with an explicit [CartesianRange](#).

Examples

1144 **julia>** A = reshape(collect(1:16), (2, 2, 2, 2)) CHAPTER 52. ARRAYS

2×2×2×2 Array{Int64,4}:

[ :, :, 1, 1] =

1 3

2 4

[ :, :, 2, 1] =

5 7

6 8

[ :, :, 1, 2] =

9 11

10 12

[ :, :, 2, 2] =

13 15

14 16

**julia>** A[CartesianIndex((1, 1, 1, 1))]

1

**julia>** A[CartesianIndex((1, 1, 1, 2))]

9

**julia>** A[CartesianIndex((1, 1, 2, 1))]

5

**source**

[Base.Iterators](#)[MD.CartesianRange](#) – Type.

CartesianRange(sz::Dims) -> R

CartesianRange(istart:istop, jstart:jstop, ...) -> R

teger indices. These are most commonly encountered in the context of iteration, where `for I in R ... end` will return `CartesianIndex` indices `I` equivalent to the nested loops

```
for j = jstart:jstop
    for i = istart:istop
        ...
    end
end
```

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

Examples

```
julia> foreach(println, CartesianRange((2, 2, 2)))
CartesianIndex(1, 1, 1)
CartesianIndex(2, 1, 1)
CartesianIndex(1, 2, 1)
CartesianIndex(2, 2, 1)
CartesianIndex(1, 1, 2)
CartesianIndex(2, 1, 2)
CartesianIndex(1, 2, 2)
CartesianIndex(2, 2, 2)
```

`source`

`Base.to_indices` – Function.

```
| to_indices(A, I::Tuple)
```

Convert the tuple `I` to a tuple of indices for use in indexing into array `A`.

The returned tuple must only contain either `Ints` or `AbstractArrays` of scalar indices that are supported by array `A`. It will error upon encountering a novel index type that it does not know how to process.

1146 For simple index types, it defers to the unexported `Base.to_index`.

i) to process each index `i`. While this internal function is not intended to be called directly, `Base.to_index` may be extended by custom array or index types to provide custom indexing behaviors.

More complicated index types may require more context about the dimension into which they index. To support those cases, `to_indices(A, I)` calls `to_indices(A, indices(A), I)`, which then recursively walks through both the given tuple of indices and the dimensional indices of `A` in tandem. As such, not all index types are guaranteed to propagate to `Base.to_index`.

#### source

[Base.checkbounds](#) – Function.

```
| checkbounds(Bool, A, I...)
```

Return `true` if the specified indices `I` are in bounds for the given array `A`. Subtypes of `AbstractArray` should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on `A`'s indices and [checkindex](#).

See also [checkindex](#).

#### Examples

```
julia> A = rand(3, 3);
```

```
julia> checkbounds(Bool, A, 2)
true
```

```
julia> checkbounds(Bool, A, 3, 4)
false
```

```
true
```

```
julia> checkbounds(Bool, A, 1:3)
```

```
false
```

**source**

```
| checkbounds(A, I...)
```

Throw an error if the specified indices `I` are not in bounds for the given array `A`.

**source**

`Base.checkindex` – Function.

```
| checkindex(Bool, inds::AbstractUnitRange, index)
```

Return `true` if the given `index` is within the bounds of `inds`. Custom types that would like to behave as indices for all arrays can extend this method in order to provide a specialized bounds checking implementation.

Examples

```
julia> checkindex(Bool, 1:20, 8)
```

```
true
```

```
julia> checkindex(Bool, 1:20, 21)
```

```
false
```

**source**

## 52.5 Views (SubArrays and other view types)

`Base.view` – Function.

1148 `view(A, inds...)`

CHAPTER 52. ARRAYS

Like `getindex`, but returns a view into the parent array `A` with the given indices instead of making a copy. Calling `getindex` or `setindex!` on the returned `SubArray` computes the indices to the parent array on the fly without checking bounds.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = view(A, :, 1)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A # Note A has changed even though we modified b
2×2 Array{Int64,2}:
 0  2
 0  4
```

`source`

`Base.@view` – Macro.

```
@view A[inds...]
```

## 52.5. VIEWS (SSA ARRAYS AND OTHER VIEW TYPES)

This can only be applied directly to a reference expression (e.g. `@view A[1,2:end]`), and should not be used as the target of an assignment (e.g. `@view(A[1,2:end]) = ...`). See also `@views` to switch an entire block of code to use views for slicing.

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> b = @view A[:, 1]
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 1
 3

julia> fill!(b, 0)
2-element view(::Array{Int64,2}, :, 1) with eltype Int64:
 0
 0

julia> A
2×2 Array{Int64,2}:
 0  2
 0  4

source
```

`Base.@views` – Macro.

```
| @views expression
```

Convert every array-slicing operation in the given expression (which may be a begin/end block, loop, function, etc.) to return a view. Scalar in-

1150dices, non-array types, and explicit `getindex` calls<sup>1</sup> that proposed ARRAYS  
CHAPTER 52 ARRAYS  
`ray[...]`) are unaffected.

Note that the `@views` macro only affects `array[...]` expressions that appear explicitly in the given `expression`, not array slicing that occurs in functions called by that code.

`source`

`Base.parent` – Function.

```
| parent(A)
```

Returns the “parent array” of an array view type (e.g., `SubArray`), or the array itself if it is not a view.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> s_a = Symmetric(a)
2×2 Symmetric{Int64,Array{Int64,2}}:
 1  2
 2  4

julia> parent(s_a)
2×2 Array{Int64,2}:
 1  2
 3  4
```

`source`

`Base.parentindexes` – Function.

From an array view A, returns the corresponding indexes in the parent.

`source`

`Base.slicedim` – Function.

```
| slicedim(A, d::Integer, i)
```

Return all the data of A where the index for dimension d equals i. Equivalent to A[:, :, ..., i, :, :, ...] where i is in position d.

Examples

```
julia> A = [1 2 3 4; 5 6 7 8]
2×4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8
```

```
julia> slicedim(A, 2, 3)
2-element Array{Int64,1}:
 3
 7
```

`source`

`Base.reinterpret` – Function.

```
| reinterpret(type, A)
```

Change the type-interpretation of a block of memory. For arrays, this constructs a view of the array with the same binary data as the given array, but with the specified element type. For example, `reinterpret(Float32, UInt32(7))` interprets the 4 bytes corresponding to `UInt32(7)` as a `Float32`.

Examples

```
1152 julia> reinterpret(Float32, UInt32(7))
```

CHAPTER 52. ARRAYS

```
1.0f-44
```

```
julia> reinterpret(Float32, UInt32[1 2 3 4 5])
```

```
1×5 reinterpret(Float32, ::Array{UInt32,2}):  
 1.4013e-45 2.8026e-45 4.2039e-45 5.60519e-45 7.00649e-45
```

source

[Base.reshape](#) – Function.

```
| reshape(A, dims...) -> R  
| reshape(A, dims) -> R
```

Return an array R with the same data as A, but with different dimension sizes or number of dimensions. The two arrays share the same underlying data, so that setting elements of R alters the values of A and vice versa.

The new dimensions may be specified either as a list of arguments or as a shape tuple. At most one dimension may be specified with a :, in which case its length is computed such that its product with all the specified dimensions is equal to the length of the original array A. The total number of elements must not change.

```
julia> A = collect(1:16)  
16-element Array{Int64,1}:  
 1  
 2  
 3  
 4  
 5  
 6  
 7  
 8
```

```
10  
11  
12  
13  
14  
15  
16
```

```
julia> reshape(A, (4, 4))  
4×4 Array{Int64,2}:  
 1  5   9  13  
 2  6  10  14  
 3  7  11  15  
 4  8  12  16
```

```
julia> reshape(A, 2, :)  
2×8 Array{Int64,2}:  
 1  3   5   7   9   11  13  15  
 2  4   6   8   10  12  14  16
```

source

[Base.squeeze](#) – Function.

```
| squeeze(A, dims)
```

Remove the dimensions specified by `dims` from array `A`. Elements of `dims` must be unique and within the range `1:ndims(A)`. `size(A, i)` must equal 1 for all `i` in `dims`.

Examples

```
julia> a = reshape(collect(1:4), (2,2,1,1))  
2×2×1×1 Array{Int64,4}:
```

```
1154[:, :, 1, 1] =  
1 3  
2 4
```

## CHAPTER 52. ARRAYS

```
julia> squeeze(a,3)  
2×2×1 Array{Int64,3}:  
[:, :, 1] =  
1 3  
2 4
```

source

[Base.vec](#) – Function.

```
| vec(a::AbstractArray) -> Vector
```

Reshape the array `a` as a one-dimensional column vector. The resulting array shares the same underlying data as `a`, so modifying one will also modify the other.

Examples

```
julia> a = [1 2 3; 4 5 6]  
2×3 Array{Int64,2}:  
1 2 3  
4 5 6
```

```
julia> vec(a)  
6-element Array{Int64,1}:  
1  
4  
2  
5  
3  
6
```

**source**

## 52.6 Concatenation and permutation

**Base.cat** – Function.

```
| cat(dims, A...)
```

Concatenate the input arrays along the specified dimensions in the iterable `dims`. For dimensions not in `dims`, all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in `dims`, the size of the output array is the sum of the sizes of the input arrays along that dimension. If `dims` is a single number, the different arrays are tightly stacked along that dimension. If `dims` is an iterable containing several dimensions, this allows one to construct block diagonal matrices and their higher-dimensional analogues by simultaneously increasing several dimensions for every new input array and putting zero blocks elsewhere. For example, `cat([1,2], matrices...)` builds a block diagonal matrix, i.e. a block matrix with `matrices[1]`, `matrices[2]`, ... as diagonal blocks and matching zero blocks away from the diagonal.

**source**

**Base.vcat** – Function.

```
| vcat(A...)
```

Concatenate along dimension 1.

Examples

```
| julia> a = [1 2 3 4 5]
| 1×5 Array{Int64,2}:
```

1156 1 2 3 4 5

CHAPTER 52. ARRAYS

```
julia> b = [6 7 8 9 10; 11 12 13 14 15]
```

```
2×5 Array{Int64,2}:
 6   7   8   9   10
 11  12  13  14  15
```

```
julia> vcat(a,b)
```

```
3×5 Array{Int64,2}:
 1   2   3   4   5
 6   7   8   9   10
 11  12  13  14  15
```

```
julia> c = ([1 2 3], [4 5 6])
```

```
([1 2 3], [4 5 6])
```

```
julia> vcat(c...)
```

```
2×3 Array{Int64,2}:
 1   2   3
 4   5   6
```

source

[Base.hcat](#) – Function.

```
| hcat(A...)
```

Concatenate along dimension 2.

Examples

```
julia> a = [1; 2; 3; 4; 5]
```

```
5-element Array{Int64,1}:
 1
```

```
2  
3  
4  
5
```

```
julia> b = [6 7; 8 9; 10 11; 12 13; 14 15]
```

```
5×2 Array{Int64,2}:
```

```
6 7  
8 9  
10 11  
12 13  
14 15
```

```
julia> hcat(a,b)
```

```
5×3 Array{Int64,2}:
```

```
1 6 7  
2 8 9  
3 10 11  
4 12 13  
5 14 15
```

```
julia> c = ([1; 2; 3], [4; 5; 6])
```

```
([1, 2, 3], [4, 5, 6])
```

```
julia> hcat(c...)
```

```
3×2 Array{Int64,2}:
```

```
1 4  
2 5  
3 6
```

source

```
| hvcat(rows::Tuple{Vararg{Int}}, values...)
```

Horizontal and vertical concatenation in one call. This function is called for block matrix syntax. The first argument specifies the number of arguments to concatenate in each block row.

Examples

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)
```

```
julia> [a b c; d e f]
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

```
julia> hvcat((3,3), a,b,c,d,e,f)
2×3 Array{Int64,2}:
 1  2  3
 4  5  6
```

```
julia> [a b;c d; e f]
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

```
julia> hvcat((2,2,2), a,b,c,d,e,f)
3×2 Array{Int64,2}:
 1  2
 3  4
 5  6
```

52.6 If the concatenation and permutation then all block rows are assumed to have  $n$  block columns.

[source](#)

[Base.flipdim](#) – Function.

```
| flipdim(A, d::Integer)
```

Reverse  $A$  in dimension  $d$ .

Examples

```
julia> b = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> flipdim(b,2)
2×2 Array{Int64,2}:
 2  1
 4  3
```

[source](#)

[Base.circshift](#) – Function.

```
| circshift(A, shifts)
```

Circularly shift, i.e. rotate, the data in an array. The second argument is a tuple or vector giving the amount to shift in each dimension, or an integer to shift only in the first dimension.

Examples

```
julia> b = reshape(collect(1:16), (4,4))
4×4 Array{Int64,2}:
 1  5   9  13
```

```
1160 2 6 10 14  
3 7 11 15  
4 8 12 16
```

## CHAPTER 52. ARRAYS

```
julia> circshift(b, (0,2))
```

```
4×4 Array{Int64,2}:  
 9 13 1 5  
10 14 2 6  
11 15 3 7  
12 16 4 8
```

```
julia> circshift(b, (-1,0))
```

```
4×4 Array{Int64,2}:  
2 6 10 14  
3 7 11 15  
4 8 12 16  
1 5 9 13
```

```
julia> a = BitArray([true, true, false, false, true])
```

```
5-element BitArray{1}:  
 true  
 true  
 false  
 false  
 true
```

```
julia> circshift(a, 1)
```

```
5-element BitArray{1}:  
 true  
 true  
 true
```

```
false
```

```
false
```

```
julia> circshift(a, -1)
```

```
5-element BitArray{1}:
```

```
 true
```

```
false
```

```
false
```

```
 true
```

```
 true
```

See also [circshift!](#).

[source](#)

[Base.circshift!](#) – Function.

```
| circshift!(dest, src, shifts)
```

Circularly shift, i.e. rotate, the data in `src`, storing the result in `dest`.  
`shifts` specifies the amount to shift in each dimension.

The `dest` array must be distinct from the `src` array (they cannot alias each other).

See also [circshift](#).

[source](#)

[Base.circcopy!](#) – Function.

```
| circcopy!(dest, src)
```

Copy `src` to `dest`, indexing each dimension modulo its length. `src` and `dest` must have the same size, but can be offset in their indices; any offset results in a (circular) wraparound. If the arrays have overlapping indices, then on the domain of the overlap `dest` agrees with `src`.

```
julia> src = reshape(collect(1:16), (4,4))  
4×4 Array{Int64,2}:  
 1  5  9  13  
 2  6  10 14  
 3  7  11 15  
 4  8  12 16  
  
julia> dest = OffsetArray{Int}((0:3,2:5))  
  
julia> circcopy!(dest, src)  
OffsetArrays.OffsetArray{Int64,2,Array{Int64,2}} with indices  
→ 0:3×2:5:  
 8  12  16  4  
 5  9  13  1  
 6  10  14  2  
 7  11  15  3  
  
julia> dest[1:3,2:4] == src[1:3,2:4]  
true
```

source

[Base.find](#) – Method.

| `find(A)`

Return a vector of the linear indices of the `true` values in `A`. To search for other kinds of values, pass a predicate as the first argument.

Examples

```
julia> A = [true false; false true]  
2×2 Array{Bool,2}:
```

```
true false
```

```
false true
```

```
julia> find(A)
```

```
2-element Array{Int64,1}:
```

```
1
```

```
4
```

```
julia> find(falses(3))
```

```
0-element Array{Int64,1}
```

source

[Base.find](#) – Method.

```
| find(f::Function, A)
```

Return a vector  $\mathbf{I}$  of the linear indexes of  $A$  where  $f(A[\mathbf{I}])$  returns `true`.

If there are no such elements of  $A$ , `find` returns an empty array.

Examples

```
julia> A = [1 2 0; 3 4 0]
```

```
2×3 Array{Int64,2}:
```

```
1 2 0
```

```
3 4 0
```

```
julia> find(isodd, A)
```

```
2-element Array{Int64,1}:
```

```
1
```

```
2
```

```
julia> find(!iszero, A)
```

```
4-element Array{Int64,1}:
```

```
1164 1  
2  
3  
4
```

## CHAPTER 52. ARRAYS

```
julia> find(isodd, [2, 4])  
0-element Array{Int64,1}
```

**source**

[Base.findn](#) – Function.

```
| findn(A)
```

Return a vector of indexes for each dimension giving the locations of the non-zeros in A (determined by  $A[i] \neq 0$ ). If there are no non-zero elements of A, `findn` returns a 2-tuple of empty arrays.

Examples

```
julia> A = [1 2 0; 0 0 3; 0 4 0]  
3×3 Array{Int64,2}:  
1 2 0  
0 0 3  
0 4 0
```

```
julia> findn(A)  
([1, 1, 3, 2], [1, 2, 2, 3])
```

```
julia> A = zeros(2,2)  
2×2 Array{Float64,2}:  
0.0 0.0  
0.0 0.0
```

```
julia> findn(A)  
(Int64[], Int64[])
```

**source**

[Base.findnz](#) – Function.

```
| findnz(A)
```

Return a tuple ( $I$ ,  $J$ ,  $V$ ) where  $I$  and  $J$  are the row and column indexes of the non-zero values in matrix  $A$ , and  $V$  is a vector of the non-zero values.

Examples

```
julia> A = [1 2 0; 0 0 3; 0 4 0]  
3×3 Array{Int64,2}:  
 1 2 0  
 0 0 3  
 0 4 0  
  
julia> findnz(A)  
([1, 1, 3, 2], [1, 2, 2, 3], [1, 2, 4, 3])
```

**source**

[Base.findfirst](#) – Method.

```
| findfirst(A)
```

Return the linear index of the first `true` value in  $A$ . Returns `0` if no such value is found. To search for other kinds of values, pass a predicate as the first argument.

Examples

```
julia> A = [false false; true false]  
2×2 Array{Bool,2}:
```

```
1166 false  false  
      true  false
```

## CHAPTER 52. ARRAYS

```
julia> findfirst(A)  
2  
  
julia> findfirst(falses(3))  
0
```

source

[Base.findfirst](#) – Method.

```
| findfirst(predicate::Function, A)
```

Return the linear index of the first element of `A` for which `predicate` returns `true`. Returns `0` if there is no such element.

Examples

```
julia> A = [1 4; 2 2]  
2×2 Array{Int64,2}:  
 1  4  
 2  2
```

```
julia> findfirst(iseven, A)  
2
```

```
julia> findfirst(x -> x>10, A)  
0
```

```
julia> findfirst(equalto(4), A)  
3
```

source

```
| findlast(A)
```

Return the linear index of the last `true` value in `A`. Returns `0` if there is no `true` value in `A`.

Examples

```
julia> A = [true false; true false]
2×2 Array{Bool,2}:
 true  false
 true  false
```

```
julia> findlast(A)
```

```
2
```

```
julia> A = falses(2,2);
```

```
julia> findlast(A)
```

```
0
```

`source`

`Base.findlast` – Method.

```
| findlast(predicate::Function, A)
```

Return the linear index of the last element of `A` for which `predicate` returns `true`. Returns `0` if there is no such element.

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
```

```
julia> findlast(isodd, A)  
2  
  
julia> findlast(x -> x > 5, A)  
0
```

source

[Base.findnext](#) – Method.

```
| findnext(A, i::Integer)
```

Find the next linear index  $\geq i$  of a `true` element of `A`, or `0` if not found.

Examples

```
julia> A = [false false; true false]  
2×2 Array{Bool,2}:  
  false  false  
  true  false
```

```
julia> findnext(A, 1)
```

```
2
```

```
julia> findnext(A, 3)
```

```
0
```

source

[Base.findnext](#) – Method.

```
| findnext(predicate::Function, A, i::Integer)
```

Find the next linear index  $\geq i$  of an element of `A` for which `predicate` returns `true`, or `0` if not found.

```
julia> A = [1 4; 2 2]
2×2 Array{Int64,2}:
 1  4
 2  2

julia> findnext(isodd, A, 1)
1

julia> findnext(isodd, A, 2)
0
```

source

[Base.findprev](#) – Method.

```
findprev(A, i::Integer)
```

Find the previous linear index  $\leq i$  of a `true` element of `A`, or `0` if not found.

Examples

```
julia> A = [false false; true true]
2×2 Array{Bool,2}:
 false  false
  true   true

julia> findprev(A,2)
2

julia> findprev(A,1)
0
```

source

```
| findprev(predicate::Function, A, i::Integer)
```

Find the previous linear index ( $\leq i$ ) of an element of `A` for which `predicate` returns `true`, or `0` if not found.

Examples

```
| julia> A = [4 6; 1 2]
| 2×2 Array{Int64,2}:
|   4  6
|   1  2
|
| julia> findprev(isodd, A, 1)
| 0
|
| julia> findprev(isodd, A, 3)
| 2
```

`source`

```
| permutedims(A, perm)
```

Permute the dimensions of array `A`. `perm` is a vector specifying a permutation of length `ndims(A)`. This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to `permutedims(A, [2, 1])`.

See also: [PermutedDimsArray](#).

Examples

```
| julia> A = reshape(collect(1:8), (2,2,2))
| 2×2×2 Array{Int64,3}:
```

```
1 3  
2 4  
  
[:, :, 2] =  
5 7  
6 8  
  
julia> permutedims(A, [3, 2, 1])  
2×2×2 Array{Int64,3}:  
[:, :, 1] =  
1 3  
5 7  
  
[:, :, 2] =  
2 4  
6 8
```

**source**

[Base.permutedims!](#) – Function.

```
| permutedims!(dest, src, perm)
```

Permute the dimensions of array `src` and store the result in the array `dest`. `perm` is a vector specifying a permutation of length `ndims(src)`. The preallocated array `dest` should have `size(dest) == size(src)[perm]` and is completely overwritten. No in-place permutation is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

See also [permutedims](#).

**source**

```
| PermutedDimsArray(A, perm) -> B
```

Given an AbstractArray A, create a view B such that the dimensions appear to be permuted. Similar to `permutedims`, except that no copying occurs (B shares storage with A).

See also: [permutedims](#).

Examples

```
julia> A = rand(3,5,4);

julia> B = PermutedDimsArray(A, (3,1,2));

julia> size(B)
(4, 3, 5)

julia> B[3,1,2] == A[1,2,3]
true
```

[source](#)

[Base.promote\\_shape](#) – Function.

```
| promote_shape(s1, s2)
```

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

```
julia> a = ones(3,4,1,1,1);

julia> b = ones(3,4);

julia> promote_shape(a,b)
```

52.7.1173 | **ARRAY FUNCTIONS**  
| (Base.OneTo(3), Base.OneTo(4), Base.OneTo(1), Base.OneTo(1),  
| → Base.OneTo(1))

```
| julia> promote_shape((2,3,1,4), (2, 3, 1, 4, 1))  
| (2, 3, 1, 4, 1)
```

[source](#)

## 52.7 Array functions

[Base.accumulate](#) – Method.

```
| accumulate(op, A, dim=1)
```

Cumulative operation `op` along a dimension `dim`. See also [accumulate!](#) to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow). For common operations there are specialized variants of `accumulate`, see: [cumsum](#), [cumprod](#)

```
| julia> accumulate(+, [1,2,3])  
| 3-element Array{Int64,1}:  
|   1  
|   3  
|   6
```

```
| julia> accumulate(*, [1,2,3])  
| 3-element Array{Int64,1}:  
|   1  
|   2  
|   6
```

[source](#)

```
| accumulate(op, v0, A)
```

1174 like `accumulate`, but using a starting element `v0`. The result will be `op(v0, first(A))`.

Examples

```
julia> accumulate(+, 100, [1,2,3])
```

```
3-element Array{Int64,1}:
```

```
101
```

```
103
```

```
106
```

```
julia> accumulate(min, 0, [1,2,-1])
```

```
3-element Array{Int64,1}:
```

```
0
```

```
0
```

```
-1
```

`source`

`Base.accumulate!` – Function.

```
| accumulate!(op, B, A, dim=1)
```

Cumulative operation `op` on `A` along a dimension, storing the result in `B`.

See also `accumulate`.

`source`

`Base.cumprod` – Function.

```
| cumprod(A, dim=1)
```

Cumulative product along a dimension `dim`. See also `cumprod!` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

```
julia> a = [1 2 3; 4 5 6]
```

```
2×3 Array{Int64,2}:
```

```
1 2 3  
4 5 6
```

```
julia> cumprod(a,1)
```

```
2×3 Array{Int64,2}:
```

```
1 2 3  
4 10 18
```

```
julia> cumprod(a,2)
```

```
2×3 Array{Int64,2}:
```

```
1 2 6  
4 20 120
```

`source`

`Base.cumprod!` – Function.

```
| cumprod!(B, A, dim::Integer=1)
```

Cumulative product of `A` along a dimension, storing the result in `B`. See also `cumprod`.

`source`

`Base.cumsum` – Function.

```
| cumsum(A, dim=1)
```

Cumulative sum along a dimension `dim`. See also `cumsum!` to use a pre-allocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

```
julia> a = [1 2 3; 4 5 6]
```

```
2×3 Array{Int64,2}:
```

```
1 2 3  
4 5 6
```

```
julia> cumsum(a,1)
2×3 Array{Int64,2}:
 1  2  3
 5  7  9
```

```
julia> cumsum(a,2)
2×3 Array{Int64,2}:
 1  3  6
 4  9  15
```

[source](#)

[Base.cumsum!](#) – Function.

```
| cumsum!(B, A, dim::Integer=1)
```

Cumulative sum of A along a dimension, storing the result in B. See also [cumsum](#).

[source](#)

[Base.cumsum\\_kbn](#) – Function.

```
| cumsum_kbn(A, [dim::Integer=1])
```

Cumulative sum along a dimension, using the Kahan–Babuska–Neumaier compensated summation algorithm for additional accuracy. The dimension defaults to 1.

[source](#)

[Base.LinAlg.diff](#) – Function.

```
| diff(A, [dim::Integer=1])
```

Finite difference operator of matrix or vector A. If A is a matrix, compute the finite difference over a dimension `dim` (default 1).

```
julia> a = [2 4; 6 16]
2×2 Array{Int64,2}:
 2   4
 6  16
```

```
julia> diff(a,2)
2×1 Array{Int64,2}:
 2
 10
```

source

[Base.repeat](#) – Method.

```
repeat(A::AbstractArray; inner=ntuple(x->1, ndims(A)), outer=
      ntuple(x->1, ndims(A)))
```

Construct an array by repeating the entries of A. The i-th element of `inner` specifies the number of times that the individual entries of the i-th dimension of A should be repeated. The i-th element of `outer` specifies the number of times that a slice along the i-th dimension of A should be repeated. If `inner` or `outer` are omitted, no repetition is performed.

Examples

```
julia> repeat(1:2, inner=2)
4-element Array{Int64,1}:
 1
 1
 2
 2
```

```
julia> repeat(1:2, outer=2)
```

```
1178 4-element Array{Int64,1}:
```

```
1  
2  
1  
2
```

```
julia> repeat([1 2; 3 4], inner=(2, 1), outer=(1, 3))
```

```
4×6 Array{Int64,2}:
```

```
1 2 1 2 1 2  
1 2 1 2 1 2  
3 4 3 4 3 4  
3 4 3 4 3 4
```

`source`

`Base.rot180` – Function.

```
| rot180(A)
```

Rotate matrix A 180 degrees.

Examples

```
julia> a = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
1 2  
3 4
```

```
julia> rot180(a)
```

```
2×2 Array{Int64,2}:
```

```
4 3  
2 1
```

`source`

```
| rot180(A, k)
```

52.7 **ROTATE FUNCTIONS** rotates a matrix  $k$  degrees an integer  $k$  number of times. If  $k$  is even, this is equivalent to a copy.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> rot180(a,1)
2×2 Array{Int64,2}:
 4  3
 2  1
```

```
julia> rot180(a,2)
2×2 Array{Int64,2}:
 1  2
 3  4
```

[source](#)

[Base.rotl90](#) – Function.

```
| rotl90(A)
```

Rotate matrix A left 90 degrees.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> rotl90(a)
```

```
1180 2×2 Array{Int64,2}:
  2  4
  1  3
```

## CHAPTER 52. ARRAYS

source

```
| rotl90(A, k)
```

Rotate matrix A left 90 degrees an integer k number of times. If k is zero or a multiple of four, this is equivalent to a `copy`.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
  1  2
  3  4
```

```
julia> rotl90(a,1)
2×2 Array{Int64,2}:
  2  4
  1  3
```

```
julia> rotl90(a,2)
2×2 Array{Int64,2}:
  4  3
  2  1
```

```
julia> rotl90(a,3)
2×2 Array{Int64,2}:
  3  1
  4  2
```

```
julia> rotl90(a,4)
```

52.7 | 2×2 Array{Int64,2}:

```
1 2  
3 4
```

**source**

[Base.rot90](#) – Function.

| **rot90(A)**

Rotate matrix A right 90 degrees.

Examples

**julia>** a = [1 2; 3 4]

2×2 Array{Int64,2}:

```
1 2  
3 4
```

**julia>** rot90(a)

2×2 Array{Int64,2}:

```
3 1  
4 2
```

**source**

| **rot90(A, k)**

Rotate matrix A right 90 degrees an integer **k** number of times. If **k** is zero or a multiple of four, this is equivalent to a **copy**.

Examples

**julia>** a = [1 2; 3 4]

2×2 Array{Int64,2}:

```
1 2  
3 4
```

```

julia> rot90(a,1)
2×2 Array{Int64,2}:
 3  1
 4  2

julia> rot90(a,2)
2×2 Array{Int64,2}:
 4  3
 2  1

julia> rot90(a,3)
2×2 Array{Int64,2}:
 2  4
 1  3

julia> rot90(a,4)
2×2 Array{Int64,2}:
 1  2
 3  4

```

[source](#)

`Base.reducedim` – Function.

```
| reducedim(f, A, region[, v0])
```

Reduce 2-argument function `f` along dimensions of `A`. `region` is a vector specifying the dimensions to reduce, and `v0` is the initial value to use in the reductions. For `+`, `*`, `max` and `min` the `v0` argument is optional.

The associativity of the reduction is implementation-dependent; if you need a particular associativity, e.g. left-to-right, you should write your own loop. See documentation for `reduce`.

```
julia> a = reshape(collect(1:16), (4,4))  
4×4 Array{Int64,2}:  
 1  5   9  13  
 2  6   10 14  
 3  7   11 15  
 4  8   12 16  
  
julia> reducedim(max, a, 2)  
4×1 Array{Int64,2}:  
 13  
 14  
 15  
 16  
  
julia> reducedim(max, a, 1)  
1×4 Array{Int64,2}:  
 4  8  12  16
```

source

[Base.mapreducedim](#) – Function.

```
| mapreducedim(f, op, A, region[, v0])
```

Evaluates to the same as `reducedim(op, map(f, A), region, f(v0))`, but is generally faster because the intermediate array is avoided.

Examples

```
julia> a = reshape(collect(1:16), (4,4))  
4×4 Array{Int64,2}:  
 1  5   9  13  
 2  6   10 14
```

```
1184 3 7 11 15  
4 8 12 16
```

## CHAPTER 52. ARRAYS

```
julia> mapreducedim(isodd, *, a, 1)  
1×4 Array{Bool,2}:  
false false false false  
  
julia> mapreducedim(isodd, |, a, 1, true)  
1×4 Array{Bool,2}:  
true true true true
```

source

[Base.mapslices](#) – Function.

```
|mapslices(f, A, dims)
```

Transform the given dimensions of array  $A$  using function  $f$ .  $f$  is called on each slice of  $A$  of the form  $A[\dots, :, \dots, :, \dots]$ .  $\text{dims}$  is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if  $\text{dims}$  is  $[1, 2]$  and  $A$  is 4-dimensional,  $f$  is called on  $A[:, :, i, j]$  for all  $i$  and  $j$ .

Examples

```
julia> a = reshape(collect(1:16), (2,2,2,2))  
2×2×2×2 Array{Int64,4}:  
[:, :, 1, 1] =  
1 3  
2 4  
  
[:, :, 2, 1] =  
5 7  
6 8
```

```
[ :, :, 1, 2] =  
    9  11  
   10  12  
  
[ :, :, 2, 2] =  
  13  15  
  14  16  
  
julia> mapslices(sum, a, [1,2])  
1×1×2×2 Array{Int64,4}:  
[ :, :, 1, 1] =  
    10  
  
[ :, :, 2, 1] =  
    26  
  
[ :, :, 1, 2] =  
    42  
  
[ :, :, 2, 2] =  
    58
```

**source**

[Base.sum\\_kbn](#) – Function.

| **sum\_kbn(A)**

Returns the sum of all elements of A, using the Kahan–Babuska–Neumaier compensated summation algorithm for additional accuracy.

**source**

`Base.Random.randperm` – Function.

```
randperm([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random permutation of length `n`. The optional `rng` argument specifies a random number generator (see [Random Numbers](#)). To randomly permute an arbitrary vector, see `shuffle` or `shuffle!`.

Examples

```
julia> randperm(MersenneTwister(1234), 4)
4-element Array{Int64,1}:
 2
 1
 4
 3
```

`source`

`Base.Random.randperm!` – Function.

```
randperm!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in `A` a random permutation of length `length(A)`. The optional `rng` argument specifies a random number generator (see [Random Numbers](#)). To randomly permute an arbitrary vector, see `shuffle` or `shuffle!`.

Examples

```
julia> randperm!(MersenneTwister(1234), Vector{Int}(4))
4-element Array{Int64,1}:
 2
 1
```

source

Base.invperm – Function.

| invperm(v)

Return the inverse permutation of v. If  $B = A[v]$ , then  $A == B[\text{invperm}(v)]$ .

Examples

| julia> v = [2; 4; 3; 1];

| julia> invperm(v)

4-element Array{Int64,1}:

4

1

3

2

| julia> A = ['a', 'b', 'c', 'd'];

| julia> B = A[v]

4-element Array{Char,1}:

'b'

'd'

'c'

'a'

| julia> B[invperm(v)]

4-element Array{Char,1}:

```
1188 'a'  
     'b'  
     'c'  
     'd'
```

## CHAPTER 52. ARRAYS

**source**

[Base.isperm](#) – Function.

```
| isperm(v) -> Bool
```

Returns **true** if  $v$  is a valid permutation.

Examples

```
| julia> isperm([1; 2])
```

```
true
```

```
| julia> isperm([1; 3])
```

```
false
```

**source**

[Base.permute!](#) – Method.

```
| permute!(v, p)
```

Permute vector  $v$  in-place, according to permutation  $p$ . No checking is done to verify that  $p$  is a permutation.

To return a new permutation, use  $v[p]$ . Note that this is generally faster than  $\text{permute}!(v, p)$  for large vectors.

See also [ipermute!](#).

Examples

```
| julia> A = [1, 1, 3, 4];
```

52.8| **julia>** COMBINATORICS perm = [2, 4, 3, 1];

1189

**julia>** permute!(A, perm);

**julia>** A

4-element Array{Int64,1}:

1

4

3

1

**source**

**Base.ipermute!** – Function.

| **ipermute!(v, p)**

Like **permute!**, but the inverse of the given permutation is applied.

Examples

**julia>** A = [1, 1, 3, 4];

**julia>** perm = [2, 4, 3, 1];

**julia>** ipermute!(A, perm);

**julia>** A

4-element Array{Int64,1}:

4

1

3

1

**source**

```
| randcycle([rng=GLOBAL_RNG,] n::Integer)
```

Construct a random cyclic permutation of length `n`. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Examples

```
julia> randcycle(MersenneTwister(1234), 6)
```

```
6-element Array{Int64,1}:
```

```
3  
5  
4  
6  
1  
2
```

[source](#)

```
| randcycle!([rng=GLOBAL_RNG,] A::Array{<:Integer})
```

Construct in `A` a random cyclic permutation of length `length(A)`. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Examples

```
julia> randcycle!(MersenneTwister(1234), Vector{Int}(6))
```

```
6-element Array{Int64,1}:
```

```
3  
5  
4  
6
```

[source](#)

[Base.Random.shuffle](#) – Function.

```
| shuffle([rng=GLOBAL_RNG,] v::AbstractArray)
```

Return a randomly permuted copy of  $v$ . The optional `rng` argument specifies a random number generator (see [Random Numbers](#)). To permute  $v$  in-place, see [shuffle!](#). To obtain randomly permuted indices, see [randperm](#).

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> shuffle(rng, collect(1:10))
```

```
10-element Array{Int64,1}:
```

```
 6  
 1  
 10  
 2  
 3  
 9  
 5  
 7  
 4  
 8
```

[source](#)

[Base.Random.shuffle!](#) – Function.

```
| shuffle!([rng=GLOBAL_RNG,] v::AbstractArray)
```

1192n-place version of [shuffle](#): randomly permute CHAPTER 5. ARRAYS  
supplying the random-number generator `rng`.

### Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> shuffle!(rng, collect(1:16))
```

```
16-element Array{Int64,1}:
```

```
 2  
 15  
 5  
 14  
 1  
 9  
 10  
 6  
 11  
 3  
 16  
 7  
 4  
 12  
 8  
 13
```

[source](#)

[Base.reverse](#) – Function.

```
| reverse(v [, start=1 [, stop=length(v) ]] )
```

Return a copy of `v` reversed from `start` to `stop`. See also [Iterators.reverse](#) for reverse-order iteration without making a copy.

### Examples

```
5-element Array{Int64,1}:
```

```
1  
2  
3  
4  
5
```

```
julia> reverse(A)
```

```
5-element Array{Int64,1}:
```

```
5  
4  
3  
2  
1
```

```
julia> reverse(A, 1, 4)
```

```
5-element Array{Int64,1}:
```

```
4  
3  
2  
1  
5
```

```
julia> reverse(A, 3, 5)
```

```
5-element Array{Int64,1}:
```

```
1  
2  
5  
4  
3
```

```
| reverse(s::AbstractString) -> AbstractString
```

Reverses a string.

Technically, this function reverses the codepoints in a string, and its main utility is for reversed-order string processing, especially for reversed regular-expression searches. See also `reverseind` to convert indices in `s` to indices in `reverse(s)` and vice-versa, and `graphemes` to operate on user-visible "characters" (graphemes) rather than codepoints. See also `Iterators.reverse` for reverse-order iteration without making a copy.

Examples

```
julia> reverse("JuliaLang")
```

```
"gnaLailuJ"
```

```
julia> reverse("axe") # combining characters can lead to
```

```
    → surprising results
```

```
"exa"
```

```
julia> join(reverse(collect(graphemes("axe")))) # reverses
```

```
    → graphemes
```

```
"exa"
```

`source`

`Base.reverseind` – Function.

```
| reverseind(v, i)
```

Given an index `i` in `reverse(v)`, return the corresponding index in `v` so that `v[reverseind(v, i)] == reverse(v)[i]`. (This can be nontrivial in cases where `v` contains non-ASCII characters.)

Examples

52.9 | **julia>** r = reverse("Julia")  
"ailuJ"

1195

**julia>** for i in 1:length(r)  
    print(r[reverseind("Julia", i)])

end

Julia

source

**Base.reverse!** – Function.

| reverse!(v [, start=1 [, stop=length(v) ]]) -> v

In-place version of **reverse**.

source

## 52.9 BitArrays

**BitArrays** are space-efficient “packed” boolean arrays, which store one bit per boolean value. They can be used similarly to **Array{Bool}** arrays (which store one byte per boolean value), and can be converted to/from the latter via **Array(bitarray)** and **BitArray(array)**, respectively.

**Base.flipbits!** – Function.

| flipbits!(B::BitArray{N}) -> BitArray{N}

Performs a bitwise not operation on B. See [~](#).

Examples

**julia>** A = trues(2,2)  
2×2 BitArray{2}:

```
1196 true  true  
      true  true
```

CHAPTER 52. ARRAYS

```
julia> flipbits!(A)  
2×2 BitArray{2}:  
  false  false  
  false  false
```

`source`

## 52.10 Sparse Vectors and Matrices

Sparse vectors and matrices largely support the same set of operations as their dense counterparts. The following functions are specific to sparse arrays.

[Base.SparseArrays.SparseVector](#) – Type.

```
| SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,Ti}
```

Vector type for storing sparse vectors.

`source`

[Base.SparseArrays.SparseMatrixCSC](#) – Type.

```
| SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrix{Tv,Ti}
```

Matrix type for storing sparse matrices in the [Compressed Sparse Column](#) format.

`source`

[Base.SparseArrays.sparse](#) – Function.

```
| sparse(A)
```

Convert an `AbstractMatrix A` into a sparse matrix.

Examples

```
julia> A = Matrix(1.0I, 3, 3)
```

3×3 Array{Float64,2}:

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

```
julia> sparse(A)
```

3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:

[1, 1]	=	1.0
[2, 2]	=	1.0
[3, 3]	=	1.0

**source**

```
| sparse(I, J, V, [ m, n, combine])
```

Create a sparse matrix  $S$  of dimensions  $m \times n$  such that  $S[I[k], J[k]] = V[k]$ . The `combine` function is used to combine duplicates. If  $m$  and  $n$  are not specified, they are set to `maximum(I)` and `maximum(J)` respectively. If the `combine` function is not supplied, `combine` defaults to `+` unless the elements of  $V$  are Booleans in which case `combine` defaults to `|`. All elements of  $I$  must satisfy  $1 \leq I[k] \leq m$ , and all elements of  $J$  must satisfy  $1 \leq J[k] \leq n$ . Numerical zeros in  $(I, J, V)$  are retained as structural nonzeros; to drop numerical zeros, use [dropzeros!](#).

For additional documentation and an expert driver, see `Base.SparseArrays.sparse!`.

Examples

```
julia> Is = [1; 2; 3];
```

```
julia> Js = [1; 2; 3];
```

```
1198 julia> Vs = [1; 2; 3];
```

CHAPTER 52. ARRAYS

```
julia> sparse(Is, Js, Vs)
```

```
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
```

```
[1, 1] = 1  
[2, 2] = 2  
[3, 3] = 3
```

source

[Base.SparseArrays.sparsevec](#) – Function.

```
| sparsevec(I, V, [m, combine])
```

Create a sparse vector  $S$  of length  $m$  such that  $S[I[k]] = V[k]$ . Duplicates are combined using the `combine` function, which defaults to `+` if no `combine` argument is provided, unless the elements of  $V$  are Booleans in which case `combine` defaults to `|`.

Examples

```
julia> II = [1, 3, 3, 5]; V = [0.1, 0.2, 0.3, 0.2];
```

```
julia> sparsevec(II, V)
```

```
5-element SparseVector{Float64, Int64} with 3 stored entries:
```

```
[1] = 0.1  
[3] = 0.5  
[5] = 0.2
```

```
julia> sparsevec(II, V, 8, -)
```

```
8-element SparseVector{Float64, Int64} with 3 stored entries:
```

```
[1] = 0.1  
[3] = -0.1  
[5] = 0.2
```

```
julia> sparsevec([1, 3, 1, 2, 2], [true, true, false, false,
    ↵   false])
3-element SparseVector{Bool,Int64} with 3 stored entries:
 [1] = true
 [2] = false
 [3] = true
```

### source

```
| sparsevec(d::Dict, [m])
```

Create a sparse vector of length  $m$  where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

### Examples

```
julia> sparsevec(Dict(1 => 3, 2 => 2))
2-element SparseVector{Int64,Int64} with 2 stored entries:
 [1] = 3
 [2] = 2
```

### source

```
| sparsevec(A)
```

Convert a vector  $A$  into a sparse vector of length  $m$ .

### Examples

```
julia> sparsevec([1.0, 2.0, 0.0, 0.0, 3.0, 0.0])
6-element SparseVector{Float64,Int64} with 3 stored entries:
 [1] = 1.0
 [2] = 2.0
 [5] = 3.0
```

[Base.SparseArrays.issparse](#) – Function.

```
| issparse(S)
```

Returns `true` if `S` is sparse, and `false` otherwise.

[source](#)

[Base.SparseArrays.nnz](#) – Function.

```
| nnz(A)
```

Returns the number of stored (filled) elements in a sparse array.

Examples

```
julia> A = sparse(2I, 3, 3)
```

```
3×3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
```

```
[1, 1] = 2
[2, 2] = 2
[3, 3] = 2
```

```
julia> nnz(A)
```

```
3
```

[source](#)

[Base.SparseArrays.spzeros](#) – Function.

```
| spzeros([type, ]m[, n])
```

Create a sparse vector of length `m` or sparse matrix of size `m` × `n`. This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to `Float64` if not specified.

Examples

```
julia> spzeros(3, 3)
```

```
3×3 SparseMatrixCSC{Float64, Int64} with 0 stored entries
```

```
julia> spzeros(Float32, 4)
```

```
4-element SparseVector{Float32, Int64} with 0 stored entries
```

source

[Base.SparseArrays.spones](#) – Function.

```
| spones(S)
```

Create a sparse array with the same structure as that of S, but with every nonzero element having the value 1.0.

Examples

```
julia> A = sparse([1,2,3,4],[2,4,3,1],[5.,4.,3.,2.])
```

```
4×4 SparseMatrixCSC{Float64, Int64} with 4 stored entries:
```

```
[4, 1] = 2.0  
[1, 2] = 5.0  
[3, 3] = 3.0  
[2, 4] = 4.0
```

```
julia> spones(A)
```

```
4×4 SparseMatrixCSC{Float64, Int64} with 4 stored entries:
```

```
[4, 1] = 1.0  
[1, 2] = 1.0  
[3, 3] = 1.0  
[2, 4] = 1.0
```

source

[Base.SparseArrays.spdiagm](#) – Function.

```
| spdiagm(kv::Pair{<:Integer, <:AbstractVector}...)
```

120 Construct a square sparse diagonal matrix from CHAPTER 52 VECTORS AND ARRAYS diagonals. Vector `kv.second` will be placed on the `kv.first` diagonal.

Examples

```
| julia> spdiagm(-1 => [1,2,3,4], 1 => [4,3,2,1])
5×5 SparseMatrixCSC{Int64,Int64} with 8 stored entries:
 [2, 1]  =  1
 [1, 2]  =  4
 [3, 2]  =  2
 [2, 3]  =  3
 [4, 3]  =  3
 [3, 4]  =  2
 [5, 4]  =  4
 [4, 5]  =  1
```

[source](#)

`Base.SparseArrays.sprand` – Function.

```
| sprand([rng],[type],m,[n],p::AbstractFloat,[rfn])
```

Create a random length `m` sparse vector or `m` by `n` sparse matrix, in which the probability of any element being nonzero is independently given by `p` (and hence the mean density of nonzeros is also exactly `p`). Nonzero values are sampled from the distribution specified by `rfn` and have the type `type`. The uniform distribution is used in case `rfn` is not specified. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Examples

```
| julia> rng = MersenneTwister(1234);

| julia> sprand(rng, Bool, 2, 2, 0.5)
```

52.10 SPARSE VECTORS AND MATRICES 1203  
2×2 SparseMatrixCSC{Bool, Int64} with 2 stored entries:

```
[1, 1] = true
[2, 1] = true
```

```
julia> sprand(rng, Float64, 3, 0.75)
```

3-element SparseVector{Float64, Int64} with 1 stored entry:

```
[3] = 0.298614
```

[source](#)

[Base.SparseArrays.sprandn](#) – Function.

```
| sprandn([rng], m[,n],p::AbstractFloat)
```

Create a random sparse vector of length  $m$  or sparse matrix of size  $m$  by  $n$  with the specified (independent) probability  $p$  of any entry being nonzero, where nonzero values are sampled from the normal distribution. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Examples

```
julia> rng = MersenneTwister(1234);
```

```
julia> sprandn(rng, 2, 2, 0.75)
```

2×2 SparseMatrixCSC{Float64, Int64} with 3 stored entries:

```
[1, 1] = 0.532813
[2, 1] = -0.271735
[2, 2] = 0.502334
```

[source](#)

[Base.SparseArrays.nonzeros](#) – Function.

```
| nonzeros(A)
```

1204Return a vector of the structural nonzero values in [CHAPTER 52. SPARSE ARRAYS](#)  
includes zeros that are explicitly stored in the sparse array. The returned  
vector points directly to the internal nonzero storage of A, and any mod-  
ifications to the returned vector will mutate A as well. See [rowvals](#) and  
[nzrange](#).

## Examples

```
julia> A = sparse(2I, 3, 3)
3x3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
 [1, 1] = 2
 [2, 2] = 2
 [3, 3] = 2

julia> nonzeros(A)
3-element Array{Int64,1}:
 2
 2
 2
```

## source

[Base.SparseArrays.rowvals](#) – Function.

```
| rowvals(A::SparseMatrixCSC)
```

Return a vector of the row indices of A. Any modifications to the returned  
vector will mutate A as well. Providing access to how the row indices are  
stored internally can be useful in conjunction with iterating over structural  
nonzero values. See also [nonzeros](#) and [nzrange](#).

## Examples

```
julia> A = sparse(2I, 3, 3)
3x3 SparseMatrixCSC{Int64,Int64} with 3 stored entries:
```

```
[1, 1] = 2  
[2, 2] = 2  
[3, 3] = 2
```

```
julia> rowvals(A)  
3-element Array{Int64,1}:  
1  
2  
3
```

source

[Base.SparseArrays.nzrange](#) – Function.

```
nzrange(A::SparseMatrixCSC, col::Integer)
```

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with [nonzeros](#) and [rowvals](#), this allows for convenient iterating over a sparse matrix :

```
A = sparse(I,J,V)  
rows = rowvals(A)  
vals = nonzeros(A)  
m, n = size(A)  
for i = 1:n  
    for j in nzrange(A, i)  
        row = rows[j]  
        val = vals[j]  
        # perform sparse wizardry...  
    end  
end
```

source

[Base.SparseArrays.dropzeros!](#) – Method.

1206 `dropzeros!(A::SparseMatrixCSC, trim::Bool = true)` CHAPTER 52. ARRAYS

Removes stored numerical zeros from A, optionally trimming resulting excess space from A.`rowval` and A.`nzval` when `trim` is `true`.

For an out-of-place version, see [dropzeros](#). For algorithmic information, see [fkeep!](#).

[source](#)

[Base.SparseArrays.dropzeros](#) – Method.

`dropzeros(A::SparseMatrixCSC, trim::Bool = true)`

Generates a copy of A and removes stored numerical zeros from that copy, optionally trimming excess space from the result's `rowval` and `nzval` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see [dropzeros!](#).

Examples

```
julia> A = sparse([1, 2, 3], [1, 2, 3], [1.0, 0.0, 1.0])
3×3 SparseMatrixCSC{Float64, Int64} with 3 stored entries:
 [1, 1]  =  1.0
 [2, 2]  =  0.0
 [3, 3]  =  1.0
```

```
julia> dropzeros(A)
3×3 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 [1, 1]  =  1.0
 [3, 3]  =  1.0
```

[source](#)

[Base.SparseArrays.dropzeros!](#) – Method.

`dropzeros!(x::SparseVector, trim::Bool = true)`

52.1 REMOVE SPARSE VECTORS AND MATRICES 1207  
cess space from `x.nzind` and `x.nzval` when `trim` is `true`.

For an out-of-place version, see [dropzeros](#). For algorithmic information, see [fkeep!](#).

[source](#)

`Base.SparseArrays.dropzeros` – Method.

```
dropzeros(x::SparseVector, trim::Bool = true)
```

Generates a copy of `x` and removes numerical zeros from that copy, optionally trimming excess space from the result's `nzind` and `nzval` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see [dropzeros!](#).

Examples

```
julia> A = sparsevec([1, 2, 3], [1.0, 0.0, 1.0])
3-element SparseVector{Float64,Int64} with 3 stored entries:
 [1]  =  1.0
 [2]  =  0.0
 [3]  =  1.0

julia> dropzeros(A)
3-element SparseVector{Float64,Int64} with 2 stored entries:
 [1]  =  1.0
 [3]  =  1.0
```

[source](#)

`Base.SparseArrays.permute` – Function.

```
permute(A::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer}
 },
 q::AbstractVector{<:Integer}) where {Tv,Ti}
```

120 Bilaterally permute A, returning PAQ ( $A[p, q]$ ). CHAPTER 52: ARRAYS

length must match A's column count (`length(q) == A.n`). Row-permutation p's length must match A's row count (`length(p) == A.m`).

For expert drivers and additional information, see [permute!](#).

## Examples

```
julia> A = spdiagm(0 => [1, 2, 3, 4], 1 => [5, 6, 7])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
[1, 1] = 1
[1, 2] = 5
[2, 2] = 2
[2, 3] = 6
[3, 3] = 3
[3, 4] = 7
[4, 4] = 4

julia> permute(A, [4, 3, 2, 1], [1, 2, 3, 4])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
[4, 1] = 1
[3, 2] = 2
[4, 2] = 5
[2, 3] = 3
[3, 3] = 6
[1, 4] = 4
[2, 4] = 7

julia> permute(A, [1, 2, 3, 4], [4, 3, 2, 1])
4×4 SparseMatrixCSC{Int64,Int64} with 7 stored entries:
[3, 1] = 7
[4, 1] = 4
[2, 2] = 6
```

```
[3, 2] = 3
[1, 3] = 5
[2, 3] = 2
[1, 4] = 1
```

**source**

[Base.permute!](#) – Method.

```
permute!(X::SparseMatrixCSC{Tv,Ti}, A::SparseMatrixCSC{Tv,Ti},
         p::AbstractVector{<:Integer}, q::AbstractVector{<:
         Integer},
         [C::SparseMatrixCSC{Tv,Ti}]) where {Tv,Ti}
```

Bilaterally permute A, storing result PAQ ( $A[p, q]$ ) in X. Stores intermediate result  $(AQ)^T$  (`transpose(A[:, q])`) in optional argument C if present. Requires that none of X, A, and, if present, C alias each other; to store result PAQ back into A, use the following method lacking X:

```
permute!(A::SparseMatrixCSC{Tv,Ti}, p::AbstractVector{<:Integer
    },
         q::AbstractVector{<:Integer}[, C::SparseMatrixCSC{Tv,
         Ti},
         [workcolptr::Vector{Ti}]]) where {Tv,Ti}
```

X's dimensions must match those of A (`X.m == A.m` and `X.n == A.n`), and X must have enough storage to accommodate all allocated entries in A (`length(X.rowval) >= nnz(A)` and `length(X.nzval) >= nnz(A)`). Column-permutation q's length must match A's column count (`length(q) == A.n`). Row-permutation p's length must match A's row count (`length(p) == A.m`).

C's dimensions must match those of `transpose(A)` (`C.m == A.n` and `C.n == A.m`), and C must have enough storage to accommodate all allo-

located entries in A (`length(C.rowval) >= nnz(A)`) and length of arrays `val`  $\geq \text{nnz}(A)$ .

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `unchecked_noalias_mute!` and `unchecked_aliasing_permute!`.

See also: [permute](#).

[source](#)

## Chapter 53

# Tasks and Parallel Computing

### 53.1 Tasks

`Core.Task` – Type.

```
| Task(func)
```

Create a **Task** (i.e. coroutine) to execute the given function `func` (which must be callable with no arguments). The task exits when this function returns.

Examples

```
| julia> a() = det(rand(1000, 1000));
```

```
| julia> b = Task(a);
```

In this example, `b` is a runnable **Task** that hasn't started yet.

`source`

`Base.current_task` – Function.

```
| current_task()
```

Get the currently running **Task**.

`source`

## Base.istaskdone – Function

CHAPTER 53. TASKS AND PARALLEL COMPUTING

```
| istaskdone(t::Task) -> Bool
```

Determine whether a task has exited.

```
julia> a2() = det(rand(1000, 1000));
```

```
julia> b = Task(a2);
```

```
julia> istaskdone(b)
```

```
false
```

```
julia> schedule(b);
```

```
julia> yield();
```

```
julia> istaskdone(b)
```

```
true
```

`source`

## Base.istaskstarted – Function.

```
| istaskstarted(t::Task) -> Bool
```

Determine whether a task has started executing.

```
julia> a3() = det(rand(1000, 1000));
```

```
julia> b = Task(a3);
```

```
julia> istaskstarted(b)
```

```
false
```

`source`

```
| yield()
```

Switch to the scheduler to allow another scheduled task to run. A task that calls this function is still runnable, and will be restarted immediately if there are no other runnable tasks.

**source**

```
| yield(t::Task, arg = nothing)
```

A fast, unfair-scheduling version of `schedule(t, arg); yield()` which immediately yields to `t` before calling the scheduler.

**source**

Base.`yieldto` – Function.

```
| yieldto(t::Task, arg = nothing)
```

Switch to the given task. The first time a task is switched to, the task's function is called with no arguments. On subsequent switches, `arg` is returned from the task's last call to `yieldto`. This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

**source**

Base.`task_local_storage` – Method.

```
| task_local_storage(key)
```

Look up the value of a key in the current task's task-local storage.

**source**

Base.`task_local_storage` – Method.

```
| task_local_storage(key, value)
```

**source**

[Base.task\\_local\\_storage](#) – Method.

| `task_local_storage(body, key, value)`

Call the function **body** with a modified task-local storage, in which **value** is assigned to **key**; the previous value of **key**, or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

**source**

[Base.Condition](#) – Type.

| `Condition()`

Create an edge-triggered event source that tasks can wait for. Tasks that call **wait** on a **Condition** are suspended and queued. Tasks are woken up when **notify** is later called on the **Condition**. Edge triggering means that only tasks waiting at the time **notify** is called can be woken up. For level-triggered notifications, you must keep extra state to keep track of whether a notification has happened. The [Channel](#) type does this, and so can be used for level-triggered events.

**source**

[Base.notify](#) – Function.

| `notify(condition, val=nothing; all=true, error=false)`

Wake up tasks waiting for a condition, passing them **val**. If **all** is **true** (the default), all waiting tasks are woken, otherwise only one is. If **error** is **true**, the passed value is raised as an exception in the woken tasks.

Returns the count of tasks woken up. Returns 0 if no tasks are waiting on **condition**.

`Base.schedule` – Function.

```
| schedule(t::Task, [val]; error=false)
```

Add a `Task` to the scheduler's queue. This causes the task to run constantly when the system is otherwise idle, unless the task performs a blocking operation such as `wait`.

If a second argument `val` is provided, it will be passed to the task (via the return value of `yieldto`) when it runs again. If `error` is `true`, the value is raised as an exception in the woken task.

```
julia> a5() = det(rand(1000, 1000));
```

```
julia> b = Task(a5);
```

```
julia> istaskstarted(b)
```

```
false
```

```
julia> schedule(b);
```

```
julia> yield();
```

```
julia> istaskstarted(b)
```

```
true
```

```
julia> istaskdone(b)
```

```
true
```

`source`

`Base.@schedule` – Macro.

```
| @schedule
```

1216 Wrap an expression in [CHAPTER 3 TASKS AND PARALLEL COMPUTING](#)

queue. Similar to `@async` except that an enclosing `@sync` does NOT wait for tasks started with an `@schedule`.

[source](#)

`Base.@task` – Macro.

| `@task`

Wrap an expression in a `Task` without executing it, and return the `Task`. This only creates a task, and does not run it.

```
julia> a1() = det(rand(1000, 1000));
```

```
julia> b = @task a1();
```

```
julia> istaskstarted(b)
```

```
false
```

```
julia> schedule(b);
```

```
julia> yield();
```

```
julia> istaskdone(b)
```

```
true
```

[source](#)

`Base.sleep` – Function.

| `sleep(seconds)`

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0.001`.

[source](#)

```
| Channel{T}(sz:Int)
```

Constructs a **Channel** with an internal buffer that can hold a maximum of **sz** objects of type **T**. **put!** calls on a full channel block until an object is removed with **take!**.

**Channel(0)** constructs an unbuffered channel. **put!** blocks until a matching **take!** is called. And vice-versa.

Other constructors:

**Channel(Inf)**: equivalent to **Channel{Any}(typemax(Int))**

**Channel(sz)**: equivalent to **Channel{Any}(sz)**

**source**

**Base.put!** – Method.

```
| put!(c:Channel, v)
```

Appends an item **v** to the channel **c**. Blocks if the channel is full.

For unbuffered channels, blocks until a **take!** is performed by a different task.

**source**

**Base.take!** – Method.

```
| take!(c:Channel)
```

Removes and returns a value from a **Channel**. Blocks until data is available.

For unbuffered channels, blocks until a **put!** is performed by a different task.

**source**

## `Base.isready` – Method. CHAPTER 53. TASKS AND PARALLEL COMPUTING

```
| isready(c::Channel)
```

Determine whether a `Channel` has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns `true` if there are tasks waiting on a `put!`.

`source`

## `Base.fetch` – Method.

```
| fetch(c::Channel)
```

Waits for and gets the first available item from the channel. Does not remove the item. `fetch` is unsupported on an unbuffered (0-size) channel.

`source`

## `Base.close` – Method.

```
| close(c::Channel)
```

Closes a channel. An exception is thrown by:

`put!` on a closed channel.

`take!` and `fetch` on an empty, closed channel.

`source`

## `Base.bind` – Method.

```
| bind(chnl::Channel, task::Task)
```

Associates the lifetime of `chnl` with a task. Channel `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

53.1. The `TASKS` object can be explicitly closed independent of task termination.

Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

Examples

```
julia> c = Channel(0);

julia> task = @schedule foreach(i->put!(c, i), 1:4);

julia> bind(c, task);

julia> for i in c
           @show i

           end;
i = 1
i = 2
i = 3
i = 4

julia> isopen(c)
false

julia> c = Channel(0);

julia> task = @schedule (put!(c,1);error("foo"));

julia> bind(c, task);
```

1220 **julia>** take!(c)

CHAPTER 53. TASKS AND PARALLEL COMPUTING

1

**julia>** put!(c, 1);

ERROR: foo

Stacktrace:

[ ... ]

**source**

[Base.asyncmap](#) – Function.

| `asyncmap(f, c...; ntasks=0, batch_size=nothing)`

Uses multiple concurrent tasks to map `f` over a collection (or multiple equal length collections). For multiple collection arguments, `f` is applied elementwise.

`ntasks` specifies the number of tasks to run concurrently. Depending on the length of the collections, if `ntasks` is unspecified, up to 100 tasks will be used for concurrent mapping.

`ntasks` can also be specified as a zero-arg function. In this case, the number of tasks to run in parallel is checked before processing every element and a new task started if the value of `ntasks_func` is less than the current number of tasks.

If `batch_size` is specified, the collection is processed in batch mode. `f` must then be a function that must accept a `Vector` of argument tuples and must return a vector of results. The input vector will have a length of `batch_size` or less.

The following examples highlight execution in different tasks by returning the `object_id` of the tasks in which the mapping function is executed.

53.1 First, with `ntasks` undefined, each element is processed in a different task.

```
julia> tskoid() = object_id(current_task());  
  
julia> asyncmap(x->tskoid(), 1:5)  
5-element Array{UInt64,1}:  
    0x6e15e66c75c75853  
    0x440f8819a1baa682  
    0x9fb3eeadd0c83985  
    0xebd3e35fe90d4050  
    0x29efc93edce2b961  
  
julia> length(unique(asyncmap(x->tskoid(), 1:5)))  
5
```

With `ntasks=2` all elements are processed in 2 tasks.

```
julia> asyncmap(x->tskoid(), 1:5; ntasks=2)  
5-element Array{UInt64,1}:  
    0x027ab1680df7ae94  
    0xa23d2f80cd7cf157  
    0x027ab1680df7ae94  
    0xa23d2f80cd7cf157  
    0x027ab1680df7ae94  
  
julia> length(unique(asyncmap(x->tskoid(), 1:5; ntasks=2)))  
2
```

With `batch_size` defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. `map` is used in the modified mapping function to achieve this.

```
julia> batch_func(input) = map(x->string("args_tuple: ", x, ", "  
    "element_val: ", x[1], ", task: ", tskoid()), input)
```

1222 batch\_func (generic function with 1 method) CHAPTER 53. TASKS AND PARALLEL COMPUTING

```
julia> asyncmap(batch_func, 1:5; ntasks=2, batch_size=2)
5-element Array{String,1}:
 "args_tuple: (1,), element_val: 1, task: 9118321258196414413"
 "args_tuple: (2,), element_val: 2, task: 4904288162898683522"
 "args_tuple: (3,), element_val: 3, task: 9118321258196414413"
 "args_tuple: (4,), element_val: 4, task: 4904288162898683522"
 "args_tuple: (5,), element_val: 5, task: 9118321258196414413"
```

### Note

Currently, all tasks in Julia are executed in a single OS thread cooperatively. Consequently, `asyncmap` is beneficial only when the mapping function involves any I/O – disk, network, remote worker invocation, etc.

### source

[Base.asyncmap!](#) – Function.

```
| asyncmap!(f, results, c...; ntasks=0, batch_size=nothing)
```

Like `asyncmap`, but stores output in `results` rather than returning a collection.

### source

## 53.2 General Parallel Computing Support

[Base.Distributed.addprocs](#) – Function.

```
| addprocs(manager::ClusterManager; kwargs...) -> List of process
| identifiers
```

Launches worker processes via the specified cluster manager.

For example, `addprocs` can be used to support via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

#### `source`

```
| addprocs(machines; tunnel=false, sshflags=``, max_parallel=10,
|         kwargs...) -> List of process identifiers
```

Add processes on remote machines via SSH. Requires `julia` to be installed in the same location on each node, or to be available via a shared file system.

`machines` is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string `machine_spec` or a tuple - `(machine_spec, count)`.

`machine_spec` is a string of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

`count` is the number of workers to be launched on the specified host. If specified as `:auto` it will launch as many workers as the number of cores on the specific host.

Keyword arguments:

`tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.

1224    **sshflags**: specifies CHAPTER 5: SSH TUNNELS AND PARALLEL COMPUTING  
  `- /foo/bar.pem'

**max\_parallel**: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.

**dir**: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)

**enable\_threaded\_blas**: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.

**exename**: name of the `julia` executable. Defaults to "`$JULIA_HOME/julia`" or "`$JULIA_HOME/julia-debug`" as the case may be.

**exeflags**: additional flags passed to the worker processes.

**topology**: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.

- **topology=:all\_to\_all**: All processes are connected to each other. The default.
- **topology=:master\_slave**: Only the driver process, i.e. `pid 1` connects to the workers. The workers do not connect to each other.
- **topology=:custom**: The `launch` method of the cluster manager specifies the connection topology via fields `ident` and `connect_ids` in `WorkerConfig`. A worker with a cluster manager identity `ident` will connect to all workers specified in `connect_ids`.

**lazy**: Applicable only with `topology=:all_to_all`. If `true`, worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is `true`.

Environment variables :

If the master process fails to establish a connection with a newly launched worker within 60.0 seconds, the worker treats it as a fatal situation and

`LIA_WORKER_TIMEOUT`. The value of `JULIA_WORKER_TIMEOUT` on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

**source**

```
| addprocs(; kwargs...) -> List of process identifiers
```

Equivalent to `addprocs(Sys.CPU_CORES; kwargs...)`

Note that workers do not run a `.juliarc.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

**source**

```
| addprocs(np::Integer; restrict=true, kwargs...) -> List of
|   process identifiers
```

Launches workers using the in-built `LocalManager` which only launches workers on the local host. This can be used to take advantage of multiple cores. `addprocs(4)` will add 4 processes on the local machine. If `restrict` is `true`, binding is restricted to `127.0.0.1`. Keyword args `dir`, `exename`, `exeflags`, `topology`, `lazy` and `enable_threaded_blas` have the same effect as documented for `addprocs(machines)`.

**source**

[Base.Distributed.nprocs](#) – Function.

```
| nprocs()
```

Get the number of available processes.

**source**

[Base.Distributed.nworkers](#) – Function.

1226 `nworkers()`

## CHAPTER 53. TASKS AND PARALLEL COMPUTING

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

`source`

`Base.Distributed.procs` – Method.

| `procs()`

Returns a list of all process identifiers.

`source`

`Base.Distributed.procs` – Method.

| `procs(pid::Integer)`

Returns a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as `pid` are returned.

`source`

`Base.Distributed.workers` – Function.

| `workers()`

Returns a list of all worker process identifiers.

`source`

`Base.Distributed.rmparts` – Function.

| `rmparts(pids...; waitfor=typemax(Int))`

Removes the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down: - If unspecified, `rmparts` will wait until all requested `pids` are

53.2 GENERAL API FOR EXECUTING SUPPORT 1227

nated before the requested `waitfor` seconds. - With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

**source**

[Base.Distributed.interrupt](#) – Function.

```
| interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

**source**

```
| interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

**source**

[Base.Distributed.myid](#) – Function.

```
| myid()
```

Get the id of the current process.

**source**

[Base.Distributed.pmap](#) – Function.

```
| pmap([::AbstractWorkerPool], f, c...; distributed=true,
      batch_size=1, on_error=nothing, retry_delays=[], retry_check
      =nothing) -> collection
```

workers and tasks.

For multiple collection arguments, apply `f` elementwise.

Note that `f` must be made available to all worker processes; see [Code Availability and Loading Packages](#) for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, `pmap` distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using `asyncmap`. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f, c; ntasks=()->nworkers())`

`pmap` can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops `pmap` from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4;
   ↵   on_error=identity)
```

```
| 4-element Array{Any,1}:
|   1
|  ErrorException("foo")
|   3
|  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4;
→  on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
 3
 0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `f` on an element a maximum of 3 times without any delay between retries.

```
| pmap(f, c; retry_delays = zeros(3))
```

Example: Retry `f` only if the exception is not of type `InexactError`, with exponentially increasing delays up to 3 times. Return a `NaN` in place for all `InexactError` occurrences.

```
| pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN :
→   rethrow(e)), retry_delays = ExponentialBackOff(n = 3))
```

1230 **source**

## CHAPTER 53. TASKS AND PARALLEL COMPUTING

**Base.Distributed.RemoteException** – Type.

| **RemoteException(captured)**

Exceptions on remote computations are captured and rethrown locally. A **RemoteException** wraps the **pid** of the worker and a captured exception. A **CapturedException** captures the remote exception and a serializable form of the call stack when the exception was raised.

**source**

**Base.Distributed.Future** – Type.

| **Future(pid::Integer=myid())**

Create a **Future** on process **pid**. The default **pid** is the current process.

**source**

**Base.Distributed.RemoteChannel** – Method.

| **RemoteChannel(pid::Integer=myid())**

Make a reference to a **Channel{Any}(1)** on process **pid**. The default **pid** is the current process.

**source**

**Base.Distributed.RemoteChannel** – Method.

| **RemoteChannel(f::Function, pid::Integer=myid())**

Create references to remote channels of a specific size and type. **f** is a function that when executed on **pid** must return an implementation of an **AbstractChannel**.

For example, **RemoteChannel(()->Channel{Int}(10), pid)**, will return a reference to a channel of type **Int** and size 10 on **pid**.

[source](#)

`Base.wait` – Function.

| `wait([x])`

Block the current task until some event occurs, depending on the type of the argument:

`RemoteChannel`: Wait for a value to become available on the specified remote channel.

`Future`: Wait for a value to become available for the specified future.

`Channel`: Wait for a value to be appended to the channel.

`Condition`: Wait for `notify` on a condition.

`Process`: Wait for a process or process chain to exit. The `exitcode` field of a process can be used to determine success or failure.

`Task`: Wait for a `Task` to finish, returning its result value. If the task fails with an exception, the exception is propagated (re-thrown in the task that called `wait`).

`RawFD`: Wait for changes on a file descriptor (see the `FileWatching` package).

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to `schedule` or `yieldto`.

Often `wait` is called within a `while` loop to ensure a waited-for condition is met before proceeding.

[source](#)

`Base.fetch` – Method.

| `fetch(x)`

**Future**: Wait for and get the value of a Future. The fetched value is cached locally. Further calls to **fetch** on the same reference return the cached value. If the remote value is an exception, throws a **RemoteException** which captures the remote exception and backtrace.

**RemoteChannel**: Wait for and get the value of a remote reference. Exceptions raised are same as for a Future .

Does not remove the item fetched.

**source**

[Base.Distributed.remotecall](#) – Method.

```
| remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function **f** asynchronously on the given arguments on the specified process. Returns a **Future**. Keyword arguments, if any, are passed through to **f**.

**source**

[Base.Distributed.remotecall\\_wait](#) – Method.

```
| remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster `wait(remotecall(...))` in one message on the **Worker** specified by worker id **id**. Keyword arguments, if any, are passed through to **f**.

See also [wait](#) and [remotecall](#).

**source**

[Base.Distributed.remotecall\\_fetch](#) – Method.

```
| remotecall_fetch(f, id::Integer, args...; kwargs...)
```

53.2 PERFORMANT PARALLEL COMPUTING SUPPORT message. Keyword arguments, if any, are passed through to `f`. Any remote exceptions are captured in a `RemoteException` and thrown.

See also `fetch` and `remotecall`.

`source`

`Base.Distributed.remote_do` – Method.

```
| remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes `f` on worker `id` asynchronously. Unlike `remotecall`, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive `remotecalls` to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2)` will serialize the call to `f1`, followed by `f2` and `f3` in that order. However, it is not guaranteed that `f1` is executed before `f3` on worker 2.

Any exceptions thrown by `f` are printed to `STDERR` on the remote worker.

Keyword arguments, if any, are passed through to `f`.

`source`

`Base.put!` – Method.

```
| put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Returns its first argument.

`source`

`Base.put!` – Method. CHAPTER 53. TASKS AND PARALLEL COMPUTING

```
| put!(rr::Future, v)
```

Store a value to a `Future` `rr`. `Futures` are write-once remote references. A `put!` on an already set `Future` throws an `Exception`. All asynchronous remote calls return `Futures` and set the value to the return value of the call upon completion.

`source`

`Base.take!` – Method.

```
| take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a `RemoteChannel` `rr`, removing the value(s) in the process.

`source`

`Base.isready` – Method.

```
| isready(rr::RemoteChannel, args...)
```

Determine whether a `RemoteChannel` has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a `Future` since they are assigned only once.

`source`

`Base.isready` – Method.

```
| isready(rr::Future)
```

Determine whether a `Future` has a value stored to it.

If the argument `Future` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `rr` in a separate task instead or to use a local `Channel` as a proxy:

53.2 c GENERAL PARALLEL COMPUTING SUPPORT

1235

```
|c = Channel(1)  
|@async put!(c, remotecall_fetch(long_computation, p))  
|isready(c) # will not block
```

**source**

[Base.Distributed.WorkerPool](#) – Type.

```
|WorkerPool(workers::Vector{Int})
```

Create a WorkerPool from a vector of worker ids.

**source**

[Base.Distributed.CachingPool](#) – Type.

```
|CachingPool(workers::Vector{Int})
```

An implementation of an [AbstractWorkerPool](#). [remote](#), [remotecall\\_fetch](#), [pmap](#) (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned CachingPool object. To clear the cache earlier, use [clear!\(pool\)](#).

For global variables, only the bindings are captured in a closure, not the data. [let](#) blocks can be used to capture global data.

Examples

```
const foo = rand(10^8);  
wp = CachingPool(workers())  
let foo = foo  
    pmap(wp, i -> sum(foo) + i, 1:100);  
end
```

The above would transfer `foo` only once to each worker.

1236 `source`

## CHAPTER 53. TASKS AND PARALLEL COMPUTING

`Base.Distributed.default_worker_pool` – Function.

```
| default_worker_pool()
```

`WorkerPool` containing idle workers – used by `remote(f)` and `pmap` (by default).

`source`

`Base.Distributed.clear!` – Method.

```
| clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

`source`

`Base.Distributed.remote` – Function.

```
| remote(::AbstractWorkerPool], f) -> Function
```

Returns an anonymous function that executes function `f` on an available worker using `remotecall_fetch`.

`source`

`Base.Distributed.remotecall` – Method.

```
| remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) ->
|   Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall` on it.

`source`

`Base.Distributed.remotecall_wait` – Method.

```
| remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs
|   ...) -> Future
```

53.2 WORKER API ABSTRACT COMPUTING SUPPORT (f, pid, ....). Waits<sup>1287</sup> and takes a free worker from pool and performs a remotecall\_wait on it.

**source**

[Base.Distributed.remotecall\\_fetch](#) – Method.

```
| remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs  
|   ...) -> result
```

WorkerPool variant of remotecall\_fetch(f, pid, ....). Waits for and takes a free worker from pool and performs a remotecall\_fetch on it.

**source**

[Base.Distributed.remote\\_do](#) – Method.

```
| remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) ->  
|   nothing
```

WorkerPool variant of remote\_do(f, pid, ....). Waits for and takes a free worker from pool and performs a remote\_do on it.

**source**

[Base.timedwait](#) – Function.

```
| timedwait(testcb::Function, secs::Float64; pollint::Float64  
|   =0.1)
```

Waits until testcb returns true or for secs seconds, whichever is earlier. testcb is polled every pollint seconds.

**source**

[Base.Distributed.@spawn](#) – Macro.

1238 @spawn

## CHAPTER 53. TASKS AND PARALLEL COMPUTING

Create a closure around an expression and run it on an automatically-chosen process, returning a [Future](#) to the result.

Examples

```
julia> addprocs(3);  
  
julia> f = @spawn myid()  
Future(2, 1, 5, Nullable{Any}())
```

```
julia> fetch(f)  
2
```

```
julia> f = @spawn myid()  
Future(3, 1, 7, Nullable{Any}())
```

```
julia> fetch(f)  
3
```

`source`

[Base.Distributed.@spawnat](#) – Macro.

```
@spawnat
```

Create a closure around an expression and run the closure asynchronously on process p. Returns a [Future](#) to the result. Accepts two arguments, p and an expression.

Examples

```
julia> addprocs(1);  
  
julia> f = @spawnat 2 myid()
```

```
julia> fetch(f)
```

```
2
```

```
source
```

Base.Distributed.@fetch – Macro.

```
@fetch
```

Equivalent to `fetch(@spawn expr)`. See [fetch](#) and [@spawn](#).

Examples

```
julia> addprocs(3);
```

```
julia> @fetch myid()
```

```
2
```

```
julia> @fetch myid()
```

```
3
```

```
julia> @fetch myid()
```

```
4
```

```
julia> @fetch myid()
```

```
2
```

```
source
```

Base.Distributed.@fetchfrom – Macro.

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See [fetch](#) and [@spawnat](#).

Examples

1240 **julia>** addprocs(3); CHAPTER 53. TASKS AND PARALLEL COMPUTING

```
| julia> @fetchfrom 2 myid()
```

```
| 2
```

```
| julia> @fetchfrom 4 myid()
```

```
| 4
```

source

[Base.@async](#) – Macro.

```
| @async
```

Like `@schedule`, `@async` wraps an expression in a `Task` and adds it to the local machine's scheduler queue. Additionally it adds the task to the set of items that the nearest enclosing `@sync` waits for.

source

[Base.@sync](#) – Macro.

```
| @sync
```

Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete. All exceptions thrown by enclosed `async` operations are collected and thrown as a `CompositeException`.

source

[Base.Distributed.@parallel](#) – Macro.

```
| @parallel
```

A parallel for loop of the form :

```
| @parallel [reducer] for var = range
```

```
|   body
```

```
| end
```

In case an optional reducer function is specified, `@parallel` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@parallel` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with `@sync`, like :

```
|@sync @parallel for var = range
|    body
|end
```

`source`

`Base.Distributed.@everywhere` – Macro.

```
|@everywhere [procs()] expr
```

Execute an expression under `Main` on all `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown. For example:

```
|@everywhere bar = 1
```

will define `Main.bar` on all processes.

Unlike `@spawn` and `@spawnat`, `@everywhere` does not capture any local variables. Instead, local variables can be broadcast using interpolation:

```
|foo = 1
|@everywhere bar = $foo
```

The optional argument `procs` allows specifying a subset of all processes to have execute the expression.

Equivalent to calling `remotecall_eval(Main, procs, expr)`.

`source`

```
| clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbols`. `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

`source`

[Base.Distributed.remoteref\\_id](#) – Function.

```
| Base.remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` are identified by fields:

`where` – refers to the node where the underlying object/storage referred to by the reference actually exists.

`whence` – refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.

`id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`Base.remoteref_id` is a low-level API which returns a `Base.RRID` object that wraps `whence` and `id` values of a remote reference.

`source`

| Base.channel\_from\_id(id) -> c

A low-level API which returns the backing `AbstractChannel` for an `id` returned by `remoteref_id`. The call is valid only on the node where the backing channel exists.

`source`

[Base.Distributed.worker\\_id\\_from\\_socket](#) – Function.

| Base.worker\_id\_from\_socket(s) -> pid

A low-level API which given a `I0` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom `serialize` methods for a type, which optimizes the data written out depending on the receiving process id.

`source`

[Base.Distributed.cluster\\_cookie](#) – Method.

| Base.cluster\_cookie() -> cookie

Returns the cluster cookie.

`source`

[Base.Distributed.cluster\\_cookie](#) – Method.

| Base.cluster\_cookie(cookie) -> cookie

Sets the passed cookie as the cluster cookie, then returns it.

`source`

## 5243 Multi-Threading CHAPTER 53. TASKS AND PARALLEL COMPUTING

This experimental interface supports Julia's multi-threading capabilities. Types and functions described here might (and likely will) change in the future.

[Base.Threads.threadid](#) – Function.

```
| Threads.threadid()
```

Get the ID number of the current thread of execution. The master thread has ID 1.

[source](#)

[Base.Threads.nthreads](#) – Function.

```
| Threads.nthreads()
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `threadid()`.

[source](#)

[Base.Threads.@threads](#) – Macro.

```
| Threads.@threads
```

A macro to parallelize a for-loop to run with multiple threads. This spawns `nthreads()` number of threads, splits the iteration space amongst them, and iterates in parallel. A barrier is placed at the end of the loop which waits for all the threads to finish execution, and the loop returns.

[source](#)

[Base.Threads.Atomic](#) – Type.

```
| Threads.Atomic{T}
```

Holds a reference to an object of type T, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

53.3.0 ~~MULTITHREADING~~<sup>1245</sup> Types can be used atomically, namely the primitive integer and float-point types. These are `Int8...Int128`, `UInt8...UInt128`, and `Float16...Float64`.

New atomic objects can be created from a non-atomic values; if none is specified, the atomic object is initialized with zero.

Atomic objects can be accessed using the `[ ]` notation:

Examples

```
julia> x = Threads.Atomic{Int}(3)  
Base.Threads.Atomic{Int64}(3)
```

```
julia> x[] = 1  
1
```

```
julia> x[]  
1
```

Atomic operations use an `atomic_` prefix, such as `atomic_add!`, `atomic_xchg!`, etc.

`source`

`Base.Threads.atomic_cas!` – Function.

```
| Threads.atomic_cas!(x::Atomic{T}, cmp::T, newval::T) where T
```

Atomically compare-and-set `x`

Atomically compares the value in `x` with `cmp`. If equal, write `newval` to `x`. Otherwise, leaves `x` unmodified. Returns the old value in `x`. By comparing the returned value to `cmp` (via `==`) one knows whether `x` was modified and now holds the new value `newval`.

For further details, see LLVM's `cmpxchg` instruction.

the transaction, one records the value in `x`. After the transaction, the new value is stored only if `x` has not been modified in the meantime.

### Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 4, 2);

julia> x
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_cas!(x, 3, 2);

julia> x
Base.Threads.Atomic{Int64}(2)
```

### source

`Base.Threads.atomic_xchg!` – Function.

`| Threads.atomic_xchg!(x::Atomic{T}, newval::T) where T`

Atomically exchange the value in `x`

Atomically exchanges the value in `x` with `newval`. Returns the old value.

For further details, see LLVM's `atomicrmw_xchg` instruction.

### Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)

julia> Threads.atomic_xchg!(x, 2)
```

```
julia> x[ ]
```

```
2
```

source

[Base.Threads.atomic\\_add!](#) – Function.

```
| Threads.atomic_add!(x::Atomic{T}, val::T) where T
```

Atomically add `val` to `x`

Performs `x[] += val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw add` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
```

```
Base.Threads.Atomic{Int64}(3)
```

```
julia> Threads.atomic_add!(x, 2)
```

```
3
```

```
julia> x[ ]
```

```
5
```

source

[Base.Threads.atomic\\_sub!](#) – Function.

```
| Threads.atomic_sub!(x::Atomic{T}, val::T) where T
```

Atomically subtract `val` from `x`

Performs `x[] -= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw sub` instruction.

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)
```

```
julia> Threads.atomic_sub!(x, 2)
3
```

```
julia> x[]
1
```

source

[Base.Threads.atomic\\_and!](#) – Function.

```
| Threads.atomic_and!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-and `x` with `val`

Performs `x[] &= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw` and instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)
```

```
julia> Threads.atomic_and!(x, 2)
3
```

```
julia> x[]
2
```

source

[Base.Threads.atomic\\_nand!](#) – Function.

Threads.atomic\_nand!(x::Atomic{T}, val::T) where T

Atomically bitwise-and (not-and) x with val

Performs  $x[] = \sim(x[] \& val)$  atomically. Returns the old value.

For further details, see LLVM's `atomicrmw nand` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(3)
Base.Threads.Atomic{Int64}(3)
```

```
julia> Threads.atomic_nand!(x, 2)
3
```

```
julia> x[]
-3
```

source

Base.Threads.atomic\_or! – Function.

Threads.atomic\_or!(x::Atomic{T}, val::T) where T

Atomically bitwise-or x with val

Performs  $x[] |= val$  atomically. Returns the old value.

For further details, see LLVM's `atomicrmw or` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
Base.Threads.Atomic{Int64}(5)
```

```
julia> Threads.atomic_or!(x, 7)
5
```

```
julia> x[ ]
```

```
7
```

source

`Base.Threads.atomic_xor!` – Function.

```
| Threads.atomic_xor!(x::Atomic{T}, val::T) where T
```

Atomically bitwise-xor (exclusive-or) `x` with `val`

Performs `x[] $= val` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw xor` instruction.

Examples

```
julia> x = Threads.Atomic{Int}(5)
```

```
Base.Threads.Atomic{Int64}(5)
```

```
julia> Threads.atomic_xor!(x, 7)
```

```
5
```

```
julia> x[ ]
```

```
2
```

source

`Base.Threads.atomic_max!` – Function.

```
| Threads.atomic_max!(x::Atomic{T}, val::T) where T
```

Atomically store the maximum of `x` and `val` in `x`

Performs `x[] = max(x[], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw max` instruction.

Examples

53.3 | **MULTI-THREADING** `julia> x = Threads.Atomic{Int}(5)`

1251

`Base.Threads.Atomic{Int64}(5)`

`julia> Threads.atomic_max!(x, 7)`

5

`julia> x[ ]`

7

`source`

`Base.Threads.atomic_min!` – Function.

`| Threads.atomic_min!(x::Atomic{T}, val::T) where T`

Atomically store the minimum of `x` and `val` in `x`

Performs `x[ ] = min(x[ ], val)` atomically. Returns the old value.

For further details, see LLVM's `atomicrmw min` instruction.

Examples

`julia> x = Threads.Atomic{Int}(7)`

`Base.Threads.Atomic{Int64}(7)`

`julia> Threads.atomic_min!(x, 5)`

7

`julia> x[ ]`

5

`source`

`Base.Threads.atomic_fence` – Function.

`| Threads.atomic_fence()`

Inserts a memory fence with sequentially-consistent ordering semantics.

There are algorithms where this is needed, i.e. where an acquire/release ordering is insufficient.

This is likely a very expensive operation. Given that all other atomic operations in Julia already have acquire/release semantics, explicit fences should not be necessary in most cases.

For further details, see LLVM's `fence` instruction.

[source](#)

## 53.4 ccall using a threadpool (Experimental)

`Base.@threadcall` – Macro.

```
|@threadcall((cfunc, clib), rettype, (argtypes...), argvals...)
```

The `@threadcall` macro is called in the same way as `ccall` but does the work in a different thread. This is useful when you want to call a blocking C function without causing the main `julia` thread to become blocked. Concurrency is limited by size of the libuv thread pool, which defaults to 4 threads but can be increased by setting the `UV_THREADPOOL_SIZE` environment variable and restarting the `julia` process.

Note that the called function should never call back into Julia.

[source](#)

## 53.5 Synchronization Primitives

`Base.Threads.AbstractLock` – Type.

```
|AbstractLock
```

53.5 ABSYNCHRONIZATION PRIMITIVES  
Abstract synchronization primitives that implement the thread-safe synchronization primitives: **lock**, **trylock**, **unlock**, and **islocked**

[source](#)

**Base.lock** – Function.

| **lock(the\_lock)**

Acquires the lock when it becomes available. If the lock is already locked by a different task/thread, it waits for it to become available.

Each **lock** must be matched by an **unlock**.

[source](#)

**Base.unlock** – Function.

| **unlock(the\_lock)**

Releases ownership of the lock.

If this is a recursive lock which has been acquired before, it just decrements an internal counter and returns immediately.

[source](#)

**Base.trylock** – Function.

| **trylock(the\_lock) -> Success (Boolean)**

Acquires the lock if it is available, returning **true** if successful. If the lock is already locked by a different task/thread, returns **false**.

Each successful **trylock** must be matched by an **unlock**.

[source](#)

**Base.islocked** – Function.

| **islocked(the\_lock) -> Status (Boolean)**

1254 Check whether the lock used for synchronization (see instead `trylock`).

[source](#)

`Base.ReentrantLock` – Type.

`| ReentrantLock()`

Creates a reentrant lock for synchronizing Tasks. The same task can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

This lock is NOT threadsafe. See `Threads.Mutex` for a threadsafe lock.

[source](#)

`Base.Threads.Mutex` – Type.

`| Mutex()`

These are standard system mutexes for locking critical sections of logic.

On Windows, this is a critical section object, on pthreads, this is a `pthread_mutex_t`.

See also `SpinLock` for a lighter-weight lock.

[source](#)

`Base.Threads.SpinLock` – Type.

`| SpinLock()`

Creates a non-reentrant lock. Recursive use will result in a deadlock. Each `lock` must be matched with an `unlock`.

Test-and-test-and-set spin locks are quickest up to about 30ish contending threads. If you have more contention than that, perhaps a lock is the wrong way to synchronize.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

[source](#)

[Base.Threads.RecursiveSpinLock](#) – Type.

| `RecursiveSpinLock()`

Creates a reentrant lock. The same thread can acquire the lock as many times as required. Each `lock` must be matched with an `unlock`.

See also SpinLock for a slightly faster version.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

[source](#)

[Base.Semaphore](#) – Type.

| `Semaphore(sem_size)`

Creates a counting semaphore that allows at most `sem_size` acquires to be in use at any time. Each acquire must be matched with a release.

This construct is NOT threadsafe.

[source](#)

[Base.acquire](#) – Function.

| `acquire(s::Semaphore)`

Wait for one of the `sem_size` permits to be available, blocking until one can be acquired.

[source](#)

[Base.release](#) – Function.

```
1256 release(s)::Semaphore) CHAPTER 53. TASKS AND PARALLEL COMPUTING
```

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

[source](#)

## 53.6 Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in `Base`: `LocalManager`, for launching additional workers on the same host, and `SSHManager`, for launching on remote hosts via `ssh`. TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

[`Base.Distributed.launch`](#) – Function.

```
launch(manager::ClusterManager, params::Dict, launched::Array,  
       launch_ntfy::Condition)
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

[source](#)

[`Base.Distributed.manage`](#) – Function.

```
manage(manager::ClusterManager, id::Integer, config::  
       WorkerConfig, op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate `op` values:

53.6. ~~CLUSTER MANAGER INTERFACE~~<sup>1257</sup> when a worker is added / removed from the Julia worker pool.

with :interrupt when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.

with :finalize for cleanup purposes.

`source`

`Base.kill` – Method.

`| kill(manager::ClusterManager, pid::Int, config::WorkerConfig)`

Implemented by cluster managers. It is called on the master process, by `rmprefs`. It should cause the remote worker specified by `pid` to exit. `kill(manager::ClusterManager.....)` executes a remote `exit()` on `pid`.

`source`

`Base.connect` – Method.

`| connect(manager::ClusterManager, pid::Int, config::WorkerConfig  
| ) -> (instrm::IO, outstrm::IO)`

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id `pid`, specified by `config` and return a pair of `IO` objects. Messages from `pid` to current process will be read off `instrm`, while messages to be sent to `pid` will be written to `outstrm`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `connect(manager::ClusterManager.....)` sets up TCP/IP socket connections in-between workers.

`source`

## Base.Distributed.init\_worker – Function

```
init_worker(cookie::AbstractString, manager::ClusterManager=
           DefaultClusterManager())
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--worker[=<cookie>]` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `cookie` is a [cluster\\_cookie](#).

[source](#)

## Base.Distributed.start\_worker – Function.

```
start_worker(out::IO=STDOUT)
start_worker(cookie::AbstractString)
start_worker(out::IO, cookie::AbstractString)
```

`Base.start_worker` is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

If the cookie is unspecified, the worker tries to read it from its STDIN.

host:port information is written to stream `out` (defaults to STDOUT).

The function closes STDIN (after reading the cookie if required), redirects STDERR to STDOUT, listens on a free port (or if specified, the port in the `--bind-to` command line option) and schedules tasks to process incoming TCP connections and requests.

It does not return.

[source](#)

## Base.Distributed.process\_messages – Function.

```
Base.process_messages(r_stream::IO, w_stream::IO, incoming::
                      Bool=true)
```

53.6. Called by `CLUSTER_MANAGER_INTERFACE` custom transports. It should be called  
when the custom transport implementation receives the first message from  
a remote worker. The custom transport must manage a logical connec-  
tion to the remote worker and provide two `I0` objects, one for incoming  
messages and the other for messages addressed to the remote worker.  
If `incoming` is `true`, the remote peer initiated the connection. Whichever  
of the pair initiates the connection sends the cluster cookie and its Julia  
version number to perform the authentication handshake.

See also [cluster\\_cookie](#).

[source](#)



# Chapter 54

## Linear Algebra

### 54.1 Standard Functions

Linear algebra functions in Julia are largely implemented by calling functions from [LAPACK](#). Sparse factorizations call functions from [SuiteSparse](#).

`Base. :*` – Method.

```
| *(A::AbstractMatrix, B::AbstractMatrix)
```

Matrix multiplication.

Examples

```
julia> [1 1; 0 1] * [1 0; 1 1]
2×2 Array{Int64,2}:
 2  1
 1  1
```

`source`

```
| *(x, y...)
```

Multiplication operator. `x*y*z*...` calls this function with all arguments, i.e. `*(x, y, z, ...)`.

Examples

```
1262 julia> 2 * 7 * 8
```

## CHAPTER 54. LINEAR ALGEBRA

```
112
```

```
julia> *(2, 7, 8)
```

```
112
```

**source**

**Base.:\ - Method.**

```
\(A, B)
```

Matrix division using a polyalgorithm. For input matrices  $A$  and  $B$ , the result  $X$  is such that  $A*X == B$  when  $A$  is square. The solver that is used depends upon the structure of  $A$ . If  $A$  is upper or lower triangular (or diagonal), no factorization of  $A$  is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.

For rectangular  $A$  the result is the minimum-norm least squares solution computed by a pivoted QR factorization of  $A$  and a rank estimate of  $A$  based on the R factor.

When  $A$  is sparse, a similar polyalgorithm is used. For indefinite matrices, the `LDLt` factorization does not use pivoting during the numerical factorization and therefore the procedure can fail even for invertible matrices.

Examples

```
julia> A = [1 0; 1 -2]; B = [32; -4];
```

```
julia> X = A \ B
```

```
2-element Array{Float64,1}:
```

```
 32.0
```

```
 18.0
```

```
julia> A * X == B  
true
```

[source](#)

[Base.LinAlg.dot](#) – Function.

```
dot(x, y)  
(x, y)
```

Compute the dot product between two vectors. For complex vectors, the first vector is conjugated. When the vectors have equal lengths, calling `dot` is semantically equivalent to `sum(vx'vy for (vx,vy) in zip(x,y))`.

Examples

```
julia> dot([1; 1], [2; 3])  
5  
  
julia> dot([im; im], [1; 1])  
0 - 2im
```

[source](#)

[Base.LinAlg.vecdot](#) – Function.

```
vecdot(x, y)
```

For any iterable containers `x` and `y` (including arrays of any dimension) of numbers (or any element type for which `dot` is defined), compute the Euclidean dot product (the sum of `dot(x[i],y[i])`) as if they were vectors.

Examples

```
1264 julia> vecdot(1:5, 2:6)  
70
```

## CHAPTER 54. LINEAR ALGEBRA

```
julia> x = fill(2., (5,5));  
  
julia> y = fill(3., (5,5));  
  
julia> vecdot(x, y)  
150.0
```

source

[Base.LinAlg.cross](#) – Function.

```
| cross(x, y)  
| x(x,y)
```

Compute the cross product of two 3-vectors.

Examples

```
julia> a = [0;1;0]  
3-element Array{Int64,1}:  
 0  
 1  
 0
```

```
julia> b = [0;0;1]  
3-element Array{Int64,1}:  
 0  
 0  
 1
```

```
julia> cross(a,b)
```

```
1  
0  
0
```

source

[Base.LinAlg.factorize](#) – Function.

```
| factorize(A)
```

Compute a convenient factorization of A, based upon the type of the input matrix. `factorize` checks A to see if it is symmetric/triangular/etc. if A is passed as a generic matrix. `factorize` checks every element of A to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example: `A=factorize(A); x=A\b; y=A\C.`

Properties of A	type of factorization
Positive-definite	Cholesky (see <a href="#">cholfact</a> )
Dense Symmetric/Hermitian	Bunch-Kaufman (see <a href="#">bkfact</a> )
Sparse Symmetric/Hermitian	LDLt (see <a href="#">ldltfact</a> )
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see <a href="#">lufact</a> )
Symmetric real tridiagonal	LDLt (see <a href="#">ldltfact</a> )
General square	LU (see <a href="#">lufact</a> )
General non-square	QR (see <a href="#">qrfact</a> )

If `factorize` is called on a Hermitian positive-definite matrix, for instance, then `factorize` will return a Cholesky factorization.

Examples

```
| julia> A = Array(Bidiagonal(ones(5, 5), :U))  
| 5×5 Array{Float64,2}:
```

```
1266 1.0 1.0 0.0 0.0 0.0  
      0.0 1.0 1.0 0.0 0.0  
      0.0 0.0 1.0 1.0 0.0  
      0.0 0.0 0.0 1.0 1.0  
      0.0 0.0 0.0 0.0 1.0
```

## CHAPTER 54. LINEAR ALGEBRA

```
julia> factorize(A) # factorize will check to see that A is  
→ already factorized  
5×5 Bidiagonal{Float64,Array{Float64,1}}:  
1.0 1.0  
1.0 1.0  
1.0 1.0  
1.0 1.0  
1.0
```

This returns a `5×5 Bidiagonal{Float64}`, which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for `Bidiagonal` types.

[source](#)

`Base.LinAlg.Diagonal` – Type.

```
| Diagonal(A::AbstractMatrix)
```

Construct a matrix from the diagonal of A.

Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]  
3×3 Array{Int64,2}:  
1 2 3  
4 5 6  
7 8 9
```

```
julia> Diagonal(A)
3×3 Diagonal{Int64, Array{Int64, 1}}:
 1
 5
 9
```

**source**

```
Diagonal(V::AbstractVector)
```

Construct a matrix with V as its diagonal.

Examples

```
julia> V = [1, 2]
2-element Array{Int64, 1}:
 1
 2
```

```
julia> Diagonal(V)
2×2 Diagonal{Int64, Array{Int64, 1}}:
 1
 2
```

**source**

[Base.LinAlg.Bidiagonal](#) – Type.

```
Bidiagonal(dv::V, ev::V, uplo::Symbol) where V <:
    AbstractVector
```

Constructs an upper (`uplo=:U`) or lower (`uplo=:L`) bidiagonal matrix using the given diagonal (`dv`) and off-diagonal (`ev`) vectors. The result is of type `Bidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or

1268 Array(\_) for short). The length of `ev` must be less than or equal to the length of `dv`.

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```
julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9
```

```
julia> Bu = Bidiagonal(dv, ev, :U) # ev is on the first
           ↳ superdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  7
 2  8
 3  9
        4
```

```
julia> Bl = Bidiagonal(dv, ev, :L) # ev is on the first
           ↳ subdiagonal
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1
 7  2
 8  3
 9  4
```

```
| Bidiagonal(A, uplo::Symbol)
```

Construct a **Bidiagonal** matrix from the main diagonal of A and its first super- (if `uplo=:U`) or sub-diagonal (if `uplo=:L`).

Examples

```
julia> A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4]
4×4 Array{Int64,2}:
 1  1  1  1
 2  2  2  2
 3  3  3  3
 4  4  4  4

julia> Bidiagonal(A, :U) # contains the main diagonal and first
   ↳ superdiagonal of A
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1  1
 2  2
 3  3
 4

julia> Bidiagonal(A, :L) # contains the main diagonal and first
   ↳ subdiagonal of A
4×4 Bidiagonal{Int64,Array{Int64,1}}:
 1
 2  2
 3  3
 4  4
```

`source`

`Base.LinAlg.SymTridiagonal` – Type.

1270 SymTridiagonal(dv::V, ev::V) where V<:AbstractVector

CHAPTER 54. LINEAR ALGEBRA  
Construct a symmetric tridiagonal matrix from the diagonal (dv) and first sub/super-diagonal (ev), respectively. The result is of type `SymTridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Examples

```
julia> dv = [1, 2, 3, 4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> ev = [7, 8, 9]
3-element Array{Int64,1}:
 7
 8
 9

julia> SymTridiagonal(dv, ev)
4×4 SymTridiagonal{Int64,Array{Int64,1}}:
 1  7
 7  2  8
   8  3  9
       9  4
```

[source](#)

```
| SymTridiagonal(A::AbstractMatrix)
```

54.1. STANDARD FUNCTIONS

Construct a tridiagonal matrix from the diagonal and first subdiagonal, and first superdiagonal, of the symmetric matrix A.

Examples

```
julia> A = [1 2 3; 2 4 5; 3 5 6]
3×3 Array{Int64,2}:
 1  2  3
 2  4  5
 3  5  6

julia> SymTridiagonal(A)
3×3 SymTridiagonal{Int64,Array{Int64,1}}:
 1  2
 2  4  5
      6
```

source

[Base.LinAlg.Tridiagonal](#) – Type.

```
| Tridiagonal(dl::V, d::V, du::V) where V <: AbstractVector
```

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `convert(Array, _)` (or `Array(_)` for short). The lengths of `dl` and `du` must be one less than the length of `d`.

Examples

```
julia> dl = [1, 2, 3];
julia> du = [4, 5, 6];
```

```
1272 julia> d = [7, 8, 9, 0];
```

CHAPTER 54. LINEAR ALGEBRA

```
julia> Tridiagonal(dl, d, du)
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 7  4
 1  8  5
 2  9  6
 3  0
```

[source](#)

```
| Tridiagonal(A)
```

Construct a tridiagonal matrix from the first sub-diagonal, diagonal and first super-diagonal of the matrix A.

Examples

```
julia> A = [1 2 3 4; 1 2 3 4; 1 2 3 4; 1 2 3 4]
4×4 Array{Int64,2}:
 1  2  3  4
 1  2  3  4
 1  2  3  4
 1  2  3  4
```

```
julia> Tridiagonal(A)
```

```
4×4 Tridiagonal{Int64,Array{Int64,1}}:
 1  2
 1  2  3
 2  3  4
 3  4
```

[source](#)

[Base.LinAlg.Symmetric](#) – Type.

Construct a `Symmetric` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix A.

Examples

```
julia> A = [1 0 2 0 3; 0 4 0 5 0; 6 0 7 0 8; 0 9 0 1 0; 2 0 3
           ↵ 0 4]
5×5 Array{Int64,2}:
 1  0  2  0  3
 0  4  0  5  0
 6  0  7  0  8
 0  9  0  1  0
 2  0  3  0  4

julia> Supper = Symmetric(A)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  2  0  3
 0  4  0  5  0
 2  0  7  0  8
 0  5  0  1  0
 3  0  8  0  4

julia> Slower = Symmetric(A, :L)
5×5 Symmetric{Int64,Array{Int64,2}}:
 1  0  6  0  2
 0  4  0  9  0
 6  0  7  0  3
 0  9  0  1  0
 2  0  3  0  4
```

Note that `Supper` will not be equal to `Slower` unless A is itself symmetric

1274 e.g. if  $A == A'$ ).

## CHAPTER 54. LINEAR ALGEBRA

`source`

`Base.LinAlg.Hermitian` – Type.

`| Hermitian(A, uplo=:U)`

Construct a `Hermitian` view of the upper (if `uplo = :U`) or lower (if `uplo = :L`) triangle of the matrix `A`.

Examples

```
julia> A = [1 0 2+2im 0 3-3im; 0 4 0 5 0; 6-6im 0 7 0 8+8im; 0
    ↳ 9 0 1 0; 2+2im 0 3-3im 0 4];
```

```
julia> Hupper = Hermitian(A)
```

```
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  2+2im  0+0im  3-3im
 0+0im  4+0im  0+0im  5+0im  0+0im
 2-2im  0+0im  7+0im  0+0im  8+8im
 0+0im  5+0im  0+0im  1+0im  0+0im
 3+3im  0+0im  8-8im  0+0im  4+0im
```

```
julia> Hlower = Hermitian(A, :L)
```

```
5×5 Hermitian{Complex{Int64},Array{Complex{Int64},2}}:
 1+0im  0+0im  6+6im  0+0im  2-2im
 0+0im  4+0im  0+0im  9+0im  0+0im
 6-6im  0+0im  7+0im  0+0im  3+3im
 0+0im  9+0im  0+0im  1+0im  0+0im
 2+2im  0+0im  3-3im  0+0im  4+0im
```

Note that `Hupper` will not be equal to `Hlower` unless `A` is itself Hermitian (e.g. if  $A == A'$ ).

All non-real parts of the diagonal will be ignored.

54.1 | STANDARD FUNCTIONS  
Hermitian(fill(Complex(1,1), 1, 1)) == fill(1, 1, 1)

1275

[source](#)

[Base.LinAlg.LowerTriangular](#) – Type.

`LowerTriangular(A::AbstractMatrix)`

Construct a `LowerTriangular` view of the the matrix A.

Examples

**julia>** A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]

3×3 Array{Float64,2}:

```
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0
```

**julia>** LowerTriangular(A)

3×3 LowerTriangular{Float64,Array{Float64,2}}:

```
1.0
4.0  5.0
7.0  8.0  9.0
```

[source](#)

[Base.LinAlg.UpperTriangular](#) – Type.

`UpperTriangular(A::AbstractMatrix)`

Construct an `UpperTriangular` view of the the matrix A.

Examples

**julia>** A = [1.0 2.0 3.0; 4.0 5.0 6.0; 7.0 8.0 9.0]

3×3 Array{Float64,2}:

```
1.0  2.0  3.0
```

```
1276 4.0 5.0 6.0  
      7.0 8.0 9.0
```

## CHAPTER 54. LINEAR ALGEBRA

```
julia> UpperTriangular(A)  
3×3 UpperTriangular{Float64, Array{Float64, 2}}:  
 1.0 2.0 3.0  
      5.0 6.0  
          9.0
```

`source`

[Base.LinAlg.UniformScaling](#) – Type.

```
| UniformScaling{T<:Number}
```

Generically sized uniform scaling operator defined as a scalar times the identity operator,  $\lambda * \mathbf{I}$ . See also [I](#).

Examples

```
julia> J = UniformScaling(2.)  
UniformScaling{Float64}  
2.0*I
```

```
julia> A = [1. 2.; 3. 4.]  
2×2 Array{Float64, 2}:  
 1.0 2.0  
 3.0 4.0
```

```
julia> J*A  
2×2 Array{Float64, 2}:  
 2.0 4.0  
 6.0 8.0
```

`source`

```
| lu(A, pivot=Val(true)) -> L, U, p
```

Compute the LU factorization of A, such that  $A[p, :] = L * U$ . By default, pivoting is used. This can be overridden by passing `Val(false)` for the second argument.

See also [lufact](#).

Examples

```
julia> A = [4. 3.; 6. 3.]
2×2 Array{Float64,2}:
 4.0  3.0
 6.0  3.0

julia> L, U, p = lu(A)
([1.0 0.0; 0.666667 1.0], [6.0 3.0; 0.0 1.0], [2, 1])

julia> A[p, :] == L * U
true
```

[source](#)

[Base.LinAlg.lufact](#) – Function.

```
| lufact(A, pivot=Val(true)) -> F::LU
```

Compute the LU factorization of A.

In most cases, if A is a subtype S of `AbstractMatrix{T}` with an element type T supporting +, -, \*, and /, the return type is `LU{T, S{T}}`. If pivoting is chosen (default) the element type should also support `abs` and <.

The individual components of the factorization F can be accessed by indexing:

<code>F[ :L ]</code>	L (lower triangular) part of LU
<code>F[ :U ]</code>	U (upper triangular) part of LU
<code>F[ :p ]</code>	(right) permutation <b>Vector</b>
<code>F[ :P ]</code>	(right) permutation <b>Matrix</b>

The relationship between F and A is

$$F[ :L ] * F[ :U ] == A[ F[ :p ], : ]$$

F further supports the following functions:

Supported function	LU	LU{T, Tridiagonal{T}}
/		
\		
<code>inv</code>		
<code>det</code>		
<code>logdet</code>		
<code>logabsdet</code>		
<code>size</code>		

Examples

```
julia> A = [4 3; 6 3]
2×2 Array{Int64,2}:
 4  3
 6  3

julia> F = lufact(A)
Base.LinAlg.LU{Float64,Array{Float64,2}}
L factor:
2×2 Array{Float64,2}:
 1.0  0.0
 1.5  1.0
U factor:
2×2 Array{Float64,2}:
 4.0   3.0
 0.0  -1.5
```

54.1 | **julia>** F[ :L]\*F[ :U] = A[F[ :p], :]  
true

1279

**source**

| lufact(A::SparseMatrixCSC) -> F::UmfpackLU

Compute the LU factorization of a sparse matrix A.

For sparse A with real or complex element type, the return type of F is **UmfpackLU**{Tv, Ti}, with Tv = **Float64** or **Complex128** respectively and Ti is an integer type (**Int32** or **Int64**).

The individual components of the factorization F can be accessed by indexing:

Component	Description
F[ :L]	L (lower triangular) part of LU
F[ :U]	U (upper triangular) part of LU
F[ :p]	right permutation Vector
F[ :q]	left permutation Vector
F[ :Rs]	Vector of scaling factors
F[ :( :) ]	(L, U, p, q, Rs) components

The relation between F and A is

$$F[ :L]*F[ :U] == (F[ :Rs] . * A)[F[ :p], F[ :q]]$$

F further supports the following functions:

\  
cond  
det

Note

**lufact(A::SparseMatrixCSC)** uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with **Float64** or **Complex128** elements, **lufact** converts A into a

1280 copy that is of type `SparseMatrixCSC{Complex128}` or `SparseMatrixCSC{Complex64}` as appropriate.

`source`

[Base.LinAlg.lufact!](#) – Function.

```
| lufact!(A, pivot=Val(true)) -> LU
```

`lufact!` is the same as `lufact`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

`source`

[Base.LinAlg.chol](#) – Function.

```
| chol(A) -> U
```

Compute the Cholesky factorization of a positive definite matrix `A` and return the `UpperTriangular` matrix `U` such that  $A = U'U$ .

Examples

```
julia> A = [1. 2.; 2. 50.]
```

```
2×2 Array{Float64,2}:
```

```
1.0  2.0
2.0  50.0
```

```
julia> U = chol(A)
```

```
2×2 UpperTriangular{Float64,Array{Float64,2}}:
```

```
1.0  2.0
       6.78233
```

```
julia> U'U
```

```
1.0  2.0
2.0  50.0
```

source

```
| chol(x::Number) -> y
```

Compute the square root of a non-negative number x.

Examples

```
| julia> chol(16)
| 4.0
```

source

```
| chol(J::UniformScaling) -> C
```

Compute the square root of a non-negative UniformScaling J.

Examples

```
| julia> chol(16I)
| UniformScaling{Float64}
| 4.0*I
```

source

[Base.LinAlg.cholfact](#) – Function.

```
| cholfact(A, Val(false)) -> Cholesky
```

Compute the Cholesky factorization of a dense symmetric positive definite matrix A and return a Cholesky factorization. The matrix A can either be a [Symmetric](#) or [Hermitian StridedMatrix](#) or a perfectly symmetric or Hermitian [StridedMatrix](#). The triangular Cholesky factor can be obtained from the factorization F with: F[ :L] and F[ :U]. The following

128 functions are available for Cholesky objects. See [CHAPTER 54, LINEAR ALGEBRA](#) and [isposdef](#).

## Examples

```
julia> A = [4. 12. -16.; 12. 37. -43.; -16. -43. 98.]  
3×3 Array{Float64,2}:  
 4.0 12.0 -16.0  
 12.0 37.0 -43.0  
 -16.0 -43.0 98.0  
  
julia> C = cholfact(A)  
Base.LinAlg.Cholesky{Float64,Array{Float64,2}}  
U factor:  
3×3 UpperTriangular{Float64,Array{Float64,2}}:  
 2.0 6.0 -8.0  
 1.0 5.0  
 3.0  
  
julia> C[:U]  
3×3 UpperTriangular{Float64,Array{Float64,2}}:  
 2.0 6.0 -8.0  
 1.0 5.0  
 3.0  
  
julia> C[:L]  
3×3 LowerTriangular{Float64,Array{Float64,2}}:  
 2.0  
 6.0 1.0  
 -8.0 5.0 3.0  
  
julia> C[:L] * C[:U] == A  
true
```

```
| cholfact(A, Val(true); tol = 0.0) -> CholeskyPivoted
```

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix A and return a **CholeskyPivoted** factorization.

The matrix A can either be a **Symmetric** or **Hermitian StridedMatrix** or a perfectly symmetric or Hermitian **StridedMatrix**. The triangular Cholesky factor can be obtained from the factorization F with: F[ :L] and F[ :U]. The following functions are available for **PivotedCholesky** objects: **size**, **\**, **inv**, **det**, and **rank**. The argument **tol** determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

**source**

```
| cholfact(A; shift = 0.0, perm = Int[]) -> CHOLMOD.Factor
```

Compute the Cholesky factorization of a sparse positive definite matrix A. A must be a **SparseMatrixCSC** or a **Symmetric/Hermitian** view of a **SparseMatrixCSC**. Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. F = **cholfact**(A) is most frequently used to solve systems of equations with F\ b, but also the methods **diag**, **det**, and **logdet** are defined for F. You can also extract individual factors from F, using F[ :L]. However, since pivoting is on by default, the factorization is internally represented as A == P' \* L \* L' \* P with a permutation matrix P; using just L without accounting for P will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like PtL = F[ :PtL] (the equivalent of P' \* L) and LtP = F[ :UP] (the equivalent of L' \* P).

Setting the optional **shift** keyword argument computes the factorization of A+shift\*I instead of A. If the **perm** argument is nonempty, it should

1284 be a permutation of `1:size(A, 1)` giving CHOLMOD's default AMD ordering).

### Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

### source

`Base.LinAlg.cholfact!` – Function.

```
| cholfact!(A, Val(false)) -> Cholesky
```

The same as `cholfact`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

### Examples

```
julia> A = [1 2; 2 50]
2×2 Array{Int64,2}:
 1  2
 2  50

julia> cholfact!(A)
ERROR: InexactError: convert(Int64, 6.782329983125268)
```

### source

```
| cholfact!(A, Val(true); tol = 0.0) -> CholeskyPivoted
```

54.1. The **STANDARD FUNCTIONS** [cholfact](#)<sup>285</sup> but saves space by overwriting the input A instead of creating a copy. An [InexactError](#) exception is thrown if the factorization produces a number not representable by the element type of A, e.g. for integer types.

[source](#)

```
| cholfact!(F::Factor, A; shift = 0.0) -> CHOLMOD.Factor
```

Compute the Cholesky ( $LL'$ ) factorization of A, reusing the symbolic factorization F. A must be a [SparseMatrixCSC](#) or a [Symmetric](#)/[Hermitian](#) view of a [SparseMatrixCSC](#). Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian.

See also [cholfact](#).

Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to [SparseMatrixCSC{Float64}](#) or [SparseMatrixCSC{Complex128}](#) as appropriate.

[source](#)

[Base.LinAlg.lowrankupdate](#) – Function.

```
| lowrankupdate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

Update a Cholesky factorization C with the vector v. If  $A = C[:U]'C[:U]$  then  $CC = cholfact(C[:U]'C[:U] + v*v')$  but the computation of CC only uses  $O(n^2)$  operations.

[source](#)

[Base.LinAlg.lowrankdowndate](#) – Function.

```
| lowrankdowndate(C::Cholesky, v::StridedVector) -> CC::Cholesky
```

1286 Downdate a Cholesky factorization C with the vector v. If A = C[ :U]'C[ :U]

then CC = cholfact(C[ :U]'C[ :U] - v\*v') but the computation of CC only uses O(n^2) operations.

[source](#)

[Base.LinAlg.lowrankupdate!](#) – Function.

`lowrankupdate!(C::Cholesky, v::StridedVector) -> CC::Cholesky`

Update a Cholesky factorization C with the vector v. If A = C[ :U]'C[ :U] then CC = cholfact(C[ :U]'C[ :U] + v\*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

[source](#)

[Base.LinAlg.lowrankdowndate!](#) – Function.

`lowrankdowndate!(C::Cholesky, v::StridedVector) -> CC::Cholesky`

Downdate a Cholesky factorization C with the vector v. If A = C[ :U]'C[ :U] then CC = cholfact(C[ :U]'C[ :U] - v\*v') but the computation of CC only uses O(n^2) operations. The input factorization C is updated in place such that on exit C == CC. The vector v is destroyed during the computation.

[source](#)

[Base.LinAlg.ldltfact](#) – Function.

`ldltfact(S::SymTridiagonal) -> LDLt`

Compute an LDLt factorization of the real symmetric tridiagonal matrix S such that S = L\*Diagonal(d)\*L' where L is a unit lower triangular matrix and d is a vector. The main use of an LDLt factorization F = ldltfact(S) is to solve the linear system of equations Sx = b with F\b.

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 3.0  1.0
 1.0  4.0  2.0
 2.0  5.0

julia> ldltS = ldltfact(S);

julia> b = [6., 7., 8.];

julia> ldltS \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255

julia> S \ b
3-element Array{Float64,1}:
 1.7906976744186047
 0.627906976744186
 1.3488372093023255
```

source

```
|ldltfact(A; shift = 0.0, perm=Int[]) -> CHOLMOD.Factor
```

Compute the  $LDL'$  factorization of a sparse matrix A. A must be a [SparseMatrixCSC](#) or a [Symmetric/Hermitian](#) view of a [SparseMatrixCSC](#). Note that even if A doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. F = ldltfact(A) is most frequently used to solve systems of equations A\*x = b with F\b.

CHAPTER 54. LINEAR ALGEBRA

However, since pivoting is on by default, the factorization is internally represented as  $A == P' * L * D * L' * P$  with a permutation matrix  $P$ ; using just  $L$  without accounting for  $P$  will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like  $PtL = F[ :PtL]$  (the equivalent of  $P' * L$ ) and  $LtP = F[ :UP]$  (the equivalent of  $L' * P$ ). The complete list of supported factors is `:L`, `:PtL`, `:D`, `:UP`, `:U`, `:LD`, `:DU`, `:PtLD`, `:DUP`.

Setting the optional `shift` keyword argument computes the factorization of  $A + shift * I$  instead of  $A$ . If the `perm` argument is nonempty, it should be a permutation of `1:size(A, 1)` giving the ordering to use (instead of CHOLMOD's default AMD ordering).

### Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to `SparseMatrixCSC{Float64}` or `SparseMatrixCSC{Complex128}` as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the `Base.SparseArrays.CHOLMOD` module.

### source

`Base.LinAlg.ldltfact!` – Function.

```
|ldltfact!(S::SymTridiagonal) -> LDLt
```

Same as `ldltfact`, but saves space by overwriting the input  $S$ , instead of creating a copy.

### Examples

```
julia> S = SymTridiagonal([3., 4., 5.], [1., 2.])
```

```
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
```

```
 3.0  1.0  
 1.0  4.0  2.0  
 2.0  5.0
```

```
julia> ldltS = ldltfact!(S);
```

```
julia> ldltS === S
```

```
false
```

```
julia> S
```

```
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
```

```
 3.0      0.333333  
 0.333333  3.66667   0.545455  
          0.545455  3.90909
```

`source`

```
| ldltfact!(F::Factor, A; shift = 0.0) -> CHOLMOD.Factor
```

Compute the  $LDL'$  factorization of  $A$ , reusing the symbolic factorization  $F$ .  $A$  must be a [SparseMatrixCSC](#) or a [Symmetric/Hermitian](#) view of a [SparseMatrixCSC](#). Note that even if  $A$  doesn't have the type tag, it must still be symmetric or Hermitian.

See also [ldltfact](#).

#### Note

This method uses the CHOLMOD library from SuiteSparse, which only supports doubles or complex doubles. Input matrices not of those element types will be converted to [SparseMatrixCSC{Float64}](#) or [SparseMatrixCSC{Complex128}](#) as appropriate.

1290 `source`

CHAPTER 54. LINEAR ALGEBRA

`Base.LinAlg.qr` – Function.

```
| qr(A, pivot=Val(false); full::Bool = false) -> Q, R, [p]
```

Compute the (pivoted) QR factorization of  $A$  such that either  $A = Q*R$  or  $A[:, p] = Q*R$ . Also see `qrfact`. The default is to compute a “thin” factorization. Note that  $R$  is not extended with zeros when a full/square orthogonal factor  $Q$  is requested (via `full = true`).

`source`

```
| qr(v::AbstractVector) -> w, r
```

Computes the polar decomposition of a vector. Returns  $w$ , a unit vector in the direction of  $v$ , and  $r$ , the norm of  $v$ .

See also `normalize`, `normalize!`, and `LinAlg.qr!`.

Examples

```
julia> v = [1; 2]
```

```
2-element Array{Int64,1}:
```

```
1
```

```
2
```

```
julia> w, r = qr(v)
```

```
([0.447214, 0.894427], 2.23606797749979)
```

```
julia> w*r == v
```

```
true
```

`source`

`Base.LinAlg.qr!` – Function.

```
| LinAlg.qr!(v::AbstractVector) -> w, r
```

54.1. ~~COMPOSITION~~<sup>STANDARD DEFINITIONS</sup> Decomposition of a vector. Instead of returning a new vector as `qr(v::AbstractVector)`, this function mutates the input vector `v` in place. Returns `w`, a unit vector in the direction of `v` (this is a mutation of `v`), and `r`, the norm of `v`.

See also [normalize](#), [normalize!](#), and [qr](#).

Examples

```
julia> v = [1.; 2.]  
2-element Array{Float64,1}:  
 1.0  
 2.0  
  
julia> w, r = Base.LinAlg.qr!(v)  
([0.447214, 0.894427], 2.23606797749979)  
  
julia> w === v  
true
```

[source](#)

[Base.LinAlg.qrfact](#) – Function.

```
|qrfact(A, pivot=Val(false)) -> F
```

Compute the QR factorization of the matrix `A`: an orthogonal (or unitary if `A` is complex-valued) matrix `Q`, and an upper triangular matrix `R` such that

$$A = QR$$

The returned object `F` stores the factorization in a packed format:

if `pivot == Val(true)` then `F` is a [QRPivoted](#) object,

1292 otherwise if the element type of A is a [CHAPTER 54 LINEAR ALGEBRA](#)  
[Complex64](#) or [Complex128](#)), then F is a [QRCompactWY](#) object,  
otherwise F is a [QR](#) object.

The individual components of the factorization F can be accessed by indexing with a symbol:

F[ :Q]: the orthogonal/unitary matrix Q

F[ :R]: the upper triangular matrix R

F[ :p]: the permutation vector of the pivot ([QRPivoted](#) only)

F[ :P]: the permutation matrix of the pivot ([QRPivoted](#) only)

The following functions are available for the QR objects: [inv](#), [size](#), and [\](#). When A is rectangular, \ will return a least squares solution and if the solution is not unique, the one with smallest norm is returned.

Multiplication with respect to either full/square or non-full/square Q is allowed, i.e. both F[ :Q]\*F[ :R] and F[ :Q]\*A are supported. A Q matrix can be converted into a regular matrix with [Matrix](#).

Examples

```
julia> A = [3.0 -6.0; 4.0 -8.0; 0.0 1.0]
3x2 Array{Float64,2}:
 3.0  -6.0
 4.0  -8.0
 0.0   1.0

julia> F = qrfact(A)
Base.LinAlg.QRCompactWY{Float64,Array{Float64,2}} with factors
→ Q and R:
[-0.6 0.0 0.8; -0.8 0.0 -0.6; 0.0 -1.0 0.0]
[-5.0 10.0; 0.0 -1.0]
```

```
julia> F[ :Q] * F[ :R] == A  
true
```

Note

`qrfact` returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the `Q` and `R` matrices can be stored compactly rather as two separate dense matrices.

#### source

```
| qrfact(A) -> QRSpars
```

Compute the QR factorization of a sparse matrix `A`. Fill-reducing row and column permutations are used such that  $F[ :R] = F[ :Q]'^*A[F[ :prow], F[ :pcol]]$ . The main application of this type is to solve least squares or underdetermined problems with `\`. The function calls the C library SPQR.

#### Examples

```
julia> A = sparse([1,2,3,4], [1,1,2,2], ones(4))  
4x2 SparseMatrixCSC{Float64, Int64} with 4 stored entries:  
[1, 1] = 1.0  
[2, 1] = 1.0  
[3, 2] = 1.0  
[4, 2] = 1.0
```

```
julia> qrfact(A)  
Base.SparseArrays.SPQR.QRSpars{Float64, Int64}  
Q factor:  
4x4 Base.SparseArrays.SPQR.QRSparsQ{Float64, Int64}:  
-0.707107 0.0 0.0 -0.707107
```

```
1294 0.0      -0.707107 -0.707107 0.0
      0.0      -0.707107  0.707107 0.0
     -0.707107  0.0       0.0      0.707107
R factor:
2×2 SparseMatrixCSC{Float64, Int64} with 2 stored entries:
 [1, 1] = -1.41421
 [2, 2] = -1.41421
Row permutation:
4-element Array{Int64,1}:
1
3
4
2
Column permutation:
2-element Array{Int64,1}:
1
2
```

## source

[Base.LinAlg.qrfact!](#) – Function.

```
| qrfact!(A, pivot=Val(false))
```

`qrfact!` is the same as `qrfact` when `A` is a subtype of `StridedMatrix`, but saves space by overwriting the input `A`, instead of creating a copy. An `InexactError` exception is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

## Examples

```
julia> a = [1. 2.; 3. 4.]
2×2 Array{Float64,2}:
 1.0  2.0
```

```
julia> qrfact!(a)
Base.LinAlg.QRCompactWY{Float64,Array{Float64,2}} with factors
→ Q and R:
[-0.316228 -0.948683; -0.948683 0.316228]
[-3.16228 -4.42719; 0.0 -0.632456]

julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> qrfact!(a)
ERROR: InexactError: convert(Int64, -3.1622776601683795)
Stacktrace:
[...]
```

**source**

[Base.LinAlg.QR](#) – Type.

QR <: Factorization

A QR matrix factorization stored in a packed format, typically obtained from [qrfact](#). If  $A$  is an  $m \times n$  matrix, then

$$A = QR$$

where  $Q$  is an orthogonal/unitary matrix and  $R$  is upper triangular. The matrix  $Q$  is stored as a sequence of Householder reflectors  $v_i$  and coefficients  $\tau_i$  where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has two fields:

**factors** is an  $m \times n$  matrix.

- The upper triangular part contains the elements of  $R$ , that is  $R = \text{triu}(F.\text{factors})$  for a QR object  $F$ .
- The subdiagonal part contains the reflectors  $v_i$  stored in a packed format where  $v_i$  is the  $i$ th column of the matrix  $V = I + \text{tril}(F.\text{factors}, -1)$ .

$\tau$  is a vector of length  $\min(m, n)$  containing the coefficients  $a u_i$ .

**source**

[Base.LinAlg.QRCompactWY](#) – Type.

QRCompactWY <: Factorization

A QR matrix factorization stored in a compact blocked format, typically obtained from [qrfact](#). If  $A$  is an  $m \times n$  matrix, then

$$A = QR$$

where  $Q$  is an orthogonal/unitary matrix and  $R$  is upper triangular. It is similar to the [QR](#) format except that the orthogonal/unitary matrix  $Q$  is stored in Compact WY format<sup>1</sup>, as a lower trapezoidal matrix  $V$  and an upper triangular matrix  $T$  where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

54.1 STANDARD FUNCTIONS of  $V$ , and  $a_{ii}$  is the  $i$ th diagonal element<sup>1297</sup>.

The object has two fields:

**factors**, as in the **QR** type, is an  $m \times n$  matrix.

- The upper triangular part contains the elements of  $R$ , that is  $R = \text{triu}(F.\text{factors})$  for a QR object  $F$ .
- The subdiagonal part contains the reflectors  $v_i$  stored in a packed format such that  $V = I + \text{tril}(F.\text{factors}, -1)$ .

$T$  is a square matrix with  $\min(m, n)$  columns, whose upper triangular part gives the matrix  $T$  above (the subdiagonal elements are ignored).

Note

This format should not to be confused with the older WY representation<sup>2</sup>.

**source**

[Base.LinAlg.QRPivoted](#) – Type.

**QRPivoted** <: Factorization

A QR matrix factorization with column pivoting in a packed format, typically obtained from [qrfact](#). If  $A$  is an  $m \times n$  matrix, then

$$AP = QR$$

where  $P$  is a permutation matrix,  $Q$  is an orthogonal/unitary matrix and  $R$  is upper triangular. The matrix  $Q$  is stored as a sequence of Householder reflectors:

---

<sup>2</sup>C Bischof and C Van Loan, "The WY representation for products of Householder matrices", SIAM J Sci Stat Comput 8 (1987), s2-s13. [doi:10.1137/0908009](#)

<sup>1</sup>R Schreiber and C Van Loan, "A storage-efficient WY representation for products of Householder transformations", SIAM J Sci Stat Comput 10 (1989), 53-57. [doi:10.1137/0910005](#)

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has three fields:

**factors** is an  $m \times n$  matrix.

- The upper triangular part contains the elements of  $R$ , that is  $R = \text{triu}(F.\text{factors})$  for a QR object  $F$ .
- The subdiagonal part contains the reflectors  $v_i$  stored in a packed format where  $v_i$  is the  $i$ th column of the matrix  $V = I + \text{tril}(F.\text{factors}, -1)$ .

**tau** is a vector of length  $\min(m, n)$  containing the coefficients  $a u_i$ .

**jpv**t is an integer vector of length  $n$  corresponding to the permutation  $P$ .

**source**

[Base.LinAlg.lqfact!](#) – Function.

| lqfact!(A) -> LQ

Compute the LQ factorization of  $A$ , using the input matrix as a workspace.

See also [lq](#).

**source**

[Base.LinAlg.lqfact](#) – Function.

| lqfact(A) -> LQ

Compute the LQ factorization of  $A$ . See also [lq](#).

**source**

[Base.LinAlg.lq](#) – Function.

Perform an LQ factorization of A such that  $A = L*Q$ . The default (`full = false`) computes a factorization with possibly-rectangular L and Q, commonly the "thin" factorization. The LQ factorization is the QR factorization of  $A^\top$ . If the explicit, full/square form of Q is requested via `full = true`, L is not extended with zeros.

#### Note

While in QR factorization the "thin" factorization is so named due to yielding either a square or "tall"/"thin" rectangular factor Q, in LQ factorization the "thin" factorization somewhat confusingly produces either a square or "short"/"wide" rectangular factor Q. "Thin" factorizations more broadly are also referred to as "reduced" factorizations.

#### source

[Base.LinAlg.bkfact](#) – Function.

```
bkfact(A, uplo::Symbol=:U, symmetric::Bool=issymmetric(A), rook
      ::Bool=false) -> BunchKaufman
```

Compute the Bunch-Kaufman<sup>3</sup> factorization of a symmetric or Hermitian matrix A and return a `BunchKaufman` object. `uplo` indicates which triangle of matrix A to reference. If `symmetric` is `true`, A is assumed to be symmetric. If `symmetric` is `false`, A is assumed to be Hermitian. If `rook` is `true`, rook pivoting is used. If `rook` is `false`, rook pivoting is not used. The following functions are available for `BunchKaufman` objects: `size`, \, `inv`, `issymmetric`, `ishermitian`.

#### source

---

<sup>3</sup>J R Bunch and L Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of Computation 31:137 (1977), 163–179. [url](#).

`Base.LinAlg.bkfact!` – Function.

CHAPTER 54. LINEAR ALGEBRA

```
bkfact!(A, uplo::Symbol=:U, symmetric::Bool=issymmetric(A),
        rook::Bool=false) -> BunchKaufman
```

`bkfact!` is the same as `bkfact`, but saves space by overwriting the input `A`, instead of creating a copy.

`source`

`Base.LinAlg.eig` – Function.

```
eig(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange::UnitRange) -> D, V
eig(A::Union{SymTridiagonal, Hermitian, Symmetric}, v1::Real, vu::Real) -> D, V
eig(A, permute::Bool=true, scale::Bool=true) -> D, V
```

Computes eigenvalues (`D`) and eigenvectors (`V`) of `A`. See `eigfact` for details on the `irange`, `v1`, and `vu` arguments (for `SymTridiagonal`, `Hermitian`, and `Symmetric` matrices) and the `permute` and `scale` keyword arguments. The eigenvectors are returned columnwise.

Examples

```
julia> eig([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
([1.0, 3.0, 18.0], [1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])
```

`eig` is a wrapper around `eigfact`, extracting all parts of the factorization to a tuple; where possible, using `eigfact` is recommended.

`source`

```
eig(A, B) -> D, V
```

Computes generalized eigenvalues (`D`) and vectors (`V`) of `A` with respect to `B`.

54.1 STANDARD FUNCTIONS [eigfact](#), extracting all parts of the factorization to a tuple; where possible, using [eigfact](#) is recommended.

Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0   1
 1   0

julia> eig(A, B)
(Complex{Float64}[0.0+1.0im, 0.0-1.0im],
 → Complex{Float64}[0.0-1.0im 0.0+1.0im; -1.0-0.0im
 → -1.0+0.0im])
```

[source](#)

[Base.LinAlg.eigvals](#) – Function.

```
| eigvals(A; permute::Bool=true, scale::Bool=true) -> values
```

Returns the eigenvalues of A.

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

[source](#)

```
| eigvals(A, B) -> values
```

130 Computes the generalized eigenvalues of **CHAPTER 54. LINEAR ALGEBRA**

## Examples

```
julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1

julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0   1
 1   0

julia> eigvals(A,B)
2-element Array{Complex{Float64},1}:
 0.0 + 1.0im
 0.0 - 1.0im
```

## source

```
eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange
       ::UnitRange) -> values
```

Returns the eigenvalues of A. It is possible to calculate only a subset of the eigenvalues by specifying a **UnitRange** **irange** covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
3×3 SymTridiagonal{Float64,Array{Float64,1}}:
 1.0  2.0
 2.0  2.0  3.0
 3.0  1.0

julia> eigvals(A, 2:2)
```

```
| 1-element Array{Float64,1}:
```

```
|   0.999999999999996
```

```
| julia> eigvals(A)
```

```
| 3-element Array{Float64,1}:
```

```
|   -2.1400549446402604
```

```
|   1.0000000000000002
```

```
|   5.140054944640259
```

```
| source
```

```
| eigvals(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::
```

```
|   Real, vu::Real) -> values
```

Returns the eigenvalues of A. It is possible to calculate only a subset of the eigenvalues by specifying a pair `vl` and `vu` for the lower and upper boundaries of the eigenvalues.

```
| julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
```

```
| 3×3 SymTridiagonal{Float64,Array{Float64,1}}:
```

```
|   1.0  2.0
```

```
|   2.0  2.0  3.0
```

```
|   3.0  1.0
```

```
| julia> eigvals(A, -1, 2)
```

```
| 1-element Array{Float64,1}:
```

```
|   1.000000000000009
```

```
| julia> eigvals(A)
```

```
| 3-element Array{Float64,1}:
```

```
|   -2.1400549446402604
```

```
|   1.0000000000000002
```

```
|   5.140054944640259
```

`Base.LinAlg.eigvals!` – Function.

```
| eigvals!(A; permute::Bool=true, scale::Bool=true) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm.

`source`

```
| eigvals!(A, B) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A` (and `B`), instead of creating copies.

`source`

```
| eigvals!(A)::Union{SymTridiagonal, Hermitian, Symmetric}, irange  
  ::UnitRange) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `irange` is a range of eigenvalue indices to search for – for instance, the 2nd to 8th eigenvalues.

`source`

```
| eigvals!(A)::Union{SymTridiagonal, Hermitian, Symmetric}, vl::  
  Real, vu::Real) -> values
```

Same as `eigvals`, but saves space by overwriting the input `A`, instead of creating a copy. `vl` is the lower bound of the interval to search for eigenvalues, and  `is the upper bound.`

`source`

`Base.LinAlg.eigmax` – Function.

Returns the largest eigenvalue of A. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

### Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmax(A)
1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 -1+0im  0+0im

julia> eigmax(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im
→  0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

[source](#)

[Base.LinAlg.eigmin](#) – Function.

1306 `eigmin(A; permute::Bool=true, scale::Bool=true)` CHAPTER 54 LINEAR ALGEBRA

Returns the smallest eigenvalue of A. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of A are complex, this method will fail, since complex numbers cannot be sorted.

## Examples

```
julia> A = [0 im; -im 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 0-1im  0+0im

julia> eigmin(A)
-1.0

julia> A = [0 im; -1 0]
2×2 Array{Complex{Int64},2}:
 0+0im  0+1im
 -1+0im  0+0im

julia> eigmin(A)
ERROR: DomainError with Complex{Int64}[0+0im 0+1im; -1+0im
    ↵ 0+0im]:
`A` cannot have complex eigenvalues.
Stacktrace:
[...]
```

[source](#)

[Base.LinAlg.eigvecs](#) – Function.

Returns a matrix  $M$  whose columns are the eigenvectors of  $A$ . (The  $k$ th eigenvector can be obtained from the slice  $M[:, k]$ .)

If the optional vector of eigenvalues `eigvals` is specified, `eigvecs` returns the specific corresponding eigenvectors.

### Examples

```
julia> A = SymTridiagonal([1.; 2.; 1.], [2.; 3.])
```

```
3×3 SymTridiagonal{Float64, Array{Float64, 1}}:
```

```
1.0  2.0  
2.0  2.0  3.0  
      3.0  1.0
```

```
julia> eigvals(A)
```

```
3-element Array{Float64, 1}:
```

```
-2.1400549446402604  
1.0000000000000002  
5.140054944640259
```

```
julia> eigvecs(A)
```

```
3×3 Array{Float64, 2}:
```

```
0.418304  -0.83205       0.364299  
-0.656749  -7.39009e-16  0.754109  
0.627457    0.5547        0.546448
```

```
julia> eigvecs(A, [1.])
```

```
3×1 Array{Float64, 2}:
```

```
0.8320502943378438  
4.263514128092366e-17  
-0.5547001962252291
```

```
| eigvecs(A; permute::Bool=true, scale::Bool=true) -> Matrix
```

Returns a matrix  $M$  whose columns are the eigenvectors of  $A$ . (The  $k$ th eigenvector can be obtained from the slice  $M[:, k]$ .) The `permute` and `scale` keywords are the same as for [eigfact](#).

Examples

```
| julia> eigvecs([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
3×3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

source

```
| eigvecs(A, B) -> Matrix
```

Returns a matrix  $M$  whose columns are the generalized eigenvectors of  $A$  and  $B$ . (The  $k$ th eigenvector can be obtained from the slice  $M[:, k]$ .)

Examples

```
| julia> A = [1 0; 0 -1]
2×2 Array{Int64,2}:
 1   0
 0  -1
```

```
| julia> B = [0 1; 1 0]
2×2 Array{Int64,2}:
 0   1
 1   0
```

```
| julia> eigvecs(A, B)
```

54.1| 2×2 Array{Complex{Float64},2}:

0.0-1.0im	0.0+1.0im
-1.0-0.0im	-1.0+0.0im

1309

**source**

[Base.LinAlg.eigfact](#) – Function.

| `eigfact(A; permute::Bool=true, scale::Bool=true) -> Eigen`

Computes the eigenvalue decomposition of `A`, returning an `Eigen` factorization object `F` which contains the eigenvalues in `F[:values]` and the eigenvectors in the columns of the matrix `F[:vectors]`. (The  $k$ th eigenvector can be obtained from the slice `F[:vectors][:, k]`.)

The following functions are available for `Eigen` objects: [inv](#), [det](#), and [isposdef](#).

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option `permute=true` permutes the matrix to become closer to upper triangular, and `scale=true` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `true` for both options.

Examples

```
julia> F = eigfact([1.0 0.0 0.0; 0.0 3.0 0.0; 0.0 0.0 18.0])
Base.LinAlg.Eigen{Float64,Float64,Array{Float64,2},Array{Float64,1}}([1.0
→ 3.0, 18.0], [1.0 0.0 0.0; 0.0 1.0 0.0; 0.0 0.0 1.0])
```

```
julia> F[:values]
```

```
3-element Array{Float64,1}:
```

1.0
3.0
18.0

```
julia> F[:vectors]
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

[source](#)

```
| eigfact(A, B) -> GeneralizedEigen
```

Computes the generalized eigenvalue decomposition of A and B, returning a `GeneralizedEigen` factorization object F which contains the generalized eigenvalues in F[:values] and the generalized eigenvectors in the columns of the matrix F[:vectors]. (The kth generalized eigenvector can be obtained from the slice F[:vectors][:, k].)

[source](#)

```
| eigfact(A::Union{SymTridiagonal, Hermitian, Symmetric}, irange
      ::UnitRange) -> Eigen
```

Computes the eigenvalue decomposition of A, returning an `Eigen` factorization object F which contains the eigenvalues in F[:values] and the eigenvectors in the columns of the matrix F[:vectors]. (The kth eigenvector can be obtained from the slice F[:vectors][:, k].)

The following functions are available for `Eigen` objects: `inv`, `det`, and `isposdef`.

The `UnitRange` `irange` specifies indices of the sorted eigenvalues to search for.

Note

If `irange` is not `1:n`, where `n` is the dimension of A, then the returned factorization will be a truncated factorization.

```
| eigfact(A::Union{SymTridiagonal, Hermitian, Symmetric}, vl::  
|     Real, vu::Real) -> Eigen
```

Computes the eigenvalue decomposition of A, returning an **Eigen** factorization object F which contains the eigenvalues in F[ :values] and the eigenvectors in the columns of the matrix F[ :vectors]. (The kth eigenvector can be obtained from the slice F[ :vectors][ :, k].)

The following functions are available for **Eigen** objects: **inv**, **det**, and **isposdef**.

**vl** is the lower bound of the window of eigenvalues to search for, and **vu** is the upper bound.

#### Note

If [vl, vu] does not contain all eigenvalues of A, then the returned factorization will be a truncated factorization.

**source**

[Base.LinAlg.eigfact!](#) – Function.

```
| eigfact!(A, [B])
```

Same as **eigfact**, but saves space by overwriting the input A (and B), instead of creating a copy.

**source**

[Base.LinAlg.hessfact](#) – Function.

```
| hessfact(A) -> Hessenberg
```

Compute the Hessenberg decomposition of A and return a Hessenberg object. If F is the factorization object, the unitary matrix can be accessed

1312with `F[ :Q]` and the Hessenberg matrix with `CHAPTER 5: LINEAR ALGEBRA`,  
the resulting type is the `HessenbergQ` object, and may be converted to  
a regular matrix with `convert(Array, _)` (or `Array(_)` for short).

Examples

```
julia> A = [4. 9. 7.; 4. 4. 1.; 4. 3. 2.]  
3×3 Array{Float64,2}:  
 4.0 9.0 7.0  
 4.0 4.0 1.0  
 4.0 3.0 2.0  
  
julia> F = hessfact(A);  
  
julia> F[ :Q] * F[ :H] * F[ :Q]'  
3×3 Array{Float64,2}:  
 4.0 9.0 7.0  
 4.0 4.0 1.0  
 4.0 3.0 2.0
```

source

`Base.LinAlg.hessfact!` – Function.

```
| hessfact!(A) -> Hessenberg
```

`hessfact!` is the same as `hessfact`, but saves space by overwriting the  
input `A`, instead of creating a copy.

source

`Base.LinAlg.schurfact` – Function.

```
| schurfact(A::StridedMatrix) -> F::Schur
```

Computes the Schur factorization of the matrix `A`. The (quasi) triangular Schur factor can be obtained from the `Schur` object `F` with either

54.1 F [STANDARD FUNCTIONS] [View source](#)  
and the orthogonal/unitary Schur vectors can be obtained with `F[:vectors]` or `F[:Z]` such that  $A = F[:vectors] * F[:Schur] * F[:vectors]'$ . The eigenvalues of  $A$  can be obtained with `F[:values]`.

Examples

```
julia> A = [5. 7.; -2. -4.]
```

```
2×2 Array{Float64,2}:
```

```
 5.0  7.0
```

```
-2.0 -4.0
```

```
julia> F = schurfact(A)
```

```
Base.LinAlg.Schur{Float64,Array{Float64,2}} with factors T and  
→ Z:
```

```
[3.0 9.0; 0.0 -2.0]
```

```
[0.961524 0.274721; -0.274721 0.961524]
```

```
and values:
```

```
[3.0, -2.0]
```

```
julia> F[:vectors] * F[:Schur] * F[:vectors]'
```

```
2×2 Array{Float64,2}:
```

```
 5.0  7.0
```

```
-2.0 -4.0
```

[source](#)

```
schurfact(A::StridedMatrix, B::StridedMatrix) -> F::  
    GeneralizedSchur
```

Computes the Generalized Schur (or QZ) factorization of the matrices  $A$  and  $B$ . The (quasi) triangular Schur factors can be obtained from the `Schur` object  $F$  with `F[:S]` and `F[:T]`, the left unitary/orthogonal Schur vectors can be obtained with `F[:left]` or `F[:Q]` and the right uni-

1314ary/orthogonal Schur vectors can be obtained with `schurfact!`

such that  $A = F[:left] * F[:S] * F[:right]'$  and  $B = F[:left] * F[:T] * F[:right]'$ .

The generalized eigenvalues of A and B can be obtained with `F[:alpha] ./ F[:beta]`.

`source`

`Base.LinAlg.schurfact!` – Function.

`| schurfact!(A::StridedMatrix) -> F::Schur`

Same as `schurfact` but uses the input argument as workspace.

`source`

`| schurfact!(A::StridedMatrix, B::StridedMatrix) -> F::GeneralizedSchur`

Same as `schurfact` but uses the input matrices A and B as workspace.

`source`

`Base.LinAlg.schur` – Function.

`| schur(A::StridedMatrix) -> T::Matrix, Z::Matrix, λ::Vector`

Computes the Schur factorization of the matrix A. The methods return the (quasi) triangular Schur factor T and the orthogonal/unitary Schur vectors Z such that  $A = Z * T * Z'$ . The eigenvalues of A are returned in the vector  $\lambda$ .

See `schurfact`.

Examples

```
julia> A = [5. 7.; -2. -4.]  
2×2 Array{Float64,2}:  
 5.0  7.0  
 -2.0 -4.0
```

```
julia> T, Z, lambda = schur(A)
([3.0 9.0; 0.0 -2.0], [0.961524 0.274721; -0.274721 0.961524],
 ↵ [3.0, -2.0])

julia> Z * Z'
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> Z * T * Z'
2×2 Array{Float64,2}:
 5.0   7.0
 -2.0  -4.0
```

**source**

```
schur(A::StridedMatrix, B::StridedMatrix) -> S::StridedMatrix,
T::StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, α::
Vector, β::Vector
```

See [schurfact](#).

**source**

[Base.LinAlg.ordschur](#) – Function.

```
ordschur(F::Schur, select::Union{Vector{Bool}, BitVector}) -> F
::Schur
```

Reorders the Schur factorization  $F$  of a matrix  $A = Z \cdot T \cdot Z'$  according to the logical array `select` returning the reordered factorization  $F$  object. The selected eigenvalues appear in the leading diagonal of  $F[ :Schur ]$  and the corresponding leading columns of  $F[ :vectors ]$  form an orthogonal/unitary basis of the corresponding right invariant subspace. In the

1316 real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

**source**

```
ordschur(T::StridedMatrix, Z::StridedMatrix, select::Union{  
    Vector{Bool}, BitVector}) -> T::StridedMatrix, Z::  
    StridedMatrix, λ::Vector
```

Reorders the Schur factorization of a real matrix  $A = Z*T*Z'$  according to the logical array `select` returning the reordered matrices  $T$  and  $Z$  as well as the vector of eigenvalues  $\lambda$ . The selected eigenvalues appear in the leading diagonal of  $T$  and the corresponding leading columns of  $Z$  form an orthogonal/unitary basis of the corresponding right invariant subspace. In the real case, a complex conjugate pair of eigenvalues must be either both included or both excluded via `select`.

**source**

```
ordschur(F::GeneralizedSchur, select::Union{Vector{Bool},  
    BitVector}) -> F::GeneralizedSchur
```

Reorders the Generalized Schur factorization  $F$  of a matrix pair  $(A, B) = (Q*S*Z', Q*T*Z')$  according to the logical array `select` and returns a `GeneralizedSchur` object  $F$ . The selected eigenvalues appear in the leading diagonal of both  $F[ :S]$  and  $F[ :T]$ , and the left and right orthogonal/unitary Schur vectors are also reordered such that  $(A, B) = F[ :Q]*(F[ :S], F[ :T])*F[ :Z]'$  still holds and the generalized eigenvalues of  $A$  and  $B$  can still be obtained with  $F[ :alpha]./F[ :beta]$ .

**source**

```
ordschur(S::StridedMatrix, T::StridedMatrix, Q::StridedMatrix,  
    Z::StridedMatrix, select) -> S::StridedMatrix, T::  
    StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, α::Vector  
    , β::Vector
```

## 54.1 REORDERED GENERALIZED Schur factorization of a matrix pair ( $A$ , $B$ )<sup>17</sup>

( $Q*S*Z'$ ,  $Q*T*Z'$ ) according to the logical array `select` and returns the matrices  $S$ ,  $T$ ,  $Q$ ,  $Z$  and vectors  $\alpha$  and  $\beta$ . The selected eigenvalues appear in the leading diagonal of both  $S$  and  $T$ , and the left and right unitary/orthogonal Schur vectors are also reordered such that  $(A, B) = Q*(S, T)*Z'$  still holds and the generalized eigenvalues of  $A$  and  $B$  can still be obtained with  $\alpha ./ \beta$ .

**source**

`Base.LinAlg.ordschur!` – Function.

```
ordschur!(F::Schur, select::Union{Vector{Bool}, BitVector}) -> F
    ::Schur
```

Same as `ordschur` but overwrites the factorization  $F$ .

**source**

```
ordschur!(T::StridedMatrix, Z::StridedMatrix, select::Union{
    Vector{Bool}, BitVector}) -> T::StridedMatrix, Z::
    StridedMatrix, λ::Vector
```

Same as `ordschur` but overwrites the input arguments.

**source**

```
ordschur!(F::GeneralizedSchur, select::Union{Vector{Bool},
    BitVector}) -> F::GeneralizedSchur
```

Same as `ordschur` but overwrites the factorization  $F$ .

**source**

```
ordschur!(S::StridedMatrix, T::StridedMatrix, Q::StridedMatrix,
    Z::StridedMatrix, select) -> S::StridedMatrix, T::
    StridedMatrix, Q::StridedMatrix, Z::StridedMatrix, α::Vector
    , β::Vector
```

`source`

`Base.LinAlg.svdfact` – Function.

```
| svdfact(A; full::Bool = false) -> SVD
```

Compute the singular value decomposition (SVD) of  $A$  and return an SVD object.

$U$ ,  $S$ ,  $V$  and  $Vt$  can be obtained from the factorization  $F$  with  $F[ :U]$ ,  $F[ :S]$ ,  $F[ :V]$  and  $F[ :Vt]$ , such that  $A = U * \text{Diagonal}(S) * Vt$ . The algorithm produces  $Vt$  and hence  $Vt$  is more efficient to extract than  $V$ . The singular values in  $S$  are sorted in descending order.

If `full = false` (default), a "thin" SVD is returned. For a  $M \times N$  matrix  $A$ ,  $U$  is  $M \times M$  for a "full" SVD (`full = true`) and  $M \times \min(M, N)$  for a "thin" SVD.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0.
```

```
    ↵ 2. 0. 0. 0.]
```

```
4×5 Array{Float64,2}:
```

1.0	0.0	0.0	0.0	2.0
0.0	0.0	3.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.0	2.0	0.0	0.0	0.0

```
julia> F = svdfact(A);
```

```
julia> F[:U] * Diagonal(F[:S]) * F[:Vt]
```

```
4×5 Array{Float64,2}:
```

1.0	0.0	0.0	0.0	2.0
0.0	0.0	3.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.0	2.0	0.0	0.0	0.0

54.1|  $\begin{matrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 & 0.0 & 0.0 \end{matrix}$

1319

**source**

| `svdfact(A, B) -> GeneralizedSVD`

Compute the generalized SVD of A and B, returning a `GeneralizedSVD` factorization object F, such that  $A = F[ :U]*F[ :D1]*F[ :R0]*F[ :Q]'$  and  $B = F[ :V]*F[ :D2]*F[ :R0]*F[ :Q]'$ .

For an M-by-N matrix A and P-by-N matrix B,

$F[ :U]$  is a M-by-M orthogonal matrix,

$F[ :V]$  is a P-by-P orthogonal matrix,

$F[ :Q]$  is a N-by-N orthogonal matrix,

$F[ :R0]$  is a  $(K+L)$ -by-N matrix whose rightmost  $(K+L)$ -by- $(K+L)$  block is nonsingular upper block triangular,

$F[ :D1]$  is a M-by- $(K+L)$  diagonal matrix with 1s in the first K entries,

$F[ :D2]$  is a P-by- $(K+L)$  matrix whose top right L-by-L block is diagonal,

$K+L$  is the effective numerical rank of the matrix  $[A; B]$ .

The entries of  $F[ :D1]$  and  $F[ :D2]$  are related, as explained in the LAPACK documentation for the `generalized SVD` and the `xGGSVD3` routine which is called underneath (in LAPACK 3.6.0 and newer).

**source**

[Base.LinAlg.svdfact!](#) – Function.

| `svdfact!(A; full::Bool = false) -> SVD`

`svdfact!` is the same as `svdfact`, but saves space by overwriting the input A, instead of creating a copy.

1320 **source**

CHAPTER 54. LINEAR ALGEBRA

```
| svdfact!(A, B) -> GeneralizedSVD
```

**svdfact!** is the same as **svdfact**, but modifies the arguments A and B in-place, instead of making copies.

**source**

[Base.LinAlg.svd](#) – Function.

```
| svd(A; full::Bool = false) -> U, S, V
```

Computes the SVD of A, returning U, vector S, and V such that  $A == U * \text{Diagonal}(S) * V'$ . The singular values in S are sorted in descending order.

If `full = false` (default), a "thin" SVD is returned. For a  $M \times N$  matrix A, U is  $M \times M$  for a "full" SVD (`full = true`) and  $M \times \min(M, N)$  for a "thin" SVD.

`svd` is a wrapper around **svdfact**, extracting all parts of the SVD factorization to a tuple. Direct use of **svdfact** is therefore more efficient.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0.
           ↵ 2. 0. 0. 0.]
```

4×5 Array{Float64,2}:

1.0	0.0	0.0	0.0	2.0
0.0	0.0	3.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0
0.0	2.0	0.0	0.0	0.0

```
julia> U, S, V = svd(A);
```

```
julia> U * Diagonal(S) * V'
```

4×5 Array{Float64,2}:

54.1| STANDARD FUNCTIONS  
1.0 0.0 0.0 0.0 2.0  
0.0 0.0 3.0 0.0 0.0  
0.0 0.0 0.0 0.0 0.0  
0.0 2.0 0.0 0.0 0.0

1321

**source**

| svd(A, B) -> U, V, Q, D1, D2, R0

Wrapper around `svdfact` extracting all parts of the factorization to a tuple. Direct use of `svdfact` is therefore generally more efficient. The function returns the generalized SVD of A and B, returning U, V, Q, D1, D2, and R0 such that A = U\*D1\*R0\*Q' and B = V\*D2\*R0\*Q'.

**source**

[Base.LinAlg.svdvals](#) – Function.

| svdvals(A)

Returns the singular values of A in descending order.

Examples

```
julia> A = [1. 0. 0. 0. 2.; 0. 0. 3. 0. 0.; 0. 0. 0. 0. 0.; 0.  
         ↳ 2. 0. 0. 0.]  
4×5 Array{Float64,2}:  
 1.0  0.0  0.0  0.0  2.0  
 0.0  0.0  3.0  0.0  0.0  
 0.0  0.0  0.0  0.0  0.0  
 0.0  2.0  0.0  0.0  0.0  
  
julia> svdvals(A)  
4-element Array{Float64,1}:  
 3.0  
 2.23606797749979
```

```
1322 2.0  
      0.0
```

## CHAPTER 54. LINEAR ALGEBRA

**source**

```
| svdvals(A, B)
```

Return the generalized singular values from the generalized singular value decomposition of A and B. See also [svdfact](#).

**source**

[Base.LinAlg.Givens](#) – Type.

```
| LinAlg.Givens(i1,i2,c,s) -> G
```

A Givens rotation linear operator. The fields `c` and `s` represent the cosine and sine of the rotation angle, respectively. The `Givens` type supports left multiplication `G*A` and conjugated transpose right multiplication `A*G'`. The type doesn't have a `size` and can therefore be multiplied with matrices of arbitrary size as long as `i2<=size(A,2)` for `G*A` or `i2<=size(A,1)` for `A*G'`.

See also: [givens](#)

**source**

[Base.LinAlg.givens](#) – Function.

```
| givens(f::T, g::T, i1::Integer, i2::Integer) where {T} -> (G::  
      Givens, r::T)
```

Computes the Givens rotation `G` and scalar `r` such that for any vector `x` where

```
| x[i1] = f  
| x[i2] = g
```

the result of the multiplication

has the property that

$$\begin{cases} y[i1] = r \\ y[i2] = 0 \end{cases}$$

See also: [LinAlg.Givens](#)

**source**

$$\begin{cases} \text{givens}(A::\text{AbstractArray}, i1::\text{Integer}, i2::\text{Integer}, j::\text{Integer}) \\ \quad \rightarrow (G::\text{Givens}, r) \end{cases}$$

Computes the Givens rotation  $G$  and scalar  $r$  such that the result of the multiplication

$$B = G*A$$

has the property that

$$\begin{cases} B[i1, j] = r \\ B[i2, j] = 0 \end{cases}$$

See also: [LinAlg.Givens](#)

**source**

$$\begin{cases} \text{givens}(x::\text{AbstractVector}, i1::\text{Integer}, i2::\text{Integer}) \rightarrow (G:: \\ \quad \text{Givens}, r) \end{cases}$$

Computes the Givens rotation  $G$  and scalar  $r$  such that the result of the multiplication

$$B = G*x$$

has the property that

$$\begin{cases} B[i1] = r \\ B[i2] = 0 \end{cases}$$

`source`

[Base.LinAlg.triu](#) – Function.

`| triu(M)`

Upper triangle of a matrix.

Examples

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0
```

```
julia> triu(a)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
 0.0  1.0  1.0  1.0
 0.0  0.0  1.0  1.0
 0.0  0.0  0.0  1.0
```

`source`

`| triu(M, k::Integer)`

Returns the upper triangle of  $M$  starting from the  $k$ th superdiagonal.

Examples

```
julia> a = ones(4,4)
4×4 Array{Float64,2}:
 1.0  1.0  1.0  1.0
```

```
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0
```

```
julia> triu(a,3)
```

```
4×4 Array{Float64,2}:  
0.0 0.0 0.0 1.0  
0.0 0.0 0.0 0.0  
0.0 0.0 0.0 0.0  
0.0 0.0 0.0 0.0
```

```
julia> triu(a,-3)
```

```
4×4 Array{Float64,2}:  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0
```

**source**

[Base.LinAlg.triu!](#) – Function.

| **triu!(M)**

Upper triangle of a matrix, overwriting M in the process. See also [triu](#).

**source**

| **triu!(M, k::Integer)**

Returns the upper triangle of M starting from the kth superdiagonal, overwriting M in the process.

Examples

1326 CHAPTER 54. LINEAR ALGEBRA

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3  
        ↵ 4 5]  
5×5 Array{Int64,2}:  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5
```

```
julia> triu!(M, 1)
```

```
5×5 Array{Int64,2}:  
0 2 3 4 5  
0 0 3 4 5  
0 0 0 4 5  
0 0 0 0 5  
0 0 0 0 0
```

source

[Base.LinAlg.tril](#) – Function.

```
| tril(M)
```

Lower triangle of a matrix.

Examples

```
julia> a = ones(4,4)  
4×4 Array{Float64,2}:  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0  
1.0 1.0 1.0 1.0
```

```
julia> tril(a)
```

4×4 Array{Float64,2}:

1.0	0.0	0.0	0.0
1.0	1.0	0.0	0.0
1.0	1.0	1.0	0.0
1.0	1.0	1.0	1.0

source

```
| tril(M, k::Integer)
```

Returns the lower triangle of  $M$  starting from the  $k$ th superdiagonal.

Examples

```
julia> a = ones(4,4)
```

4×4 Array{Float64,2}:

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0

```
julia> tril(a,3)
```

```
4×4 Array{Float64,2}:
```

1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0
1.0	1.0	1.0	1.0

```
julia> tril(a,-3)
```

```
4×4 Array{Float64,2}:
```

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

```
1328 | 0.0  0.0  0.0  0.0  
| 1.0  0.0  0.0  0.0
```

## CHAPTER 54. LINEAR ALGEBRA

**source**

[Base.LinAlg.tril!](#) – Function.

```
| tril!(M)
```

Lower triangle of a matrix, overwriting **M** in the process. See also [tril](#).

**source**

```
| tril!(M, k::Integer)
```

Returns the lower triangle of **M** starting from the **k**th superdiagonal, overwriting **M** in the process.

Examples

```
julia> M = [1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3 4 5; 1 2 3  
↪ 4 5]
```

```
5×5 Array{Int64,2}:
```

```
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5
```

```
julia> tril!(M, 2)
```

```
5×5 Array{Int64,2}:
```

```
1 2 3 0 0  
1 2 3 4 0  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5
```

[Base.LinAlg.diagind](#) – Function.

```
| diagind(M, k::Integer=0)
```

An `AbstractRange` giving the indices of the  $k$ th diagonal of the matrix `M`.

Examples

```
| julia> A = [1 2 3; 4 5 6; 7 8 9]
| 3×3 Array{Int64,2}:
|   1  2  3
|   4  5  6
|   7  8  9
|
| julia> diagind(A, -1)
| 2:4:6
```

[source](#)

[Base.LinAlg.diag](#) – Function.

```
| diag(M, k::Integer=0)
```

The  $k$ th diagonal of a matrix, as a vector.

See also: [diagm](#)

Examples

```
| julia> A = [1 2 3; 4 5 6; 7 8 9]
| 3×3 Array{Int64,2}:
|   1  2  3
|   4  5  6
|   7  8  9
|
| julia> diag(A, 1)
```

```
1330 2-element Array{Int64,1}:
  2
  6
```

## CHAPTER 54. LINEAR ALGEBRA

[source](#)

[Base.LinAlg.diagm](#) – Function.

```
diagm(kv::Pair{<:Integer,<:AbstractVector}...)
```

Construct a square matrix from **Pairs** of diagonals and vectors. Vector `kv.second` will be placed on the `kv.first` diagonal. `diagm` constructs a full matrix; if you want storage-efficient versions with fast arithmetic, see [Diagonal](#), [Bidiagonal](#) [Tridiagonal](#) and [SymTridiagonal](#).

See also: [spdiagm](#)

Examples

```
julia> diagm(1 => [1,2,3])
4×4 Array{Int64,2}:
 0  1  0  0
 0  0  2  0
 0  0  0  3
 0  0  0  0
```

```
julia> diagm(1 => [1,2,3], -1 => [4,5])
4×4 Array{Int64,2}:
 0  1  0  0
 4  0  2  0
 0  5  0  3
 0  0  0  0
```

[source](#)

[Base.LinAlg.scale!](#) – Function.

```
scale!(A, b)
```

```
scale!(b, A)
```

Scale an array  $A$  by a scalar  $b$  overwriting  $A$  in-place.

If  $A$  is a matrix and  $b$  is a vector, then `scale!(A, b)` scales each column  $i$  of  $A$  by  $b[i]$  (similar to  $A * \text{Diagonal}(b)$ ), while `scale!(b, A)` scales each row  $i$  of  $A$  by  $b[i]$  (similar to  $\text{Diagonal}(b) * A$ ), again operating in-place on  $A$ . An `InexactError` exception is thrown if the scaling produces a number not representable by the element type of  $A$ , e.g. for integer types.

Examples

```
julia> a = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> b = [1; 2]
2-element Array{Int64,1}:
 1
 2
```

```
julia> scale!(a,b)
2×2 Array{Int64,2}:
 1  4
 3  8
```

```
julia> a = [1 2; 3 4];
```

```
julia> b = [1; 2];
```

```
1332 julia> scale!(b,a)
2×2 Array{Int64,2}:
 1  2
 6  8
```

## CHAPTER 54. LINEAR ALGEBRA

[source](#)

[Base.LinAlg.rank](#) – Function.

```
| rank(A[, tol::Real])
```

Compute the rank of a matrix by counting how many singular values of A have magnitude greater than `tol`\* $\sigma$  where  $\sigma$  is A's largest singular values. By default, the value of `tol` is the smallest dimension of A multiplied by the `eps` of the `eltype` of A.

Examples

```
julia> rank(Matrix(I, 3, 3))
3

julia> rank(diagm(0 => [1, 0, 2]))
2

julia> rank(diagm(0 => [1, 0.001, 2]), 0.1)
2

julia> rank(diagm(0 => [1, 0.001, 2]), 0.00001)
3
```

[source](#)

[Base.LinAlg.norm](#) – Function.

```
| norm(A::AbstractArray, p::Real=2)
```

54.1. STANDARD FUNCTIONS a vector or the operator norm of a matrix<sup>1838</sup>, defaulting to the 2-norm.

```
norm(A::AbstractVector, p::Real=2)
```

For vectors, this is equivalent to `vecnorm` and equal to:

$$\|A\|_p = \left( \sum_{i=1}^n |a_i|^p \right)^{1/p}$$

with  $a_i$  the entries of  $A$  and  $n$  its length.

$p$  can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `norm(A, Inf)` returns the largest value in `abs(A)`, whereas `norm(A, -Inf)` returns the smallest.

Examples

```
julia> v = [3, -2, 6]
3-element Array{Int64,1}:
 3
 -2
  6
```

```
julia> norm(v)
7.0
```

```
julia> norm(v, Inf)
6.0
```

`source`

```
norm(A::AbstractMatrix, p::Real=2)
```

1334For matrices, the matrix norm induced by the vector norm  $\|\cdot\|_p$  is called the  **$\|A\|_p$** . CHAPTER 5 AND LINEAR ALGEBRA

valid values of  $p$  are 1, 2, or Inf. (Note that for sparse matrices,  $p=2$  is currently not implemented.) Use **vecnorm** to compute the Frobenius norm.

When  $p=1$ , the matrix norm is the maximum absolute column sum of  $A$ :

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$$

with  $a_{ij}$  the entries of  $A$ , and  $m$  and  $n$  its dimensions.

When  $p=2$ , the matrix norm is the spectral norm, equal to the largest singular value of  $A$ .

When  $p=\text{Inf}$ , the matrix norm is the maximum absolute row sum of  $A$ :

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$$

Examples

```
julia> A = [1 -2 -3; 2 3 -1]
2×3 Array{Int64,2}:
 1  -2  -3
 2   3  -1
```

```
julia> norm(A, Inf)
6.0
```

source

```
| norm(x::Number, p::Real=2)
```

For numbers, return  $(|x|^p)^{1/p}$ . This is equivalent to **vecnorm**.

source

```
| norm(A::RowVector, q::Real=2)
```

54.1 STANDARD FUNCTIONS 1325

The  $q$ -norm of  $A$ , which is equivalent to the  $p$ -norm with value  $p = q/(q-1)$ . They coincide at  $p = q = 2$ .

The difference in norm between a vector space and its dual arises to preserve the relationship between duality and the inner product, and the result is consistent with the  $p$ -norm of  $1 \times n$  matrix.

Examples

```
julia> v = [1; im];
```

```
julia> vc = v';
```

```
julia> norm(vc, 1)
```

```
1.0
```

```
julia> norm(v, 1)
```

```
2.0
```

```
julia> norm(vc, 2)
```

```
1.4142135623730951
```

```
julia> norm(v, 2)
```

```
1.4142135623730951
```

```
julia> norm(vc, Inf)
```

```
2.0
```

```
julia> norm(v, Inf)
```

```
1.0
```

source

[Base.LinAlg.vecnorm](#) – Function.

```
1336 vecnorm(A, p::Real=2)
```

## CHAPTER 54. LINEAR ALGEBRA

For any iterable container A (including arrays of any dimension) of numbers (or any element type for which `norm` is defined), compute the  $p$ -norm (defaulting to  $p=2$ ) as if A were a vector of the corresponding length.

The  $p$ -norm is defined as:

$$\|A\|_p = \left( \sum_{i=1}^n |a_i|^p \right)^{1/p}$$

with  $a_i$  the entries of  $A$  and  $n$  its length.

$p$  can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `vecnorm(A, Inf)` returns the largest value in `abs(A)`, whereas `vecnorm(A, -Inf)` returns the smallest. If A is a matrix and  $p=2$ , then this is equivalent to the Frobenius norm.

Examples

```
julia> vecnorm([1 2 3; 4 5 6; 7 8 9])
```

```
16.881943016134134
```

```
julia> vecnorm([1 2 3 4 5 6 7 8 9])
```

```
16.881943016134134
```

`source`

```
| vecnorm(x::Number, p::Real=2)
```

For numbers, return  $(|x|^p)^{1/p}$ .

Examples

```
julia> vecnorm(2, 1)
```

```
2
```

```
julia> vecnorm(-2, 1)
```

```
2
```

```
julia> vecnorm(2, 2)
```

```
2
```

```
julia> vecnorm(-2, 2)
```

```
2
```

```
julia> vecnorm(2, Inf)
```

```
2
```

```
julia> vecnorm(-2, Inf)
```

```
2
```

`source`

`Base.LinAlg.normalize!` – Function.

```
|normalize!(v::AbstractVector, p::Real=2)
```

Normalize the vector `v` in-place so that its `p`-norm equals unity, i.e. `norm(v, p) == 1`. See also `normalize` and `vecnorm`.

`source`

`Base.LinAlg.normalize` – Function.

```
|normalize(v::AbstractVector, p::Real=2)
```

Normalize the vector `v` so that its `p`-norm equals unity, i.e. `norm(v, p) == vecnorm(v, p) == 1`. See also `normalize!` and `vecnorm`.

Examples

1338 **julia>** a = [1, 2, 4];

CHAPTER 54. LINEAR ALGEBRA

```
julia> b = normalize(a)
3-element Array{Float64,1}:
 0.2182178902359924
 0.4364357804719848
 0.8728715609439696
```

```
julia> norm(b)
```

```
1.0
```

```
julia> c = normalize(a, 1)
3-element Array{Float64,1}:
 0.14285714285714285
 0.2857142857142857
 0.5714285714285714
```

```
julia> norm(c, 1)
```

```
1.0
```

**source**

[Base.LinAlg.cond](#) – Function.

```
| cond(M, p::Real=2)
```

Condition number of the matrix M, computed using the operator p-norm.

Valid values for p are 1, 2 (default), or Inf.

**source**

[Base.LinAlg.condskeel](#) – Function.

```
| condskeel(M, [x, p::Real=Inf])
```

$$\kappa_S(M, p) = \left\| |M| |M^{-1}| \right\|_p$$

$$\kappa_S(M, x, p) = \left\| |M| |M^{-1}| |x| \right\|_p$$

Skeel condition number  $\kappa_S$  of the matrix  $M$ , optionally with respect to the vector  $x$ , as computed using the operator  $p$ -norm.  $|M|$  denotes the matrix of (entry wise) absolute values of  $M$ ;  $|M|_{ij} = |M_{ij}|$ . Valid values for  $p$  are 1, 2 and Inf (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

**source**

[Base.LinAlg.trace](#) – Function.

**trace( $M$ )**

Matrix trace. Sums the diagonal elements of  $M$ .

Examples

```
julia> A = [1 2; 3 4]
2×2 Array{Int64,2}:
 1  2
 3  4
```

```
julia> trace(A)
5
```

**source**

[Base.LinAlg.det](#) – Function.

**det( $M$ )**

Matrix determinant.

Examples

```
1340 julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> det(M)
2.0
```

## CHAPTER 54. LINEAR ALGEBRA

[source](#)

[Base.LinAlg.logdet](#) – Function.

```
| logdet(M)
```

Log of matrix determinant. Equivalent to `log(det(M))`, but may provide increased accuracy and/or speed.

Examples

```
julia> M = [1 0; 2 2]
2×2 Array{Int64,2}:
 1  0
 2  2

julia> logdet(M)
0.6931471805599453

julia> logdet(Matrix(I, 3, 3))
0.0
```

[source](#)

[Base.LinAlg.logabsdet](#) – Function.

```
| logabsdet(M)
```

54.1 STANDARD FUNCTIONS

matrix determinant. Equivalent to `(log(abs(det(M))), sign(det(M)))`, but may provide increased accuracy and/or speed.

`source`

`Base.inv` – Method.

`| inv(M)`

Matrix inverse. Computes matrix  $N$  such that  $M * N = I$ , where  $I$  is the identity matrix. Computed by solving the left-division  $N = M \setminus I$ .

Examples

`julia> M = [2 5; 1 3]`

`2×2 Array{Int64,2}:`

`2 5  
1 3`

`julia> N = inv(M)`

`2×2 Array{Float64,2}:`

`3.0 -5.0  
-1.0 2.0`

`julia> M*N == N*M == Matrix(I, 2, 2)`

`true`

`source`

`Base.LinAlg.pinv` – Function.

`| pinv(M[, tol::Real])`

Computes the Moore–Penrose pseudoinverse.

For matrices  $M$  with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values above a given threshold, `tol`.

tended application of the pseudoinverse. The default value of `tol` is `eps(real(float(one(eltype(M))))) * maximum(size(A))`, which is essentially machine epsilon for the real part of a matrix element multiplied by the larger matrix dimension. For inverting dense ill-conditioned matrices in a least-squares sense, `tol = sqrt(eps(real(float(one(eltype(M))))))` is recommended.

For more information, see <sup>4</sup>, <sup>5</sup>, <sup>6</sup>, <sup>7</sup>.

## Examples

```
julia> M = [1.5 1.3; 1.2 1.9]
```

```
2×2 Array{Float64,2}:
```

```
1.5 1.3
```

```
1.2 1.9
```

```
julia> N = pinv(M)
```

```
2×2 Array{Float64,2}:
```

```
1.47287 -1.00775
```

```
-0.930233 1.16279
```

```
julia> M * N
```

```
2×2 Array{Float64,2}:
```

```
1.0 -2.22045e-16
```

```
4.44089e-16 1.0
```

<sup>4</sup>Issue 8859, "Fix least squares", <https://github.com/JuliaLang/julia/pull/8859>

<sup>5</sup>Kruse Björck, "Numerical Methods for Least Squares Problems", SIAM Press, Philadelphia, 1996, "Other Titles in Applied Mathematics", Vol. 51. doi:[10.1137/1.9781611971484](https://doi.org/10.1137/1.9781611971484)

<sup>6</sup>G. W. Stewart, "Rank Degeneracy", SIAM Journal on Scientific and Statistical Computing, 5(2), 1984, 403–413. doi:[10.1137/0905030](https://doi.org/10.1137/0905030)

<sup>7</sup>Konstantinos Konstantinides and Kung Yao, "Statistical analysis of effective singular values in matrix rank determination", IEEE Transactions on Acoustics, Speech and Signal Processing, 36(5), 1988, 757–763. doi:[10.1109/29.1585](https://doi.org/10.1109/29.1585)

`Base.LinAlg.nullspace` – Function.

```
| nullspace(M)
```

Basis for nullspace of `M`.

Examples

```
julia> M = [1 0 0; 0 1 0; 0 0 0]
3×3 Array{Int64,2}:
 1  0  0
 0  1  0
 0  0  0

julia> nullspace(M)
3×1 Array{Float64,2}:
 0.0
 0.0
 1.0
```

`source`

`Base.repmat` – Function.

```
| repmat(A, m::Integer, n::Integer=1)
```

Construct a matrix by repeating the given matrix (or vector) `m` times in dimension 1 and `n` times in dimension 2.

Examples

```
julia> repmat([1, 2, 3], 2)
6-element Array{Int64,1}:
 1
 2
```

```
3  
1  
2  
3
```

```
julia> repmat([1, 2, 3], 2, 3)
```

```
6×3 Array{Int64,2}:
```

```
1 1 1  
2 2 2  
3 3 3  
1 1 1  
2 2 2  
3 3 3
```

source

Base.kron – Function.

```
|kron(A, B)
```

Kronecker tensor product of two vectors or two matrices.

Examples

```
julia> A = [1 2; 3 4]
```

```
2×2 Array{Int64,2}:
```

```
1 2  
3 4
```

```
julia> B = [im 1; 1 -im]
```

```
2×2 Array{Complex{Int64},2}:
```

```
0+1im 1+0im  
1+0im 0-1im
```

```
julia> kron(A, B)
```

4×4 Array{Complex{Int64},2}:

0+1im	1+0im	0+2im	2+0im
1+0im	0-1im	2+0im	0-2im
0+3im	3+0im	0+4im	4+0im
3+0im	0-3im	4+0im	0-4im

[source](#)

[Base.SparseArrays.blkdiag](#) – Function.

```
| blkdiag(A...)
```

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Examples

```
julia> blkdiag(sparse(2I, 3, 3), sparse(4I, 2, 2))
```

5×5 SparseMatrixCSC{Int64, Int64} with 5 stored entries:

[1, 1] = 2
[2, 2] = 2
[3, 3] = 2
[4, 4] = 4
[5, 5] = 4

[source](#)

[Base.LinAlg.linreg](#) – Function.

```
| linreg(x, y)
```

Perform simple linear regression using Ordinary Least Squares. Returns a and b such that a + b\*x is the closest straight line to the given points (x, y), i.e., such that the squared error between y and a + b\*x is minimized.

Examples

1346 **using** PyPlot

CHAPTER 54. LINEAR ALGEBRA

```
x = 1.0:12.0
y = [5.5, 6.3, 7.6, 8.8, 10.9, 11.79, 13.48, 15.02, 17.77,
    ↵ 20.81, 22.0, 22.99]
a, b = linreg(x, y)          # Linear regression
plot(x, y, "o")             # Plot (x, y) points
plot(x, a + b*x)            # Plot line determined by linear
    ↵ regression
```

See also:

\, cov, std, mean.

source

Base.exp – Method.

```
| exp(A::AbstractMatrix)
```

Compute the matrix exponential of A, defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian A, an eigendecomposition ([eigfact](#)) is used, otherwise the scaling and squaring algorithm (see <sup>8</sup>) is chosen.

Examples

```
julia> A = Matrix(1.0I, 2, 2)
2×2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0
```

---

<sup>8</sup>Nicholas J. Higham, "The squaring and scaling method for the matrix exponential revisited", SIAM Journal on Matrix Analysis and Applications, 26(4), 2005, 1179–1193.  
doi:[10.1137/090768539](https://doi.org/10.1137/090768539)

```
julia> exp(A)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828
```

**source**

**Base.log** – Method.

```
| log(A{T}::StridedMatrix{T})
```

If  $A$  has no negative real eigenvalue, compute the principal matrix logarithm of  $A$ , i.e. the unique matrix  $x$  such that  $e^x = A$  and  $-\pi < \text{Im}(\lambda) < \pi$  for all the eigenvalues  $\lambda$  of  $x$ . If  $A$  has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If  $A$  is symmetric or Hermitian, its eigendecomposition (**eigfact**) is used, if  $A$  is triangular an improved version of the inverse scaling and squaring method is employed (see <sup>9</sup> and <sup>10</sup>). For general matrices, the complex Schur form (**schur**) is computed and the triangular algorithm is used on the triangular factor.

Examples

```
julia> A = Matrix(2.7182818*I, 2, 2)
2×2 Array{Float64,2}:
 2.71828  0.0
 0.0      2.71828

julia> log(A)
```

---

<sup>9</sup>Awad H. Al-Mohy and Nicholas J. Higham, "Improved inverse scaling and squaring algorithms for the matrix logarithm", SIAM Journal on Scientific Computing, 34(4), 2012, C153–C169. [doi:10.1137/110852553](https://doi.org/10.1137/110852553)

<sup>10</sup>Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Frchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394–C410. [doi:10.1137/120885991](https://doi.org/10.1137/120885991)

```
1348 2×2 Array{Float64,2}:
```

1.0	0.0
0.0	1.0

## CHAPTER 54. LINEAR ALGEBRA

**source**

[Base.sqrt](#) – Method.

```
| sqrt(A::AbstractMatrix)
```

If  $A$  has no negative real eigenvalues, compute the principal matrix square root of  $A$ , that is the unique matrix  $x$  with eigenvalues having positive real part such that  $x^2 = A$ . Otherwise, a nonprincipal square root is returned.

If  $A$  is symmetric or Hermitian, its eigendecomposition ([eigfact](#)) is used to compute the square root. Otherwise, the square root is determined by means of the Björck–Hammarling method<sup>11</sup>, which computes the complex Schur form ([schur](#)) and then the complex square root of the triangular factor.

Examples

```
julia> A = [4 0; 0 4]
```

```
2×2 Array{Int64,2}:
```

4	0
0	4

```
julia> sqrt(A)
```

```
2×2 Array{Float64,2}:
```

2.0	0.0
0.0	2.0

**source**

---

<sup>11</sup>Krige Björck and Sven Hammarling, "A Schur method for the square root of a matrix", Linear Algebra and its Applications, 52–53, 1983, 127–140. doi:[10.1016/0024-3795\(83\)80010-X](https://doi.org/10.1016/0024-3795(83)80010-X)

```
| cos(A::AbstractMatrix)
```

Compute the matrix cosine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition ([eigfact](#)) is used to compute the cosine. Otherwise, the cosine is determined by calling [exp](#).

Examples

```
| julia> cos(ones(2, 2))  
2×2 Array{Float64,2}:  
    0.291927  -0.708073  
   -0.708073   0.291927
```

[source](#)

[Base.sin](#) – Method.

```
| sin(A::AbstractMatrix)
```

Compute the matrix sine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition ([eigfact](#)) is used to compute the sine. Otherwise, the sine is determined by calling [exp](#).

Examples

```
| julia> sin(ones(2, 2))  
2×2 Array{Float64,2}:  
    0.454649  0.454649  
   0.454649  0.454649
```

[source](#)

[Base.Math.sinCos](#) – Method.

```
| sinCos(A::AbstractMatrix)
```

## 1350 Compute the matrix sine and cosine of a square matrix

### CHAPTER 54 LINEAR ALGEBRA

Examples

```
julia> S, C = sincos(ones(2, 2));
```

```
julia> S
```

```
2×2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649
```

```
julia> C
```

```
2×2 Array{Float64,2}:
 0.291927  -0.708073
 -0.708073  0.291927
```

source

[Base.tan](#) – Method.

```
| tan(A::AbstractMatrix)
```

Compute the matrix tangent of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition ([eigfact](#)) is used to compute the tangent. Otherwise, the tangent is determined by calling [exp](#).

Examples

```
julia> tan(ones(2, 2))
```

```
2×2 Array{Float64,2}:
 -1.09252  -1.09252
 -1.09252  -1.09252
```

source

```
| sec(A::AbstractMatrix)
```

Compute the matrix secant of a square matrix A.

**source**

**Base.Math.sec** – Method.

```
| csc(A::AbstractMatrix)
```

Compute the matrix cosecant of a square matrix A.

**source**

**Base.Math.csc** – Method.

```
| cot(A::AbstractMatrix)
```

Compute the matrix cotangent of a square matrix A.

**source**

**Base.cosh** – Method.

```
| cosh(A::AbstractMatrix)
```

Compute the matrix hyperbolic cosine of a square matrix A.

**source**

**Base.sinh** – Method.

```
| sinh(A::AbstractMatrix)
```

Compute the matrix hyperbolic sine of a square matrix A.

**source**

**Base.tanh** – Method.

```
| tanh(A::AbstractMatrix)
```

**source**

**Base.Math.sech** – Method.

| **sech(A::AbstractMatrix)**

Compute the matrix hyperbolic secant of square matrix A.

**source**

**Base.Math.csch** – Method.

| **csch(A::AbstractMatrix)**

Compute the matrix hyperbolic cosecant of square matrix A.

**source**

**Base.Math.coth** – Method.

| **coth(A::AbstractMatrix)**

Compute the matrix hyperbolic cotangent of square matrix A.

**source**

**Base.acos** – Method.

| **acos(A::AbstractMatrix)**

Compute the inverse matrix cosine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (**eigfact**) is used to compute the inverse cosine. Otherwise, the inverse cosine is determined by using **log** and **sqrt**. For the theory and logarithmic formulas used to compute this function, see <sup>12</sup>.

### Examples

---

<sup>12</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

54.1 STANDARD FUNCTIONS  
**julia>** `acos(cos([0.5 0.1; -0.2 0.3]))`

1353

```
2×2 Array{Complex{Float64},2}:
  0.5-8.32667e-17im  0.1-2.77556e-17im
  -0.2+2.77556e-16im  0.3-3.46945e-16im
```

**source**

**Base.asin** – Method.

```
| asin(A::AbstractMatrix)
```

Compute the inverse matrix sine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (**eigfact**) is used to compute the inverse sine. Otherwise, the inverse sine is determined by using **log** and **sqrt**. For the theory and logarithmic formulas used to compute this function, see <sup>13</sup>.

Examples

**julia>** `asin(sin([0.5 0.1; -0.2 0.3]))`

```
2×2 Array{Complex{Float64},2}:
  0.5-4.16334e-17im  0.1-5.55112e-17im
  -0.2+9.71445e-17im  0.3-1.249e-16im
```

**source**

**Base.atan** – Method.

```
| atan(A::AbstractMatrix)
```

Compute the inverse matrix tangent of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (**eigfact**) is used to compute the inverse tangent. Otherwise, the inverse tangent is de-

---

<sup>13</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

1354etermined by using [log](#). For the theory and algorithms, see <sup>14</sup>.

## CHAPTER 154 LINEAR ALGEBRA

compute this function, see <sup>14</sup>.

Examples

```
julia> atan(tan([0.5 0.1; -0.2 0.3]))  
2×2 Array{Complex{Float64},2}:  
 0.5+1.38778e-17im  0.1-2.77556e-17im  
 -0.2+6.93889e-17im  0.3-4.16334e-17im
```

[source](#)

[Base.Math.asec](#) – Method.

```
asec(A::AbstractMatrix)
```

Compute the inverse matrix secant of A.

[source](#)

[Base.Math.acsc](#) – Method.

```
acsc(A::AbstractMatrix)
```

Compute the inverse matrix cosecant of A.

[source](#)

[Base.Math.acot](#) – Method.

```
acot(A::AbstractMatrix)
```

Compute the inverse matrix cotangent of A.

[source](#)

[Base.acosh](#) – Method.

---

<sup>14</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

Compute the inverse hyperbolic matrix cosine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see <sup>15</sup>.

**source**

**Base.asinh** – Method.

| **asinh(A::AbstractMatrix)**

Compute the inverse hyperbolic matrix sine of a square matrix A. For the theory and logarithmic formulas used to compute this function, see <sup>16</sup>.

**source**

**Base.atanh** – Method.

| **atanh(A::AbstractMatrix)**

Compute the inverse hyperbolic matrix tangent of a square matrix A. For the theory and logarithmic formulas used to compute this function, see <sup>17</sup>.

**source**

**Base.Math.asech** – Method.

| **asech(A::AbstractMatrix)**

Compute the inverse matrix hyperbolic secant of A.

**source**

---

<sup>15</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

<sup>16</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

<sup>17</sup>Mary Aprahamian and Nicholas J. Higham, "Matrix Inverse Trigonometric and Inverse Hyperbolic Functions: Theory and Algorithms", MIMS EPrint: 2016.4. <https://doi.org/10.1137/16M1057577>

[Base.Math.acsch](#) – Method.

CHAPTER 54. LINEAR ALGEBRA

```
| acsch(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cosecant of A.

**source**

[Base.Math.acoth](#) – Method.

```
| acoth(A::AbstractMatrix)
```

Compute the inverse matrix hyperbolic cotangent of A.

**source**

[Base.LinAlg.lyap](#) – Function.

```
| lyap(A, C)
```

Computes the solution X to the continuous Lyapunov equation  $AX + XA' + C = 0$ , where no eigenvalue of A has a zero real part and no two eigenvalues are negative complex conjugates of each other.

**source**

[Base.LinAlg.sylvester](#) – Function.

```
| sylvester(A, B, C)
```

Computes the solution X to the Sylvester equation  $AX + XB + C = 0$ , where A, B and C have compatible dimensions and A and -B have no eigenvalues with equal real part.

**source**

[Base.LinAlg.issuccess](#) – Function.

```
| issuccess(F::Factorization)
```

Test that a factorization of a matrix succeeded.

```
julia> F = cholfact([1 0; 0 1]);  
  
julia> LinAlg.issuccess(F)  
true  
  
julia> F = lufact([1 0; 0 0]);  
  
julia> LinAlg.issuccess(F)  
false
```

source

[Base.LinAlg.issymmetric](#) – Function.

```
| issymmetric(A) -> Bool
```

Test whether a matrix is symmetric.

Examples

```
julia> a = [1 2; 2 -1]  
2×2 Array{Int64,2}:  
 1  2  
 2 -1  
  
julia> issymmetric(a)  
true  
  
julia> b = [1 im; -im 1]  
2×2 Array{Complex{Int64},2}:  
 1+0im  0+1im  
 0-1im  1+0im  
  
julia> issymmetric(b)  
false
```

`Base.LinAlg.isposdef` – Function.

```
| isposdef(A) -> Bool
```

Test whether a matrix is positive definite by trying to perform a Cholesky factorization of `A`. See also [isposdef!](#)

Examples

```
| julia> A = [1 2; 2 50]
```

```
2×2 Array{Int64,2}:
```

```
 1  2
```

```
 2  50
```

```
| julia> isposdef(A)
```

```
true
```

`source`

`Base.LinAlg.isposdef!` – Function.

```
| isposdef!(A) -> Bool
```

Test whether a matrix is positive definite by trying to perform a Cholesky factorization of `A`, overwriting `A` in the process. See also [isposdef](#).

Examples

```
| julia> A = [1. 2.; 2. 50.];
```

```
| julia> isposdef!(A)
```

```
true
```

```
| julia> A
```

```
2×2 Array{Float64,2}:
```

1.0	2.0
2.0	6.78233

source

[Base.LinAlg.istril](#) – Function.

```
| istril(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is lower triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
```

```
2×2 Array{Int64,2}:
```

```
1 2
```

```
2 -1
```

```
julia> istril(a)
```

```
false
```

```
julia> istril(a, 1)
```

```
true
```

```
julia> b = [1 0; -im -1]
```

```
2×2 Array{Complex{Int64},2}:
```

```
1+0im 0+0im
```

```
0-1im -1+0im
```

```
julia> istril(b)
```

```
true
```

```
julia> istril(b, -1)
```

```
false
```

[Base.LinAlg.istrilu](#) – Function.

```
| istrilu(A::AbstractMatrix, k::Integer = 0) -> Bool
```

Test whether A is upper triangular starting from the kth superdiagonal.

Examples

```
julia> a = [1 2; 2 -1]
```

```
2×2 Array{Int64,2}:
```

```
1 2  
2 -1
```

```
julia> istrilu(a)
```

```
false
```

```
julia> istrilu(a, -1)
```

```
true
```

```
julia> b = [1 im; 0 -1]
```

```
2×2 Array{Complex{Int64},2}:
```

```
1+0im 0+1im  
0+0im -1+0im
```

```
julia> istrilu(b)
```

```
true
```

```
julia> istrilu(b, 1)
```

```
false
```

[source](#)

[Base.LinAlg.isdiag](#) – Function.

Test whether a matrix is diagonal.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1

julia> isdiag(a)
false

julia> b = [im 0; 0 -im]
2×2 Array{Complex{Int64},2}:
 0+1im  0+0im
 0+0im  0-1im

julia> isdiag(b)
true
```

source

[Base.LinAlg.ishermitian](#) – Function.

```
| ishermitian(A) → Bool
```

Test whether a matrix is Hermitian.

Examples

```
julia> a = [1 2; 2 -1]
2×2 Array{Int64,2}:
 1   2
 2  -1
```

```
julia> ishermitian(a)
true

julia> b = [1 im; -im 1]
2×2 Array{Complex{Int64},2}:
 1+0im  0+1im
 0-1im  1+0im

julia> ishermitian(b)
true
```

[source](#)

[Base.LinAlg.RowVector](#) – Type.

| [RowVector\(vector\)](#)

A lazy-view wrapper of an [AbstractVector](#), which turns a length- $n$  vector into a  $1 \times n$  shaped row vector and represents the transpose of a vector (the elements are also transposed recursively). This type is usually constructed (and unwrapped) via the [transpose](#) function or `.'` operator (or related [adjoint](#) or `'` operator).

By convention, a vector can be multiplied by a matrix on its left ( $A * v$ ) whereas a row vector can be multiplied by a matrix on its right (such that  $v.' * A = (A.' * v).'$ ). It differs from a  $1 \times n$ -sized matrix by the facts that its transpose returns a vector and the inner product  $v1.' * v2$  returns a scalar, but will otherwise behave similarly.

[source](#)

[Base.LinAlg.ConjArray](#) – Type.

| [ConjArray\(array\)](#)

plex conjugate. This type is usually constructed (and unwrapped) via the `conj` function (or related `adjoint`), but currently this is the default behavior for `RowVector` only. For other arrays, the `ConjArray` constructor can be used directly.

## Examples

```
julia> [1+im, 1-im]'  
1×2 RowVector{Complex{Int64},1}  
→ [1+im, 1-im]  
1-1im 1+1im
```

```
julia> ConjArray([1+im 0; 0 1-im])  
2×2 ConjArray{Complex{Int64},2}  
1-1im 0+0im  
0+0im 1+1im
```

`source`

`Base.transpose` – Function.

```
transpose(A::AbstractMatrix)
```

The transposition operator (`.'`).

## Examples

```
julia> A = [1 2 3; 4 5 6; 7 8 9]  
3×3 Array{Int64,2}:  
1 2 3  
4 5 6  
7 8 9
```

```
julia> transpose(A)
```

```
1364 3×3 Array{Int64,2}:
```

1	4	7
2	5	8
3	6	9

## CHAPTER 54. LINEAR ALGEBRA

**source**

```
| transpose(v::AbstractVector)
```

The transposition operator (`.'`).

Examples

```
julia> v = [1,2,3]
```

```
3-element Array{Int64,1}:
```

1
2
3

```
julia> transpose(v)
```

```
1×3 RowVector{Int64,Array{Int64,1}}:
```

1	2	3
---	---	---

**source**

[Base.LinAlg.transpose!](#) – Function.

```
| transpose!(dest,src)
```

Transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2), size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

**source**

[Base.adjoint](#) – Function.

The conjugate transposition operator (').

Examples

```
julia> A = [3+2im 9+2im; 8+7im 4+6im]
2×2 Array{Complex{Int64},2}:
 3+2im  9+2im
 8+7im  4+6im

julia> adjoint(A)
2×2 Array{Complex{Int64},2}:
 3-2im  8-7im
 9-2im  4-6im
```

source

[Base.LinAlg.adjoint!](#) – Function.

```
| adjoint!(dest,src)
```

Conjugate transpose array `src` and store the result in the preallocated array `dest`, which should have a size corresponding to `(size(src,2), size(src,1))`. No in-place transposition is supported and unexpected results will happen if `src` and `dest` have overlapping memory regions.

source

[Base.LinAlg.eigs](#) – Method.

```
| eigs(A; nev=6, ncv=max(20,2*nev+1), which=:LM, tol=0.0, maxiter
|   =300, sigma=nothing, ritzvec=true, v0=zeros((0,))) -> (d,[v
|   ,],nconv,niter,nmult,resid)
```

136 CHAPTER 5. LINEAR ALGEBRA  
 Computes eigenvalues  $d$  of  $A$  using implicitly restarted Lanczos iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

**nev**: Number of eigenvalues

**ncv**: Number of Krylov vectors used in the computation; should satisfy  $\text{nev}+1 \leq \text{ncv} \leq n$  for real symmetric problems and  $\text{nev}+2 \leq \text{ncv} \leq n$  for other problems, where  $n$  is the size of the input matrix  $A$ . The default is  $\text{ncv} = \max(20, 2*\text{nev}+1)$ . Note that these restrictions limit the input matrix  $A$  to be of dimension at least 2.

**which**: type of eigenvalues to compute. See the note below.

which	type of eigenvalues
:LM	eigenvalues of largest magnitude (default)
:SM	eigenvalues of smallest magnitude
:LR	eigenvalues of largest real part
:SR	eigenvalues of smallest real part
:LI	eigenvalues of largest imaginary part (nonsymmetric or complex $A$ only)
:SI	eigenvalues of smallest imaginary part (nonsymmetric or complex $A$ only)
:BE	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric $A$ only)

**tol**: parameter defining the relative tolerance for convergence of Ritz values (eigenvalue estimates). A Ritz value  $\lambda$  is considered converged when its associated residual is less than or equal to the product of  $\text{tol}$  and  $\max(2^{1/3}, \|\lambda\|)$ , where  $\|\lambda\| = \text{eps}(\text{real}(\text{eltype}(A)))^{1/2}$  is LAPACK's machine epsilon. The residual associated with  $\lambda$  and its corresponding Ritz vector  $v$  is defined as the norm  $\|Av - v\|$ . The specified value of  $\text{tol}$  should be positive; otherwise, it is ignored and  $\text{eps}(\text{real}(\text{eltype}(A)))^{1/2}$  is used instead. Default:  $\text{eps}(\text{real}(\text{eltype}(A)))^{1/2}$ .

**maxiter**: Maximum number of iterations (default = 300)

## 54.1. STANDARD FUNCTIONS

**sigma**: level shift used in inverse iteration. If **nothing** (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to **sigma** using shift and invert iterations.

**ritzvec**: Returns the Ritz vectors **v** (eigenvectors) if **true**

**v0**: starting vector from which to start the iterations

**eigs** returns the **nev** requested eigenvalues in **d**, the corresponding Ritz vectors **v** (only if **ritzvec=true**), the number of converged eigenvalues **nconv**, the number of iterations **niter** and the number of matrix vector multiplications **nmult**, as well as the final residual vector **resid**.

Examples

```
julia> A = Diagonal(1:4);  
  
julia> λ, v = eigs(A, nev = 2);  
  
julia> λ  
2-element Array{Float64,1}:  
 4.0  
 3.0
```

Note

The **sigma** and **which** keywords interact: the description of eigenvalues searched for by **which** do not necessarily refer to the eigenvalues of **A**, but rather the linear operator constructed by the specification of the iteration mode implied by **sigma**.

<b>sigma</b>	iteration mode	<b>which</b> refers to eigenvalues of
<b>nothing</b>	ordinary (forward)	$A$
real or complex	inverse with level shift <b>sigma</b>	$(A - \sigma I)^{-1}$

Although `tol` has a default value, the best choice depends strongly on the matrix `A`. We recommend that users `_always_` specify a value for `tol` which suits their specific needs.

For details of how the errors in the computed eigenvalues are estimated, see:

B. N. Parlett, "The Symmetric Eigenvalue Problem", SIAM: Philadelphia, 2/e (1998), Ch. 13.2, "Accessing Accuracy in Lanczos Problems", pp. 290–292 ff.

R. B. Lehoucq and D. C. Sorensen, "Deflation Techniques for an Implicitly Restarted Arnoldi Iteration", SIAM Journal on Matrix Analysis and Applications (1996), 17(4), 789–821. doi:10.1137/S089547989528

### source

`Base.LinAlg.eigs` – Method.

```
eigs(A, B; nev=6, ncv=max(20,2*nev+1), which=:LM, tol=0.0,
      maxiter=300, sigma=nothing, ritzvec=true, v0=zeros((0,))) ->
      (d,[v,],nconv,niter,nmult,resid)
```

Computes generalized eigenvalues `d` of `A` and `B` using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

`nev`: Number of eigenvalues

`ncv`: Number of Krylov vectors used in the computation; should satisfy `nev+1 <= ncv <= n` for real symmetric problems and `nev+2 <= ncv <= n` for other problems, where `n` is the size of the input matrices `A` and `B`. The default is `ncv = max(20,2*nev+1)`. Note that these restrictions limit the input matrix `A` to be of dimension at least 2.

which	type of eigenvalues
:LM	eigenvalues of largest magnitude (default)
:SM	eigenvalues of smallest magnitude
:LR	eigenvalues of largest real part
:SR	eigenvalues of smallest real part
:LI	eigenvalues of largest imaginary part (nonsymmetric or complex A only)
:SI	eigenvalues of smallest imaginary part (nonsymmetric or complex A only)
:BE	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric A only)

**tol**: relative tolerance used in the convergence criterion for eigenvalues, similar to **tol** in the [eigs\(A\)](#) method for the ordinary eigenvalue problem, but effectively for the eigenvalues of  $B^{-1}A$  instead of  $A$ . See the documentation for the ordinary eigenvalue problem in [eigs\(A\)](#) and the accompanying note about **tol**.

**maxiter**: Maximum number of iterations (default = 300)

**sigma**: Specifies the level shift used in inverse iteration. If **nothing** (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to **sigma** using shift and invert iterations.

**ritzvec**: Returns the Ritz vectors **v** (eigenvectors) if **true**

**v0**: starting vector from which to start the iterations

**eigs** returns the **nev** requested eigenvalues in **d**, the corresponding Ritz vectors **v** (only if **ritzvec=true**), the number of converged eigenvalues **nconv**, the number of iterations **niter** and the number of matrix vector multiplications **nmult**, as well as the final residual vector **resid**.

Examples

```
julia> A = sparse(1.0I, 4, 4); B = Diagonal(1:4);
```

1370 **julia>**  $\lambda$ , = eigs(A, B, nev = 2); CHAPTER 54. LINEAR ALGEBRA

```
julia> λ
2-element Array{Float64,1}:
 1.0
 0.4999999999999999
```

Note

The **sigma** and **which** keywords interact: the description of eigenvalues searched for by **which** do not necessarily refer to the eigenvalue problem  $Av = Bv\lambda$ , but rather the linear operator constructed by the specification of the iteration mode implied by **sigma**.

<b>sigma</b>	iteration mode	<b>which</b> refers to the problem
<b>nothing</b>	ordinary (forward)	$Av = Bv\lambda$
real or complex	inverse with level shift <b>sigma</b>	$(A - \sigma B)^{-1}B = v\nu$

source

[Base.LinAlg.svds](#) – Function.

```
svds(A; nsv=6, ritzvec=true, tol=0.0, maxiter=1000, ncv=2*nsv,
      v0=zeros((0,))) -> (SVD([left_sv,] s, [right_sv,]), nconv,
      niter, nmult, resid)
```

Computes the largest singular values **s** of **A** using implicitly restarted Lanczos iterations derived from [eigs](#).

Inputs

**A**: Linear operator whose singular values are desired. **A** may be represented as a subtype of **AbstractArray**, e.g., a sparse matrix, or any other type supporting the four methods **size(A)**, **eltype(A)**, **A \* vector**, and **A' \* vector**.

**ritzvec**: If `true`, return the left and right singular vectors `left_sv` and `right_sv`. If `false`, omit the singular vectors. Default: `true`.

**tol**: tolerance, see [eigs](#).

**maxiter**: Maximum number of iterations, see [eigs](#). Default: 1000.

**ncv**: Maximum size of the Krylov subspace, see [eigs](#) (there called `nev`). Default: `2*nsv`.

**v0**: Initial guess for the first Krylov vector. It may have length `min(size(A) ...)`, or 0.

## Outputs

**svd**: An SVD object containing the left singular vectors, the requested values, and the right singular vectors. If `ritzvec = false`, the left and right singular vectors will be empty.

**nconv**: Number of converged singular values.

**niter**: Number of iterations.

**nmult**: Number of matrix – vector products used.

**resid**: Final residual vector.

## Examples

```
julia> A = Diagonal(1:4);  
  
julia> s = svds(A, nsv = 2)[1];  
  
julia> s[:]  
2-element Array{Float64,1}:  
 4.0  
 2.999999999999996
```

`svds(A)` is formally equivalent to calling `eigs` to perform implicitly restarted Lanczos tridiagonalization on the Hermitian matrix  $A'A$  or  $AA'$  such that the size is smallest.

`source`

`Base.LinAlg.peakflops` – Function.

| `peakflops(n::Integer=2000; parallel::Bool=false)`

`peakflops` computes the peak flop rate of the computer by using double precision `gemm!`. By default, if no arguments are specified, it multiplies a matrix of size  $n \times n$ , where  $n = 2000$ . If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS.set_num_threads(n)`.

If the keyword argument `parallel` is set to `true`, `peakflops` is run in parallel on all the worker processors. The flop rate of the entire parallel computer is returned. When running in parallel, only 1 BLAS thread is used. The argument `n` still refers to the size of the problem that is solved on each processor.

`source`

`Base.LinAlg.stride1` – Function.

| `stride1(A) -> Int`

Returns the distance between successive array elements in dimension 1 in units of element size.

Examples

| `julia> A = [1,2,3,4]`  
| 4-element Array{Int64,1}:

```
2  
3  
4
```

```
julia> Base.LinAlg.stride1(A)  
1  
  
julia> B = view(A, 2:2:4)  
2-element view(::Array{Int64,1}, 2:2:4) with eltype Int64:  
2  
4  
  
julia> Base.LinAlg.stride1(B)  
2
```

[source](#)

## 54.2 Low-level matrix operations

Matrix operations involving transpositions operations like  $A' \setminus B$  are converted by the Julia parser into calls to specially named functions like [Ac\\_ldiv\\_B](#). If you want to overload these operations for your own types, then it is useful to know the names of these functions.

Also, in many cases there are in-place versions of matrix operations that allow you to supply a pre-allocated output vector or matrix. This is useful when optimizing critical code in order to avoid the overhead of repeated allocations. These in-place operations are suffixed with ! below (e.g. [A\\_mul\\_B!](#)) according to the usual Julia convention.

[Base.LinAlg.A\\_ldiv\\_B!](#) – Function.

```
| A_ldiv_B!([Y,] A, B) -> Y
```

1374 Compute  $A \setminus B$  in-place and store the result in  $B$ .

If only two arguments are passed, then `A_ldiv_B!(A, B)` overwrites  $B$  with the result.

The argument  $A$  should not be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `factorize` or `chol-fact`). The reason for this is that factorization itself is both expensive and typically allocates memory (although it can also be done in-place via, e.g., `lufact!`), and performance-critical situations requiring `A_ldiv_B!` usually also require fine-grained control over the factorization of  $A$ .

`source`

`Base.A_ldiv_Bc` – Function.

`| A_ldiv_Bc(A, B)`

For matrices or vectors  $A$  and  $B$ , calculates  $A \setminus B$ .

`source`

`Base.A_ldiv_Bt` – Function.

`| A_ldiv_Bt(A, B)`

For matrices or vectors  $A$  and  $B$ , calculates  $A \setminus B$ .

`source`

`Base.LinAlg.A_mul_B!` – Function.

`| A_mul_B!(Y, A, B) -> Y`

Calculates the matrix-matrix or matrix-vector product  $AB$  and stores the result in  $Y$ , overwriting the existing value of  $Y$ . Note that  $Y$  must not be aliased with either  $A$  or  $B$ .

Examples

54.2 LOW-LEVEL MATRIX OPERATIONS 1375

```
julia> A=[1.0 2.0; 3.0 4.0], B=[1.0 1.0; 1.0 1.0]; Y =
```

```
↪ similar(B); A_mul_B!(Y, A, B);
```

```
julia> Y
```

```
2×2 Array{Float64,2}:
```

```
3.0 3.0
```

```
7.0 7.0
```

```
source
```

```
| A_mul_B!(A, B)
```

Calculate the matrix-matrix product  $AB$ , overwriting one of  $A$  or  $B$  (but not both), and return the result (the overwritten argument).

```
source
```

[Base.A\\_mul\\_Bc](#) – Function.

```
| A_mul_Bc(A, B)
```

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

```
source
```

[Base.A\\_mul\\_Bt](#) – Function.

```
| A_mul_Bt(A, B)
```

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

```
source
```

[Base.A\\_rdiv\\_Bc](#) – Function.

```
| A_rdiv_Bc(A, B)
```

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

```
source
```

[B276](#).[A\\_rdiv\\_Bt](#) – Function.

CHAPTER 54. LINEAR ALGEBRA

| [A\\_rdiv\\_Bt\(A, B\)](#)

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

[source](#)

[Base.Ac\\_ldiv\\_B](#) – Function.

| [Ac\\_ldiv\\_B\(A, B\)](#)

For matrices or vectors  $A$  and  $B$ , calculates  $A \setminus B$ .

[source](#)

[Base.LinAlg.Ac\\_ldiv\\_B!](#) – Function.

| [Ac\\_ldiv\\_B!\(\[Y,\] A, B\) -> Y](#)

Similar to [A\\_ldiv\\_B!](#), but return  $A \setminus B$ , computing the result in-place in  $Y$  (or overwriting  $B$  if  $Y$  is not supplied).

[source](#)

[Base.Ac\\_ldiv\\_Bc](#) – Function.

| [Ac\\_ldiv\\_Bc\(A, B\)](#)

For matrices or vectors  $A$  and  $B$ , calculates  $A \setminus B$ .

[source](#)

[Base.Ac\\_mul\\_B](#) – Function.

| [Ac\\_mul\\_B\(A, B\)](#)

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

[source](#)

[Base.Ac\\_mul\\_Bc](#) – Function.

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

**source**

[Base.Ac\\_rdiv\\_B](#) – Function.

| Ac\_rdiv\_B(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

**source**

[Base.Ac\\_rdiv\\_Bc](#) – Function.

| Ac\_rdiv\_Bc(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

**source**

[Base.At\\_ldiv\\_B](#) – Function.

| At\_ldiv\_B(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $A \nabla B$ .

**source**

[Base.LinAlg.At\\_ldiv\\_B!](#) – Function.

| At\_ldiv\_B!([Y,] A, B) -> Y

Similar to [A\\_ldiv\\_B!](#), but return  $A \nabla B$ , computing the result in-place in  $Y$  (or overwriting  $B$  if  $Y$  is not supplied).

**source**

[Base.At\\_ldiv\\_Bt](#) – Function.

| At\_ldiv\_Bt(A, B)

137 For matrices or vectors  $A$  and  $B$ , calculate

CHAPTER 54. LINEAR ALGEBRA

**source**

**Base.At\_mul\_B** – Function.

| At\_mul\_B(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

**source**

**Base.At\_mul\_Bt** – Function.

| At\_mul\_Bt(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $AB$ .

**source**

**Base.At\_rdiv\_B** – Function.

| At\_rdiv\_B(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

**source**

**Base.At\_rdiv\_Bt** – Function.

| At\_rdiv\_Bt(A, B)

For matrices or vectors  $A$  and  $B$ , calculates  $A/B$ .

**source**

### 54.3 BLAS Functions

In Julia (as in much of scientific computation), dense linear-algebra operations are based on the [LAPACK library](#), which in turn is built on top of basic linear-algebra building-blocks known as the [BLAS](#). There are highly optimized implementations of BLAS available for every computer architecture, and

Sometimes it is more efficient to call [BLAS functions](#) directly.

`Base.LinAlg.BLAS` provides wrappers for some of the BLAS functions. Those BLAS functions that overwrite one of the input arrays have names ending in '!'. Usually, a BLAS function has four methods defined, for `Float64`, `Float32`, `Complex128`, and `Complex64` arrays.

## BLAS Character Arguments

Many BLAS functions accept arguments that determine whether to transpose an argument (`trans`), which triangle of a matrix to reference (`uplo` or `ul`), whether the diagonal of a triangular matrix can be assumed to be all ones (`dA`) or which side of a matrix multiplication the input argument belongs on (`side`). The possibilities are:

### Multiplication Order

side	Meaning
'L'	The argument goes on the left side of a matrix-matrix operation.
'R'	The argument goes on the right side of a matrix-matrix operation.

### Triangle Referencing

uplo/ul	Meaning
'U'	Only the upper triangle of the matrix will be used.
'L'	Only the lower triangle of the matrix will be used.

### Transposition Operation

trans/tX	Meaning
'N'	The input matrix X is not transposed or conjugated.
'T'	The input matrix X will be transposed.
'C'	The input matrix X will be conjugated and transposed.

### Unit Diagonal

`Base.LinAlg.BLAS.dotu` – Function.

<code>diag/dX</code>	Meaning	CHAPTER 54. LINEAR ALGEBRA
'N'	The diagonal values of the matrix X will be read.	
'U'	The diagonal of the matrix X is assumed to be all ones.	

`dotu(n, X, incx, Y, incy)`

Dot function for two complex vectors consisting of n elements of array X with stride `incx` and n elements of array Y with stride `incy`.

Examples

```
julia> Base.BLAS.dotu(10, im*ones(10), 1, complex.(ones(20),
    ↪ ones(20)), 2)
-10.0 + 10.0im
```

`source`

`Base.LinAlg.BLAS.dotc` – Function.

`dotc(n, X, incx, U, incy)`

Dot function for two complex vectors, consisting of n elements of array X with stride `incx` and n elements of array U with stride `incy`, conjugating the first vector.

Examples

```
julia> Base.BLAS.dotc(10, im*ones(10), 1, complex.(ones(20),
    ↪ ones(20)), 2)
10.0 - 10.0im
```

`source`

`Base.LinAlg.BLAS.blascopy!` – Function.

`blascopy!(n, X, incx, Y, incy)`

Copy n elements of array X with stride `incx` to array Y with stride `incy`. Returns Y.

`Base.LinAlg.BLAS.nrm2` – Function.

```
| nrm2(n, X, incx)
```

2-norm of a vector consisting of `n` elements of array `X` with stride `incx`.

Examples

```
| julia> Base.BLAS.nrm2(4, ones(8), 2)
```

```
2.0
```

```
| julia> Base.BLAS.nrm2(1, ones(8), 2)
```

```
1.0
```

`source`

`Base.LinAlg.BLAS.asum` – Function.

```
| asum(n, X, incx)
```

Sum of the absolute values of the first `n` elements of array `X` with stride `incx`.

Examples

```
| julia> Base.BLAS.asum(5, im*ones(10), 2)
```

```
5.0
```

```
| julia> Base.BLAS.asum(2, im*ones(10), 5)
```

```
2.0
```

`source`

`Base.LinAlg.axpy!` – Function.

```
| axpy!(a, X, Y)
```

138 Overwrite Y with  $a \cdot X + Y$ , where  $a$  is a scalar.

CHAPTER 54 LINEAR ALGEBRA

Examples

```
julia> x = [1; 2; 3];  
  
julia> y = [4; 5; 6];  
  
julia> Base.BLAS.axpy!(2, x, y)  
3-element Array{Int64,1}:  
 6  
 9  
 12
```

source

`Base.LinAlg.BLAS.scal!` – Function.

```
| scal!(n, a, X, incx)
```

Overwrite  $X$  with  $a \cdot X$  for the first  $n$  elements of array  $X$  with stride  $incx$ .

Returns  $X$ .

source

`Base.LinAlg.BLAS.scal` – Function.

```
| scal(n, a, X, incx)
```

Returns  $X$  scaled by  $a$  for the first  $n$  elements of array  $X$  with stride  $incx$ .

source

`Base.LinAlg.BLAS.ger!` – Function.

```
| ger!(alpha, x, y, A)
```

Rank-1 update of the matrix  $A$  with vectors  $x$  and  $y$  as  $\text{alpha} \cdot x \cdot y' + A$ .

`Base.LinAlg.BLAS.syr!` – Function.

```
| syr!(uplo, alpha, x, A)
```

Rank-1 update of the symmetric matrix `A` with vector `x` as `alpha*x*x.'` + `A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`source`

`Base.LinAlg.BLAS.syrk!` – Function.

```
| syrk!(uplo, trans, alpha, A, beta, C)
```

Rank-k update of the symmetric matrix `C` as `alpha*A*A.'` + `beta*C` or `alpha*A.*A + beta*C` according to `trans`. Only the `uplo` triangle of `C` is used. Returns `C`.

`source`

`Base.LinAlg.BLAS.syrk` – Function.

```
| syrk(uplo, trans, alpha, A)
```

Returns either the upper triangle or the lower triangle of `A`, according to `uplo`, of `alpha*A*A.'` or `alpha*A.*A`, according to `trans`.

`source`

`Base.LinAlg.BLAS.her!` – Function.

```
| her!(uplo, alpha, x, A)
```

Methods for complex arrays only. Rank-1 update of the Hermitian matrix `A` with vector `x` as `alpha*x*x'` + `A`. `uplo` controls which triangle of `A` is updated. Returns `A`.

`source`

## [Base4.LinAlg.BLAS.herk!](#) – Function. CHAPTER 54. LINEAR ALGEBRA

```
|herk!(uplo, trans, alpha, A, beta, C)
```

Methods for complex arrays only. Rank-k update of the Hermitian matrix C as `alpha*A*A' + beta*C` or `alpha*A'*A + beta*C` according to `trans`. Only the `uplo` triangle of C is updated. Returns C.

[source](#)

## [Base.LinAlg.BLAS.herk](#) – Function.

```
|herk(uplo, trans, alpha, A)
```

Methods for complex arrays only. Returns the `uplo` triangle of `alpha*A*A'` or `alpha*A'*A`, according to `trans`.

[source](#)

## [Base.LinAlg.BLAS.gbmv!](#) – Function.

```
|gbmv!(trans, m, kl, ku, alpha, A, x, beta, y)
```

Update vector y as `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` according to `trans`. The matrix A is a general band matrix of dimension `m` by `size(A, 2)` with `kl` sub-diagonals and `ku` super-diagonals. `alpha` and `beta` are scalars. Returns the updated y.

[source](#)

## [Base.LinAlg.BLAS.gbmv](#) – Function.

```
|gbmv(trans, m, kl, ku, alpha, A, x)
```

Returns `alpha*A*x` or `alpha*A'*x` according to `trans`. The matrix A is a general band matrix of dimension `m` by `size(A, 2)` with `kl` sub-diagonals and `ku` super-diagonals, and `alpha` is a scalar.

[source](#)

```
| sbmv!(uplo, k, alpha, A, x, beta, y)
```

Update vector  $y$  as  $\alpha * A * x + \beta * y$  where  $A$  is a symmetric band matrix of order  $\text{size}(A, 2)$  with  $k$  super-diagonals stored in the argument  $A$ . The storage layout for  $A$  is described in the reference BLAS module, level-2 BLAS at <http://www.netlib.org/lapack/explore-html/>. Only the `uplo` triangle of  $A$  is used.

Returns the updated  $y$ .

`source`

```
| sbmv(uplo, k, alpha, A, x)
```

Returns  $\alpha * A * x$  where  $A$  is a symmetric band matrix of order  $\text{size}(A, 2)$  with  $k$  super-diagonals stored in the argument  $A$ . Only the `uplo` triangle of  $A$  is used.

`source`

```
| sbmv(uplo, k, A, x)
```

Returns  $A * x$  where  $A$  is a symmetric band matrix of order  $\text{size}(A, 2)$  with  $k$  super-diagonals stored in the argument  $A$ . Only the `uplo` triangle of  $A$  is used.

`source`

```
| gemm!(tA, tB, alpha, A, B, beta, C)
```

Update  $C$  as  $\alpha * A * B + \beta * C$  or the other three variants according to `tA` and `tB`. Returns the updated  $C$ .

`Base.LinAlg.BLAS.gemm` – Method.

```
| gemm(tA, tB, alpha, A, B)
```

Returns  $\alpha \cdot A \cdot B$  or the other three variants according to `tA` and `tB`.

`source`

`Base.LinAlg.BLAS.gemm` – Method.

```
| gemm(tA, tB, A, B)
```

Returns  $A \cdot B$  or the other three variants according to `tA` and `tB`.

`source`

`Base.LinAlg.BLAS.gemv!` – Function.

```
| gemv!(tA, alpha, A, x, beta, y)
```

Update the vector `y` as  $\alpha \cdot A \cdot x + \beta \cdot y$  or  $\alpha \cdot A' \cdot x + \beta \cdot y$  according to `tA`. `alpha` and `beta` are scalars. Returns the updated `y`.

`source`

`Base.LinAlg.BLAS.gemv` – Method.

```
| gemv(tA, alpha, A, x)
```

Returns  $\alpha \cdot A \cdot x$  or  $\alpha \cdot A' \cdot x$  according to `tA`. `alpha` is a scalar.

`source`

`Base.LinAlg.BLAS.gemv` – Method.

```
| gemv(tA, A, x)
```

Returns  $A \cdot x$  or  $A' \cdot x$  according to `tA`.

`source`

```
| symm!(side, ul, alpha, A, B, beta, C)
```

Update  $C$  as  $\alpha * A * B + \beta * C$  or  $\alpha * B * A + \beta * C$  according to `side`.  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used. Returns the updated  $C$ .

`source`

```
| symm(side, ul, alpha, A, B)
```

Returns  $\alpha * A * B$  or  $\alpha * B * A$  according to `side`.  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used.

`source`

```
| symm(side, ul, A, B)
```

Returns  $A * B$  or  $B * A$  according to `side`.  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used.

`source`

```
| symv!(ul, alpha, A, x, beta, y)
```

Update the vector  $y$  as  $\alpha * A * x + \beta * y$ .  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used. `alpha` and `beta` are scalars. Returns the updated  $y$ .

`source`

1388 `symv`

## CHAPTER 54. LINEAR ALGEBRA

Returns  $\alpha * A * x$ .  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used. `alpha` is a scalar.

`source`

`Base.LinAlg.BLAS.symv` – Method.

`| symv`

Returns  $A * x$ .  $A$  is assumed to be symmetric. Only the `ul` triangle of  $A$  is used.

`source`

`Base.LinAlg.BLAS.trmm!` – Function.

`| trmm!(side, ul, tA, dA, alpha, A, B)`

Update  $B$  as  $\alpha * A * B$  or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of  $A$  is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated  $B$ .

`source`

`Base.LinAlg.BLAS.trmm` – Function.

`| trmm(side, ul, tA, dA, alpha, A, B)`

Returns  $\alpha * A * B$  or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of  $A$  is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

`source`

`Base.LinAlg.BLAS.trsm!` – Function.

Overwrite B with the solution to  $A*X = \text{alpha}*B$  or one of the other three variants determined by `side` and `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones. Returns the updated B.

[source](#)

`Base.LinAlg.BLAS.trsm` – Function.

| `trsm(side, ul, tA, dA, alpha, A, B)`

Returns the solution to  $A*X = \text{alpha}*B$  or one of the other three variants determined by determined by `side` and `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

[source](#)

`Base.LinAlg.BLAS.trmv!` – Function.

| `trmv!(ul, tA, dA, A, b)`

Returns  $\text{op}(A)*b$ , where `op` is determined by `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on `b`.

[source](#)

`Base.LinAlg.BLAS.trmv` – Function.

| `trmv(ul, tA, dA, A, b)`

Returns  $\text{op}(A)*b$ , where `op` is determined by `tA`. Only the `ul` triangle of A is used. `dA` determines if the diagonal values are read or are assumed to be all ones.

[source](#)

[Base.LinAlg.BLAS.trsv!](#) – Function.

CHAPTER 54. LINEAR ALGEBRA

```
| trsv!(ul, tA, dA, A, b)
```

Overwrite **b** with the solution to  $A \cdot x = b$  or one of the other two variants determined by **tA** and **ul**. **dA** determines if the diagonal values are read or are assumed to be all ones. Returns the updated **b**.

[source](#)

[Base.LinAlg.BLAS.trsv](#) – Function.

```
| trsv!(ul, tA, dA, A, b)
```

Returns the solution to  $A \cdot x = b$  or one of the other two variants determined by **tA** and **ul**. **dA** determines if the diagonal values are read or are assumed to be all ones.

[source](#)

[Base.LinAlg.BLAS.set\\_num\\_threads](#) – Function.

```
| set_num_threads(n)
```

Set the number of threads the BLAS library should use.

[source](#)

[Base.LinAlg.I](#) – Constant.

```
| I
```

An object of type [UniformScaling](#), representing an identity matrix of any size.

Examples

```
| julia> ones(5, 6) * I == ones(5, 6)
```

```
| true
```

54.4 LAPACK FUNCTIONS  
julia> [1 2im 3; 1im 2 3] \* I

1391

```
2×3 Array{Complex{Int64},2}:
 1+0im  0+2im  3+0im
 0+1im  2+0im  3+0im
```

[source](#)

## 54.4 LAPACK Functions

`Base.LinAlg.LAPACK` provides wrappers for some of the LAPACK functions for linear algebra. Those functions that overwrite one of the input arrays have names ending in '!'.

Usually a function has 4 methods defined, one each for `Float64`, `Float32`, `Complex128` and `Complex64` arrays.

Note that the LAPACK API provided by Julia can and will change in the future. Since this API is not user-facing, there is no commitment to support/deprecate this specific set of functions in future releases.

`Base.LinAlg.LAPACK.gbtrf!` – Function.

```
|gbtrf!(kl, ku, m, AB) -> (AB, ipiv)
```

Compute the LU factorization of a banded matrix `AB`. `kl` is the first sub-diagonal containing a nonzero band, `ku` is the last superdiagonal containing one, and `m` is the first dimension of the matrix `AB`. Returns the LU factorization in-place and `ipiv`, the vector of pivots used.

[source](#)

`Base.LinAlg.LAPACK.gbtrs!` – Function.

```
|gbtrs!(trans, kl, ku, m, AB, ipiv, B)
```

Solve the equation  $AB \times X = B$ . `trans` determines the orientation of `AB`. It may be `N` (no transpose), `T` (transpose), or `C` (conjugate transpose). `kl`

1392s the first subdiagonal containing a nonzero, **CHAPTER 54. LINEAR ALGEBRA**  
diagonal containing one, and **m** is the first dimension of the matrix **AB**.  
**ipiv** is the vector of pivots returned from **gbtrf!**. Returns the vector or  
matrix **X**, overwriting **B** in-place.

**source**

[Base.LinAlg.LAPACK.gebal!](#) – Function.

```
| gebal!(job, A) -> (ilo, ihi, scale)
```

Balance the matrix **A** before computing its eigensystem or Schur factorization. **job** can be one of **N** (**A** will not be permuted or scaled), **P** (**A** will only be permuted), **S** (**A** will only be scaled), or **B** (**A** will be both permuted and scaled). Modifies **A** in-place and returns **ilo**, **ihi**, and **scale**. If permuting was turned on,  $A[i, j] = 0$  if  $j > i$  and  $1 < j < ilo$  or  $j > ihi$ . **scale** contains information about the scaling/permuations performed.

**source**

[Base.LinAlg.LAPACK.gebak!](#) – Function.

```
| gebak!(job, side, ilo, ihi, scale, V)
```

Transform the eigenvectors **V** of a matrix balanced using **gebal!** to the unscaled/unpermuted eigenvectors of the original matrix. Modifies **V** in-place. **side** can be **L** (left eigenvectors are transformed) or **R** (right eigenvectors are transformed).

**source**

[Base.LinAlg.LAPACK.gebrd!](#) – Function.

```
| gebrd!(A) -> (A, d, e, tauq, taup)
```

54.4 LAPACK FUNCTIONS

bidiagonal form  $A = QBP'$ . Returns A, containing the bidiagonal matrix B; d, containing the diagonal elements of B; e, containing the off-diagonal elements of B; tauq, containing the elementary reflectors representing Q; and taup, containing the elementary reflectors representing P.

**source**

[Base.LinAlg.LAPACK.gelqf!](#) – Function.

| `gelqf!(A, tau)`

Compute the LQ factorization of A,  $A = LQ$ . **tau** contains scalars which parameterize the elementary reflectors of the factorization. **tau** must have length greater than or equal to the smallest dimension of A.

Returns A and **tau** modified in-place.

**source**

| `gelqf!(A) -> (A, tau)`

Compute the LQ factorization of A,  $A = LQ$ .

Returns A, modified in-place, and **tau**, which contains scalars which parameterize the elementary reflectors of the factorization.

**source**

[Base.LinAlg.LAPACK.geqlf!](#) – Function.

| `geqlf!(A, tau)`

Compute the QL factorization of A,  $A = QL$ . **tau** contains scalars which parameterize the elementary reflectors of the factorization. **tau** must have length greater than or equal to the smallest dimension of A.

Returns A and **tau** modified in-place.

**source**

1394 `geqlf!(A) -> (A, tau)`

## CHAPTER 54. LINEAR ALGEBRA

Compute the QL factorization of A,  $A = QL$ .

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

**source**

[Base.LinAlg.LAPACK.geqrf!](#) – Function.

| `geqrf!(A, tau)`

Compute the QR factorization of A,  $A = QR$ . tau contains scalars which parameterize the elementary reflectors of the factorization. tau must have length greater than or equal to the smallest dimension of A.

Returns A and tau modified in-place.

**source**

| `geqrf!(A) -> (A, tau)`

Compute the QR factorization of A,  $A = QR$ .

Returns A, modified in-place, and tau, which contains scalars which parameterize the elementary reflectors of the factorization.

**source**

[Base.LinAlg.LAPACK.geqp3!](#) – Function.

| `geqp3!(A, jpvt, tau)`

Compute the pivoted QR factorization of A,  $AP = QR$  using BLAS level 3. P is a pivoting matrix, represented by jpvt. tau stores the elementary reflectors. jpvt must have length length greater than or equal to n if A is an ( $m \times n$ ) matrix. tau must have length greater than or equal to the smallest dimension of A.

**source**

```
| geqp3!(A, jpvt) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A,  $AP = QR$  using BLAS level 3. P is a pivoting matrix, represented by **jpvt**. **jpvt** must have length greater than or equal to n if A is an ( $m \times n$ ) matrix.

Returns A and **jpvt**, modified in-place, and **tau**, which stores the elementary reflectors.

**source**

```
| geqp3!(A) -> (A, jpvt, tau)
```

Compute the pivoted QR factorization of A,  $AP = QR$  using BLAS level 3.

Returns A, modified in-place, **jpvt**, which represents the pivoting matrix P, and **tau**, which stores the elementary reflectors.

**source**

[Base.LinAlg.LAPACK.gerqf!](#) – Function.

```
| gerqf!(A, tau)
```

Compute the RQ factorization of A,  $A = RQ$ . **tau** contains scalars which parameterize the elementary reflectors of the factorization. **tau** must have length greater than or equal to the smallest dimension of A.

Returns A and **tau** modified in-place.

**source**

```
| gerqf!(A) -> (A, tau)
```

Compute the RQ factorization of A,  $A = RQ$ .

Returns A, modified in-place, and **tau**, which contains scalars which parameterize the elementary reflectors of the factorization.

[Base.LinAlg.LAPACK.geqrt!](#) – Function.

| [geqrt!](#)(A, T)

Compute the blocked QR factorization of A,  $A = QR$ . T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

[source](#)

| [geqrt!](#)(A, nb) -> (A, T)

Compute the blocked QR factorization of A,  $A = QR$ . nb sets the block size and it must be between 1 and n, the second dimension of A.

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

[source](#)

[Base.LinAlg.LAPACK.geqrt3!](#) – Function.

| [geqrt3!](#)(A, T)

Recursively computes the blocked QR factorization of A,  $A = QR$ . T contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of T sets the block size and it must be between 1 and n. The second dimension of T must equal the smallest dimension of A.

Returns A and T modified in-place.

```
| geqrt3!(A) -> (A, T)
```

Recursively computes the blocked QR factorization of A,  $A = QR$ .

Returns A, modified in-place, and T, which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

**source**

[Base.LinAlg.LAPACK.getrf!](#) – Function.

```
| getrf!(A) -> (A, ipiv, info)
```

Compute the pivoted LU factorization of A,  $A = LU$ .

Returns A, modified in-place, **ipiv**, the pivoting information, and an **info** code which indicates success (**info** = 0), a singular value in U (**info** = i, in which case  $U[i, i]$  is singular), or an error code (**info** < 0).

**source**

[Base.LinAlg.LAPACK.tzrzf!](#) – Function.

```
| tzrzf!(A) -> (A, tau)
```

Transforms the upper trapezoidal matrix A to upper triangular form in-place. Returns A and **tau**, the scalar parameters for the elementary reflectors of the transformation.

**source**

[Base.LinAlg.LAPACK.ormrz!](#) – Function.

```
| ormrz!(side, trans, A, tau, C)
```

Multiplies the matrix C by Q from the transformation supplied by **tzrzf!**. Depending on **side** or **trans** the multiplication can be left-sided (**side**

1398= L, Q\*C) or right-sided (`side = R`, CHAPTER 54: LINEAR ALGEBRA  
`(trans = N)`, transposed (`trans = T`), or conjugate transposed (`trans = C`). Returns matrix C which is modified in-place with the result of the multiplication.

`source`

[Base.LinAlg.LAPACK.gels!](#) – Function.

| `gels!(trans, A, B) -> (F, B, ssr)`

Solves the linear equation  $A * X = B$ ,  $A.^* X = B$ , or  $A'^* X = B$  using a QR or LQ factorization. Modifies the matrix/vector B in place with the solution. A is overwritten with its QR or LQ factorization. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose). `gels!` searches for the minimum norm/least squares solution. A may be under or over determined. The solution is returned in B.

`source`

[Base.LinAlg.LAPACK.gesv!](#) – Function.

| `gesv!(A, B) -> (B, A, ipiv)`

Solves the linear equation  $A * X = B$  where A is a square matrix using the LU factorization of A. A is overwritten with its LU factorization and B is overwritten with the solution X. `ipiv` contains the pivoting information for the LU factorization of A.

`source`

[Base.LinAlg.LAPACK.getrs!](#) – Function.

| `getrs!(trans, A, ipiv, B)`

Solves the linear equation  $A * X = B$ ,  $A.^* X = B$ , or  $A'^* X = B$  for square A. Modifies the matrix/vector B in place with the solution. A is the LU

54.4 fact LAPACK FUNCTIONS 1309  
getrf!, with `ipiv` the pivoting information. `trans` may be one of `N` (no modification), `T` (transpose), or `C` (conjugate transpose).

#### source

[Base.LinAlg.LAPACK.getri!](#) – Function.

```
| getri!(A, ipiv)
```

Computes the inverse of `A`, using its LU factorization found by `getrf!`. `ipiv` is the pivot information output and `A` contains the LU factorization of `getrf!`. `A` is overwritten with its inverse.

#### source

[Base.LinAlg.LAPACK.gesvx!](#) – Function.

```
| gesvx!(fact, trans, A, AF, ipiv, equed, R, C, B) -> (X, equed,  
|   R, C, B, rcond, ferr, berr, work)
```

Solves the linear equation  $A * X = B$  (`trans = N`),  $A.' * X = B$  (`trans = T`), or  $A' * X = B$  (`trans = C`) using the LU factorization of `A`. `fact` may be `E`, in which case `A` will be equilibrated and copied to `AF`; `F`, in which case `AF` and `ipiv` from a previous LU factorization are inputs; or `N`, in which case `A` will be copied to `AF` and then factored. If `fact = F`, `equed` may be `N`, meaning `A` has not been equilibrated; `R`, meaning `A` was multiplied by `Diagonal(R)` from the left; `C`, meaning `A` was multiplied by `Diagonal(C)` from the right; or `B`, meaning `A` was multiplied by `Diagonal(R)` from the left and `Diagonal(C)` from the right. If `fact = F` and `equed = R` or `B` the elements of `R` must all be positive. If `fact = F` and `equed = C` or `B` the elements of `C` must all be positive.

Returns the solution `X`; `equed`, which is an output if `fact` is not `N`, and describes the equilibration that was performed; `R`, the row equilibration diagonal; `C`, the column equilibration diagonal; `B`, which may be overwritten with its equilibrated form `Diagonal(R)*B` (if `trans = N` and `equed`

1400= R, B) or Diagonal(C)\*B (if trans = `TRANSPOSE`)  
the reciprocal condition number of A after equilibrating; ferr, the forward  
error bound for each solution vector in X; berr, the forward error bound  
for each solution vector in X; and work, the reciprocal pivot growth factor.

[source](#)

```
| gesvx!(A, B)
```

The no-equilibration, no-transpose simplification of `gesvx!`.

[source](#)

`Base.LinAlg.LAPACK.gelsd!` – Function.

```
| gelsd!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of  $A * X = B$  by finding the SVD fac-  
torization of A, then dividing-and-conquering the problem. B is overwrit-  
ten with the solution X. Singular values below `rcond` will be treated as  
zero. Returns the solution in B and the effective rank of A in `rnk`.

[source](#)

`Base.LinAlg.LAPACK.gelsy!` – Function.

```
| gelsy!(A, B, rcond) -> (B, rnk)
```

Computes the least norm solution of  $A * X = B$  by finding the full QR  
factorization of A, then dividing-and-conquering the problem. B is over-  
written with the solution X. Singular values below `rcond` will be treated as  
zero. Returns the solution in B and the effective rank of A in `rnk`.

[source](#)

`Base.LinAlg.LAPACK.gglse!` – Function.

```
| gglse!(A, c, B, d) -> (X, res)
```

54.4 Solving linear equations  $\mathbf{A} \mathbf{x} = \mathbf{c}$  where  $\mathbf{x}$  is subject to the equality constraint  $\mathbf{B} \mathbf{x} = \mathbf{d}$ . Uses the formula  $\|\mathbf{c} - \mathbf{A}\mathbf{x}\|^2 = 0$  to solve. Returns  $\mathbf{X}$  and the residual sum-of-squares.

[source](#)

[Base.LinAlg.LAPACK.geev!](#) – Function.

```
| geev!(jobvl, jobvr, A) -> (W, VL, VR)
```

Finds the eigensystem of  $\mathbf{A}$ . If `jobvl` = `N`, the left eigenvectors of  $\mathbf{A}$  aren't computed. If `jobvr` = `N`, the right eigenvectors of  $\mathbf{A}$  aren't computed. If `jobvl` = `V` or `jobvr` = `V`, the corresponding eigenvectors are computed. Returns the eigenvalues in `W`, the right eigenvectors in `VR`, and the left eigenvectors in `VL`.

[source](#)

[Base.LinAlg.LAPACK.gesdd!](#) – Function.

```
| gesdd!(job, A) -> (U, S, VT)
```

Finds the singular value decomposition of  $\mathbf{A}$ ,  $\mathbf{A} = \mathbf{U} * \mathbf{S} * \mathbf{V}'$ , using a divide and conquer approach. If `job` = `A`, all the columns of  $\mathbf{U}$  and the rows of  $\mathbf{V}'$  are computed. If `job` = `N`, no columns of  $\mathbf{U}$  or rows of  $\mathbf{V}'$  are computed. If `job` = `0`,  $\mathbf{A}$  is overwritten with the columns of (thin)  $\mathbf{U}$  and the rows of (thin)  $\mathbf{V}'$ . If `job` = `S`, the columns of (thin)  $\mathbf{U}$  and the rows of (thin)  $\mathbf{V}'$  are computed and returned separately.

[source](#)

[Base.LinAlg.LAPACK.gesvd!](#) – Function.

```
| gesvd!(jobu, jobvt, A) -> (U, S, VT)
```

Finds the singular value decomposition of  $\mathbf{A}$ ,  $\mathbf{A} = \mathbf{U} * \mathbf{S} * \mathbf{V}'$ . If `jobu` = `A`, all the columns of  $\mathbf{U}$  are computed. If `jobvt` = `A` all the rows of  $\mathbf{V}'$  are

1402computed. If `jobu` = `N`, no columns of `U` are computed. If `jobv` = `N`, no rows of `V'` are computed. If `jobu` = `0`, `A` is overwritten with the columns of (thin) `U`. If `jobvt` = `0`, `A` is overwritten with the rows of (thin) `V'`. If `jobu` = `S`, the columns of (thin) `U` are computed and returned separately. If `jobvt` = `S` the rows of (thin) `V'` are computed and returned separately. `jobu` and `jobvt` can't both be `0`.

Returns `U`, `S`, and `Vt`, where `S` are the singular values of `A`.

**source**

[Base.LinAlg.LAPACK.ggsvd!](#) – Function.

```
ggsvd!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l,  
R)
```

Finds the generalized singular value decomposition of `A` and `B`,  $U' * A * Q = D1 * R$  and  $V' * B * Q = D2 * R$ . `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu` = `U`, the orthogonal/unitary matrix `U` is computed. If `jobv` = `V` the orthogonal/unitary matrix `V` is computed. If `jobq` = `Q`, the orthogonal/unitary matrix `Q` is computed. If `jobu`, `jobv` or `jobq` is `N`, that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

**source**

[Base.LinAlg.LAPACK.ggsvd3!](#) – Function.

```
ggsvd3!(jobu, jobv, jobq, A, B) -> (U, V, Q, alpha, beta, k, l,  
R)
```

Finds the generalized singular value decomposition of `A` and `B`,  $U' * A * Q = D1 * R$  and  $V' * B * Q = D2 * R$ . `D1` has `alpha` on its diagonal and `D2` has `beta` on its diagonal. If `jobu` = `U`, the orthogonal/unitary matrix `U` is computed. If `jobv` = `V` the orthogonal/unitary matrix `V` is computed. If

54.4. ~~jd~~<sup>1408</sup> LAPACK, ~~F~~<sup>1408</sup> C<sup>1408</sup> ~~T~~<sup>1408</sup> ~~S~~<sup>1408</sup> ~~o~~<sup>1408</sup>n<sup>1408</sup>onal/unitary matrix Q is computed. If jobu, jobv or jobq is N, that matrix is not computed. This function requires LAPACK 3.6.0.

## source

## Base.LinAlg.LAPACK.geevx! – Function.

```
geevx!(balanc, jobvl, jobvr, sense, A) -> (A, w, VL, VR, ilo, ihi, scale, abnrm, rconde, rcondv)
```

Finds the eigensystem of A with matrix balancing. If **jobvl** = N, the left eigenvectors of A aren't computed. If **jobvr** = N, the right eigenvectors of A aren't computed. If **jobvl** = V or **jobvr** = V, the corresponding eigenvectors are computed. If **balanc** = N, no balancing is performed. If **balanc** = P, A is permuted but not scaled. If **balanc** = S, A is scaled but not permuted. If **balanc** = B, A is permuted and scaled. If **sense** = N, no reciprocal condition numbers are computed. If **sense** = E, reciprocal condition numbers are computed for the eigenvalues only. If **sense** = V, reciprocal condition numbers are computed for the right eigenvectors only. If **sense** = B, reciprocal condition numbers are computed for the right eigenvectors and the eigenvalues. If **sense** = E, B, the right and left eigenvectors must be computed.

## source

## Base.LinAlg.LAPACK.ggev! – Function.

```
ggev!(jobvl, jobvr, A, B) -> (alpha, beta, vl, vr)
```

Finds the generalized eigendecomposition of A and B. If `jobvl` = N, the left eigenvectors aren't computed. If `jobvr` = N, the right eigenvectors aren't computed. If `jobvl` = V or `jobvr` = V, the corresponding eigenvectors are computed.

## source

[B404](#).[LinAlg.LAPACK.gtsv!](#) – Function. CHAPTER 54. LINEAR ALGEBRA

```
|gtsv!(dl, d, du, B)
```

Solves the equation  $A * X = B$  where  $A$  is a tridiagonal matrix with  $dl$  on the subdiagonal,  $d$  on the diagonal, and  $du$  on the superdiagonal.

Overwrites  $B$  with the solution  $X$  and returns it.

[source](#)

[Base](#).[LinAlg.LAPACK.gttrf!](#) – Function.

```
|gttrf!(dl, d, du) -> (dl, d, du, du2, ipiv)
```

Finds the LU factorization of a tridiagonal matrix with  $dl$  on the subdiagonal,  $d$  on the diagonal, and  $du$  on the superdiagonal.

Modifies  $dl$ ,  $d$ , and  $du$  in-place and returns them and the second super-diagonal  $du2$  and the pivoting vector  $ipiv$ .

[source](#)

[Base](#).[LinAlg.LAPACK.gttrs!](#) – Function.

```
|gttrs!(trans, dl, d, du, du2, ipiv, B)
```

Solves the equation  $A * X = B$  ( $trans = N$ ),  $A.^' * X = B$  ( $trans = T$ ), or  $A'^' * X = B$  ( $trans = C$ ) using the LU factorization computed by [gttrf!](#).  $B$  is overwritten with the solution  $X$ .

[source](#)

[Base](#).[LinAlg.LAPACK.orglq!](#) – Function.

```
|orglq!(A, tau, k = length(tau))
```

Explicitly finds the matrix  $Q$  of a LQ factorization after calling [gelqf!](#) on  $A$ . Uses the output of [gelqf!](#).  $A$  is overwritten by  $Q$ .

[source](#)

```
|orgqr!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a QR factorization after calling `geqrf!` on A. Uses the output of `geqrf!`. A is overwritten by Q.

`source`

```
|orgql!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a QL factorization after calling `geqlf!` on A. Uses the output of `geqlf!`. A is overwritten by Q.

`source`

```
|orgrq!(A, tau, k = length(tau))
```

Explicitly finds the matrix Q of a RQ factorization after calling `gerqf!` on A. Uses the output of `gerqf!`. A is overwritten by Q.

`source`

```
|ormlq!(side, trans, A, tau, C)
```

Computes  $Q * C$  (`trans = N`),  $Q.' * C$  (`trans = T`),  $Q' * C$  (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using Q from a LQ factorization of A computed using `gelqf!`. C is overwritten.

`source`

1406 `ormqr!(side, trans, A, tau, C)`

CHAPTER 54. LINEAR ALGEBRA

Computes  $Q * C$  (`trans = N`),  $Q.' * C$  (`trans = T`),  $Q' * C$  (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using  $Q$  from a QR factorization of  $A$  computed using `geqrf!`.  $C$  is overwritten.

`source`

[Base.LinAlg.LAPACK.ormql!](#) – Function.

| `ormql!(side, trans, A, tau, C)`

Computes  $Q * C$  (`trans = N`),  $Q.' * C$  (`trans = T`),  $Q' * C$  (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using  $Q$  from a QL factorization of  $A$  computed using `geqlf!`.  $C$  is overwritten.

`source`

[Base.LinAlg.LAPACK.ormrq!](#) – Function.

| `ormrq!(side, trans, A, tau, C)`

Computes  $Q * C$  (`trans = N`),  $Q.' * C$  (`trans = T`),  $Q' * C$  (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side = R` using  $Q$  from a RQ factorization of  $A$  computed using `gerqf!`.  $C$  is overwritten.

`source`

[Base.LinAlg.LAPACK.gemqrt!](#) – Function.

| `gemqrt!(side, trans, V, T, C)`

Computes  $Q * C$  (`trans = N`),  $Q.' * C$  (`trans = T`),  $Q' * C$  (`trans = C`) for `side = L` or the equivalent right-sided multiplication for `side`

54.4. = ~~LAPACK.QR~~ factorization of A computed using `geqrt!`.  
A is overwritten.

`source`

`Base.LinAlg.LAPACK.posv!` – Function.

| `posv!(uplo, A, B) -> (A, B)`

Finds the solution to  $A * X = B$  where A is a symmetric or Hermitian positive definite matrix. If `uplo` = U the upper Cholesky decomposition of A is computed. If `uplo` = L the lower Cholesky decomposition of A is computed. A is overwritten by its Cholesky decomposition. B is overwritten with the solution X.

`source`

`Base.LinAlg.LAPACK.potrf!` – Function.

| `potrf!(uplo, A)`

Computes the Cholesky (upper if `uplo` = U, lower if `uplo` = L) decomposition of positive-definite matrix A. A is overwritten and returned with an info code.

`source`

`Base.LinAlg.LAPACK.potri!` – Function.

| `potri!(uplo, A)`

Computes the inverse of positive-definite matrix A after calling `potrf!` to find its (upper if `uplo` = U, lower if `uplo` = L) Cholesky decomposition.

A is overwritten by its inverse and returned.

`source`

`Base.LinAlg.LAPACK.potrs!` – Function.

1408 `potrs!(uplo, A, B)`

## CHAPTER 54. LINEAR ALGEBRA

Finds the solution to  $A * X = B$  where  $A$  is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `potrf!`. If `uplo = U` the upper Cholesky decomposition of  $A$  was computed. If `uplo = L` the lower Cholesky decomposition of  $A$  was computed.  $B$  is overwritten with the solution  $X$ .

`source`

[Base.LinAlg.LAPACK.pstrf!](#) – Function.

| `pstrf!(uplo, A, tol) -> (A, piv, rank, info)`

Computes the (upper if `uplo = U`, lower if `uplo = L`) pivoted Cholesky decomposition of positive-definite matrix  $A$  with a user-set tolerance `tol`.  $A$  is overwritten by its Cholesky decomposition.

Returns  $A$ , the pivots `piv`, the rank of  $A$ , and an `info` code. If `info = 0`, the factorization succeeded. If `info = i > 0`, then  $A$  is indefinite or rank-deficient.

`source`

[Base.LinAlg.LAPACK.ptsv!](#) – Function.

| `ptsv!(D, E, B)`

Solves  $A * X = B$  for positive-definite tridiagonal  $A$ .  $D$  is the diagonal of  $A$  and  $E$  is the off-diagonal.  $B$  is overwritten with the solution  $X$  and returned.

`source`

[Base.LinAlg.LAPACK.pttrf!](#) – Function.

| `pttrf!(D, E)`

## 54.4 LAPACK FUNCTIONS

Factorization of a positive-definite tridiagonal matrix with D as diagonal and E as off-diagonal. D and E are overwritten and returned.

[source](#)

`Base.LinAlg.LAPACK.pttrs!` – Function.

`| pttrs!(D, E, B)`

Solves  $A * X = B$  for positive-definite tridiagonal A with diagonal D and off-diagonal E after computing A's LDLt factorization using `pttrf!`. B is overwritten with the solution X.

[source](#)

`Base.LinAlg.LAPACK.trtri!` – Function.

`| trtri!(uplo, diag, A)`

Finds the inverse of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix A. If `diag = N`, A has non-unit diagonal elements. If `diag = U`, all diagonal elements of A are one. A is overwritten with its inverse.

[source](#)

`Base.LinAlg.LAPACK.trtrs!` – Function.

`| trtrs!(uplo, trans, diag, A, B)`

Solves  $A * X = B$  (`trans = N`),  $A.^* X = B$  (`trans = T`), or  $A'^* X = B$  (`trans = C`) for (upper if `uplo = U`, lower if `uplo = L`) triangular matrix A. If `diag = N`, A has non-unit diagonal elements. If `diag = U`, all diagonal elements of A are one. B is overwritten with the solution X.

[source](#)

`Base.LinAlg.LAPACK.trcon!` – Function.

1410 `trcon!(norm, uplo, diag, A)`

CHAPTER 54. LINEAR ALGEBRA

Finds the reciprocal condition number of (upper if `uplo = U`, lower if `uplo = L`) triangular matrix `A`. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. If `norm = I`, the condition number is found in the infinity norm. If `norm = 0` or `1`, the condition number is found in the one norm.

`source`

`Base.LinAlg.LAPACK.trevc!` – Function.

```
| trevc!(side, howmny, select, T, VL = similar(T), VR = similar(T))
```

Finds the eigensystem of an upper triangular matrix `T`. If `side = R`, the right eigenvectors are computed. If `side = L`, the left eigenvectors are computed. If `side = B`, both sets are computed. If `howmny = A`, all eigenvectors are found. If `howmny = B`, all eigenvectors are found and backtransformed using `VL` and `VR`. If `howmny = S`, only the eigenvectors corresponding to the values in `select` are computed.

`source`

`Base.LinAlg.LAPACK.trrfs!` – Function.

```
| trrfs!(uplo, trans, diag, A, B, X, Ferr, Berr) -> (Ferr, Berr)
```

Estimates the error in the solution to  $A * X = B$  (`trans = N`),  $A.' * X = B$  (`trans = T`),  $A' * X = B$  (`trans = C`) for `side = L`, or the equivalent equations a right-handed `side = R`  $X * A$  after computing `X` using `trtrs!`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, `A` is lower triangular. If `diag = N`, `A` has non-unit diagonal elements. If `diag = U`, all diagonal elements of `A` are one. `Ferr` and `Berr` are optional inputs. `Ferr`

54.4 is LAPACK FUNCTIONS and **Berr** is the backward error, each componentwise.

**source**

[Base.LinAlg.LAPACK.stev!](#) – Function.

| **stev!**(**job**, **dv**, **ev**) -> (**dv**, **Zmat**)

Computes the eigensystem for a symmetric tridiagonal matrix with **dv** as diagonal and **ev** as off-diagonal. If **job** = **N** only the eigenvalues are found and returned in **dv**. If **job** = **V** then the eigenvectors are also found and returned in **Zmat**.

**source**

[Base.LinAlg.LAPACK.stebz!](#) – Function.

| **stebz!**(**range**, **order**, **vl**, **vu**, **il**, **iu**, **abstol**, **dv**, **ev**) -> (**dv**, **iblock**, **isplit**)

Computes the eigenvalues for a symmetric tridiagonal matrix with **dv** as diagonal and **ev** as off-diagonal. If **range** = **A**, all the eigenvalues are found. If **range** = **V**, the eigenvalues in the half-open interval (**vl**, **vu**] are found. If **range** = **I**, the eigenvalues with indices between **il** and **iu** are found. If **order** = **B**, eigenvalues are ordered within a block. If **order** = **E**, they are ordered across all the blocks. **abstol** can be set as a tolerance for convergence.

**source**

[Base.LinAlg.LAPACK.stegr!](#) – Function.

| **stegr!**(**jobz**, **range**, **dv**, **ev**, **vl**, **vu**, **il**, **iu**) -> (**w**, **Z**)

Computes the eigenvalues (**jobz** = **N**) or eigenvalues and eigenvectors (**jobz** = **V**) for a symmetric tridiagonal matrix with **dv** as diagonal and **ev**

1412as off-diagonal. If `range = A`, all the eigenvalues in the half-open interval (`vl`, `] are found. If range = V, the eigenvalues in the half-open interval (vl, ] are found. If range = I, the eigenvalues with indices between il and iu are found. The eigenvalues are returned in w and the eigenvectors in Z.`

`source`

`Base.LinAlg.LAPACK.stein!` – Function.

```
| stein!(dv, ev_in, w_in, iblock_in, isplit_in)
```

Computes the eigenvectors for a symmetric tridiagonal matrix with `dv` as diagonal and `ev_in` as off-diagonal. `w_in` specifies the input eigenvalues for which to find corresponding eigenvectors. `iblock_in` specifies the submatrices corresponding to the eigenvalues in `w_in`. `isplit_in` specifies the splitting points between the submatrix blocks.

`source`

`Base.LinAlg.LAPACK.syconv!` – Function.

```
| syconv!(uplo, A, ipiv) -> (A, work)
```

Converts a symmetric matrix `A` (which has been factorized into a triangular matrix) into two matrices `L` and `D`. If `uplo = U`, `A` is upper triangular. If `uplo = L`, it is lower triangular. `ipiv` is the pivot vector from the triangular factorization. `A` is overwritten by `L` and `D`.

`source`

`Base.LinAlg.LAPACK.sysv!` – Function.

```
| sysv!(uplo, A, B) -> (B, A, ipiv)
```

Finds the solution to `A * X = B` for symmetric matrix `A`. If `uplo = U`, the upper half of `A` is stored. If `uplo = L`, the lower half is stored. `B` is overwritten by the solution `X`. `A` is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`Base.LinAlg.LAPACK.sytrf!` – Function.

```
| sytrf!(uplo, A) -> (A, ipiv, info)
```

Computes the Bunch-Kaufman factorization of a symmetric matrix `A`. If `uplo` = `U`, the upper half of `A` is stored. If `uplo` = `L`, the lower half is stored.

Returns `A`, overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`source`

`Base.LinAlg.LAPACK.sytri!` – Function.

```
| sytri!(uplo, A, ipiv)
```

Computes the inverse of a symmetric matrix `A` using the results of `sytrf!`. If `uplo` = `U`, the upper half of `A` is stored. If `uplo` = `L`, the lower half is stored. `A` is overwritten by its inverse.

`source`

`Base.LinAlg.LAPACK.sytrs!` – Function.

```
| sytrs!(uplo, A, ipiv, B)
```

Solves the equation `A * X = B` for a symmetric matrix `A` using the results of `sytrf!`. If `uplo` = `U`, the upper half of `A` is stored. If `uplo` = `L`, the lower half is stored. `B` is overwritten by the solution `X`.

`source`

`Base.LinAlg.LAPACK.hesv!` – Function.

Finds the solution to  $A * X = B$  for Hermitian matrix  $A$ . If `uplo` = U, the upper half of  $A$  is stored. If `uplo` = L, the lower half is stored.  $B$  is overwritten by the solution  $X$ .  $A$  is overwritten by its Bunch-Kaufman factorization. `ipiv` contains pivoting information about the factorization.

`source`

[Base.LinAlg.LAPACK.hetrf!](#) – Function.

| `hetrf!(uplo, A) -> (A, ipiv, info)`

Computes the Bunch-Kaufman factorization of a Hermitian matrix  $A$ . If `uplo` = U, the upper half of  $A$  is stored. If `uplo` = L, the lower half is stored.

Returns  $A$ , overwritten by the factorization, a pivot vector `ipiv`, and the error code `info` which is a non-negative integer. If `info` is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position `info`.

`source`

[Base.LinAlg.LAPACK.hetri!](#) – Function.

| `hetri!(uplo, A, ipiv)`

Computes the inverse of a Hermitian matrix  $A$  using the results of `sytrf!`. If `uplo` = U, the upper half of  $A$  is stored. If `uplo` = L, the lower half is stored.  $A$  is overwritten by its inverse.

`source`

[Base.LinAlg.LAPACK.hetrs!](#) – Function.

| `hetrs!(uplo, A, ipiv, B)`

54.4 Solving Linear Equations \*  $X = B$  for a Hermitian matrix  $A$  using the results of `sytrf!`. If `uplo = U`, the upper half of  $A$  is stored. If `uplo = L`, the lower half is stored.  $B$  is overwritten by the solution  $X$ .

**source**

[Base.LinAlg.LAPACK.syev!](#) – Function.

| `syev!(jobz, uplo, A)`

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix  $A$ . If `uplo = U`, the upper triangle of  $A$  is used. If `uplo = L`, the lower triangle of  $A$  is used.

**source**

[Base.LinAlg.LAPACK.syevr!](#) – Function.

| `syevr!(jobz, range, uplo, A, v1, vu, il, iu, abstol) -> (W, Z)`

Finds the eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix  $A$ . If `uplo = U`, the upper triangle of  $A$  is used. If `uplo = L`, the lower triangle of  $A$  is used. If `range = A`, all the eigenvalues are found. If `range = V`, the eigenvalues in the half-open interval  $(v1, vu]$  are found. If `range = I`, the eigenvalues with indices between `il` and `iu` are found. `abstol` can be set as a tolerance for convergence.

The eigenvalues are returned in `W` and the eigenvectors in `Z`.

**source**

[Base.LinAlg.LAPACK.sygvd!](#) – Function.

| `sygvd!(itype, jobz, uplo, A, B) -> (w, A, B)`

Finds the generalized eigenvalues (`jobz = N`) or eigenvalues and eigenvectors (`jobz = V`) of a symmetric matrix  $A$  and symmetric positive-definite matrix  $B$ . If `uplo = U`, the upper triangles of  $A$  and  $B$  are used. If `uplo =`

1416, the lower triangles of A and B are used to solve is  $A * x = \lambda * B * x$ . If `itype` = 2, the problem to solve is  $A * B * x = \lambda * x$ . If `itype` = 3, the problem to solve is  $B * A * x = \lambda * x$ .

[source](#)

[Base.LinAlg.LAPACK.bdsqr!](#) – Function.

```
| bdsqr!(uplo, d, e_, Vt, U, C) -> (d, Vt, U, C)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal. If `uplo` = `U`, `e_` is the superdiagonal. If `uplo` = `L`, `e_` is the subdiagonal. Can optionally also compute the product  $Q' * C$ .

Returns the singular values in `d`, and the matrix `C` overwritten with  $Q' * C$ .

[source](#)

[Base.LinAlg.LAPACK.bdsdc!](#) – Function.

```
| bdsdc!(uplo, compq, d, e_) -> (d, e, u, vt, q, iq)
```

Computes the singular value decomposition of a bidiagonal matrix with `d` on the diagonal and `e_` on the off-diagonal using a divide and conquer method. If `uplo` = `U`, `e_` is the superdiagonal. If `uplo` = `L`, `e_` is the subdiagonal. If `compq` = `N`, only the singular values are found. If `compq` = `I`, the singular values and vectors are found. If `compq` = `P`, the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in `d`, and if `compq` = `P`, the compact singular vectors in `iq`.

[source](#)

[Base.LinAlg.LAPACK.gecon!](#) – Function.

Finds the reciprocal condition number of matrix A. If `normtype` = I, the condition number is found in the infinity norm. If `normtype` = 0 or 1, the condition number is found in the one norm. A must be the result of `getrf!` and `anorm` is the norm of A in the relevant norm.

`source`

`Base.LinAlg.LAPACK.gehrd!` – Function.

`| gehrd!(ilo, ihi, A) -> (A, tau)`

Converts a matrix A to Hessenberg form. If A is balanced with `gebal!` then `ilo` and `ihi` are the outputs of `gebal!`. Otherwise they should be `ilo` = 1 and `ihi` = `size(A, 2)`. `tau` contains the elementary reflectors of the factorization.

`source`

`Base.LinAlg.LAPACK.orghr!` – Function.

`| orghr!(ilo, ihi, A, tau)`

Explicitly finds Q, the orthogonal/unitary matrix from `gehrd!`. `ilo`, `ihi`, `A`, and `tau` must correspond to the input/output to `gehrd!`.

`source`

`Base.LinAlg.LAPACK.gees!` – Function.

`| gees!(jobvs, A) -> (A, vs, w)`

Computes the eigenvalues (`jobvs` = N) or the eigenvalues and Schur vectors (`jobvs` = V) of matrix A. A is overwritten by its Schur form.

Returns A, `vs` containing the Schur vectors, and `w`, containing the eigenvalues.

`source`

## [Base.LinAlg.LAPACK.gges!](#) – Function. CHAPTER 54. LINEAR ALGEBRA

```
| gges!(jobvsl, jobvsr, A, B) -> (A, B, alpha, beta, vsl, vsr)
```

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (`jobsvl = V`), or right Schur vectors (`jobvsr = V`) of `A` and `B`.

The generalized eigenvalues are returned in `alpha` and `beta`. The left Schur vectors are returned in `vsl` and the right Schur vectors are returned in `vsr`.

`source`

## [Base.LinAlg.LAPACK.trexc!](#) – Function.

```
| trexc!(compq, ifst, ilst, T, Q) -> (T, Q)
```

Reorder the Schur factorization of a matrix. If `compq = V`, the Schur vectors `Q` are reordered. If `compq = N` they are not modified. `ifst` and `ilst` specify the reordering of the vectors.

`source`

## [Base.LinAlg.LAPACK.trsen!](#) – Function.

```
| trsen!(compq, job, select, T, Q) -> (T, Q, w, s, sep)
```

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If `job = N`, no condition numbers are found. If `job = E`, only the condition number for this cluster of eigenvalues is found. If `job = V`, only the condition number for the invariant subspace is found. If `job = B` then the condition numbers for the cluster and subspace are found. If `compq = V` the Schur vectors `Q` are updated. If `compq = N` the Schur vectors are not modified. `select` determines which eigenvalues are in the cluster.

54.4 LAPACK FUNCTIONS eigenvalues in `w`, the condition number of the cluster of eigenvalues `s`, and the condition number of the invariant subspace `sep`.

`source`

`Base.LinAlg.LAPACK.tgsen!` – Function.

```
| tgsen!(select, S, T, Q, Z) -> (S, T, alpha, beta, Q, Z)
```

Reorders the vectors of a generalized Schur decomposition. `select` specifies the eigenvalues in each cluster.

`source`

`Base.LinAlg.LAPACK.trsyl!` – Function.

```
| trsyl!(transa, transb, A, B, C, isgn=1) -> (C, scale)
```

Solves the Sylvester matrix equation  $A * X +/ - X * B = \text{scale} * C$  where `A` and `B` are both quasi-upper triangular. If `transa` = `N`, `A` is not modified. If `transa` = `T`, `A` is transposed. If `transa` = `C`, `A` is conjugate transposed. Similarly for `transb` and `B`. If `isgn` = 1, the equation  $A * X + X * B = \text{scale} * C$  is solved. If `isgn` = -1, the equation  $A * X - X * B = \text{scale} * C$  is solved.

Returns `X` (overwriting `C`) and `scale`.

`source`



# Chapter 55

## Constants

`Core.nothing` – Constant.

| `nothing`

The singleton instance of type `Void`, used by convention when there is no value to return (as in a C `void` function). Can be converted to an empty `Nullable` value.

`source`

`Base.PROGRAM_FILE` – Constant.

| `PROGRAM_FILE`

A string containing the script name passed to Julia from the command line. Note that the script name remains unchanged from within included files. Alternatively see `@__FILE__`.

`source`

`Base.ARGS` – Constant.

| `ARGS`

An array of the command line arguments passed to Julia, as strings.

`source`

`Base.C_NULL` – Constant.

CHAPTER 55. CONSTANTS

| `C_NULL`

The C null pointer constant, sometimes used when calling external code.

`source`

`Base.VERSION` – Constant.

| `VERSION`

A `VersionNumber` object describing which version of Julia is in use. For details see [Version Number Literals](#).

`source`

`Base.LOAD_PATH` – Constant.

| `LOAD_PATH`

An array of paths as strings or custom loader objects for the `require` function and `using` and `import` statements to consider when loading code.

`source`

`Base.JULIA_HOME` – Constant.

| `JULIA_HOME`

A string containing the full path to the directory containing the `julia` executable.

`source`

`Base.Sys.CPU_CORES` – Constant.

| `Sys.CPU_CORES`

The number of logical CPU cores available in the system.

1423

See the `Hwloc.jl` package for extended information, including number of physical cores.

**source**

`Base.Sys.WORD_SIZE` – Constant.

| `Sys.WORD_SIZE`

Standard word size on the current machine, in bits.

**source**

`Base.Sys.KERNEL` – Constant.

| `Sys.KERNEL`

A symbol representing the name of the operating system, as returned by `uname` of the build configuration.

**source**

`Base.Sys.ARCH` – Constant.

| `Sys.ARCH`

A symbol representing the architecture of the build configuration.

**source**

`Base.Sys.MACHINE` – Constant.

| `Sys.MACHINE`

A string containing the build triple.

**source**

See also:

`STDOUT``STDERR``ENV``ENDIAN_BOM``Libc.MS_ASYNC``Libc.MS_INVALIDATE``Libc.MS_SYNC``Libdl.DL_LOAD_PATH``Libdl.RTLD_DEEPBIND``Libdl.RTLD_LOCAL``Libdl.RTLD_NOLOAD``Libdl.RTLD.LAZY``Libdl.RTLD_NOW``Libdl.RTLD_GLOBAL``Libdl.RTLD_NODELETE``Libdl.RTLD_FIRST`

# Chapter 56

## Filesystem

`Base.Filesystem.pwd` – Function.

```
| pwd() -> AbstractString
```

Get the current working directory.

`source`

`Base.Filesystem.cd` – Method.

```
| cd(dir::AbstractString=homedir())
```

Set the current working directory.

`source`

`Base.Filesystem.cd` – Method.

```
| cd(f::Function, dir::AbstractString=homedir())
```

Temporarily changes the current working directory and applies function `f` before returning.

`source`

`Base.Filesystem.readdir` – Function.

```
| readdir(dir::AbstractString=".") -> Vector{String}
```

142 Returns the files and directories in the directory **CHARTER 5 Current File System** directory if not given).

**source**

[Base.Filesystem.walkdir](#) – Function.

```
walkdir(dir; topdown=true, follow_symlinks=false, onerror=throw  
    )
```

The **walkdir** method returns an iterator that walks the directory tree of a directory. The iterator returns a tuple containing (**rootpath**, **dirs**, **files**). The directory tree can be traversed top-down or bottom-up. If **walkdir** encounters a [SystemError](#) it will rethrow the error by default. A custom error handling function can be provided through **onerror** keyword argument. **onerror** is called with a [SystemError](#) as argument.

```
for (root, dirs, files) in walkdir(".")  
    println("Directories in $root")  
    for dir in dirs  
        println(joinpath(root, dir)) # path to directories  
    end  
    println("Files in $root")  
    for file in files  
        println(joinpath(root, file)) # path to files  
    end  
end
```

**source**

[Base.Filesystem.mkdir](#) – Function.

```
mkdir(path::AbstractString, mode::Unsigned=0o777)
```

Make a new directory with name **path** and permissions **mode**. **mode** defaults to **0o777**, modified by the current file creation mask. This function

never creates more than one directory. If the directory already exists<sup>1427</sup>, or some intermediate directories do not exist, this function throws an error. See [mkpath](#) for a function which creates all required intermediate directories.

[source](#)

[Base.Filesystem.mkpath](#) – Function.

```
| mkpath(path::AbstractString, mode::Unsigned=0o777)
```

Create all directories in the given **path**, with permissions **mode**. **mode** defaults to **0o777**, modified by the current file creation mask.

[source](#)

[Base.Filesystem.symlink](#) – Function.

```
| symlink(target::AbstractString, link::AbstractString)
```

Creates a symbolic link to **target** with the name **link**.

Note

This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

[source](#)

[Base.Filesystem.readlink](#) – Function.

```
| readlink(path::AbstractString) -> AbstractString
```

Returns the target location a symbolic link **path** points to.

[source](#)

[Base.Filesystem.chmod](#) – Function.

```
| chmod(path::AbstractString, mode::Integer; recursive::Bool=false)
```

1428 Change the permissions mode of path to mode. CHAPTER 5: Filesystem

0o777) are currently supported. If recursive=true and the path is a directory all permissions in that directory will be recursively changed.

[source](#)

[Base.Filesystem.chown](#) – Function.

| chown(path::AbstractString, owner::Integer, group::Integer=-1)

Change the owner and/or group of path to owner and/or group. If the value entered for owner or group is -1 the corresponding ID will not change. Only integer owners and groups are currently supported.

[source](#)

[Base.stat](#) – Function.

| stat(file)

Returns a structure whose fields contain information about the file. The fields of the structure are:

Name	Description
size	The size (in bytes) of the file
device	ID of the device that contains the file
inode	The inode number of the file
mode	The protection mode of the file
nlink	The number of hard links to the file
uid	The user id of the owner of the file
gid	The group id of the file owner
rdev	If this file refers to a device, the ID of the device it refers to
blksize	The file-system preferred block size for the file
blocks	The number of such blocks allocated
mtime	Unix timestamp of when the file was last modified
ctime	Unix timestamp of when the file was created

[source](#)

[Base.Filesystem.lstat](#) – Function.

| lstat(file)

Like `stat`, but for symbolic links gets the info for the link itself rather than the file it refers to. This function must be called on a file path rather than a file object or a file descriptor.

`source`

`Base.Filesystem.ctime` – Function.

| `ctime(file)`

Equivalent to `stat(file).ctime`

`source`

`Base.Filesystem.mtime` – Function.

| `mtime(file)`

Equivalent to `stat(file).mtime`.

`source`

`Base.Filesystem.filemode` – Function.

| `filemode(file)`

Equivalent to `stat(file).mode`

`source`

`Base.Filesystem.filesize` – Function.

| `filesize(path...)`

Equivalent to `stat(file).size`.

`source`

`Base.Filesystem.uperm` – Function.

| `uperm(file)`

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

## CHAPTER 56. FILESYSTEM

Gets the permissions of the owner of the file as a bitfield of

For allowed arguments, see [stat](#).

[source](#)

[Base.Filesystem.gperm](#) – Function.

| `gperm(file)`

Like [uperm](#) but gets the permissions of the group owning the file.

[source](#)

[Base.Filesystem.operm](#) – Function.

| `operm(file)`

Like [uperm](#) but gets the permissions for people who neither own the file nor are a member of the group owning the file

[source](#)

[Base.Filesystem.cp](#) – Function.

| `cp(src::AbstractString, dst::AbstractString; remove_destination  
::Bool=false, follow_symlinks::Bool=false)`

Copy the file, link, or directory from `src` to `dst`. `remove_destination=true` will first remove an existing `dst`.

If `follow_symlinks=false`, and `src` is a symbolic link, `dst` will be created as a symbolic link. If `follow_symlinks=true` and `src` is a symbolic link, `dst` will be a copy of the file or directory `src` refers to.

[source](#)

```
| download(url::AbstractString, [localfile::AbstractString])
```

Download a file from the given url, optionally renaming it to the given local file name. Note that this function relies on the availability of external tools such as `curl`, `wget` or `fetch` to download the file and is provided for convenience. For production use or situations in which more options are needed, please use a package that provides the desired functionality instead.

**source**

```
| mv(src::AbstractString, dst::AbstractString; remove_destination  
|   ::Bool=false)
```

Move the file, link, or directory from `src` to `dst`. `remove_destination=true` will first remove an existing `dst`.

**source**

```
| rm(path::AbstractString; force::Bool=false, recursive::Bool=  
|   false)
```

Delete the file, link, or empty directory at the given path. If `force=true` is passed, a non-existing path is not treated as error. If `recursive=true` is passed and the path is a directory, then all contents are removed recursively.

**source**

```
| touch(path::AbstractString)
```

`source`

`Base.Filesystem.tempname` – Function.

| `tempname()`

Generate a unique temporary file path.

`source`

`Base.Filesystem.tempdir` – Function.

| `tempdir()`

Obtain the path of a temporary directory (possibly shared with other processes).

`source`

`Base.Filesystem.mktemp` – Method.

| `mktemp(parent=tempdir())`

Returns `(path, io)`, where `path` is the path of a new temporary file in `parent` and `io` is an open file object for this path.

`source`

`Base.Filesystem.mktemp` – Method.

| `mktemp(f::Function, parent=tempdir())`

Apply the function `f` to the result of `mktemp(parent)` and remove the temporary file upon completion.

`source`

`Base.Filesystem.mktempdir` – Method.

| `mktempdir(parent=tempdir())`

Create a temporary directory in the `parent` directory and return its `path`.  
If `parent` does not exist, throw an error.

[source](#)

`Base.Filesystem.mktempdir` – Method.

| `mktempdir(f::Function, parent=tempdir())`

Apply the function `f` to the result of `mktempdir(parent)` and remove the temporary directory upon completion.

[source](#)

`Base.Filesystem.isblockdev` – Function.

| `isblockdev(path) -> Bool`

Returns `true` if `path` is a block device, `false` otherwise.

[source](#)

`Base.Filesystem.ischardev` – Function.

| `ischardev(path) -> Bool`

Returns `true` if `path` is a character device, `false` otherwise.

[source](#)

`Base.Filesystem.isdir` – Function.

| `.isdir(path) -> Bool`

Returns `true` if `path` is a directory, `false` otherwise.

[source](#)

`Base.Filesystem.isfifo` – Function.

| `isfifo(path) -> Bool`

143>Returns **true** if path is a FIFO, **false** otherwise. CHAPTER 56. FILESYSTEM

[source](#)

[Base.Filesystem.isfile](#) – Function.

| `.isfile(path) -> Bool`

Returns **true** if path is a regular file, **false** otherwise.

[source](#)

[Base.Filesystem.islink](#) – Function.

| `.islink(path) -> Bool`

Returns **true** if path is a symbolic link, **false** otherwise.

[source](#)

[Base.Filesystem.ismount](#) – Function.

| `.ismount(path) -> Bool`

Returns **true** if path is a mount point, **false** otherwise.

[source](#)

[Base.Filesystem.ispath](#) – Function.

| `.ispath(path) -> Bool`

Returns **true** if path is a valid filesystem path, **false** otherwise.

[source](#)

[Base.Filesystem.issetgid](#) – Function.

| `.issetgid(path) -> Bool`

Returns **true** if path has the setgid flag set, **false** otherwise.

[source](#)

`Base.Filesystem.issetuid` – Function.

1435

```
| issetuid(path) -> Bool
```

Returns `true` if `path` has the setuid flag set, `false` otherwise.

`source`

`Base.Filesystem.issocket` – Function.

```
| issocket(path) -> Bool
```

Returns `true` if `path` is a socket, `false` otherwise.

`source`

`Base.Filesystem.issticky` – Function.

```
| issticky(path) -> Bool
```

Returns `true` if `path` has the sticky bit set, `false` otherwise.

`source`

`Base.Filesystem.homedir` – Function.

```
| homedir() -> AbstractString
```

Return the current user's home directory.

Note

`homedir` determines the home directory via `libuv`'s `uv_os_homedir`. For details (for example on how to specify the home directory via environment variables), see the [uv\\_os\\_homedir documentation](#).

`source`

`Base.Filesystem.dirname` – Function.

1436 `dirname(path::AbstractString) -> AbstractString` CHAPTER 56. FILESYSTEM

Get the directory part of a path.

Examples

```
| julia> dirname("/home/myuser")
| "/home"
```

See also: [basename](#)

[source](#)

[Base.Filesystem.basename](#) – Function.

```
| basename(path::AbstractString) -> AbstractString
```

Get the file name part of a path.

Examples

```
| julia> basename("/home/myuser/example.jl")
| "example.jl"
```

See also: [dirname](#)

[source](#)

[Base. @\\_FILE\\_\\_](#) – Macro.

```
| @_FILE__ -> AbstractString
```

`@__FILE__` expands to a string with the path to the file containing the macrocall, or an empty string if evaluated by `julia -e <expr>`. Returns `nothing` if the macro was missing parser source information. Alternatively see [PROGRAM\\_FILE](#).

[source](#)

[Base. @\\_DIR\\_\\_](#) – Macro.

```
| @__DIR__ -> AbstractString
```

1437

`@__DIR__` expands to a string with the absolute path to the directory of the file containing the macrocall. Returns the current working directory if run from a REPL or if evaluated by `julia -e <expr>`.

`source`

`Base.@__LINE__` – Macro.

```
| @__LINE__ -> Int
```

`@__LINE__` expands to the line number of the location of the macrocall. Returns `0` if the line number could not be determined.

`source`

`Base.Filesystem.isabspath` – Function.

```
| isabspath(path::AbstractString) -> Bool
```

Determines whether a path is absolute (begins at the root directory).

Examples

```
| julia> isabspath( "/home" )
```

```
true
```

```
| julia> isabspath( "home" )
```

```
false
```

`source`

`Base.Filesystem.isdirpath` – Function.

```
| isdirpath(path::AbstractString) -> Bool
```

Determines whether a path refers to a directory (for example, ends with a path separator).

```
| julia> isdirpath("/home")
| false
```

```
| julia> isdirpath("/home/")
| true
```

source

[Base.Filesystem.joinpath](#) – Function.

```
| joinpath(parts...) -> AbstractString
```

Join path components into a full path. If some argument is an absolute path or (on Windows) has a drive specification that doesn't match the drive computed for the join of the preceding paths, then prior components are dropped.

Examples

```
| julia> joinpath("/home/myuser", "example.jl")
| "/home/myuser/example.jl"
```

source

[Base.Filesystem.abspath](#) – Function.

```
| abspath(path::AbstractString) -> AbstractString
```

Convert a path to an absolute path by adding the current directory if necessary.

source

```
| abspath(path::AbstractString, paths::AbstractString...) ->
|   AbstractString
```

Convert a set of paths to an absolute path by joining them together<sup>1439</sup> adding the current directory if necessary. Equivalent to `abspath(joinpath(path, paths...))`.

`source`

`Base.Filesystem.normpath` – Function.

`| normpath(path::AbstractString) -> AbstractString`

Normalize a path, removing `"."` and `".."` entries.

Examples

`| julia> normpath("/home/myuser/../example.jl")`  
`"/home/example.jl"`

`source`

`Base.Filesystem.realpath` – Function.

`| realpath(path::AbstractString) -> AbstractString`

Canonicalize a path by expanding symbolic links and removing `"."` and `".."` entries.

`source`

`Base.Filesystem.relpwd` – Function.

`| relpath(path::AbstractString, startpath::AbstractString = ".")`  
`-> AbstractString`

Return a relative filepath to `path` either from the current directory or from an optional start directory. This is a path computation: the filesystem is not accessed to confirm the existence or nature of `path` or `startpath`.

`source`

`Base.Filesystem.expanduser` – Function.

1440 `expanduser(path::AbstractString) -> AbstractString` CHAPTER 56. FILESYSTEM

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

`source`

`Base.Filesystem.splitdir` – Function.

```
splitdir(path::AbstractString) -> (AbstractString,  
AbstractString)
```

Split a path into a tuple of the directory name and file name.

Examples

```
julia> splitdir("/home/myuser")  
("/home", "myuser")
```

`source`

`Base.Filesystem.splitdrive` – Function.

```
splitdrive(path::AbstractString) -> (AbstractString,  
AbstractString)
```

On Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.

`source`

`Base.Filesystem.splitext` – Function.

```
splitext(path::AbstractString) -> (AbstractString,  
AbstractString)
```

If the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.

```
julia> splitext("/home/myuser/example.jl")
("/home/myuser/example", ".jl")

julia> splitext("/home/myuser/example")
("/home/myuser/example", "")
```

source



# Chapter 57

## Delimited Files

[DelimitedFiles.readdlm](#) – Method.

```
readdlm(source, delim::Char, T::Type, eol::Char; header=false,  
       skipstart=0, skipblanks=true, use_mmap, quotes=true, dims,  
       comments=true, comment_char='#')
```

Read a matrix from the source where each line (separated by `eol`) gives one row, with elements separated by the given delimiter. The source can be a text file, stream or byte array. Memory mapped files can be used by passing the byte array representation of the mapped segment as source.

If `T` is a numeric type, the result is an array of that type, with any non-numeric elements as `NaN` for floating-point types, or zero. Other useful values of `T` include `String`, `AbstractString`, and `Any`.

If `header` is `true`, the first row of data will be read as header and the tuple `(data_cells, header_cells)` is returned instead of only `data_cells`.

Specifying `skipstart` will ignore the corresponding number of initial lines from the input.

If `skipblanks` is `true`, blank lines in the input will be ignored.

144f `use_mmap` is `true`, the file specified by `source` is memory mapped for potential speedups. Default is `true` except on Windows. On Windows, you may want to specify `true` if the file is large, and is only read once and not written to.

If `quotes` is `true`, columns enclosed within double-quote ("") characters are allowed to contain new lines and column delimiters. Double-quote characters within a quoted field must be escaped with another double-quote. Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If `comments` is `true`, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

`source`

[DelimitedFiles.readdlm](#) – Method.

```
| readdlm(source, delim::Char, eol::Char; options...)
```

If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

`source`

[DelimitedFiles.readdlm](#) – Method.

```
| readdlm(source, delim::Char, T::Type; options...)
```

The end of line delimiter is taken as \n.

`source`

[DelimitedFiles.readdlm](#) – Method.

```
| readdlm(source, delim::Char; options...)
```

The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

**source**

[DelimitedFiles.readdlm](#) – Method.

```
| readdlm(source, T::Type; options...)
```

The columns are assumed to be separated by one or more whitespaces.

The end of line delimiter is taken as `\n`.

**source**

[DelimitedFiles.readdlm](#) – Method.

```
| readdlm(source; options...)
```

The columns are assumed to be separated by one or more whitespaces.

The end of line delimiter is taken as `\n`. If all data is numeric, the result will be a numeric array. If some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

**source**

[DelimitedFiles.writedlm](#) – Function.

```
| writedlm(f, A, delim='\t'; opts)
```

Write `A` (a vector, matrix, or an iterable collection of iterable rows) as text to `f` (either a filename string or an `I/O` stream) using the given delimiter `delim` (which defaults to tab, but can be any printable Julia object, typically a `Char` or `AbstractString`).

For example, two vectors `x` and `y` of the same length can be written as two columns of tab-delimited text to `f` by either `writedlm(f, [x y])` or by `writedlm(f, zip(x, y))`.



# Chapter 58

## I/O and Network

### 58.1 General I/O

`Base.STDOUT` – Constant.

| `STDOUT`

Global variable referring to the standard out stream.

`source`

`Base.STDERR` – Constant.

| `STDERR`

Global variable referring to the standard error stream.

`source`

`Base.STDIN` – Constant.

| `STDIN`

Global variable referring to the standard input stream.

`source`

`Base.open` – Function.

1448 open(filename::AbstractString, [read::Bool, write::Bool, create  
    ::Bool, truncate::Bool, append::Bool]) -> IOStream

CHAPTER 58. I/O AND NETWORK  
Open a file in a mode specified by five boolean arguments. The default is to open files for reading only. Returns a stream for accessing the file.

**source**

open(filename::AbstractString, [mode::AbstractString]) ->  
IOStream

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of `mode` correspond to those from `fopen(3)` or Perl `open`, and are equivalent to setting the following boolean groups:

Mode	Description
r	read
r+	read, write
w	write, create, truncate
w+	read, write, create, truncate
a	write, create, append
a+	read, write, create, append

**source**

open(f::Function, args...)

Apply the function `f` to the result of `open(args...)` and close the resulting file descriptor upon completion.

Examples

open(f->read(f, String), "file.txt")

**source**

open(command, mode::AbstractString="r", stdio=DevNull)

58.1 Start ~~GENERAL~~ / Command asynchronously, and return a tuple (`stream`, ~~p1469~~)  
If `mode` is "r", then `stream` reads from the process's standard output and `stdio` optionally specifies the process's standard input stream. If `mode` is "w", then `stream` writes to the process's standard input and `stdio` optionally specifies the process's standard output stream.

[source](#)

```
open(f::Function, command, mode::AbstractString="r", stdio=DevNull)
```

Similar to `open(command, mode, stdio)`, but calls `f(stream)` on the resulting process stream, then closes the input stream and waits for the process to complete. Returns the value returned by `f`.

[source](#)

`Base.IOBuffer` – Type.

```
IOBuffer([data, ][readable::Bool=true, writable::Bool=false[, maxsize::Int=typemax(Int)]])
```

Create an `IOBuffer`, which may optionally operate on a pre-existing array. If the readable/writable arguments are given, they restrict whether or not the buffer may be read from or written to respectively. The last argument optionally specifies a size beyond which the buffer may not be grown.

[source](#)

```
IOBuffer() -> IOBuffer
```

Create an in-memory I/O stream.

[source](#)

```
IOBuffer(size::Integer)
```

1450 Create a fixed size IOBuffer. The buffer will grow by 1024 bytes when needed.

source

```
| IOBuffer(string::String)
```

Create a read-only IOBuffer on the data underlying the given string.

Examples

```
julia> io = IOBuffer("Haho");
```

```
julia> String(take!(io))
```

```
"Haho"
```

```
julia> String(take!(io))
```

```
"Haho"
```

source

Base.take! – Method.

```
| take!(b::IOBuffer)
```

Obtain the contents of an IOBuffer as an array, without copying. Afterwards, the IOBuffer is reset to its initial state.

source

Base.fdio – Function.

```
| fdio([name::AbstractString, ]fd::Integer[, own::Bool=false]) ->
| IOStream
```

Create an IOStream object from an integer file descriptor. If own is true, closing this object will close the underlying descriptor. By default, an IOStream is closed when it is garbage collected. name allows you to associate the descriptor with a named file.

source

```
| flush(stream)
```

Commit all currently buffered writes to the given stream.

**source**

**Base.close** – Function.

```
| close(stream)
```

Close an I/O stream. Performs a **flush** first.

**source**

**Base.write** – Function.

```
| write(stream::IO, x)
| write(filename::AbstractString, x)
```

Write the canonical binary representation of a value to the given I/O stream or file. Returns the number of bytes written into the stream.

You can write multiple values with the same **write** call. i.e. the following are equivalent:

```
| write(stream, x, y...)
| write(stream, x) + write(stream, y...)
```

**source**

**Base.read** – Function.

```
| read(stream::IO, T)
```

Read a single value of type T from **stream**, in canonical binary representation.

```
| read(stream::IO, String)
```

**source**

```
| read(filename::AbstractString, args...)
```

Open a file and read its contents. `args` is passed to `read`: this is equivalent to `open(io->read(io, args...), filename)`.

```
| read(filename::AbstractString, String)
```

Read the entire contents of a file as a string.

**source**

```
| read(s::IO, nb=typemax(Int))
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

**source**

```
| read(s::IOStream, nb::Integer; all=true)
```

Read at most `nb` bytes from `s`, returning a `Vector{UInt8}` of the bytes read.

If `all` is `true` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `all` is `false`, at most one `read` call is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `all` option.

**source**

`Base.read!` – Function.

```
| read!(stream::IO, array::Union{Array, BitArray})
```

```
| read!(filename::AbstractString, array::Union{Array, BitArray})
```

**source**

[Base.readbytes!](#) – Function.

```
| readbytes!(stream::IO, b::AbstractVector{UInt8}, nb=length(b))
```

Read at most **nb** bytes from **stream** into **b**, returning the number of bytes read. The size of **b** will be increased if needed (i.e. if **nb** is greater than **length(b)** and enough bytes could be read), but it will never be decreased.

**source**

```
| readbytes!(stream::IOStream, b::AbstractVector{UInt8}, nb=
|   length(b); all::Bool=true)
```

Read at most **nb** bytes from **stream** into **b**, returning the number of bytes read. The size of **b** will be increased if needed (i.e. if **nb** is greater than **length(b)** and enough bytes could be read), but it will never be decreased.

See [read](#) for a description of the **all** option.

**source**

[Base.unsafe\\_read](#) – Function.

```
| unsafe_read(io::IO, ref, nbytes::UInt)
```

Copy **nbytes** from the **IO** stream object into **ref** (converted to a pointer).

It is recommended that subtypes **T<:IO** override the following method signature to provide more efficient implementations: **unsafe\_read(s::T, p::Ptr{UInt8}, n::UInt)**

**source**

[Base.unsafe\\_write](#) – Function.

1454 `unsafe_write(io::IO, ref, nbytes:UInt)` CHAPTER 58. I/O AND NETWORK

Copy `nbytes` from `ref` (converted to a pointer) into the `IO` object.

It is recommended that subtypes `T<:IO` override the following method signature to provide more efficient implementations: `unsafe_write(s:T, p:Ptr{UInt8}, n:UInt)`

`source`

[Base.position](#) – Function.

`| position(s)`

Get the current position of a stream.

`source`

[Base.seek](#) – Function.

`| seek(s, pos)`

Seek a stream to the given position.

`source`

[Base.seekstart](#) – Function.

`| seekstart(s)`

Seek a stream to its beginning.

`source`

[Base.seekend](#) – Function.

`| seekend(s)`

Seek a stream to its end.

`source`

| `skip(s, offset)`

Seek a stream relative to the current position.

`source`

**Base.mark** – Function.

| `mark(s)`

Add a mark at the current position of stream `s`. Returns the marked position.

See also [unmark](#), [reset](#), [ismarked](#).

`source`

**Base.unmark** – Function.

| `unmark(s)`

Remove a mark from stream `s`. Returns `true` if the stream was marked, `false` otherwise.

See also [mark](#), [reset](#), [ismarked](#).

`source`

**Base.reset** – Function.

| `reset(s)`

Reset a stream `s` to a previously marked position, and remove the mark. Returns the previously marked position. Throws an error if the stream is not marked.

See also [mark](#), [unmark](#), [ismarked](#).

`source`

[Base.ismarked](#) – Function.

CHAPTER 58. I/O AND NETWORK

```
| ismarked(s)
```

Returns **true** if stream **s** is marked.

See also [mark](#), [unmark](#), [reset](#).

[source](#)

[Base.eof](#) – Function.

```
| eof(stream) -> Bool
```

Tests whether an I/O stream is at end-of-file. If the stream is not yet exhausted, this function will block to wait for more data if necessary, and then return **false**. Therefore it is always safe to read one byte after seeing **eof** return **false**. **eof** will return **false** as long as buffered data is still available, even if the remote end of a connection is closed.

[source](#)

[Base.isreadonly](#) – Function.

```
| isreadonly(stream) -> Bool
```

Determine whether a stream is read-only.

[source](#)

[Base.iswritable](#) – Function.

```
| iswritable(io) -> Bool
```

Returns **true** if the specified IO object is writable (if that can be determined).

[source](#)

[Base.isreadable](#) – Function.

Returns `true` if the specified IO object is readable (if that can be determined).

`source`

`Base.isopen` – Function.

| `isopen(object) -> Bool`

Determine whether an object – such as a stream, timer, or mmap – is not yet closed. Once an object is closed, it will never produce a new event. However, a closed stream may still have data to read in its buffer, use `eof` to check for the ability to read data. Use the `FileWatching` package to be notified when a stream might be writable or readable.

`source`

`Base.Serializer.serialize` – Function.

| `serialize(stream::IO, value)`

Write an arbitrary value to a stream in an opaque format, such that it can be read back by `deserialize`. The read-back value will be as identical as possible to the original. In general, this process will not work if the reading and writing are done by different versions of Julia, or an instance of Julia with a different system image. `Ptr` values are serialized as all-zero bit patterns (`NULL`).

An 8-byte identifying header is written to the stream first. To avoid writing the header, construct a `SerializationState` and use it as the first argument to `serialize` instead. See also `Serializer.writeheader`.

`source`

`Base.Serializer.deserialize` – Function.

1458 `deserialize(stream)`

CHAPTER 58. I/O AND NETWORK

Read a value written by `serialize`. `deserialize` assumes the binary data read from `stream` is correct and has been serialized by a compatible implementation of `serialize`. It has been designed with simplicity and performance as a goal and does not validate the data read. Mal-formed data can result in process termination. The caller has to ensure the integrity and correctness of data read from `stream`.

`source`

`Base.Serializer.writeheader` – Function.

| `Serializer.writeheader(s::AbstractSerializer)`

Write an identifying header to the specified serializer. The header consists of 8 bytes as follows:

Offset	Description
0	tag byte (0x37)
1-2	signature bytes "JL"
3	protocol version
4	bits 0-1: endianness: 0 = little, 1 = big
4	bits 2-3: platform: 0 = 32-bit, 1 = 64-bit
5-7	reserved

`source`

`Base.Grisu.print_shortest` – Function.

| `print_shortest(io::IO, x)`

Print the shortest possible representation, with the minimum number of consecutive non-zero digits, of number `x`, ensuring that it would parse to the exact same number.

`source`

`Base.fd` – Function.

Returns the file descriptor backing the stream or file. Note that this function only applies to synchronous `File`'s and `IOStream`'s not to any of the asynchronous streams.

`source`

`Base.redirect_stdout` – Function.

```
| redirect_stdout([stream]) -> (rd, wr)
```

Create a pipe to which all C and Julia level `STDOUT` output will be redirected. Returns a tuple `(rd, wr)` representing the pipe ends. Data written to `STDOUT` may now be read from the `rd` end of the pipe. The `wr` end is given for convenience in case the old `STDOUT` object was cached by the user and needs to be replaced elsewhere.

Note

`stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

`source`

`Base.redirect_stdout` – Method.

```
| redirect_stdout(f::Function, stream)
```

Run the function `f` while redirecting `STDOUT` to `stream`. Upon completion, `STDOUT` is restored to its prior setting.

Note

`stream` must be a `TTY`, a `Pipe`, or a `TCPSocket`.

`source`

`Base.redirect_stderr` – Function.

1460 `redirect_stderr([stream]) -> (rd, wr)` CHAPTER 58. I/O AND NETWORK

Like `redirect_stdout`, but for `STDERR`.

Note

`stream` must be a TTY, a Pipe, or a TCPSocket.

`source`

`Base.redirect_stderr` – Method.

`| redirect_stderr(f::Function, stream)`

Run the function `f` while redirecting `STDERR` to `stream`. Upon completion, `STDERR` is restored to its prior setting.

Note

`stream` must be a TTY, a Pipe, or a TCPSocket.

`source`

`Base.redirect_stdin` – Function.

`| redirect_stdin([stream]) -> (rd, wr)`

Like `redirect_stdout`, but for `STDIN`. Note that the order of the return tuple is still `(rd, wr)`, i.e. data to be read from `STDIN` may be written to `wr`.

Note

`stream` must be a TTY, a Pipe, or a TCPSocket.

`source`

`Base.redirect_stdin` – Method.

`| redirect_stdin(f::Function, stream)`

58.1 ~~Run a command f while redirecting `STDIN` to stream.~~ Upon completion, ~~1461~~  
`STDIN` is restored to its prior setting.

Note

`stream` must be a TTY, a Pipe, or a TCPSocket.

`source`

`Base.readchomp` – Function.

`| readchomp(x)`

Read the entirety of `x` as a string and remove a single trailing newline.  
Equivalent to `chomp!(read(x, String))`.

`source`

`Base.truncate` – Function.

`| truncate(file, n)`

Resize the file or buffer given by the first argument to exactly `n` bytes,  
filling previously unallocated space with ‘W0’ if the file or buffer is grown.

`source`

`Base.skipchars` – Function.

`| skipchars(io::IO, predicate; linecomment=nothing)`

Advance the stream `io` such that the next-read character will be the first  
remaining for which `predicate` returns `false`. If the keyword argument  
`linecomment` is specified, all characters from that character until the  
start of the next line are ignored.

```
julia> buf = IOBuffer("    text")
IOBuffer(data=UInt8[...], readable=true, writable=false,
         seekable=true, append=false, size=8, maxsize=Inf, ptr=1,
         mark=-1)
```

```
julia> skipchars(buf, isspace)
IOBuffer(data=UInt8[...], readable=true, writable=false,
         ↵ seekable=true, append=false, size=8, maxsize=Inf, ptr=5,
         ↵ mark=-1)

julia> String(readavailable(buf))
"text"
```

[source](#)

[Base.countlines](#) – Function.

```
| countlines(io::IO, eol::Char='\n')
```

Read `io` until the end of the stream/file and count the number of lines. To specify a file pass the filename as the first argument. EOL markers other than '\n' are supported by passing them as the second argument.

[source](#)

[Base.PipeBuffer](#) – Function.

```
| PipeBuffer(data::Vector{UInt8}=UInt8[], [maxsize::Integer=
          typemax(Int)])
```

An `IOBuffer` that allows reading and performs writes by appending. Seeking and truncating are not supported. See `IOBuffer` for the available constructors. If `data` is given, creates a `PipeBuffer` to operate on a data vector, optionally specifying a size beyond which the underlying `Array` may not be grown.

[source](#)

[Base.readavailable](#) – Function.

```
| readavailable(stream)
```

58.1 ~~Read~~<sup>GENERAL</sup> available data on the stream, blocking the task only if no ~~data~~<sup>1463</sup> is available. The result is a `Vector{UInt8, 1}`.

[source](#)

`Base.IOContext` – Type.

| `IOContext`

`IOContext` provides a mechanism for passing output configuration settings among `show` methods.

In short, it is an immutable dictionary that is a subclass of `I0`. It supports standard dictionary operations such as `getindex`, and can also be used as an I/O stream.

[source](#)

`Base.IOContext` – Method.

| `IOContext(io::I0, KV::Pair...)`

Create an `IOContext` that wraps a given stream, adding the specified `key=>value` pairs to the properties of that stream (note that `io` can itself be an `IOContext`).

`use (key => value) in dict` to see if this particular combination is in the properties set

`use get(dict, key, default)` to retrieve the most recent value for a particular key

The following properties are in common use:

`:compact`: Boolean specifying that small values should be printed more compactly, e.g. that numbers should be printed with fewer digits. This is set when printing array elements.

1464    :**limit**: Boolean specifying that CHAPTERS 58 build/DeANd NETWORK showing ... in place of most elements.

:**displaysize**: A Tuple{Int, Int} giving the size in rows and columns to use for text output. This can be used to override the display size for called functions, but to get the size of the screen use the **display-size** function.

```
julia> function f(io::IO)

    if get(io, :short, false)

        print(io, "short")

    else

        print(io, "loooooong")

    end

end

f (generic function with 1 method)

julia> f(STDOUT)
loooooong
julia> f(IOContext(STDOUT, :short => true))
short
```

source

[Base.IOContext](#) – Method.

```
| IOContext(io::IO, context::IOContext)
```

58.2.Create a `IOContext` that wraps an alternate `I0` but inherits the properties of `context`.<sup>1465</sup>

[source](#)

## 58.2 Text I/O

[Base.show](#) – Method.

`| show(x)`

Write an informative text representation of a value to the current output stream. New types should overload `show(io, x)` where the first argument is a stream. The representation used by `show` generally includes Julia-specific formatting and type information.

[source](#)

[Base.showcompact](#) – Function.

`| showcompact(x)`  
`| showcompact(io::I0, x)`

Show a compact representation of a value to `io`. If `io` is not specified, the default is to print to [STDOUT](#).

This is used for printing array elements without repeating type information (which would be redundant with that printed once for the whole array), and without line breaks inside the representation of an element.

To offer a compact representation different from its standard one, a custom type should test `get(io, :compact, false)` in its normal [show](#) method.

[source](#)

[Base.summary](#) – Function.

```
1466 summary(io::IO, x)
| str = summary(x)
```

## CHAPTER 58. I/O AND NETWORK

Print to a stream `io`, or return a string `str`, giving a brief description of a value. By default returns `string(typeof(x))`, e.g. `Int64`.

For arrays, returns a string of size and type info, e.g. `10-element Array{Int64, 1}`.

```
julia> summary(1)
"Int64"

julia> summary(zeros(2))
"2-element Array{Float64, 1}"

source
```

`Base.print` – Function.

```
| print(io::IO, x)
```

Write to `io` (or to the default output stream `STDOUT` if `io` is not given) a canonical (un-decorated) text representation of a value if there is one, otherwise call `show`. The representation used by `print` includes minimal formatting and tries to avoid Julia-specific details.

Examples

```
julia> print("Hello World!")
Hello World!
julia> io = IOBuffer();

julia> print(io, "Hello World!")

julia> String(take!(io))
"Hello World!"
```

[Base.println](#) – Function.

```
| println(io::IO, xs...)
```

Print (using [print](#)) `xs` followed by a newline. If `io` is not supplied, prints to [STDOUT](#).

Examples

```
julia> println("Hello, world")
Hello, world

julia> io = IOBuffer();

julia> println(io, "Hello, world")

julia> String(take!(io))
"Hello, world\n"
```

[source](#)

[Base.print\\_with\\_color](#) – Function.

```
| print_with_color(color::Union{Symbol, Int}, [io], xs...; bold::
|   Bool = false)
```

Print `xs` in a color specified as a symbol.

`color` may take any of the values `:normal`, `:default`, `:bold`, `:black`, `:blue`, `:cyan`, `:green`, `:light_black`, `:light_blue`, `:light_cyan`, `:light_green`, `:light_magenta`, `:light_red`, `:light_yellow`, `:magenta`, `:nothing`, `:red`, `:underline`, `:white`, or `:yellow` or an integer between 0 and 255 inclusive. Note that not all terminals support 256 colors. If the keyword `bold` is given as `true`, the result will be printed in bold.

[Base.info](#) – Function.

```
| info([io, ] msg..., [prefix="INFO: "])
```

Display an informational message. Argument `msg` is a string describing the information to be displayed. The `prefix` keyword argument can be used to override the default prepending of `msg`.

Examples

```
| julia> info("hello world")
```

```
INFO: hello world
```

```
| julia> info("hello world"; prefix="MY INFO: ")
```

```
MY INFO: hello world
```

See also [logging](#).

[source](#)

[Base.warn](#) – Function.

```
| warn([io, ] msg..., [prefix="WARNING: ", once=false, key=
```

```
nothing, bt=nothing, filename=nothing, lineno::Int=0])
```

Display a warning. Argument `msg` is a string describing the warning to be displayed. Set `once` to true and specify a `key` to only display `msg` the first time `warn` is called. If `bt` is not `nothing` a backtrace is displayed. If `filename` is not `nothing` both it and `lineno` are displayed.

See also [logging](#).

[source](#)

```
| warn(msg)
```

58.2 ~~DISPLAY~~ ~~WARNING~~ ~~tb460~~  
Argument `msg` is a string describing the warning displayed.

Examples

```
julia> warn("Beep Beep")
```

```
WARNING: Beep Beep
```

`source`

[Base.logging](#) – Function.

```
logging(io [, m [, f]][; kind=:all])  
logging(; kind=:all)
```

Stream output of informational, warning, and/or error messages to `io`, overriding what was otherwise specified. Optionally, divert stream only for module `m`, or specifically function `f` within `m`. `kind` can be `:all` (the default), `:info`, `:warn`, or `:error`. See `Base.log_{info,warn,error}_to` for the current set of redirections. Call `logging` with no arguments (or just the `kind`) to reset everything.

`source`

[Base.Printf.@printf](#) – Macro.

```
@printf([io::IOStream], "%Fmt", args...)
```

Print `args` using C `printf` style format specification string, with some caveats: `Inf` and `NaN` are printed consistently as `Inf` and `NaN` for flags `%a`, `%A`, `%e`, `%E`, `%f`, `%F`, `%g`, and `%G`. Furthermore, if a floating point number is equally close to the numeric values of two possible output strings, the output string further away from zero is chosen.

Optionally, an `IOStream` may be passed as the first argument to redirect output.

Examples

1470 **julia>** @printf("%f %F %f %F\n", Inf, Inf, NaN, NaN) CHAPTER 58. I/O AND NETWORK

Inf Inf NaN NaN

**julia>** @printf "%.0f %.1f %f\n" 0.5 0.025 -0.0078125

1 0.0 -0.007813

**source**

**Base.Printf.@sprintf** – Macro.

| @sprintf("%Fmt", args...)

Return **@printf** formatted output as string.

Examples

**julia>** s = @sprintf "this is a %s %15.1f" "test" 34.567;

**julia>** println(s)

this is a test 34.6

**source**

**Base.sprint** – Function.

| sprint(f::Function, args...)

Call the given function with an I/O stream and the supplied extra arguments. Everything written to this I/O stream is returned as a string.

Examples

**julia>** sprint(showcompact, 66.6666)

"66.6667"

**source**

```
| showerror(io, e)
```

Show a descriptive representation of an exception object.

**source**

```
| dump(x; maxdepth=8)
```

Show every part of the representation of a value.

```
julia> struct MyStruct
```

```
    x
```

```
    y
```

```
end
```

```
julia> x = MyStruct(1, (2,3));
```

```
julia> dump(x)
```

```
MyStruct
```

```
  x: Int64 1
```

```
  y: Tuple{Int64,Int64}
```

```
    1: Int64 2
```

```
    2: Int64 3
```

Nested data structures are truncated at `maxdepth`.

```
julia> struct DeeplyNested
```

```
xs::Vector{DeeplyNested}
```

```
    end;

julia> x = DeeplyNested([ ]);

julia> push!(x.xs, x);

julia> dump(x)
DeeplyNested
xs: Array{DeeplyNested}((1,))
 1: DeeplyNested

           xs: Array{DeeplyNested}((1,))

           1: DeeplyNested

           xs: Array{DeeplyNested}((1,))

           1: DeeplyNested

           xs: Array{DeeplyNested}((1,))

           1: DeeplyNested

julia> dump(x, maxdepth=2)
DeeplyNested
xs: Array{DeeplyNested}((1,))
 1: DeeplyNested

source

Base.Meta.@dump – Macro.
```

Show every part of the representation of the given expression. Equivalent to `dump(:(:expr))`.

#### source

[Base.readline](#) – Function.

```
| readline(stream::IO=STDIN; chomp::Bool=true)
| readline(filename::AbstractString; chomp::Bool=true)
```

Read a single line of text from the given I/O stream or file (defaults to `STDIN`). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with '`\n`' or "`\r\n`" or the end of an input stream. When `chomp` is true (as it is by default), these trailing newline characters are removed from the line before it is returned. When `chomp` is false, they are returned as part of the line.

#### source

[Base.readuntil](#) – Function.

```
| readuntil(stream::IO, delim)
| readuntil(filename::AbstractString, delim)
```

Read a string from an I/O stream or a file, up to and including the given delimiter byte. The text is assumed to be encoded in UTF-8.

#### source

[Base.readlines](#) – Function.

```
| readlines(stream::IO=STDIN; chomp::Bool=true)
| readlines(filename::AbstractString; chomp::Bool=true)
```

Read all lines of an I/O stream or a file as a vector of strings. Behavior is equivalent to saving the result of reading `readline` repeatedly with the same arguments and saving the resulting lines as a vector of strings.

`Base.eachline` – Function.

```
| eachline(stream::IO=STDIN; chomp::Bool=true)
| eachline(filename::AbstractString; chomp::Bool=true)
```

Create an iterable `EachLine` object that will yield each line from an I/O stream or a file. Iteration calls `readline` on the stream argument repeatedly with `chomp` passed through, determining whether trailing end-of-line characters are removed. When called with a file name, the file is opened once at the beginning of iteration and closed at the end. If iteration is interrupted, the file will be closed when the `EachLine` object is garbage collected.

`source`

`Base.displaysize` – Function.

```
| displaysize([io::IO]) -> (lines, columns)
```

Return the nominal size of the screen that may be used for rendering output to this `IO` object. If no input is provided, the environment variables `LINES` and `COLUMNS` are read. If those are not set, a default size of (24, 80) is returned.

Examples

```
julia> withenv("LINES" => 30, "COLUMNS" => 100) do
           displaysize()
       end
(30, 100)
```

To get your TTY size,

58.3 | MULTIMEDIA I/O  
julia> dispysize(STDOUT)  
(34, 147)

1475

[source](#)

## 58.3 Multimedia I/O

Just as text output is performed by `print` and user-defined types can indicate their textual representation by overloading `show`, Julia provides a standardized mechanism for rich multimedia output (such as images, formatted text, or even audio and video), consisting of three parts:

A function `display(x)` to request the richest available multimedia display of a Julia object `x` (with a plain-text fallback).

Overloading `show` allows one to indicate arbitrary multimedia representations (keyed by standard MIME types) of user-defined types.

Multimedia-capable display backends may be registered by subclassing a generic `Display` type and pushing them onto a stack of display backends via `pushdisplay`.

The base Julia runtime provides only plain-text display, but richer displays may be enabled by loading external modules or by using graphical Julia environments (such as the IPython-based IJulia notebook).

[Base.Multimedia.display](#) – Function.

display(x)  
display(d::Display, x)  
display(mime, x)  
display(d::Display, mime, x)

Display `x` using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `x`, with plain-text

1476 **STDOUT** output as a fallback. The **display** function (CHAPTER 58, VARIANTS AND NETWORK) displays **x** on the given display **d** only, throwing a **MethodError** if **d** cannot display objects of this type.

In general, you cannot assume that **display** output goes to **STDOUT** (unlike **print(x)** or **show(x)**). For example, **display(x)** may open up a separate window with an image. **display(x)** means "show **x** in the best way you can for the current output device(s)." If you want REPL-like text output that is guaranteed to go to **STDOUT**, use **show(STDOUT, "text/plain", x)** instead.

There are also two variants with a **mime** argument (a MIME type string, such as "**image/png**"), which attempt to display **x** using the requested MIME type only, throwing a **MethodError** if this type is not supported by either the display(s) or by **x**. With these variants, one can also supply the "raw" data in the requested MIME type by passing **x::AbstractString** (for MIME types with text-based storage, such as **text/html** or **application/postscript**) or **x::Vector{UInt8}** (for binary MIME types).

### source

[Base.Multimedia.redisplay](#) – Function.

```
redisplay(x)
redisplay(d::Display, x)
redisplay(mime, x)
redisplay(d::Display, mime, x)
```

By default, the **redisplay** functions simply call **display**. However, some display backends may override **redisplay** to modify an existing display of **x** (if any). Using **redisplay** is also a hint to the backend that **x** may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

### source

```
| displayable(mime) -> Bool  
| displayable(d::Display, mime) -> Bool
```

Returns a boolean value indicating whether the given `mime` type (string) is displayable by any of the displays in the current display stack, or specifically by the display `d` in the second variant.

#### source

```
| show(stream, mime, x)
```

The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O `stream` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type `T`, it is only necessary to define a new `show` method for `T`, via: `show(stream, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `stream`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(stream, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `Display` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments. Therefore,

1478 this case should be handled by defining **CHAPTER 58. I/O AND NETWORK**,  
`x::MyType)` method.

Technically, the **MIME**"`mime`" macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The first argument to `show` can be an **IOContext** specifying output format properties. See **IOContext** for details.

**source**

**Base.Multimedia.mimewritable** – Function.

```
| mimewritable(mime, x)
```

Returns a boolean value indicating whether or not the object `x` can be written as the given `mime` type. (By default, this is determined automatically by the existence of the corresponding `show` method for `typeof(x)`.)

Examples

```
julia> mimewritable(MIME("text/plain"), rand(5))
true

julia> mimewritable(MIME("img/png"), rand(5))
false
```

**source**

**Base.Multimedia.reprmime** – Function.

```
| reprmime(mime, x)
```

Returns an **AbstractString** or **Vector{UInt8}** containing the representation of `x` in the requested `mime` type, as written by `show` (throwing a **MethodError** if no appropriate `show` is available). An **AbstractString**

58.3 is ~~MULTIMEDIA~~ MIME types with textual representations (such as "`text/html`" or "`application/postscript`"), whereas binary data is returned as `Vector{UInt8}`. (The function `istextmime(mime)` returns whether or not Julia treats a given `mime` type as text.)

As a special case, if `x` is an `AbstractString` (for textual MIME types) or a `Vector{UInt8}` (for binary MIME types), the `reprmime` function assumes that `x` is already in the requested `mime` format and simply returns `x`. This special case does not apply to the "`text/plain`" MIME type. This is useful so that raw data can be passed to `display(m::MIME, x)`.

[source](#)

`Base.Multimedia.stringmime` – Function.

| `stringmime(mime, x)`

Returns an `AbstractString` containing the representation of `x` in the requested `mime` type. This is similar to `reprmime` except that binary data is base64-encoded as an ASCII string.

[source](#)

As mentioned above, one can also define new display backends. For example, a module that can display PNG images in a window can register this capability with Julia, so that calling `display(x)` on types with PNG representations will automatically display the image using the module's window.

In order to define a new display backend, one should first create a subtype `D` of the abstract class `Display`. Then, for each MIME type (`mime` string) that can be displayed on `D`, one should define a function `display(d::D, ::MIME"mime", x) = ...` that displays `x` as that MIME type, usually by calling `reprmime(mime, x)`. A `MethodError` should be thrown if `x` cannot be displayed as that MIME type; this is automatic if one calls `reprmime`. Finally, one should define a function `display(d::D, x)` that queries `mimewritable(mime,`

For the `mime` types supported by `D` and `display(x)`, a `MethodError` should be thrown if no supported MIME types are found for `x`. Similarly, some subtypes may wish to override `redisplay(d::D, ...)`. (Again, one should `import Base.display` to add new methods to `display`.) The return values of these functions are up to the implementation (since in some cases it may be useful to return a display “handle” of some type). The display functions for `D` can then be called directly, but they can also be invoked automatically from `display(x)` simply by pushing a new display onto the display-backend stack with:

`Base.Multimedia.pushdisplay` – Function.

```
| pushdisplay(d::Display)
```

Pushes a new display `d` on top of the global display-backend stack. Calling `display(x)` or `display(mime, x)` will display `x` on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a `MethodError`).

`source`

`Base.Multimedia.popdisplay` – Function.

```
| popdisplay()
| popdisplay(d::Display)
```

Pop the topmost backend off of the display-backend stack, or the topmost copy of `d` in the second variant.

`source`

`Base.Multimedia.TextDisplay` – Type.

```
| TextDisplay(io::IO)
```

58.4 Network I/O

`Display <: Display`, which displays any object as text if it is a text/plain MIME type (by default), writing the text representation to the given I/O stream. (This is how objects are printed in the Julia REPL.)

[source](#)

`Base.Multimedia.istextmime` – Function.

`| istextmime(m::MIME)`

Determine whether a MIME type is text data. MIME types are assumed to be binary data except for a set of types known to be text data (possibly Unicode).

Examples

`julia> istextmime(MIME("text/plain"))`

true

`julia> istextmime(MIME("img/png"))`

false

[source](#)

## 58.4 Network I/O

`Base.connect` – Method.

`| connect([host], port::Integer) -> TCPSocket`

Connect to the host `host` on port `port`.

[source](#)

`Base.connect` – Method.

`| connect(path::AbstractString) -> PipeEndpoint`

**source****Base.listen** – Method.

```
| listen([addr, ]port::Integer; backlog::Integer=BACKLOG_DEFAULT)
  -> TCPServer
```

Listen on port on the address specified by **addr**. By default this listens on **localhost** only. To listen on all interfaces pass **IPv4(0)** or **IPv6(0)** as appropriate. **backlog** determines how many connections can be pending (not having called **accept**) before the server will begin to reject them. The default value of **backlog** is 511.

**source****Base.listen** – Method.

```
| listen(path::AbstractString) -> PipeServer
```

Create and listen on a named pipe / UNIX domain socket.

**source****Base.getaddrinfo** – Function.

```
| getalladdrinfo(host::AbstractString, IPAddr=IPv4) -> IPAddr
```

Gets the first IP address of the **host** of the specified **IPAddr** type. Uses the operating system's underlying **getaddrinfo** implementation, which may do a DNS lookup.

**source****Base.getalladdrinfo** – Function.

```
| getalladdrinfo(host::AbstractString) -> Vector{IPAddr}
```

58.4. **getnameinfo** – Gets the IP addresses of the host. Uses the operating system's underlying getaddrinfo implementation, which may do a DNS lookup.

**source**

**Base.getnameinfo** – Function.

```
| getnameinfo(host::IPAddr) -> String
```

Performs a reverse-lookup for IP address to return a hostname and service using the operating system's underlying getnameinfo implementation.

**source**

**Base.getsockname** – Function.

```
| getsockname(sock::Union{TCPServer, TCPSocket}) -> (IPAddr,  
          UInt16)
```

Get the IP address and port that the given socket is bound to.

**source**

**Base.getpeername** – Function.

```
| getpeername(sock::TCPSocket) -> (IPAddr, UInt16)
```

Get the IP address and port of the remote endpoint that the given socket is connected to. Valid only for connected TCP sockets.

**source**

**Base.IPV4** – Type.

```
| IPv4(host::Integer) -> IPv4
```

Returns an IPv4 object from ip address **host** formatted as an **Integer**.

```
| julia> IPv4(3223256218)  
| ip"192.30.252.154"
```

`Base.IPv6` – Type.

```
| IPv6(host::Integer) -> IPv6
```

Returns an IPv6 object from ip address `host` formatted as an `Integer`.

```
| julia> IPv6(3223256218)
| ip"::c01e:fc9a"
```

`source`

`Base.nb_available` – Function.

```
| nb_available(stream)
```

Returns the number of bytes available for reading before a read from this stream or buffer will block.

`source`

`Base.accept` – Function.

```
| accept(server[,client])
```

Accepts a connection on the given server and returns a connection to the client. An uninitialized client stream may be provided, in which case it will be used instead of creating a new stream.

`source`

`Base.listenany` – Function.

```
| listenany([host::IPAddr,] port_hint) -> (UInt16, TCPServer)
```

Create a `TCPServer` on any port, using `hint` as a starting point. Returns a tuple of the actual port that the server was created on and the server itself.

`source`

```
bind(socket::Union{UDPSocket, TCPSocket}, host::IPAddr, port::  
    Integer; ipv6only=false, reuseaddr=false, kws...)
```

Bind `socket` to the given `host:port`. Note that `0.0.0.0` will listen on all devices.

The `ipv6only` parameter disables dual stack mode. If `ipv6only=true`, only an IPv6 stack is created.

If `reuseaddr=true`, multiple threads or processes can bind to the same address without error if they all set `reuseaddr=true`, but only the last to bind will receive any traffic.

#### source

```
bind(chnl::Channel, task::Task)
```

Associates the lifetime of `chnl` with a task. Channel `chnl` is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on `chnl`.

The `chnl` object can be explicitly closed independent of task termination. Terminating tasks have no effect on already closed Channel objects.

When a channel is bound to multiple tasks, the first task to terminate will close the channel. When multiple channels are bound to the same task, termination of the task will close all of the bound channels.

#### Examples

```
julia> c = Channel(0);
```

```
julia> task = @schedule foreach(i->put!(c, i), 1:4);
```

```
julia> bind(c, task);
```

```
julia> for i in c  
           @show i  
           end;  
i = 1  
i = 2  
i = 3  
i = 4  
  
julia> isopen(c)  
false  
  
julia> c = Channel(0);  
  
julia> task = @schedule (put!(c,1);error("foo"));  
  
julia> bind(c,task);  
  
julia> take!(c)  
1  
  
julia> put!(c,1);  
ERROR: foo  
Stacktrace:  
[...]
```

source

Base.send – Function.

| send(socket::UDPSocket, host, port::Integer, msg)

**source**

[Base.recv](#) – Function.

```
| recv(socket::UDPSocket)
```

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

**source**

[Base.recvfrom](#) – Function.

```
| recvfrom(socket::UDPSocket) -> (address, data)
```

Read a UDP packet from the specified socket, returning a tuple of (**address**, **data**), where **address** will be either IPv4 or IPv6 as appropriate.

**source**

[Base.setopt](#) – Function.

```
| setopt(sock::UDPSocket; multicast_loop = nothing, multicast_ttl  
=nothing, enable_broadcast=nothing, ttl=nothing)
```

Set UDP socket options.

**multicast\_loop**: loopback for multicast packets (default: **true**).

**multicast\_ttl**: TTL for multicast packets (default: **nothing**).

**enable\_broadcast**: flag must be set to **true** if socket will be used for broadcast messages, or else the UDP system will return an access error (default: **false**).

**ttl**: Time-to-live of packets sent on the socket (default: **nothing**).

**source**

[Base.ntoh](#) – Function.

CHAPTER 58. I/O AND NETWORK

| `ntoh(x)`

Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

`source`

[Base.hton](#) – Function.

| `hton(x)`

Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

`source`

[Base.ltoh](#) – Function.

| `ltoh(x)`

Converts the endianness of a value from Little-endian to that used by the Host.

`source`

[Base.htol](#) – Function.

| `htol(x)`

Converts the endianness of a value from that used by the Host to Little-endian.

`source`

[Base.ENDIAN\\_BOM](#) – Constant.

| `ENDIAN_BOM`

58.4. The ~~NETWORK~~<sup>1489</sup> byte-order-mark indicates the native byte order of the machine. Little-endian machines will contain the value 0x04030201. Big-endian machines will contain the value 0x01020304.

[source](#)



## Chapter 59

# Punctuation

Extended documentation for mathematical symbols & functions is [here](#).

Symbol	meaning
@m	invoke macro m; followed by space-separated expressions
!	prefix "not" operator
a!()	at the end of a function name, ! indicates that a function modifies its argument(s)
#	begin single line comment
#=	begin multi-line comment (these are nestable)
=#	end multi-line comment
\$	string and expression interpolation
%	remainder operator
^	exponent operator
&	bitwise and
&&	short-circuiting boolean and
	bitwise or
	short-circuiting boolean or
bitwise xor operator	
*	multiply, or matrix multiply
()	the empty tuple
~	bitwise not operator
\	backslash operator
'	complex transpose operator A
a[]	array indexing
[, ]	vertical concatenation
[ ; ]	also vertical concatenation
[ ]	with space-separated expressions, horizontal concatenation
T{ }	parametric type instantiation
;	statement separator
,	separate function arguments or tuple components
?	3-argument conditional operator (conditional ? if_true : if_false)
""	delimit string literals
''	delimit character literals
` `	delimit external process (command) specifications
...	splice arguments into a function call or declare a varargs function or type
.	access named fields in objects/modules, also prefixes elementwise operator/function calls
a:b	range a, a+1, a+2, ..., b
a:s:b	range a, a+s, a+2s, ..., b
:	index an entire dimension (1:end)
::	type annotation, depending on context
:()	quoted expression
:a	symbol a
<:	subtype operator
>:	supertype operator (reverse of subtype operator)
==	egal comparison operator

# Chapter 60

## Sorting and Related Functions

Julia has an extensive, flexible API for sorting and interacting with already-sorted arrays of values. By default, Julia picks reasonable algorithms and sorts in standard ascending order:

```
julia> sort([2,3,1])
3-element Array{Int64,1}:
 1
 2
 3
```

You can easily sort in reverse order as well:

```
julia> sort([2,3,1], rev=true)
3-element Array{Int64,1}:
 3
 2
 1
```

To sort an array in-place, use the "bang" version of the sort function:

```
julia> a = [2,3,1];
```

1494  
**julia>** sort!(a);

## CHAPTER 60. SORTING AND RELATED FUNCTIONS

```
julia> a  
3-element Array{Int64,1}:  
1  
2  
3
```

Instead of directly sorting an array, you can compute a permutation of the array's indices that puts the array into sorted order:

```
julia> v = randn(5)  
5-element Array{Float64,1}:  
0.297288  
0.382396  
-0.597634  
-0.0104452  
-0.839027
```

```
julia> p = sortperm(v)  
5-element Array{Int64,1}:  
5  
3  
4  
1  
2
```

```
julia> v[p]  
5-element Array{Float64,1}:  
-0.839027  
-0.597634  
-0.0104452
```

```
0.297288  
0.382396
```

1495

Arrays can easily be sorted according to an arbitrary transformation of their values:

```
julia> sort(v, by=abs)  
5-element Array{Float64,1}:  
-0.0104452  
0.297288  
0.382396  
-0.597634  
-0.839027
```

Or in reverse order by a transformation:

```
julia> sort(v, by=abs, rev=true)  
5-element Array{Float64,1}:  
-0.839027  
-0.597634  
0.382396  
0.297288  
-0.0104452
```

If needed, the sorting algorithm can be chosen:

```
julia> sort(v, alg=InsertionSort)  
5-element Array{Float64,1}:  
-0.839027  
-0.597634  
-0.0104452  
0.297288  
0.382396
```

1496 CHAPTER 6. SORTING AND RELATED FUNCTIONS  
The sorting and order related functions require a total order on the values to be manipulated. The `isless` function is invoked by default, but the relation can be specified via the `lt` keyword.

## 60.1 Sorting Functions

`Base.sort!` – Function.

```
sort!(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev
      ::Bool=false, order::Ordering=Forward)
```

Sort the vector `v` in place. `QuickSort` is used by default for numeric arrays while `MergeSort` is used for other arrays. You can specify an algorithm to use via the `alg` keyword (see Sorting Algorithms for available algorithms). The `by` keyword lets you provide a function that will be applied to each element before comparison; the `lt` keyword allows providing a custom “less than” function; use `rev=true` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both `by` and `lt` are specified, the `lt` function is applied to the result of the `by` function; `rev=true` reverses whatever ordering specified via the `by` and `lt` keywords.

Examples

```
julia> v = [3, 1, 2]; sort!(v); v
3-element Array{Int64,1}:
 1
 2
 3

julia> v = [3, 1, 2]; sort!(v, rev = true); v
3-element Array{Int64,1}:
 3
```

1

```
julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x ->
    ↪ x[1]); v
```

```
3-element Array{Tuple{Int64, String}, 1}:
```

```
(1, "c")
(2, "b")
(3, "a")
```

```
julia> v = [(1, "c"), (3, "a"), (2, "b")]; sort!(v, by = x ->
    ↪ x[2]); v
```

```
3-element Array{Tuple{Int64, String}, 1}:
```

```
(3, "a")
(2, "b")
(1, "c")
```

### source

[Base.sort](#) – Function.

```
sort(v; alg::Algorithm=defalg(v), lt=isless, by=identity, rev::
    Bool=false, order::Ordering=Forward)
```

Variant of [sort!](#) that returns a sorted copy of v leaving v itself unmodified.

### Examples

```
julia> v = [3, 1, 2];
```

```
julia> sort(v)
```

```
3-element Array{Int64, 1}:
```

```
1
```

1498 2

## CHAPTER 60. SORTING AND RELATED FUNCTIONS

3

**julia>** v

3-element Array{Int64,1}:

3

1

2

**source**

```
sort(A, dim::Integer; alg::Algorithm=DEFAULT_UNSTABLE, lt=
    isless, by=identity, rev::Bool=false, order::Ordering=
    Forward)
```

Sort a multidimensional array A along the given dimension. See [sort!](#) for a description of possible keyword arguments.

Examples

**julia>** A = [4 3; 1 2]

2×2 Array{Int64,2}:

4 3

1 2

**julia>** sort(A, 1)

2×2 Array{Int64,2}:

1 2

4 3

**julia>** sort(A, 2)

2×2 Array{Int64,2}:

3 4

1 2

`Base.sortperm` – Function.

```
sortperm(v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=
    identity, rev::Bool=false, order::Ordering=Forward)
```

Return a permutation vector of indices of `v` that puts it in sorted order. Specify `alg` to choose a particular sorting algorithm (see Sorting Algorithms). `MergeSort` is used by default, and since it is stable, the resulting permutation will be the lexicographically first one that puts the input array into sorted order – i.e. indices of equal elements appear in ascending order. If you choose a non-stable sorting algorithm such as `QuickSort`, a different permutation that puts the array into order may be returned. The order is specified using the same keywords as `sort!`.

See also `sortperm!`.

Examples

```
julia> v = [3, 1, 2];

julia> p = sortperm(v)
3-element Array{Int64,1}:
 2
 3
 1

julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

`source`

## Base.Sort.sortperm! – CHAPTER 60. SORTING AND RELATED FUNCTIONS

```
sortperm!(ix, v; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward, initialized::Bool=false)
```

Like `sortperm`, but accepts a preallocated index vector `ix`. If `initialized` is `false` (the default), `ix` is initialized to contain the values `1:length(v)`.

Examples

```
julia> v = [3, 1, 2]; p = zeros(Int, 3);
```

```
julia> sortperm!(p, v); p
3-element Array{Int64,1}:
 2
 3
 1
```

```
julia> v[p]
3-element Array{Int64,1}:
 1
 2
 3
```

`source`

## Base.Sort.sortrows – Function.

```
sortrows(A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Sort the rows of matrix `A` lexicographically. See `sort!` for a description of possible keyword arguments.

Examples

60.1| **julia>** sortrows([7 3 5; -1 6 4; 9 -2 8])

1501

3×3 Array{Int64,2}:

```
-1 6 4  
7 3 5  
9 -2 8
```

**julia>** sortrows([7 3 5; -1 6 4; 9 -2 8],

↪ lt=(x,y)->isless(x[2],y[2]))

3×3 Array{Int64,2}:

```
9 -2 8  
7 3 5  
-1 6 4
```

**julia>** sortrows([7 3 5; -1 6 4; 9 -2 8], rev=true)

3×3 Array{Int64,2}:

```
9 -2 8  
7 3 5  
-1 6 4
```

**source**

[Base.Sort.sortcols](#) – Function.

```
sortcols(A; alg::Algorithm=DEFAULT_UNSTABLE, lt=isless, by=identity, rev::Bool=false, order::Ordering=Forward)
```

Sort the columns of matrix A lexicographically. See [sort!](#) for a description of possible keyword arguments.

Examples

**julia>** sortcols([7 3 5; 6 -1 -4; 9 -2 8])

3×3 Array{Int64,2}:

```
3 5 7
```

```
1502 -1 -4 6  
      -2  8 9
```

## CHAPTER 60. SORTING AND RELATED FUNCTIONS

```
julia> sortcols([7 3 5; 6 -1 -4; 9 -2 8], alg=InsertionSort,  
→   lt=(x,y)->isless(x[2],y[2]))  
3×3 Array{Int64,2}:  
 5  3  7  
-4 -1  6  
 8 -2  9  
  
julia> sortcols([7 3 5; 6 -1 -4; 9 -2 8], rev=true)  
3×3 Array{Int64,2}:  
 7  5  3  
 6 -4 -1  
 9  8 -2
```

`source`

## 60.2 Order-Related Functions

`Base.issorted` – Function.

```
issorted(v, lt=isless, by=identity, rev:Bool=false, order::  
Ordering=Forward)
```

Test whether a vector is in sorted order. The `lt`, `by` and `rev` keywords modify what order is considered to be sorted just as they do for `sort`.

Examples

```
julia> issorted([1, 2, 3])  
true  
  
julia> issorted([(1, "b"), (2, "a")], by = x -> x[1])
```

```
julia> issorted([(1, "b"), (2, "a")], by = x -> x[2])
false
```

```
julia> issorted([(1, "b"), (2, "a")], by = x -> x[2],
    ↪ rev=true)
true
```

source

`Base.Sort.searchsorted` – Function.

```
searchsorted(a, x, [by=<transform>,] [lt=<comparison>,] [rev=
    false])
```

Return the range of indices of `a` which compare as equal to `x` (using binary search) according to the order specified by the `by`, `lt` and `rev` keywords, assuming that `a` is already sorted in that order. Return an empty range located at the insertion point if `a` does not contain values equal to `x`.

Examples

```
julia> a = [4, 3, 2, 1]
4-element Array{Int64,1}:
 4
 3
 2
 1
```

```
julia> searchsorted(a, 4)
5:4
```

```
julia> searchsorted(a, 4, rev=true)
1:1
```

`Base.Sort.searchsortedfirst` – Function.

```
searchsortedfirst(a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Return the index of the first value in `a` greater than or equal to `x`, according to the specified order. Return `length(a) + 1` if `x` is greater than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedfirst([1, 2, 4, 5, 14], 4)
```

```
3
```

```
julia> searchsortedfirst([1, 2, 4, 5, 14], 4, rev=true)
```

```
1
```

```
julia> searchsortedfirst([1, 2, 4, 5, 14], 15)
```

```
6
```

`source`

`Base.Sort.searchsortedlast` – Function.

```
searchsortedlast(a, x, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Return the index of the last value in `a` less than or equal to `x`, according to the specified order. Return `0` if `x` is less than all values in `a`. `a` is assumed to be sorted.

Examples

```
julia> searchsortedlast([1, 2, 4, 5, 14], 4)
```

```
3
```

```
julia> searchsortedlast([1, 2, 4, 5, 14], 4, rev=true)  
5
```

```
julia> searchsortedlast([1, 2, 4, 5, 14], -1)  
0
```

`source`

[Base.Sort.partialsort!](#) – Function.

```
partialsort!(v, k, [by=<transform>,] [lt=<comparison>,] [rev=false])
```

Partially sort the vector `v` in place, according to the order specified by `by`, `lt` and `rev` so that the value at index `k` (or range of adjacent values if `k` is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If `k` is a single index, that value is returned; if `k` is a range, an array of values at those indices is returned. Note that `partialsort!` does not fully sort the input array.

Examples

```
julia> a = [1, 2, 4, 3, 4]  
5-element Array{Int64,1}:  
1  
2  
4  
3  
4
```

```
julia> partialsort!(a, 4)  
4
```

1506 **julia>** a

## CHAPTER 60. SORTING AND RELATED FUNCTIONS

```
5-element Array{Int64,1}:
1
2
3
4
4
```

```
julia> a = [1, 2, 4, 3, 4]
```

```
5-element Array{Int64,1}:
1
2
4
3
4
```

```
julia> partialsort!(a, 4, rev=true)
```

```
2
```

```
julia> a
```

```
5-element Array{Int64,1}:
4
4
3
2
1
```

**source**

[Base.Sort.partialsort](#) – Function.

```
| partialsort(v, k, [by=<transform>,] [lt=<comparison>,] [rev=
|   false])
```

60.3.1507 **SORTING ALGORITHMS** which copies  $v$  before partially sorting it, thereby returning the same thing as `partialsort!` but leaving  $v$  unmodified.

[source](#)

`Base.Sort.partialsortperm` – Function.

```
partialsortperm(v, k, [alg=<algorithm>], [by=<transform>], [lt
    =<comparison>], [rev=false])
```

Return a partial permutation of the vector  $v$ , according to the order specified by `by`, `lt` and `rev`, so that  $v[\text{output}]$  returns the first  $k$  (or range of adjacent values if  $k$  is a range) values of a fully sorted version of  $v$ . If  $k$  is a single index, the index in  $v$  of the value which would be sorted at position  $k$  is returned; if  $k$  is a range, an array with the indices in  $v$  of the values which would be sorted in these positions is returned.

Note that this is equivalent to, but more efficient than, calling `sortperm(...)[k]`.

[source](#)

`Base.Sort.partialsortperm!` – Function.

```
partialsortperm!(ix, v, k, [alg=<algorithm>], [by=<transform>],
    [lt=<comparison>], [rev=false], [initialized=false])
```

Like `partialsortperm`, but accepts a preallocated index vector `ix`. If `initialized` is `false` (the default), `ix` is initialized to contain the values `1:length(ix)`.

[source](#)

## 60.3 Sorting Algorithms

There are currently four sorting algorithms available in base Julia:

`InsertionSort`

**PartialQuickSort(k)**

**MergeSort**

**InsertionSort** is an  $O(n^2)$  stable sorting algorithm. It is efficient for very small  $n$ , and is used internally by **QuickSort**.

**QuickSort** is an  $O(n \log n)$  sorting algorithm which is in-place, very fast, but not stable – i.e. elements which are considered equal will not remain in the same order in which they originally appeared in the array to be sorted. **QuickSort** is the default algorithm for numeric values, including integers and floats.

**PartialQuickSort(k)** is similar to **QuickSort**, but the output array is only sorted up to index  $k$  if  $k$  is an integer, or in the range of  $k$  if  $k$  is an **Ordinal-Range**. For example:

```
x = rand(1:500, 100)
k = 50
k2 = 50:100
s = sort(x; alg=QuickSort)
ps = sort(x; alg=PartialQuickSort(k))
qs = sort(x; alg=PartialQuickSort(k2))
map(issorted, (s, ps, qs))           # => (true, false, false)
map(x->issorted(x[1:k]), (s, ps, qs)) # => (true, true, false)
map(x->issorted(x[k2]), (s, ps, qs)) # => (true, false, true)
s[1:k] == ps[1:k]                   # => true
s[k2] == qs[k2]                     # => true
```

**MergeSort** is an  $O(n \log n)$  stable sorting algorithm but is not in-place – it requires a temporary array of half the size of the input array – and is typically

~~60t3qu\$@RTN6astAb\$QRITH18rt~~. It is the default algorithm for non-~~num~~<sup>1500</sup> data.

The default sorting algorithms are chosen on the basis that they are fast and stable, or appear to be so. For numeric types indeed, **QuickSort** is selected as it is faster and indistinguishable in this case from a stable sort (unless the array records its mutations in some way). The stability property comes at a non-negligible cost, so if you don't need it, you may want to explicitly specify your preferred algorithm, e.g. `sort!(v, alg=QuickSort)`.

The mechanism by which Julia picks default sorting algorithms is implemented via the `Base.Sort.defalg` function. It allows a particular algorithm to be registered as the default in all sorting functions for specific arrays. For example, here are the two default methods from `sort.jl`:

```
| defalg(v::AbstractArray) = MergeSort  
| defalg(v::AbstractArray{<:Number}) = QuickSort
```

As for numeric arrays, choosing a non-stable default algorithm for array types for which the notion of a stable sort is meaningless (i.e. when two values comparing equal can not be distinguished) may make sense.



# Chapter 61

## Package Manager Functions

All package manager functions are defined in the `Pkg` module. None of the `Pkg` module's functions are exported; to use them, you'll need to prefix each function call with an explicit `Pkg.`, e.g. `Pkg.status()` or `Pkg.dir()`.

Functions for package development (e.g. `tag`, `publish`, etc.) have been moved to the `PkgDev` package. See `PkgDev README` for the documentation of those functions.

`Base.Pkg.dir` – Function.

```
| dir() -> AbstractString
```

Returns the absolute path of the package directory. This defaults to `joinpath(homedir(),".julia","v$(VERSION.major).$(VERSION.minor)")` on all platforms (i.e. `~/.julia/v0.7` in UNIX shell syntax). If the `JULIA_PKGDIR` environment variable is set, then that path is used in the returned value as `joinpath(ENV["JULIA_PKGDIR"],"v$(VERSION.major).$(VERSION.minor)")`. If `JULIA_PKGDIR` is a relative path, it is interpreted relative to whatever the current working directory is.

`source`

```
| dir(names...) -> AbstractString
```

151 Equivalent to `normpath(Pkg.dir(pkg), joinpath(pkg, "src"))`.  
CHAPTER(61, PACKAGE MANAGER FUNCTIONS) components to the package directory and normalizes the resulting path.  
In particular, `Pkg.dir(pkg)` returns the path to the package `pkg`.

[source](#)

`Base.Pkg.init` – Function.

```
| init(meta::AbstractString=DEFAULT_META, branch::AbstractString=
| META_BRANCH)
```

Initialize `Pkg.dir()` as a package directory. This will be done automatically when the `JULIA_PKGDIR` is not set and `Pkg.dir()` uses its default value. As part of this process, clones a local METADATA git repository from the site and branch specified by its arguments, which are typically not provided. Explicit (non-default) arguments can be used to support a custom METADATA setup.

[source](#)

`Base.Pkg.resolve` – Function.

```
| resolve()
```

Determines an optimal, consistent set of package versions to install or upgrade to. The optimal set of package versions is based on the contents of `Pkg.dir("REQUIRE")` and the state of installed packages in `Pkg.dir()`. Packages that are no longer required are moved into `Pkg.dir(".trash")`.

[source](#)

`Base.Pkg.edit` – Function.

```
| edit()
```

Opens `Pkg.dir("REQUIRE")` in the editor specified by the `VISUAL` or `EDITOR` environment variables; when the editor command returns, it runs

`Pkg.resolve()` to determine and install a new optimal set of installed package versions.

`source`

`Base.Pkg.add` – Function.

| `add(pkg, vers...)`

Add a requirement entry for `pkg` to `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`. If `vers` are given, they must be `VersionNumber` objects and they specify acceptable version intervals for `pkg`.

`source`

`Base.Pkg.rm` – Function.

| `rm(pkg)`

Remove all requirement entries for `pkg` from `Pkg.dir("REQUIRE")` and call `Pkg.resolve()`.

`source`

`Base.Pkg.clone` – Function.

| `clone(pkg)`

If `pkg` has a URL registered in `Pkg.dir("METADATA")`, clone it from that URL on the default branch. The package does not need to have any registered versions.

`source`

| `clone(url, [pkg])`

Clone a package directly from the git URL `url`. The package does not need to be registered in `Pkg.dir("METADATA")`. The package repo is cloned by the name `pkg` if provided; if not provided, `pkg` is determined automatically from `url`.

[Base.Pkg.setprotocol!](#) – Function.

```
| setprotocol!(proto)
```

Set the protocol used to access GitHub-hosted packages. Defaults to 'https', with a blank `proto` delegating the choice to the package developer.

**source**

[Base.Pkg.available](#) – Function.

```
| available() -> Vector{String}
```

Returns the names of available packages.

**source**

```
| available(pkg) -> Vector{VersionNumber}
```

Returns the version numbers available for package `pkg`.

**source**

[Base.Pkg.installed](#) – Function.

```
| installed() -> Dict{String, VersionNumber}
```

Returns a dictionary mapping installed package names to the installed version number of each package.

**source**

```
| installed(pkg) -> Void | VersionNumber
```

If `pkg` is installed, return the installed version number. If `pkg` is registered, but not installed, return `nothing`.

**source**

```
|status()
```

Prints out a summary of what packages are installed and what version and state they're in.

**source**

```
|status(pkg)
```

Prints out a summary of what version and state `pkg`, specifically, is in.

**source**

```
|update(pkgs...)
```

Update the metadata repo – kept in `Pkg.dir("METADATA")` – then update any fixed packages that can safely be pulled from their origin; then call `Pkg.resolve()` to determine a new optimal set of packages versions.

Without arguments, updates all installed packages. When one or more package names are provided as arguments, only those packages and their dependencies are updated.

**source**

```
|checkout(pkg, [branch="master"]; merge=true, pull=true)
```

Checkout the `Pkg.dir(pkg)` repo to the branch `branch`. Defaults to checking out the "master" branch. To go back to using the newest compatible released version, use `Pkg.free(pkg)`. Changes are merged (fast-forward only) if the keyword argument `merge == true`, and the latest version is pulled from the upstream repo if `pull == true`.

**Base.Pkg.pin** – Function.

```
| pin(pkg)
```

Pin **pkg** at the current version. To go back to using the newest compatible released version, use **Pkg.free(pkg)**

**source**

```
| pin(pkg, version)
```

Pin **pkg** at registered version **version**.

**source**

**Base.Pkg.free** – Function.

```
| free(pkg)
```

Free the package **pkg** to be managed by the package manager again. It calls **Pkg.resolve()** to determine optimal package versions after. This is an inverse for both **Pkg.checkout** and **Pkg.pin**.

You can also supply an iterable collection of package names, e.g., **Pkg.free(("Pkg1", "Pkg2"))** to free multiple packages at once.

**source**

**Base.Pkg.build** – Function.

```
| build()
```

Run the build scripts for all installed packages in depth-first recursive order.

**source**

```
| build(pkgs...)
```

Run the build script in `deps/build.jl` for each package in `pkgs` and all of their dependencies in depth-first recursive order. This is called automatically by `Pkg.resolve()` on all installed or updated packages.

`source`

`Base.Pkg.test` – Function.

```
| test(; coverage=false)
```

Run the tests for all installed packages ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`. Coverage statistics for the packages may be generated by passing `coverage=true`. The default behavior is not to run coverage.

`source`

```
| test(pkgs...; coverage=false)
```

Run the tests for each package in `pkgs` ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its `test/runtests.jl` file and test dependencies are specified in `test/REQUIRE`. Coverage statistics for the packages may be generated by passing `coverage=true`. The default behavior is not to run coverage.

`source`

`Base.Pkg.dependents` – Function.

```
| dependents(pkg)
```

List the packages that have `pkg` as a dependency.

`source`



# Chapter 62

## Dates and Time

Functionality to handle time and dates are defined in the standard library module `Dates`. You'll need to import the module using `import Dates` and prefix each function call with an explicit `Dates.`, e.g. `Dates.dayofweek(dt)`. Alternatively, You can write `using Dates` to bring all exported functions into `Main` to be used without the `Dates.` prefix.

### 62.1 Dates and Time Types

`Dates.Period` – Type.

<code>Period</code>
<code>Year</code>
<code>Month</code>
<code>Week</code>
<code>Day</code>
<code>Hour</code>
<code>Minute</code>
<code>Second</code>
<code>Millisecond</code>
<code>Microsecond</code>
<code>Nanosecond</code>

[source](#)[Dates.CompoundPeriod](#) – Type.[CompoundPeriod](#)

A **CompoundPeriod** is useful for expressing time periods that are not a fixed multiple of smaller periods. For example, "a year and a day" is not a fixed number of days, but can be expressed using a **CompoundPeriod**. In fact, a **CompoundPeriod** is automatically generated by addition of different period types, e.g. `Year(1) + Day(1)` produces a **CompoundPeriod** result.

[source](#)[Dates.Instant](#) – Type.[Instant](#)

**Instant** types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

[source](#)[Dates.UTInstant](#) – Type.[UTInstant{T}](#)

The **UTInstant** represents a machine timeline based on UT time (1 day = one revolution of the earth). The **T** is a **Period** parameter that indicates the resolution or precision of the instant.

[source](#)[Dates.TimeType](#) – Type.[TimeType](#)

62.2.1 **DATES FUNCTIONS** Instant machine instances to provide human representations of the machine instant. Time, DateTime and Date are subtypes of TimeType.<sup>1521</sup>

**source**

[Dates.DateTime](#) – Type.

| **DateTime**

DateTime wraps a UTInstant{Millisecond} and interprets it according to the proleptic Gregorian calendar.

**source**

[Dates.Date](#) – Type.

| **Date**

Date wraps a UTInstant{Day} and interprets it according to the proleptic Gregorian calendar.

**source**

[Dates.Time](#) – Type.

| **Time**

Time wraps a Nanosecond and represents a specific moment in a 24-hour day.

**source**

## 62.2 Dates Functions

[Dates.DateTime](#) – Method.

| **DateTime(y, [m, d, h, mi, s, ms]) -> DateTime**

[Int64](#).

[source](#)

`Dates.DateTime` – Method.

```
| DateTime(periods::Period...) -> DateTime
```

Construct a `DateTime` type by `Period` type parts. Arguments may be in any order. `DateTime` parts not provided will default to the value of `Dates.default(period)`.

[source](#)

`Dates.DateTime` – Method.

```
| DateTime(f::Function, y[, m, d, h, mi, s]; step=Day(1), limit  
|   =10000) -> DateTime
```

Create a `DateTime` through the adjuster API. The starting point will be constructed from the provided `y, m, d...` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied).

Examples

```
julia> DateTime(dt -> Dates.second(dt) == 40, 2010, 10, 20,  
|   → 10; step = Dates.Second(1))  
2010-10-20T10:00:40
```

```
julia> DateTime(dt -> Dates.hour(dt) == 20, 2010, 10, 20, 10;  
|   → step = Dates.Hour(1), limit = 5)  
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
```

```
[ ... ]
```

[source](#)

[Dates.DateTime](#) – Method.

```
| DateTime(dt::Date) -> DateTime
```

Converts a **Date** to a **DateTime**. The hour, minute, second, and millisecond parts of the new **DateTime** are assumed to be zero.

[source](#)

[Dates.DateTime](#) – Method.

```
| DateTime(dt::AbstractString, format::AbstractString; locale="english") -> DateTime
```

Construct a **DateTime** by parsing the **dt** date time string following the pattern given in the **format** string.

This method creates a **DateFormat** object each time it is called. If you are parsing many date time strings of the same format, consider creating a **DateFormat** object once and using that as the second argument instead.

[source](#)

[Dates.format](#) – Function.

```
| format(io::IO, tok::AbstractDateToken, dt::TimeType, locale)
```

Format the **tok** token from **dt** and write it to **io**. The formatting can be based on **locale**.

All subtypes of **AbstractDateToken** must define this method in order to be able to print a Date / DateTime object according to a **DateFormat** containing that token.

[source](#)

```
| DateFormat(format::AbstractString, locale="english") ->
|   DateFormat
```

Construct a date formatting object that can be used for parsing date strings or formatting a date object as a string. The following character codes can be used to construct the `format` string:

Code	Matches	Comment
y	1996, 96	Returns year of 1996, 0096
Y	1996, 96	Returns year of 1996, 0096. Equivalent to y
m	1, 01	Matches 1 or 2-digit months
u	Jan	Matches abbreviated months according to the <code>locale</code> keyword
U	January	Matches full month names according to the <code>locale</code> keyword
d	1, 01	Matches 1 or 2-digit days
H	00	Matches hours
M	00	Matches minutes
S	00	Matches seconds
s	.500	Matches milliseconds
e	Mon, Tues	Matches abbreviated days of the week
E	Monday	Matches full name days of the week
yyyym- mdd	19960101	Matches fixed-width year, month, and day

Characters not listed above are normally treated as delimiters between date and time slots. For example a `dt` string of "1996-01-15T00:00:00.0" would have a `format` string like "y-m-dTH:M:S.s". If you need to use a code character as a delimiter you can escape it using backslash. The date "1995y01m" would have the format "y\Wym\Wm".

Creating a `DateFormat` object is expensive. Whenever possible, create it once and use it many times or try the `dateformat""` string macro. Using this macro creates the `DateFormat` object once at macro expansion time and reuses it later. see [@dateformat\\_str](#).

62.2 See [Date Functions](#) for how to use a `DateFormat` object to <sup>1525</sup>  
and write Date strings respectively.

[source](#)

`Dates.@dateformat_str` – Macro.

| `dateformat"Y-m-d H:M:S"`

Create a `DateFormat` object. Similar to `DateFormat("Y-m-d H:M:S")`  
but creates the `DateFormat` object once during macro expansion.

See [DateFormat](#) for details about format specifiers.

[source](#)

`Dates.DateTime` – Method.

| `DateTime(dt::AbstractString, df::DateFormat) -> DateTime`

Construct a `DateTime` by parsing the `dt` date time string following the  
pattern given in the `DateFormat` object. Similar to `DateTime(::Ab-`  
`stractString, ::AbstractString)` but more efficient when repeat-  
edly parsing similarly formatted date time strings with a pre-created  
`DateFormat` object.

[source](#)

`Dates.Date` – Method.

| `Date(y, [m, d]) -> Date`

Construct a `Date` type by parts. Arguments must be convertible to `Int64`.

[source](#)

`Dates.Date` – Method.

| `Date(period::Period...) -> Date`

1526 Construct a Date type by Period type parts in order. Date parts not provided will default to the value of Dates.default(period).

[source](#)

[Dates.Date](#) – Method.

```
| Date(f::Function, y[, m, d]; step=Day(1), limit=10000) -> Date
```

Create a Date through the adjuster API. The starting point will be constructed from the provided y, m, d arguments, and will be adjusted until f::Function returns true. The step size in adjusting can be provided manually through the step keyword. limit provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that f::Function is never satisfied).

Examples

```
| julia> Date(date -> Dates.week(date) == 20, 2010, 01, 01)  
2010-05-17
```

```
| julia> Date(date -> Dates.year(date) == 2010, 2000, 01, 01)  
2010-01-01
```

```
| julia> Date(date -> Dates.month(date) == 10, 2000, 01, 01;  
|   ↳ limit = 5)  
ERROR: ArgumentError: Adjustment limit reached: 5 iterations  
Stacktrace:  
[...]
```

[source](#)

[Dates.Date](#) – Method.

```
| Date(dt::DateTime) -> Date
```

62.2. **DATES FUNCTIONS** to a Date. The hour, minute, second, and millisecond parts of the **DateTime** are truncated, so only the year, month and day parts are used in construction.

[source](#)

**Dates.Date** – Method.

```
| Date(d::AbstractString, format::AbstractString; locale="english") -> Date
```

Construct a **Date** by parsing the **d** date string following the pattern given in the **format** string.

This method creates a **DateFormat** object each time it is called. If you are parsing many date strings of the same format, consider creating a **DateFormat** object once and using that as the second argument instead.

[source](#)

**Dates.Date** – Method.

```
| Date(d::AbstractString, df::DateFormat) -> Date
```

Parse a date from a date string **d** using a **DateFormat** object **df**.

[source](#)

**Dates.Time** – Method.

```
| Time(h, [mi, s, ms, us, ns]) -> Time
```

Construct a **Time** type by parts. Arguments must be convertible to **Int64**.

[source](#)

**Dates.Time** – Method.

```
| Time(period::TimePeriod...) -> Time
```

1528 Construct a `Time` type by `Period` type parts in `Dates` and `Time` order. `Time` parts not provided will default to the value of `Dates.default(period)`.

[source](#)

`Dates.Time` – Method.

```
Time(f::Function, h, mi=0; step::Period=Second(1), limit::Int  
      =10000)  
Time(f::Function, h, mi, s; step::Period=Millisecond(1), limit  
      ::Int=10000)  
Time(f::Function, h, mi, s, ms; step::Period=Microsecond(1),  
      limit::Int=10000)  
Time(f::Function, h, mi, s, ms, us; step::Period=Nanosecond(1),  
      limit::Int=10000)
```

Create a `Time` through the adjuster API. The starting point will be constructed from the provided `h`, `mi`, `s`, `ms`, `us` arguments, and will be adjusted until `f::Function` returns `true`. The step size in adjusting can be provided manually through the `step` keyword. `limit` provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `f::Function` is never satisfied). Note that the default step will adjust to allow for greater precision for the given arguments; i.e. if hour, minute, and second arguments are provided, the default step will be `Millisecond(1)` instead of `Second(1)`.

Examples

```
julia> Dates.Time(t -> Dates.minute(t) == 30, 20)  
20:30:00
```

```
julia> Dates.Time(t -> Dates.minute(t) == 0, 20)  
20:00:00
```

```
julia> Dates.Time(t -> Dates.hour(t) == 10, 3; limit = 5)
ERROR: ArgumentError: Adjustment limit reached: 5 iterations
Stacktrace:
[...]
```

[source](#)

[Dates.Time](#) – Method.

```
| Time(dt::DateTime) -> Time
```

Converts a **DateTime** to a **Time**. The hour, minute, second, and millisecond parts of the **DateTime** are used to create the new **Time**. Microsecond and nanoseconds are zero by default.

[source](#)

[Dates.now](#) – Method.

```
| now() -> DateTime
```

Returns a **DateTime** corresponding to the user's system time including the system timezone locale.

[source](#)

[Dates.now](#) – Method.

```
| now(::Type{UTC}) -> DateTime
```

Returns a **DateTime** corresponding to the user's system time as UTC/GMT.

[source](#)

[Base.eps](#) – Function.

```
| eps(::DateTime) -> Millisecond
| eps(::Date) -> Day
| eps(::Time) -> Nanosecond
```

1530 Returns `Millisecond(1)` for `DateTime` values, `Day(1)` for `Date` values, and `Nanosecond(1)` for `Time` values.

`source`

## Accessor Functions

`Dates.year` – Function.

```
| year(dt::TimeType) -> Int64
```

The year of a `Date` or `DateTime` as an `Int64`.

`source`

`Dates.month` – Function.

```
| month(dt::TimeType) -> Int64
```

The month of a `Date` or `DateTime` as an `Int64`.

`source`

`Dates.week` – Function.

```
| week(dt::TimeType) -> Int64
```

Return the `ISO week date` of a `Date` or `DateTime` as an `Int64`. Note that the first week of a year is the week that contains the first Thursday of the year, which can result in dates prior to January 4th being in the last week of the previous year. For example, `week(Date(2005, 1, 1))` is the 53rd week of 2004.

## Examples

```
julia> Dates.week(Date(1989, 6, 22))
```

```
25
```

```
julia> Dates.week(Date(2005, 1, 1))
```

```
julia> Dates.week(Date(2004, 12, 31))
```

```
53
```

`source`

`Dates.day` – Function.

```
| day(dt::TimeType) -> Int64
```

The day of month of a `Date` or `DateTime` as an `Int64`.

`source`

`Dates.hour` – Function.

```
| hour(dt::DateTime) -> Int64
```

The hour of day of a `DateTime` as an `Int64`.

`source`

```
| hour(t::Time) -> Int64
```

The hour of a `Time` as an `Int64`.

`source`

`Dates.minute` – Function.

```
| minute(dt::DateTime) -> Int64
```

The minute of a `DateTime` as an `Int64`.

`source`

```
| minute(t::Time) -> Int64
```

The minute of a `Time` as an `Int64`.

`source`

[D532](#).`second` – Function.

CHAPTER 62. DATES AND TIME

```
| second(dt::DateTime) -> Int64
```

The second of a `DateTime` as an [Int64](#).

[source](#)

```
| second(t::Time) -> Int64
```

The second of a `Time` as an [Int64](#).

[source](#)

[Dates](#).`millisecond` – Function.

```
| millisecond(dt::DateTime) -> Int64
```

The millisecond of a `DateTime` as an [Int64](#).

[source](#)

```
| millisecond(t::Time) -> Int64
```

The millisecond of a `Time` as an [Int64](#).

[source](#)

[Dates](#).`microsecond` – Function.

```
| microsecond(t::Time) -> Int64
```

The microsecond of a `Time` as an [Int64](#).

[source](#)

[Dates](#).`nanosecond` – Function.

```
| nanosecond(t::Time) -> Int64
```

The nanosecond of a `Time` as an [Int64](#).

[source](#)

| `Year(v)`

Construct a `Year` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`source`

`Dates.Month` – Method.

| `Month(v)`

Construct a `Month` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`source`

`Dates.Week` – Method.

| `Week(v)`

Construct a `Week` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`source`

`Dates.Day` – Method.

| `Day(v)`

Construct a `Day` object with the given `v` value. Input must be losslessly convertible to an `Int64`.

`source`

`Dates.Hour` – Method.

| `Hour(dt::DateTime) -> Hour`

The hour part of a `DateTime` as a `Hour`.

`source`

[Dates.Minute](#) – Method.

CHAPTER 62. DATES AND TIME

```
| Minute(dt::DateTime) -> Minute
```

The minute part of a DateTime as a **Minute**.

[source](#)

[Dates.Second](#) – Method.

```
| Second(dt::DateTime) -> Second
```

The second part of a DateTime as a **Second**.

[source](#)

[Dates.Millisecond](#) – Method.

```
| Millisecond(dt::DateTime) -> Millisecond
```

The millisecond part of a DateTime as a **Millisecond**.

[source](#)

[Dates.Microsecond](#) – Method.

```
| Microsecond(dt::Time) -> Microsecond
```

The microsecond part of a Time as a **Microsecond**.

[source](#)

[Dates.Nanosecond](#) – Method.

```
| Nanosecond(dt::Time) -> Nanosecond
```

The nanosecond part of a Time as a **Nanosecond**.

[source](#)

[Dates.yeardown](#) – Function.

```
| yeardown(dt::TimeType) -> (Int64, Int64)
```

## 62.2 Simpler DATE FUNCTIONS

[source](#)

[Dates.monthday](#) – Function.

```
| monthday(dt::TimeType) -> (Int64, Int64)
```

Simultaneously return the month and day parts of a Date or DateTime.

[source](#)

[Dates.yearday](#) – Function.

```
| yearday(dt::TimeType) -> (Int64, Int64, Int64)
```

Simultaneously return the year, month and day parts of a Date or DateTime.

[source](#)

## Query Functions

[Dates.dayname](#) – Function.

```
| dayname(dt::TimeType; locale="english") -> String
```

Return the full day name corresponding to the day of the week of the Date or DateTime in the given locale.

Examples

```
julia> Dates.dayname(Date("2000-01-01"))
```

```
"Saturday"
```

[source](#)

[Dates.dayabbr](#) – Function.

```
| dayabbr(dt::TimeType; locale="english") -> String
```

1536 Return the abbreviated name corresponding to the day of the week of the Date or DateTime in the given locale.

Examples

```
julia> Dates.dayabbr(Date("2000-01-01"))
"Sat"
```

source

[Dates.dayofweek](#) – Function.

```
| dayofweek(dt::TimeType) -> Int64
```

Returns the day of the week as an [Int64](#) with 1 = Monday, 2 = Tuesday, etc..

Examples

```
julia> Dates.dayofweek(Date("2000-01-01"))
6
```

source

[Dates.dayofmonth](#) – Function.

```
| dayofmonth(dt::TimeType) -> Int64
```

The day of month of a Date or DateTime as an [Int64](#).

source

[Dates.dayofweekofmonth](#) – Function.

```
| dayofweekofmonth(dt::TimeType) -> Int
```

For the day of week of dt, returns which number it is in dt's month. So if the day of the week of dt is Monday, then 1 = First Monday of the month, 2 = Second Monday of the month, etc. In the range 1:5.

```
"jldoctest julia> Dates.dayofweekofmonth(Date("2000-02-01")) 1  
julia> Dates.dayofweekofmonth(Date("2000-02-08")) 2  
julia> Dates.dayofweekofmonth(Date("2000-02-15")) 3 ""
```

**source**

**Dates.daysofweekinmonth** – Function.

```
| daysofweekinmonth(dt::TimeType) -> Int
```

For the day of week of **dt**, returns the total number of that day of the week in **dt**'s month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including **dayofweekofmonth(dt) == daysofweekinmonth(dt)** in the adjuster function.

Examples

```
| julia> Dates.daysofweekinmonth(Date("2005-01-01"))
```

```
5
```

```
| julia> Dates.daysofweekinmonth(Date("2005-01-04"))
```

```
4
```

**source**

**Dates.monthname** – Function.

```
| monthname(dt::TimeType; locale="english") -> String
```

Return the full name of the month of the **Date** or **DateTime** in the given **locale**.

Examples

```
| julia> Dates.monthname(Date("2005-01-04"))
```

```
"January"
```

[Dates.monthabbr](#) – Function.

```
| monthabbr(dt::TimeType; locale="english") -> String
```

Return the abbreviated month name of the `Date` or `DateTime` in the given `locale`.

Examples

```
| julia> Dates.monthabbr(Date("2005-01-04"))
"Jan"
```

[source](#)

[Dates.daysinmonth](#) – Function.

```
| daysinmonth(dt::TimeType) -> Int
```

Returns the number of days in the month of `dt`. Value will be 28, 29, 30, or 31.

Examples

```
| julia> Dates.daysinmonth(Date("2000-01"))
31
```

```
| julia> Dates.daysinmonth(Date("2001-02"))
28
```

```
| julia> Dates.daysinmonth(Date("2000-02"))
29
```

[source](#)

[Dates.isleapyear](#) – Function.

```
| isleapyear(dt::TimeType) -> Bool
```

Examples

```
julia> Dates.isleapyear(Date("2004"))
```

```
true
```

```
julia> Dates.isleapyear(Date("2005"))
```

```
false
```

[source](#)

[Dates.dayofyear](#) – Function.

```
| dayofyear(dt::TimeType) -> Int
```

Returns the day of the year for `dt` with January 1st being day 1.

[source](#)

[Dates.daysinyear](#) – Function.

```
| daysinyear(dt::TimeType) -> Int
```

Returns 366 if the year of `dt` is a leap year, otherwise returns 365.

Examples

```
julia> Dates.daysinyear(1999)
```

```
365
```

```
julia> Dates.daysinyear(2000)
```

```
366
```

[source](#)

[Dates.quarterofyear](#) – Function.

```
| quarterofyear(dt::TimeType) -> Int
```

154 Returns the quarter that `dt` resides in. Range of value is 1:62.

[source](#)

`Dates.dayofquarter` – Function.

```
| dayofquarter(dt::TimeType) -> Int
```

Returns the day of the current quarter of `dt`. Range of value is 1:92.

[source](#)

## Adjuster Functions

`Base.trunc` – Method.

```
| trunc(dt::TimeType, ::Type{Period}) -> TimeType
```

Truncates the value of `dt` according to the provided `Period` type.

Examples

```
| julia> trunc(Dates.DateTime("1996-01-01T12:30:00"), Dates.Day)
| 1996-01-01T00:00:00
```

[source](#)

`Dates.firstdayofweek` – Function.

```
| firstdayofweek(dt::TimeType) -> TimeType
```

Adjusts `dt` to the Monday of its week.

Examples

```
| julia> Dates.firstdayofweek(DateTime("1996-01-05T12:30:00"))
| 1996-01-01T00:00:00
```

[source](#)

`Dates.lastdayofweek` – Function.

Adjusts `dt` to the Sunday of its week.

Examples

```
julia> Dates.lastdayofweek(DateTime("1996-01-05T12:30:00"))
1996-01-07T00:00:00
```

source

[Dates.firstdayofmonth](#) – Function.

```
| firstdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its month.

Examples

```
julia> Dates.firstdayofmonth(DateTime("1996-05-20"))
1996-05-01T00:00:00
```

source

[Dates.lastdayofmonth](#) – Function.

```
| lastdayofmonth(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its month.

Examples

```
julia> Dates.lastdayofmonth(DateTime("1996-05-20"))
1996-05-31T00:00:00
```

source

[Dates.firstdayofyear](#) – Function.

```
| firstdayofyear(dt::TimeType) -> TimeType
```

Examples

```
| julia> Dates.firstdayofyear(DateTime("1996-05-20"))
| 1996-01-01T00:00:00
```

source

`Dates.lastdayofyear` – Function.

```
| lastdayofyear(dt::TimeType) -> TimeType
```

Adjusts `dt` to the last day of its year.

Examples

```
| julia> Dates.lastdayofyear(DateTime("1996-05-20"))
| 1996-12-31T00:00:00
```

source

`Dates.firstdayofquarter` – Function.

```
| firstdayofquarter(dt::TimeType) -> TimeType
```

Adjusts `dt` to the first day of its quarter.

Examples

```
| julia> Dates.firstdayofquarter(DateTime("1996-05-20"))
| 1996-04-01T00:00:00
```

```
| julia> Dates.firstdayofquarter(DateTime("1996-08-20"))
| 1996-07-01T00:00:00
```

source

`Dates.lastdayofquarter` – Function.

Adjusts `dt` to the last day of its quarter.

Examples

```
julia> Dates.lastdayofquarter(DateTime("1996-05-20"))
```

```
1996-06-30T00:00:00
```

```
julia> Dates.lastdayofquarter(DateTime("1996-08-20"))
```

```
1996-09-30T00:00:00
```

[source](#)

[Dates.tonext](#) – Method.

```
| tonext(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the next day of week corresponding to `dow` with 1 = Monday, 2 = Tuesday, etc. Setting `same=true` allows the current `dt` to be considered as the next `dow`, allowing for no adjustment to occur.

[source](#)

[Dates.toprev](#) – Method.

```
| toprev(dt::TimeType, dow::Int; same::Bool=false) -> TimeType
```

Adjusts `dt` to the previous day of week corresponding to `dow` with 1 = Monday, 2 = Tuesday, etc. Setting `same=true` allows the current `dt` to be considered as the previous `dow`, allowing for no adjustment to occur.

[source](#)

[Dates.tofirst](#) – Function.

```
| tofirst(dt::TimeType, dow::Int; of=Month) -> TimeType
```

154 Adjusts `dt` to the first dow of its month. Alternatively, `of=Year` will adjust to the first dow of the year.

[source](#)

`Dates.tolast` – Function.

```
| tolast(dt::TimeType, dow::Int; of=Month) -> TimeType
```

Adjusts `dt` to the last dow of its month. Alternatively, `of=Year` will adjust to the last dow of the year.

[source](#)

`Dates.tonext` – Method.

```
| tonext(func::Function, dt::TimeType; step=Day(1), limit=10000,  
|   same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

[source](#)

`Dates.toprev` – Method.

```
| toprev(func::Function, dt::TimeType; step=Day(-1), limit=10000,  
|   same=false) -> TimeType
```

Adjusts `dt` by iterating at most `limit` iterations by `step` increments until `func` returns `true`. `func` must take a single `TimeType` argument and return a `Bool`. `same` allows `dt` to be considered in satisfying `func`.

[source](#)

Periods

`Dates.Period` – Method.

## 62.2. DATES FUNCTIONS

Year(v)  
Month(v)  
Week(v)  
Day(v)  
Hour(v)  
Minute(v)  
Second(v)  
Millisecond(v)  
Microsecond(v)  
Nanosecond(v)

1545

Construct a `Period` type with the given v value. Input must be losslessly convertible to an `Int64`.

`source`

`Dates.CompoundPeriod` – Method.

CompoundPeriod(periods) -> CompoundPeriod

Construct a `CompoundPeriod` from a `Vector` of `Periods`. All `Periods` of the same type will be added together.

Examples

```
julia> Dates.CompoundPeriod(Dates.Hour(12), Dates.Hour(13))
```

```
25 hours
```

```
julia> Dates.CompoundPeriod(Dates.Hour(-1), Dates.Minute(1))
```

```
-1 hour, 1 minute
```

```
julia> Dates.CompoundPeriod(Dates.Month(1), Dates.Week(-2))
```

```
1 month, -2 weeks
```

1546 **julia>** Dates.CompoundPeriod(Dates.Minute(50000)) CHAPTER 62) DATES AND TIME  
50000 minutes

[source](#)

[Dates.default](#) – Function.

| `default(p::Period) -> Period`

Returns a sensible “default” value for the input Period by returning T(1) for Year, Month, and Day, and T(0) for Hour, Minute, Second, and Millisecond.

[source](#)

## Rounding Functions

**Date** and **Datetime** values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with **floor**, **ceil**, or **round**.

[Base.floor](#) – Method.

| `floor(dt::TimeType, p::Period) -> TimeType`

Returns the nearest **Date** or **Datetime** less than or equal to **dt** at resolution **p**.

For convenience, **p** may be a type instead of a value: **floor(dt, Dates.Hour)** is a shortcut for **floor(dt, Dates.Hour(1))**.

**julia>** `floor(Date(1985, 8, 16), Dates.Month)`

1985-08-01

**julia>** `floor(DateTime(2013, 2, 13, 0, 31, 20),  
→ Dates.Minute(15))`

2013-02-13T00:30:00

```
2016-08-06T00:00:00
```

[source](#)

[Base.ceil](#) – Method.

```
| ceil(dt::TimeType, p::Period) -> TimeType
```

Returns the nearest `Date` or `DateTime` greater than or equal to `dt` at resolution `p`.

For convenience, `p` may be a type instead of a value: `ceil(dt, Dates.Hour)` is a shortcut for `ceil(dt, Dates.Hour(1))`.

```
julia> ceil(Date(1985, 8, 16), Dates.Month)
```

```
1985-09-01
```

```
julia> ceil(DateTime(2013, 2, 13, 0, 31, 20),
    ↪ Dates.Minute(15))
```

```
2013-02-13T00:45:00
```

```
julia> ceil(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)
```

```
2016-08-07T00:00:00
```

[source](#)

[Base.round](#) – Method.

```
| round(dt::TimeType, p::Period, [r::RoundingMode]) -> TimeType
```

Returns the `Date` or `DateTime` nearest to `dt` at resolution `p`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, `p` may be a type instead of a value: `round(dt, Dates.Hour)` is a shortcut for `round(dt, Dates.Hour(1))`.

1548 **julia>** round(Date(1985, 8, 16), Dates.Month) CHAPTER 62. DATES AND TIME

1985-08-01

**julia>** round(DateTime(2013, 2, 13, 0, 31, 20),  
→ Dates.Minute(15))

2013-02-13T00:30:00

**julia>** round(DateTime(2016, 8, 6, 12, 0, 0), Dates.Day)  
2016-08-07T00:00:00

Valid rounding modes for `round(::TimeType, ::Period, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

[source](#)

Most `Period` values can also be rounded to a specified resolution:

[Base.floor](#) – Method.

```
| floor(x::Period, precision::T) where T <: Union{TimePeriod,  
| Week, Day} -> T
```

Rounds `x` down to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `floor(x, Dates.Hour)` is a shortcut for `floor(x, Dates.Hour(1))`.

**julia>** floor(Dates.Day(16), Dates.Week)

2 weeks

**julia>** floor(Dates.Minute(44), Dates.Minute(15))

30 minutes

```
julia> floor(Dates.Hour(36), Dates.Day)  
1 day
```

Rounding to a precision of Months or Years is not supported, as these Periods are of inconsistent length.

[source](#)

[Base.ceil](#) – Method.

```
ceil(x::Period, precision::T) where T <: Union{TimePeriod, Week  
, Day} -> T
```

Rounds `x` up to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`.

For convenience, `precision` may be a type instead of a value: `ceil(x, Dates.Hour)` is a shortcut for `ceil(x, Dates.Hour(1))`.

```
julia> ceil(Dates.Day(16), Dates.Week)
```

```
3 weeks
```

```
julia> ceil(Dates.Minute(44), Dates.Minute(15))
```

```
45 minutes
```

```
julia> ceil(Dates.Hour(36), Dates.Day)
```

```
3 days
```

Rounding to a precision of Months or Years is not supported, as these Periods are of inconsistent length.

[source](#)

[Base.round](#) – Method.

1550 round(x::Period, precision::T, [r::RoundingMode]) where T<!CHAPTER 62. DATES AND TIME  
Union{TimePeriod, Week, Day} -> T

Rounds `x` to the nearest multiple of `precision`. If `x` and `precision` are different subtypes of `Period`, the return value will have the same type as `precision`. By default (`RoundNearestTiesUp`), ties (e.g., rounding 90 minutes to the nearest hour) will be rounded up.

For convenience, `precision` may be a type instead of a value: `round(x, Dates.Hour)` is a shortcut for `round(x, Dates.Hour(1))`.

```
julia> round(Dates.Day(16), Dates.Week)
```

2 weeks

```
julia> round(Dates.Minute(44), Dates.Minute(15))
```

45 minutes

```
julia> round(Dates.Hour(36), Dates.Day)
```

3 days

Valid rounding modes for `round(::Period, ::T, ::RoundingMode)` are `RoundNearestTiesUp` (default), `RoundDown` (`floor`), and `RoundUp` (`ceil`).

Rounding to a `precision` of `Months` or `Years` is not supported, as these `Periods` are of inconsistent length.

### source

The following functions are not exported:

`Dates.floorceil` – Function.

```
| floorceil(dt::TimeType, p::Period) -> (TimeType, TimeType)
```

Simultaneously return the `floor` and `ceil` of a `Date` or `DateTime` at resolution `p`. More efficient than calling both `floor` and `ceil` individually.

```
| floorceil(x::Period, precision::T) where T <: Union{TimePeriod,  
| Week, Day} -> (T, T)
```

Simultaneously return the `floor` and `ceil` of Period at resolution p.  
More efficient than calling both `floor` and `ceil` individually.

[source](#)

[Dates.epochdays2date](#) – Function.

```
| epochdays2date(days) -> Date
```

Takes the number of days since the rounding epoch (`0000-01-01T00:00:00`)  
and returns the corresponding `Date`.

[source](#)

[Dates.epochms2datetime](#) – Function.

```
| epochms2datetime(milliseconds) -> DateTime
```

Takes the number of milliseconds since the rounding epoch (`0000-01-01T00:00:00`)  
and returns the corresponding `DateTime`.

[source](#)

[Dates.date2epochdays](#) – Function.

```
| date2epochdays(dt::Date) -> Int64
```

Takes the given `Date` and returns the number of days since the rounding  
epoch (`0000-01-01T00:00:00`) as an `Int64`.

[source](#)

[Dates.datetime2epochms](#) – Function.

```
| datetime2epochms(dt::DateTime) -> Int64
```

155 Takes the given `DateTime` and returns the rounding epoch (0000-01-01T00:00:00) as an `Int64`.

[source](#)

## Conversion Functions

`Dates.today` – Function.

| `today() -> Date`

Returns the date portion of `now()`.

[source](#)

`Dates.unix2datetime` – Function.

| `unix2datetime(x) -> DateTime`

Takes the number of seconds since unix epoch 1970-01-01T00:00:00 and converts to the corresponding `DateTime`.

[source](#)

`Dates.datetime2unix` – Function.

| `datetime2unix(dt::DateTime) -> Float64`

Takes the given `DateTime` and returns the number of seconds since the unix epoch 1970-01-01T00:00:00 as a `Float64`.

[source](#)

`Dates.julian2datetime` – Function.

| `julian2datetime(julian_days) -> DateTime`

Takes the number of Julian calendar days since epoch -4713-11-24T12:00:00 and returns the corresponding `DateTime`.

[source](#)

```
| datetime2julian(dt::DateTime) -> Float64
```

Takes the given **DateTime** and returns the number of Julian calendar days since the julian epoch **-4713-11-24T12:00:00** as a **Float64**.

**source**

Dates.rata2datetime – Function.

```
| rata2datetime(days) -> DateTime
```

Takes the number of Rata Die days since epoch **0000-12-31T00:00:00** and returns the corresponding **DateTime**.

**source**

Dates.datetime2rata – Function.

```
| datetime2rata(dt::TimeType) -> Int64
```

Returns the number of Rata Die days since epoch from the given **Date** or **DateTime**.

**source**

## Constants

Days of the Week:

Variable	Abbr.	Value (Int)
Monday	Mon	1
Tuesday	Tue	2
Wednesday	Wed	3
Thursday	Thu	4
Friday	Fri	5
Saturday	Sat	6
Sunday	Sun	7

Months of the Year:

Variable	Abbr.	Value (Int)
January	Jan	1
February	Feb	2
March	Mar	3
April	Apr	4
May	May	5
June	Jun	6
July	Jul	7
August	Aug	8
September	Sep	9
October	Oct	10
November	Nov	11
December	Dec	12

# Chapter 63

## Iteration utilities

`Base.Iterators.zip` – Function.

```
| zip(iters...)
```

For a set of iterable objects, returns an iterable of tuples, where the `i`th tuple contains the `i`th component of each input iterable.

Examples

```
julia> a = 1:5
```

```
1:5
```

```
julia> b = ["e", "d", "b", "c", "a"]
```

```
5-element Array{String,1}:
```

```
  "e"  
  "d"  
  "b"  
  "c"  
  "a"
```

```
julia> c = zip(a,b)
```

```
Base.Iterators.Zip2{UnitRange{Int64},Array{String,1}}(1:5,  
→  ["e", "d", "b", "c", "a"])
```

```
julia> length(c)
```

```
5
```

```
julia> first(c)
```

```
(1, "e")
```

```
source
```

`Base.Iterators.enumerate` – Function.

```
| enumerate(iter)
```

An iterator that yields (`i`, `x`) where `i` is a counter starting at 1, and `x` is the `i`th value from the given iterator. It's useful when you need not only the values `x` over which you are iterating, but also the number of iterations so far. Note that `i` may not be valid for indexing `iter`; it's also possible that `x != iter[i]`, if `iter` has indices that do not start at 1. See the `enumerate(IndexLinear(), iter)` method if you want to ensure that `i` is an index.

Examples

```
julia> a = ["a", "b", "c"];
```

```
julia> for (index, value) in enumerate(a)
```

```
    println("$index $value")
```

```
end
```

```
1 a
```

```
2 b
```

```
3 c
```

```
source
```

```
| rest(iter, state)
```

An iterator that yields the same elements as `iter`, but starting at the given `state`.

Examples

```
julia> collect(Iterators.rest([1,2,3,4], 2))
3-element Array{Any,1}:
 2
 3
 4
```

source

```
| countfrom(start=1, step=1)
```

An iterator that counts forever, starting at `start` and incrementing by `step`.

Examples

```
julia> for v in Iterators.countfrom(5, 2)

        v > 10 && break

        println(v)

    end

5
7
9
```

`Base.Iterators.take` – Function.

```
| take(iter, n)
```

An iterator that generates at most the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
```

```
1:2:11
```

```
julia> collect(a)
```

```
6-element Array{Int64,1}:
```

```
1
```

```
3
```

```
5
```

```
7
```

```
9
```

```
11
```

```
julia> collect(Iterators.take(a,3))
```

```
3-element Array{Int64,1}:
```

```
1
```

```
3
```

```
5
```

`source`

`Base.Iterators.drop` – Function.

```
| drop(iter, n)
```

An iterator that generates all but the first `n` elements of `iter`.

Examples

```
julia> a = 1:2:11
1:2:11

julia> collect(a)
6-element Array{Int64,1}:
 1
 3
 5
 7
 9
 11

julia> collect(Iterators.drop(a,4))
2-element Array{Int64,1}:
 9
 11
```

source

`Base.Iterators.cycle` – Function.

```
| cycle(iter)
```

An iterator that cycles through `iter` forever.

Examples

```
julia> for (i, v) in enumerate(Iterators.cycle("hello"))

        print(v)

        i > 10 && break

    end
hellohellloh
```

`Base.Iterators.repeated` – Function.

```
| repeated(x[, n::Int])
```

An iterator that generates the value `x` forever. If `n` is specified, generates `x` that many times (equivalent to `take(repeated(x), n)`).

Examples

```
julia> a = Iterators.repeated([1 2], 4);
```

```
julia> collect(a)
```

```
4-element Array{Array{Int64,2},1}:
 [1 2]
 [1 2]
 [1 2]
 [1 2]
```

source

`Base.Iterators.product` – Function.

```
| product(iters...)
```

Returns an iterator over the product of several iterators. Each generated element is a tuple whose `i`th element comes from the `i`th argument iterator. The first iterator changes the fastest.

Examples

```
julia> collect(Iterators.product(1:2,3:5))
```

```
2×3 Array{Tuple{Int64,Int64},2}:
 (1, 3)  (1, 4)  (1, 5)
 (2, 3)  (2, 4)  (2, 5)
```

source

[Base.Iterators.flatten](#) – Function.

1561

```
| flatten(iter)
```

Given an iterator that yields iterators, return an iterator that yields the elements of those iterators. Put differently, the elements of the argument iterator are concatenated.

Examples

```
| julia> collect(Iterators.flatten((1:2, 8:9)))
4-element Array{Int64,1}:
 1
 2
 8
 9
```

[source](#)

[Base.Iterators.partition](#) – Function.

```
| partition(collection, n)
```

Iterate over a collection  $n$  elements at a time.

Examples

```
| julia> collect(Iterators.partition([1,2,3,4,5], 2))
3-element Array{Array{Int64,1},1}:
 [1, 2]
 [3, 4]
 [5]
```

[source](#)

[Base.Iterators.filter](#) – Function.

```
| Iterators.filter(flt, itr)
```

156 Given a predicate function `flt` and `itr` an iterable object which upon iteration yields the elements `x` of `itr` that satisfy `flt(x)`. The order of the original iterator is preserved.

This function is lazy; that is, it is guaranteed to return in  $O(1)$  time and use  $O(1)$  additional space, and `flt` will not be called by an invocation of `filter`. Calls to `flt` will be made when iterating over the returned iterable object. These calls are not cached and repeated calls will be made when reiterating.

See [Base.filter](#) for an eager implementation of filtering for arrays.

Examples

```
julia> f = Iterators.filter(isodd, [1, 2, 3, 4, 5])
Base.Iterators.Filter{Base.#isodd, Array{Int64,1}}(isodd, [1,
→ 2, 3, 4, 5])

julia> foreach(println, f)
1
3
5
```

[source](#)

[Base.Iterators.reverse](#) – Function.

```
| Iterators.reverse(itr)
```

Given an iterator `itr`, then `reverse(itr)` is an iterator over the same collection but in the reverse order.

This iterator is “lazy” in that it does not make a copy of the collection in order to reverse it; see [Base.reverse](#) for an eager implementation.

Not all iterator types `T` support reverse-order iteration. If `T` doesn’t, then iterating over `Iterators.reverse(itr::T)` will throw a [MethodError](#).

`odError` because of the missing `start`, `next`, and `done` methods<sup>1563</sup>  
`Iterators.Reverse{T}`. (To implement these methods, the original iterator `itr::T` can be obtained from `r = Iterators.reverse(itr)` by `r.itr`.)

`source`



# Chapter 64

## Unit Testing

### 64.1 Testing Base Julia

Julia is under rapid development and has an extensive test suite to verify functionality across multiple platforms. If you build Julia from source, you can run this test suite with `make test`. In a binary install, you can run the test suite using `Base.runtests()`.

`Base.runtests` – Function.

```
Base.runtests(tests=["all"], numcores=ceil(Int, Sys.CPU_CORES /  
2);  
              exit_on_error=false, [seed])
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `numcores` processors. If `exit_on_error` is `false`, when one test fails, all remaining tests in other files will still be run; they are otherwise discarded, when `exit_on_error == true`. If a seed is provided via the keyword argument, it is used to seed the global RNG in the context where the tests are run; otherwise the seed is chosen randomly.

[source](#)

The `Test` module provides simple unit testing functionality. Unit testing is a way to see if your code is correct by checking that the results are what you expect. It can be helpful to ensure your code still works after you make changes, and can be used when developing as a way of specifying the behaviors your code should have when complete.

Simple unit testing can be performed with the `@test` and `@test_throws` macros:

`Test.@test` – Macro.

```
| @test ex  
| @test f(args...) key=val ...
```

Tests that the expression `ex` evaluates to `true`. Returns a `Pass Result` if it does, a `Fail Result` if it is `false`, and an `Error Result` if it could not be evaluated.

The `@test f(args...) key=val...` form is equivalent to writing `@test f(args..., key=val...)` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
| @test a ≈ b atol=ε
```

This is equivalent to the uglier test `@test ≈(a, b, atol=ε)`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments (`k=v`).

`source`

`Test.@test_throws` – Macro.

```
| @test_throws exception expr
```

Tests that the expression `expr` throws `exception`. The exception may specify either a type, or a value (which will be tested for equality by com-

64.2 PASS/FAIL TESTS

NOTE that @test\_throws does not support a trailing keyword form.

### source

For example, suppose we want to check our new function `foo(x)` works as expected:

```
julia> using Test

julia> foo(x) = length(x)^2
foo (generic function with 1 method)
```

If the condition is true, a **Pass** is returned:

```
julia> @test foo("bar") == 9
Test Passed

julia> @test foo("fizz") >= 10
Test Passed
```

If the condition is false, then a **Fail** is returned and an exception is thrown:

```
julia> @test foo("f") == 20
Test Failed at none:1
    Expression: foo("f") == 20
    Evaluated: 1 == 20
ERROR: There was an error during testing
```

If the condition could not be evaluated because an exception was thrown, which occurs in this case because `length` is not defined for symbols, an **Error** object is returned and an exception is thrown:

```
|1568 julia> @test foo(:cat) == 1
```

## CHAPTER 64. UNIT TESTING

Error During Test

Test threw an exception of type MethodError

Expression: foo(:cat) == 1

MethodError: no method matching length(::Symbol)

Closest candidates are:

length(::SimpleVector) at essentials.jl:256

length(::Base.MethodList) at reflection.jl:521

length(::MethodTable) at reflection.jl:597

...

Stacktrace:

[...]

ERROR: There was an error during testing

If we expect that evaluating an expression should throw an exception, then we can use `@test_throws` to check that this occurs:

```
julia> @test_throws MethodError foo(:cat)
```

Test Passed

### 64.3 Working with Test Sets

Typically a large number of tests are used to make sure functions work correctly over a range of inputs. In the event a test fails, the default behavior is to throw an exception immediately. However, it is normally preferable to run the rest of the tests first to get a better picture of how many errors there are in the code being tested.

The `@testset` macro can be used to group tests into sets. All the tests in a test set will be run, and at the end of the test set a summary will be printed. If any of the tests failed, or could not be evaluated due to an error, the test

`Test.@testset` – Macro.

```
@testset [CustomTestSet] [option=val ...] ["description"]
    begin ... end
@testset [CustomTestSet] [option=val ...] ["description $v"]
    for v in (...) ... end
@testset [CustomTestSet] [option=val ...] ["description $v, $w"]
    " ] for v in (...), w in (...) ... end
```

Starts a new test set, or multiple test sets if a `for` loop is provided.

If no custom testset type is given it defaults to creating a `DefaultTestSet`. `DefaultTestSet` records all the results and, if there are any `Fails` or `Errors`, throws an exception at the end of the top-level (non-nested) test set, along with a summary of the test results.

Any custom testset type (subtype of `AbstractTestSet`) can be given and it will also be used for any nested `@testset` invocations. The given options are only applied to the test set where they are given. The default test set type does not take any options.

The description string accepts interpolation from the loop indices. If no description is provided, one is constructed based on the variables.

By default the `@testset` macro will return the testset object itself, though this behavior can be customized in other testset types. If a `for` loop is used then the macro collects and returns a list of the return values of the `finish` method, which by default will return a list of the testset objects used in each iteration.

### source

We can put our tests for the `foo(x)` function in a test set:

```
|1570 julia> @testset "Foo Tests" begin
```

CHAPTER 64. UNIT TESTING

```
    @test foo("a") == 1

    @test foo("ab") == 4

    @test foo("abc") == 9

end;

Test Summary: | Pass  Total
Foo Tests     |      3      3
```

Test sets can also be nested:

```
julia> @testset "Foo Tests" begin
    @testset "Animals" begin
        @test foo("cat") == 9

        @test foo("dog") == foo("cat")

    end

    @testset "Arrays $i" for i in 1:3
        @test foo(zeros(i)) == i^2

        @test foo(ones(i)) == i^2

    end
```

```
Test Summary: | Pass  Total  
Foo Tests    |     8      8
```

In the event that a nested test set has no failures, as happened here, it will be hidden in the summary. If we do have a test failure, only the details for the failed test sets will be shown:

```
julia> @testset "Foo Tests" begin  
          @testset "Animals" begin  
            @testset "Felines" begin  
              @test foo("cat") == 9  
            end  
            @testset "Canines" begin  
              @test foo("dog") == 9  
            end  
          end  
          @testset "Arrays" begin  
            @test foo(zeros(2)) == 4  
            @test foo(ones(4)) == 15
```

```
    end
```

Arrays: Test Failed

Expression: foo(ones(4)) == 15

Evaluated: 16 == 15

[...]

Test Summary: | Pass Fail Total

	Pass	Fail	Total
Foo Tests	3	1	4
Animals	2		2
Arrays	1	1	2

ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0  
→ broken.

## 64.4 Other Test Macros

As calculations on floating-point values can be imprecise, you can perform approximate equality checks using either `@test a ≈ b` (where `≈`, typed via tab completion of `\approx`, is the `isapprox` function) or use `isapprox` directly.

```
julia> @test 1 ≈ 0.999999999
```

Test Passed

```
julia> @test 1 ≈ 0.999999
```

Test Failed at none:1

Expression: 1 ≈ 0.999999

Evaluated: 1 ≈ 0.999999

ERROR: There was an error during testing

`Test.@inferred` – Macro.

Tests that the call expression `f(x)` returns a value of the same type inferred by the compiler. It is useful to check for type stability.

`f(x)` can be any call expression. Returns the result of `f(x)` if the types match, and an **Error Result** if it finds different types.

```
julia> using Test

julia> f(a,b,c) = b > 1 ? 1 : 1.0
f (generic function with 1 method)

julia> typeof(f(1,2,3))
Int64

julia> @code_warntype f(1,2,3)
Variables:
    a<optimized out>
    b::Int64
    c<optimized out>

Body:
begin
    end::UNION{FLOAT64, INT64}

        unless (Base.slt_int)(1, b::Int64)::Bool goto 3

    return 1

    3:


```

1574

```
    return 1.0
```

CHAPTER 64. UNIT TESTING

```
julia> @inferred f(1,2,3)
```

ERROR: return type Int64 does not match inferred return type

  ↳ Union{Float64, Int64}

[...]

```
julia> @inferred max(1,2)
```

2

[source](#)

`Test.@test_warn` – Macro.

```
|@test_warn msg expr
```

Test whether evaluating `expr` results in `STDERR` output that contains the `msg` string or matches the `msg` regular expression. If `msg` is a boolean function, tests whether `msg(output)` returns `true`. If `msg` is a tuple or array, checks that the error output contains/matches each item in `msg`. Returns the result of evaluating `expr`.

See also `@test_nowarn` to check for the absence of error output.

[source](#)

`Test.@test_nowarn` – Macro.

```
|@test_nowarn expr
```

Test whether evaluating `expr` results in empty `STDERR` output (no warnings or other messages). Returns the result of evaluating `expr`.

[source](#)

If a test fails consistently it can be changed to use the `@test_broken` macro. This will denote the test as **Broken** if the test continues to fail and alerts the user via an **Error** if the test succeeds.

[Test.`@test\_broken`](#) – Macro.

```
| @test_broken ex  
| @test_broken f(args...) key=val ...
```

Indicates a test that should pass but currently consistently fails. Tests that the expression `ex` evaluates to `false` or causes an exception. Returns a **Broken Result** if it does, or an **Error Result** if the expression evaluates to `true`.

The `@test_broken f(args...) key=val...` form works as for the `@test` macro.

[source](#)

`@test_skip` is also available to skip a test without evaluation, but counting the skipped test in the test set reporting. The test will not run but gives a **Broken Result**.

[Test.`@test\_skip`](#) – Macro.

```
| @test_skip ex  
| @test_skip f(args...) key=val ...
```

Marks a test that should not be executed but should be included in test summary reporting as **Broken**. This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality.

The `@test_skip f(args...) key=val...` form works as for the `@test` macro.

[source](#)

## 64.6 Creating Custom `AbstractTestSet` Types

Packages can create their own `AbstractTestSet` subtypes by implementing the `record` and `finish` methods. The subtype should have a one-argument constructor taking a description string, with any options passed in as keyword arguments.

`Test.record` – Function.

```
| record(ts::AbstractTestSet, res::Result)
```

Record a result to a testset. This function is called by the `@testset` infrastructure each time a contained `@test` macro completes, and is given the test result (which could be an `Error`). This will also be called with an `Error` if an exception is thrown inside the test block but outside of a `@test` context.

`source`

`Test.finish` – Function.

```
| finish(ts::AbstractTestSet)
```

Do any final processing necessary for the given testset. This is called by the `@testset` infrastructure after a test block executes. One common use for this function is to record the testset to the parent's results list, using `get_testset`.

`source`

`Test` takes responsibility for maintaining a stack of nested testsets as they are executed, but any result accumulation is the responsibility of the `AbstractTestSet` subtype. You can access this stack with the `get_testset` and `get_testset_depth` methods. Note that these functions are not exported.

`Test.get_testset` – Function.

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

**source**

[Test.get\\_testset\\_depth](#) – Function.

```
| get_testset_depth()
```

Returns the number of active test sets, not including the default test set

**source**

`Test` also makes sure that nested `@testset` invocations use the same `AbstractTestSet` subtype as their parent unless it is set explicitly. It does not propagate any properties of the testset. Option inheritance behavior can be implemented by packages using the stack infrastructure that `Test` provides.

Defining a basic `AbstractTestSet` subtype might look like:

```
import Test: record, finish
using Test: AbstractTestSet, Result, Pass, Fail, Error
using Test: get_testset_depth, get_testset
struct CustomTestSet <: Test.AbstractTestSet
    description::AbstractString
    foo::Int
    results::Vector
    # constructor takes a description string and options keyword
    → arguments
    CustomTestSet(desc; foo=1) = new(desc, foo, [])
end

record(ts::CustomTestSet, child::AbstractTestSet) =
    → push!(ts.results, child)
```

```
1578 record(ts::CustomTestSet, res::Result) = push!(ts.results, res)
CHAPTER 64. UNIT TESTING
function finish(ts::CustomTestSet)
    # just record if we're not the top-level parent
    if get_testset_depth() > 0
        record(get_testset(), ts)
    end
    ts
end
```

And using that testset looks like:

```
@testset CustomTestSet foo=4 "custom testset inner 2" begin
    # this testset should inherit the type, but not the argument.
    @testset "custom testset inner" begin
        @test true
    end
end
```

# Chapter 65

## C Interface

`ccall` – Keyword.

```
| ccall((function_name, library), returntype, (argtype1, ...),
|       argvalue1, ...)
| ccall(function_pointer, returntype, (argtype1, ...), argvalue1,
|       ...)
```

Call a function in a C-exported shared library, specified by the tuple `(function_name, library)`, where each component is either a string or symbol. Alternatively, `ccall` may also be used to call a function pointer `function_pointer`, such as one returned by `dlsym`.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `argvalue` to the `ccall` will be converted to the corresponding `argtype`, by automatic insertion of calls to `unsafe_convert(argtype, cconvert(argtype, argvalue))`. (See also the documentation for each of these functions for further details.) In most cases, this simply results in a call to `convert(argtype, argvalue)`.

`source`

`Core.Intrinsics.cglobal` – Function.

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in [ccall](#). Returns a `Ptr{Type}`, defaulting to `Ptr{Void}` if no `Type` argument is supplied. The values can be read or written by [unsafe\\_load](#) or [unsafe\\_store!](#), respectively.

### source

`Base.cfunction` – Function.

```
cfunction(f::Function, returntype::Type, argtypes::Type) -> Ptr
    {Void}
```

Generate C-callable function pointer from the Julia function `f`. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

### Examples

```
julia> function foo(x::Int, y::Int)

        return x + y

    end

julia> cfunction(foo, Int, Tuple{Int,Int})
Ptr{Void} @0x000000001b82fc0d0

source
```

`Base.unsafe_convert` – Function.

```
unsafe_convert(T, x)
```

Convert `x` to a C argument of type `T` where the input `x` must be the return value of `cconvert(T, ...)`.

In cases where [convert](#) would need to take a Julia object and turn it into a [Ptr](#), this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to `x` exists as long as the result of this function will be used. Accordingly, the argument `x` to this function should never be an expression, only a variable name or field reference. For example, `x=a.b.c` is acceptable, but `x=[a,b,c]` is not.

The `unsafe` prefix on this function indicates that using the result of this function after the `x` argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

See also [cconvert](#)

[source](#)

[Base.cconvert](#) – Function.

`|cconvert(T,x)`

Convert `x` to a value to be passed to C code as type `T`, typically by calling `convert(T, x)`.

In cases where `x` cannot be safely converted to `T`, unlike [convert](#), [cconvert](#) may return an object of a type different from `T`, which however is suitable for [unsafe\\_convert](#) to handle. The result of this function should be kept valid (for the GC) until the result of [unsafe\\_convert](#) is not needed anymore. This can be used to allocate memory that will be accessed by the `ccall`. If multiple objects need to be allocated, a tuple of the objects can be used as return value.

Neither `convert` nor `cconvert` should take a Julia object and turn it into a `Ptr`.

[source](#)

## `Base.unsafe_load` – Function.

## CHAPTER 65. C INTERFACE

```
| unsafe_load(p::Ptr{T}, i::Integer=1)
```

Load a value of type T from the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1]`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

`source`

## `Base.unsafe_store!` – Function.

```
| unsafe_store!(p::Ptr{T}, x, i::Integer=1)
```

Store a value of type T to the address of the `i`th element (1-indexed) starting at `p`. This is equivalent to the C expression `p[i-1] = x`.

The `unsafe` prefix on this function indicates that no validation is performed on the pointer `p` to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

`source`

## `Base.unsafe_copy!` – Method.

```
| unsafe_copy!(dest::Ptr{T}, src::Ptr{T}, N)
```

Copy `N` elements from a source pointer to a destination, with no checking. The size of an element is determined by the type of the pointers.

The `unsafe` prefix on this function indicates that no validation is performed on the pointers `dest` and `src` to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

1583

[Base.unsafe\\_copy!](#) – Method.

| `unsafe_copy!(dest::Array, do, src::Array, so, N)`

Copy **N** elements from a source array to a destination, starting at offset **so** in the source and **do** in the destination (1-indexed).

The **unsafe** prefix on this function indicates that no validation is performed to ensure that **N** is inbounds on either array. Incorrect usage may corrupt or segfault your program, in the same manner as C.

source

[Base.copy!](#) – Function.

| `copy!(dest, do, src, so, N)`

Copy **N** elements from collection **src** starting at offset **so**, to array **dest** starting at offset **do**. Returns **dest**.

source

| `copy!(dest, src) -> dest`

Copy all elements from collection **src** to array **dest**.

source

| `copy!(dest, Rdest::CartesianRange, src, Rsrc::CartesianRange)`  
|   `-> dest`

Copy the block of **src** in the range of **Rsrc** to the block of **dest** in the range of **Rdest**. The sizes of the two regions must match.

source

[Base.pointer](#) – Function.

| `pointer(array [, index])`

1584 Get the native address of an array or string element. CHAPTER 65. C INTERFACE

that a Julia reference to `a` exists as long as this pointer will be used. This function is "unsafe" like `unsafe_convert`.

Calling `Ref(array[, index])` is generally preferable to this function.

`source`

`Base.unsafe_wrap` – Method.

```
| unsafe_wrap(Array, pointer::Ptr{T}, dims, own=false)
```

Wrap a Julia `Array` object around the data at the address given by `pointer`, without making a copy. The pointer element type `T` determines the array element type. `dims` is either an integer (for a 1d array) or a tuple of the array dimensions. `own` optionally specifies whether Julia should take ownership of the memory, calling `free` on the pointer when the array is no longer referenced.

This function is labelled "unsafe" because it will crash if `pointer` is not a valid memory address to data of the requested length.

`source`

`Base.pointer_from_objref` – Function.

```
| pointer_from_objref(x)
```

Get the memory address of a Julia object as a `Ptr`. The existence of the resulting `Ptr` will not protect the object from garbage collection, so you must ensure that the object remains referenced for the whole time that the `Ptr` will be used.

`source`

`Base.unsafe_pointer_to_objref` – Function.

```
| unsafe_pointer_to_objref(p::Ptr)
```

Convert a `Ptr` to an object reference. Assumes the pointer refers to a valid heap-allocated Julia object. If this is not the case, undefined behavior results, hence this function is considered "unsafe" and should be used with care.

[source](#)

`Base.disable_sigint` – Function.

```
| disable_sigint(f::Function)
```

Disable Ctrl-C handler during execution of a function on the current task, for calling external code that may call julia code that is not interrupt safe. Intended to be called using `do` block syntax as follows:

```
| disable_sigint() do
|     # interrupt-unsafe code
|     ...
| end
```

This is not needed on worker threads (`Threads.threadid() != 1`) since the `InterruptException` will only be delivered to the master thread. External functions that do not call julia code or julia runtime automatically disable sigint during their execution.

[source](#)

`Base.reenable_sigint` – Function.

```
| reenable_sigint(f::Function)
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

[source](#)

`Base.systemerror` – Function.

1586 `systemerror(sysfunc, iftrue)`

CHAPTER 65. C INTERFACE

Raises a `SystemError` for `errno` with the descriptive string `sysfunc` if `iftrue` is true

`source`

`Core.Ptr` – Type.

| `Ptr{T}`

A memory address referring to data of type T. However, there is no guarantee that the memory is actually valid, or that it actually represents data of the specified type.

`source`

`Core.Ref` – Type.

| `Ref{T}`

An object that safely references data of type T. This type is guaranteed to point to valid, Julia-allocated memory of the correct type. The underlying data is protected from freeing by the garbage collector as long as the `Ref` itself is referenced.

When passed as a `ccall` argument (either as a `Ptr` or `Ref` type), a `Ref` object will be converted to a native pointer to the data it references.

There is no invalid (NULL) `Ref`.

`source`

`Base.Cchar` – Type.

| `Cchar`

Equivalent to the native `char` c-type.

`source`

[Base.Cuchar](#) – Type.

1587

| Cuchar

Equivalent to the native `unsigned char` c-type ([UInt8](#)).

[source](#)

[Base.Cshort](#) – Type.

| Cshort

Equivalent to the native `signed short` c-type ([Int16](#)).

[source](#)

[Base.Cushort](#) – Type.

| Cushort

Equivalent to the native `unsigned short` c-type ([UInt16](#)).

[source](#)

[Base.Cint](#) – Type.

| Cint

Equivalent to the native `signed int` c-type ([Int32](#)).

[source](#)

[Base.Cuint](#) – Type.

| Cuint

Equivalent to the native `unsigned int` c-type ([UInt32](#)).

[source](#)

[Base.Clong](#) – Type.

| Clong

158 Equivalent to the native `signed long` c-type CHAPTER 65. C INTERFACE

`source`

[Base.Culong](#) – Type.

| `Culong`

Equivalent to the native `unsigned long` c-type.

`source`

[Base.Clonglong](#) – Type.

| `Clonglong`

Equivalent to the native `signed long long` c-type ([Int64](#)).

`source`

[Base.Culonglong](#) – Type.

| `Culonglong`

Equivalent to the native `unsigned long long` c-type ([UInt64](#)).

`source`

[Base.Cintmax\\_t](#) – Type.

| `Cintmax_t`

Equivalent to the native `intmax_t` c-type ([Int64](#)).

`source`

[Base.Cuintmax\\_t](#) – Type.

| `Cuintmax_t`

Equivalent to the native `uintmax_t` c-type ([UInt64](#)).

`source`

`Base.Csize_t` – Type.

1589

| `Csize_t`

Equivalent to the native `size_t` c-type (`UInt`).

`source`

`Base.Cssize_t` – Type.

| `Cssize_t`

Equivalent to the native `ssize_t` c-type.

`source`

`Base.Cptrdiff_t` – Type.

| `Cptrdiff_t`

Equivalent to the native `ptrdiff_t` c-type (`Int`).

`source`

`Base.Cwchar_t` – Type.

| `Cwchar_t`

Equivalent to the native `wchar_t` c-type (`Int32`).

`source`

`Base.Cfloat` – Type.

| `Cfloat`

Equivalent to the native `float` c-type (`Float32`).

`source`

`Base.Cdouble` – Type.

| `Cdouble`

1590 Equivalent to the native `double` c-type ([Float](#))  
CHAPTER 65. C INTERFACE

[source](#)

# Chapter 66

## LLVM Interface

`Core.Intrinsics.llvmcall` – Function.

```
llvmcall(IR::String, ReturnType, (ArgumentType1, ...),
         ArgumentValue1, ...)
llvmcall((declarations::String, IR::String), ReturnType, (
         ArgumentType1, ...), ArgumentValue1, ...)
```

Call LLVM IR string in the first argument. Similar to an LLVM function `define` block, arguments are available as consecutive unnamed SSA variables (%0, %1, etc.).

The optional declarations string contains external functions declarations that are necessary for llvm to compile the IR string. Multiple declarations can be passed in by separating them with line breaks.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression.

Each `ArgumentValue` to `llvmcall` will be converted to the corresponding `ArgumentType`, by automatic insertion of calls to `unsafe_convert(ArgumentType, cconvert(ArgumentType, ArgumentValue))`. (see also the documentation for each of these functions for further details). In most

1592 cases, this simply results in a call to `convertType` (see [CHAPTER 6: LLVM INTERFACE](#)).

See `test/llvmpcall.jl` for usage examples.

`source`

# Chapter 67

## C Standard Library

`Base.Libc.malloc` – Function.

```
| malloc(size::Integer) -> Ptr{Void}
```

Call `malloc` from the C standard library.

`source`

`Base.Libc.calloc` – Function.

```
| calloc(num::Integer, size::Integer) -> Ptr{Void}
```

Call `calloc` from the C standard library.

`source`

`Base.Libc.realloc` – Function.

```
| realloc(addr::Ptr, size::Integer) -> Ptr{Void}
```

Call `realloc` from the C standard library.

See warning in the documentation for `free` regarding only using this on memory originally obtained from `malloc`.

`source`

`Base.Libc.free` – Function.

1594 `free(addr::Ptr)`

## CHAPTER 67. C STANDARD LIBRARY

Call `free` from the C standard library. Only use this on memory obtained from `malloc`, not on pointers retrieved from other C libraries. `Ptr` objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple `libc` libraries exist on the system.

`source`

[Base.Libc\(errno\)](#) – Function.

`|errno([code])`

Get the value of the C library's `errno`. If an argument is specified, it is used to set the value of `errno`.

The value of `errno` is only valid immediately after a `ccall` to a C library routine that sets it. Specifically, you cannot call `errno` at the next prompt in a REPL, because lots of code is executed between prompts.

`source`

[Base.Libc.strerror](#) – Function.

`|strerror(n=errno())`

Convert a system call error code to a descriptive string

`source`

[Base.Libc.GetLastError](#) – Function.

`|GetLastError()`

Call the Win32 `GetLastError` function [only available on Windows].

`source`

[Base.Libc.FormatMessage](#) – Function.

| FormatMessage(n=GetLastError())

1595

Convert a Win32 system call error code to a descriptive string [only available on Windows].

**source**

[Base.Libc.time](#) – Method.

| time(t::TmStruct)

Converts a **TmStruct** struct to a number of seconds since the epoch.

**source**

[Base.Libc.strftime](#) – Function.

| strftime([format], time)

Convert time, given as a number of seconds since the epoch or a **TmStruct**, to a formatted string using the given format. Supported formats are the same as those in the standard C library.

**source**

[Base.Libc.strptime](#) – Function.

| strptime([format], timestr)

Parse a formatted time string into a **TmStruct** giving the seconds, minute, hour, date, etc. Supported formats are the same as those in the standard C library. On some platforms, timezones will not be parsed correctly. If the result of this function will be passed to **time** to convert it to seconds since the epoch, the **isdst** field should be filled in manually. Setting it to -1 will tell the C library to use the current system settings to determine the timezone.

**source**

```
| TmStruct([seconds])
```

Convert a number of seconds since the epoch to broken-down format, with fields `sec`, `min`, `hour`, `mday`, `month`, `year`, `wday`, `yday`, and `isdst`.

[source](#)

[Base.Libc.flush\\_cstdio](#) – Function.

```
| flush_cstdio()
```

Flushes the C `stdout` and `stderr` streams (which may have been written to by external C code).

[source](#)

# Chapter 68

## Dynamic Linker

The names in `Base.Libdl` are not exported and need to be called e.g. as `Libdl.dlopen`.

`Base.Libdl.dlopen` – Function.

```
| dlopen(libfile::AbstractString [, flags::Integer])
```

Load a shared library, returning an opaque handle.

The extension given by the constant `dlext` (.so, .dll, or .dylib) can be omitted from the `libfile` string, as it is automatically appended if needed. If `libfile` is not an absolute path name, then the paths in the array `DL_LOAD_PATH` are searched for `libfile`, followed by the system load path.

The optional `flags` argument is a bitwise-or of zero or more of `RTLD_LOCAL`, `RTLD_GLOBAL`, `RTLD.LAZY`, `RTLD_NOW`, `RTLD_NODELETE`, `RTLD_NOLOAD`, `RTLD_DEEPBIND`, and `RTLD_FIRST`. These are converted to the corresponding flags of the POSIX (and/or GNU libc and/or MacOS) `dlopen` command, if possible, or are ignored if the specified functionality is not available on the current platform. The default flags are platform specific. On MacOS the default `dlopen` flags are `RTLD.LAZY|RTLD_DEEPBIND|RTLD_GLOBAL` while on other platforms the defaults are `RTLD.LAZY|RTLD_DEEPBIND`.

159 BIND | RTLD\_LOCAL. An important usage CHAPTER 16.2. DYNAMIC LINKER

default behavior for when the dynamic library loader binds library references to exported symbols and if the bound references are put into process local or global scope. For instance RTLD\_LAZY|RTLD\_DEEPBIND|RTLD\_GLOBAL allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

[source](#)

[Base.Libdl.dlopen\\_e](#) – Function.

```
| dlopen_e(libfile::AbstractString [, flags::Integer])
```

Similar to [dlopen](#), except returns a NULL pointer instead of raising errors.

[source](#)

[Base.Libdl.RTLD\\_NOW](#) – Constant.

```
RTLD_DEEPBIND  
RTLD_FIRST  
RTLD_GLOBAL  
RTLD_LAZY  
RTLD_LOCAL  
RTLD_NODELETE  
RTLD_NOLOAD  
RTLD_NOW
```

Enum constant for [dlopen](#). See your platform man page for details, if applicable.

[source](#)

[Base.Libdl.dlsym](#) – Function.

```
| dlsym(handle, sym)
```

Look up a symbol from a shared library handle, return callable function pointer on success.

[source](#)

[Base.Libdl.dlsym\\_e](#) – Function.

| `dlsym_e(handle, sym)`

Look up a symbol from a shared library handle, silently return `NULL` pointer on lookup failure.

[source](#)

[Base.Libdl.dlclose](#) – Function.

| `dlclose(handle)`

Close shared library referenced by handle.

[source](#)

[Base.Libdl.dlext](#) – Constant.

| `dlext`

File extension for dynamic libraries (e.g. `dll`, `dylib`, `so`) on the current platform.

[source](#)

[Base.Libdl.find\\_library](#) – Function.

| `find_library(names, locations)`

Searches for the first library in `names` in the paths in the `locations` list, `DL_LOAD_PATH`, or system library paths (in that order) which can successfully be `dlopen`'d. On success, the return value will be one of the names (potentially prefixed by one of the paths in `locations`). This string can

be assigned to a `global const` and used in `ccall's`. On failure, it returns the empty string.

[source](#)

`Base.Libdl.DL_LOAD_PATH` – Constant.

`| DL_LOAD_PATH`

When calling `dlopen`, the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

[source](#)

# Chapter 69

## Profiling

`Profile.@profile` – Macro.

|  
| @profile

`@profile <expression>` runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

`source`

The methods in `Profile` are not exported and need to be called e.g. as `Profile.print()`.

`Profile.clear` – Function.

|  
| clear()

Clear any existing backtraces from the internal buffer.

`source`

`Profile.print` – Function.

|  
| print([io:::IO = STDOUT,] [data:::Vector]; kwargs...)

Prints profiling results to `io` (by default, `STDOUT`). If you do not supply a `data` vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

1602    **format** – Determines whether backtraces are printed with PROFOUNDING, :tree) or without (:flat) indentation indicating tree structure.

C – If true, backtraces from C and Fortran code are shown (normally they are excluded).

combine – If true (default), instruction pointers are merged that correspond to the same line of code.

maxdepth – Limits the depth higher than maxdepth in the :tree format.

sortedby – Controls the order in :flat format. :filefuncline (default) sorts by the source line, whereas :count sorts in order of number of collected samples.

noisefloor – Limits frames that exceed the heuristic noise floor of the sample (only applies to format :tree). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which  $n \leq noisefloor * \sqrt{N}$ , where n is the number of samples on this line, and N is the number of samples for the callee.

mincount – Limits the printout to only those lines with at least mincount occurrences.

#### source

```
| print([io::IO = STDOUT,] data::Vector, lidict::LineInfoDict;
|       kwargs...)
```

Prints profiling results to io. This variant is used to examine results exported by a previous call to [retrieve](#). Supply the vector data of backtraces and a dictionary lidict of line information.

See [Profile.print\(\[io\], data\)](#) for an explanation of the valid keyword arguments.

#### source

## [Profile.init](#) – Function.

1603

```
| init(; n::Integer, delay::Float64)
```

Configure the `delay` between backtraces (measured in seconds), and the number `n` of instruction pointers that may be stored. Each instruction pointer corresponds to a single line of code; backtraces generally consist of a long list of instruction pointers. Default settings can be obtained by calling this function with no arguments, and each can be set independently using keywords or in the order (`n, delay`).

[source](#)

## [Profile.fetch](#) – Function.

```
| fetch() -> data
```

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `clear`, can affect `data` unless you first make a copy. Note that the values in `data` have meaning only on this machine in the current session, because it depends on the exact memory addresses used in JIT-compiling. This function is primarily for internal use; `retrieve` may be a better choice for most users.

[source](#)

## [Profile.retrieve](#) – Function.

```
| retrieve() -> data, lidict
```

“Exports” profiling results in a portable format, returning the set of all backtraces (`data`) and a dictionary that maps the (session-specific) instruction pointers in `data` to `LineInfo` values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

[source](#)

```
| callers(funcname, [data, lidict], [filename=<filename>], [  
|   linerange=<start:stop>]) -> Vector{Tuple{count, lineinfo}}
```

Given a previous profiling run, determine who called a particular function. Supplying the filename (and optionally, range of line numbers over which the function is defined) allows you to disambiguate an overloaded method. The returned value is a vector containing a count of the number of calls and line information about the caller. One can optionally supply backtrace `data` obtained from `retrieve`; otherwise, the current internal profile buffer is used.

`source`

```
| clear_malloc_data()
```

Clears any stored memory allocation data when running julia with `--track-allocation`. Execute the command(s) you want to test (to force JIT-compilation), then call `clear_malloc_data`. Then execute your command(s) again, quit Julia, and examine the resulting `*.mem` files.

`source`

# Chapter 70

## StackTraces

[Base.StackTraces.StackFrame](#) – Type.

### StackFrame

Stack information representing execution context, with the following fields:

#### func::Symbol

The name of the function containing the execution context.

#### linfo::Union{Core.MethodInstance, CodeInfo, Void}

The MethodInstance containing the execution context (if it could be found).

#### file::Symbol

The path to the file containing the execution context.

#### line::Int

The line number in the file containing the execution context.

#### from\_c::Bool

True if the code is from C.

#### inlined::Bool

True if the code is from an inlined frame.

1606 `pointer::UInt64`

CHAPTER 70. STACKTRACES

Representation of the pointer to the execution context as returned by `backtrace`.

`source`

`Base.StackTraces.StackTrace` – Type.

`| StackTrace`

An alias for `Vector{StackFrame}` provided for convenience; returned by calls to `stacktrace` and `catch_stacktrace`.

`source`

`Base.StackTraces.stacktrace` – Function.

`| stacktrace([trace::Vector{Ptr{Void}}], [c_funcs::Bool=false])`  
`-> StackTrace`

Returns a stack trace in the form of a vector of `StackFrames`. (By default `stacktrace` doesn't return C functions, but this can be enabled.) When called without specifying a trace, `stacktrace` first calls `backtrace`.

`source`

`Base.StackTraces.catch_stacktrace` – Function.

`| catch_stacktrace([c_funcs::Bool=false]) -> StackTrace`

Returns the stack trace for the most recent error thrown, rather than the current execution context.

`source`

The following methods and types in `Base.StackTraces` are not exported and need to be called e.g. as `StackTraces.lookup(ptr)`.

`Base.StackTraces.lookup` – Function.

`lookup(pointer::Union<Ptr{Void}, UInt}) -> Vector{StackFrame}`<sup>1607</sup>

Given a pointer to an execution context (usually generated by a call to `backtrace`), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

`source`

`Base.StackTraces.remove_frames!` – Function.

`remove_frames!(stack::StackTrace, name::Symbol)`

Takes a `StackTrace` (a vector of `StackFrames`) and a function name (a `Symbol`) and removes the `StackFrame` specified by the function name from the `StackTrace` (also removing all frames above the specified function). Primarily used to remove `StackTraces` functions from the `StackTrace` prior to returning it.

`source`

`remove_frames!(stack::StackTrace, m::Module)`

Returns the `StackTrace` with all `StackFrames` from the provided `Module` removed.

`source`



# Chapter 71

## SIMD Support

Type `VecElement{T}` is intended for building libraries of SIMD operations. Practical use of it requires using `llvmcall`. The type is defined as:

```
struct VecElement{T}
    value::T
end
```

It has a special compilation rule: a homogeneous tuple of `VecElement{T}` maps to an LLVM `vector` type when `T` is a primitive bits type and the tuple length is in the set {2-6,8-10,16}.

At `-O3`, the compiler might automatically vectorize operations on such tuples. For example, the following program, when compiled with `julia -O3` generates two SIMD addition instructions (`addps`) on x86 systems:

```
const m128 = NTuple{4,VecElement{Float32}}}

function add(a::m128, b::m128)
    (VecElement(a[1].value+b[1].value),
     VecElement(a[2].value+b[2].value),
     VecElement(a[3].value+b[3].value),
     VecElement(a[4].value+b[4].value))
```

1610  
**end**

## CHAPTER 71. SIMD SUPPORT

```
triple(c::m128) = add(add(c,c),c)
```

```
code_native(triple,(m128,))
```

However, since the automatic vectorization cannot be relied upon, future use will mostly be via libraries that use `llvmcall`.

# Chapter 72

## Base64

[Base64](#).[Base64EncodePipe](#) – Type.

```
| Base64EncodePipe(ostream)
```

Return a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to `ostream`. Calling `close` on the `Base64EncodePipe` stream is necessary to complete the encoding (but does not close `ostream`).

Examples

```
julia> io = IOBuffer();  
  
julia> iob64_encode = Base64EncodePipe(io);  
  
julia> write(iob64_encode, "Hello!")  
6  
  
julia> close(iob64_encode);  
  
julia> str = String(take!(io))  
"SGVsbG8h"
```

```
1612 julia> String(base64decode(str))  
      "Hello!"
```

CHAPTER 72. BASE64

[source](#)

[Base64.base64encode](#) – Function.

```
base64encode(writefunc, args...)  
base64encode(args...)
```

Given a `write`-like function `writefunc`, which takes an I/O stream as its first argument, `base64encode(writefunc, args...)` calls `writefunc` to write `args...` to a base64-encoded string, and returns the string. `base64encode(args...)` is equivalent to `base64encode(write, args...)`: it converts its arguments into bytes using the standard `write` functions and returns the base64-encoded string.

See also [base64decode](#).

[source](#)

[Base64.Base64DecodePipe](#) – Type.

```
Base64DecodePipe(istream)
```

Return a new read-only I/O stream, which decodes base64-encoded data read from `istream`.

Examples

```
julia> io = IOBuffer();  
  
julia> iob64_decode = Base64DecodePipe(io);  
  
julia> write(io, "SGVsbG8h")
```

```
julia> seekstart(io);  
  
julia> String(read(iob64_decode))  
"Hello!"
```

[source](#)

[Base64.base64decode](#) – Function.

```
base64decode(string)
```

Decode the base64-encoded `string` and returns a `Vector{UInt8}` of the decoded bytes.

See also [base64encode](#).

Examples

```
julia> b = base64decode("SGVsbG8h")  
6-element Array{UInt8,1}:  
 0x48  
 0x65  
 0x6c  
 0x6c  
 0x6f  
 0x21  
  
julia> String(b)  
"Hello!"
```

[source](#)



# Chapter 73

## Memory-mapped I/O

[Mmap.Anonymous](#) – Type.

```
Mmap.Anonymous(name, readonly, create)
```

Create an `I0`-like object for creating zeroed-out mmapped-memory that is not tied to a file for use in `Mmap.mmap`. Used by `SharedArray` for creating shared memory arrays.

[source](#)

[Mmap.mmap](#) – Function.

```
Mmap.mmap(io::Union{IOStream,AbstractString,Mmap.AnonymousMmap
}[, type::Type{Array{T,N}}, dims, offset]; grow::Bool=true,
shared::Bool=true)
Mmap.mmap(type::Type{Array{T,N}}, dims)
```

Create an `Array` whose values are linked to a file, using memory-mapping. This provides a convenient way of working with data too large to fit in the computer's memory.

The type is an `Array{T,N}` with a bits-type element of `T` and dimension `N` that determines how the bytes of the array are interpreted. Note that

the file must be stored in binary format. CHAPTER 7.30. MEMORY MAPPED FILE

possible (this is a limitation of operating systems, not Julia).

`dims` is a tuple or single `Integer` specifying the size or length of the array.

The file is passed via the `stream` argument, either as an open `I0Stream` or filename string. When you initialize the stream, use "`r`" for a "read-only" array, and "`w+`" to create a new array used to write values to disk.

If no `type` argument is specified, the default is `Vector{UInt8}`.

Optionally, you can specify an offset (in bytes) if, for example, you want to skip over a header in the file. The default value for the offset is the current stream position for an `I0Stream`.

The `grow` keyword argument specifies whether the disk file should be grown to accommodate the requested size of array (if the total file size is < requested array size). Write privileges are required to grow the file.

The `shared` keyword argument specifies whether the resulting `Array` and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```
# Create a file for mmaping
# (you could alternatively use mmap to do this step, too)
A = rand(1:20, 5, 30)
s = open("/tmp/mmap.bin", "w+")
# We'll write the dimensions of the array as the first two Ints
# in the file
write(s, size(A,1))
write(s, size(A,2))
# Now write the data
write(s, A)
```

`close(s)`

1617

```
# Test by reading it back in
s = open("/tmp/mmap.bin")    # default is read-only
m = read(s, Int)
n = read(s, Int)
A2 = Mmap.mmap(s, Matrix{Int}, (m,n))
```

creates a `m`-by-`n` `Matrix{Int}`, linked to the file associated with stream `s`.

A more portable file would need to encode the word size – 32 bit or 64 bit – and endianness information in the header. In practice, consider encoding binary data using standard formats like HDF5 (which can be used with memory-mapping).

`source`

```
Mmap.mmap(io, BitArray, [dims, offset])
```

Create a `BitArray` whose values are linked to a file, using memory-mapping; it has the same purpose, works in the same way, and has the same arguments, as `mmap`, but the byte representation is different.

Example: `B = Mmap.mmap(s, BitArray, (25,30000))`

This would create a 25-by-30000 `BitArray`, linked to the file associated with stream `s`.

`source`

`Mmap.sync!` – Function.

```
Mmap.sync!(array)
```

Forces synchronization between the in-memory version of a memory-mapped `Array` or `BitArray` and the on-disk version.



# Chapter 74

## Shared Arrays

`SharedArrays.SharedArray` – Type.

```
| SharedArray{T}(dims::NTuple; init=false, pids=Int[])
| SharedArray{T,N}(...)
```

Construct a `SharedArray` of a bits type `T` and size `dims` across the processes specified by `pids` – all of which have to be on the same host. If `N` is specified by calling `SharedArray{T,N}(dims)`, then `N` must match the length of `dims`.

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindexes` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the `SharedArray` object exists on the node which created the mapping.

```
| SharedArray{T}(filename::AbstractString, dims::NTuple, [offset
|   =0]; mode=nothing, init=false, pids=Int[])
```

1620 `SharedArray{T,N}(...)`

## CHAPTER 74. SHARED ARRAYS

Construct a `SharedArray` backed by the file `filename`, with element type `T` (must be a `bits` type) and size `dims`, across the processes specified by `pids` - all of which have to be on the same host. This file is mmaped into the host memory, with the following consequences:

The array data must be represented in binary format (e.g., an ASCII format like CSV cannot be supported)

Any changes you make to the array values (e.g., `A[3] = 0`) will also change the values on disk

If `pids` is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, `localindexes` and `indexpids` will only refer to worker processes. This facilitates work distribution code to use workers for actual computation with the master process acting as a driver.

`mode` must be one of "`r`", "`r+`", "`w+`", or "`a+`", and defaults to "`r+`" if the file specified by `filename` already exists, or "`w+`" if not. If an `init` function of the type `initfn(S::SharedArray)` is specified, it is called on all the participating workers. You cannot specify an `init` function if the file is not writable.

`offset` allows you to skip the specified number of bytes at the beginning of the file.

`source`

[Base.Distributed.procs](#) – Method.

| `procs(S::SharedArray)`

Get the vector of processes mapping the shared array.

`source`

`SharedArrays.sdata` – Function.

1621

```
| sdata(S::SharedArray)
```

Returns the actual `Array` object backing `S`.

`source`

`SharedArrays.indexpids` – Function.

```
| indexpids(S::SharedArray)
```

Returns the current worker's index in the list of workers mapping the `SharedArray` (i.e. in the same list returned by `procs(S)`), or 0 if the `SharedArray` is not mapped locally.

`source`

`SharedArrays.localindexes` – Function.

```
| localindexes(S::SharedArray)
```

Returns a range describing the "default" indexes to be handled by the current process. This range should be interpreted in the sense of linear indexing, i.e., as a sub-range of `1:length(S)`. In multi-process contexts, returns an empty range in the parent process (or any process for which `indexpids` returns 0).

It's worth emphasizing that `localindexes` exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a `SharedArray`, all indexes should be equally fast for each worker process.

`source`



# Chapter 75

## File Events

[FileWatching.poll\\_fd](#) – Function.

```
| poll_fd(fd, timeout_s::Real=-1; readable=false, writable=false)
```

Monitor a file descriptor **fd** for changes in the read or write availability, and with a timeout given by **timeout\_s** seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to **true**.

The returned value is an object with boolean fields **readable**, **writable**, and **timedout**, giving the result of the polling.

[source](#)

[FileWatching.poll\\_file](#) – Function.

```
| poll_file(path::AbstractString, interval_s::Real=5.007,  
|           timeout_s::Real=-1) -> (previous::StatStruct, current)
```

Monitor a file for changes by polling every **interval\_s** seconds until a change occurs or **timeout\_s** seconds have elapsed. The **interval\_s** should be a long period; the default is 5.007 seconds.

Returns a pair of status objects (**previous**, **current**) when a change is detected. The **previous** status is always a **StatStruct**, but it may

1624 have all of the fields zeroed (indicating the file CHAPTER 75 previously existed, wasn't previously accessible).

The `current` status object may be a `StatStruct`, an `E0FError` (indicating the timeout elapsed), or some other `Exception` subtype (if the `stat` operation failed - for example, if the path does not exist).

To determine when a file was modified, compare `current isa StatStruct && mtime(prev) != mtime(current)` to detect notification of changes. However, using `watch_file` for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

`source`

[FileWatching.watch\\_file](#) – Function.

```
| watch_file(path::AbstractString, timeout_s::Real=-1)
```

Watch file or directory `path` for changes until a change occurs or `timeout_s` seconds have elapsed.

The returned value is an object with boolean fields `changed`, `renamed`, and `timedout`, giving the result of watching the file.

This behavior of this function varies slightly across platforms. See [https://nodejs.org/api/fs.html#fs\\_caveats](https://nodejs.org/api/fs.html#fs_caveats) for more detailed information.

`source`

# Chapter 76

## CRC32c

[CRC32c.crc32c](#) – Function.

```
|crc32c(data, crc::UInt32=0x00000000)
```

Compute the CRC-32c checksum of the given `data`, which can be an `Array{UInt8}`, a contiguous subarray thereof, or a `String`. Optionally, you can pass a starting `crc` integer to be mixed in with the checksum. The `crc` parameter can be used to compute a checksum on data divided into chunks: performing `crc32c(data2, crc32c(data1))` is equivalent to the checksum of `[data1; data2]`. (Technically, a little-endian checksum is computed.)

There is also a method `crc32c(io, nb, crc)` to checksum `nb` bytes from a stream `io`, or `crc32c(io, crc)` to checksum all the remaining bytes. Hence you can do `open(crc32c, filename)` to checksum an entire file, or `crc32c(seekstart(buf))` to checksum an `IOBuffer` without calling `take!`.

For a `String`, note that the result is specific to the UTF-8 encoding (a different checksum would be obtained from a different Unicode encoding). To checksum an `a::Array` of some other bitstype, you can do `crc32c(reinterpret(UInt8,a))`, but note that the result may be

[source](#)

[CRC32c.crc32c](#) – Method.

```
|crc32c(io::IO, [nb::Integer,] crc::UInt32=0x00000000)
```

Read up to **nb** bytes from **io** and return the CRC-32c checksum, optionally mixed with a starting **crc** integer. If **nb** is not supplied, then **io** will be read until the end of the stream.

[source](#)

## Part V

# Developer Documentation



# Chapter 77

## Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

### 77.1 Module bindings

The exported names for a `Module` are available using `names(m::Module)`, which will return an array of `Symbol` elements representing the exported bindings. `names(m::Module, true)` returns symbols for all bindings in `m`, regardless of export status.

### 77.2 DataType fields

The names of `DataType` fields may be interrogated using `fieldnames`. For example, given the following type, `fieldnames(Point)` returns an arrays of `Symbol` elements representing the field names:

```
julia> struct Point
```

```
    x::Int
```

```
    y
```

```
1630     end
```

## CHAPTER 77. REFLECTION AND INTROSPECTION

```
julia> fieldnames(Point)
2-element Array{Symbol,1}:
 :x
 :y
```

The type of each field in a `Point` object is stored in the `types` field of the `Point` variable itself:

```
julia> Point.types
svec(Int64, Any)
```

While `x` is annotated as an `Int`, `y` was unannotated in the type definition, therefore `y` defaults to the `Any` type.

Types are themselves represented as a structure called `DataType`:

```
julia> typeof(Point)
DataType
```

Note that `fieldnames(DataType)` gives the names for each field of `DataType` itself, and one of these fields is the `types` field observed in the example above.

### 77.3 Subtypes

The direct subtypes of any `DataType` may be listed using `subtypes`. For example, the abstract `DataType` `AbstractFloat` has four (concrete) subtypes:

```
julia> subtypes(AbstractFloat)
4-element Array{Union{DataType, UnionAll},1}:
 BigFloat
```

```
|   Float16
```

```
|   Float32
```

```
|   Float64
```

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of [subtypes](#) may be used to inspect the full type tree.

## 77.4 DataType layout

The internal representation of a `DataType` is critically important when interfacing with C code and several functions are available to inspect these details. `isbits(T::DataType)` returns true if `T` is stored with C-compatible alignment. `fieldoffset(T::DataType, i::Integer)` returns the (byte) offset for field `i` relative to the start of the type.

## 77.5 Function methods

The methods of any generic function may be listed using [methods](#). The method dispatch table may be searched for methods accepting a given type using [methodswith](#).

## 77.6 Expansion and lowering

As discussed in the [Metaprogramming](#) section, the `macroexpand` function gives the unquoted and interpolated expression (`Expr`) form for a given macro. To use `macroexpand`, quote the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

```
julia> macroexpand(@__MODULE__, :((@edit println("")))
:(((Base.edit)(println, (Base.typesof)(""))))
```

T632functions Base.Meta.show<sup>CHAPTER 77</sup>~~CHAPTER 77~~REFLECTION AND INSPECTION  
style views and depth-nested detail views for any expression.

Finally, the `Meta.lower` function gives the lowered form of any expression and is of particular interest for understanding both macros and top-level statements such as function declarations and variable assignments:

```
julia> Meta.lower(@__MODULE__, :(f() = 1) )  
:(begin  
    end), false)  
end)
```

## 77.7 Intermediate and compiled representations

Inspecting the lowered form for functions requires selection of the specific method to display, because generic functions may have many methods with different type signatures. For this purpose, method-specific code-lowering is available using `code_lowered(f::Function, (Argtypes...))`, and the type-inferred form is available using `code_typed(f::Function, (Argtypes...))`. `code_warntype(f::Function, (Argtypes...))` adds highlighting to the output of `code_typed` (see `@code_warntype`).

Closer to the machine, the LLVM intermediate representation of a function may be printed using by `code_llvm(f::Function, (Argtypes...))`, and finally the compiled machine code is available using `code_native(f::Function, (Argtypes...))` (this will trigger JIT compilation/code generation for any function which has not previously been called).

For convenience, there are macro versions of the above functions which take standard function calls and expand argument types automatically:

```
julia> @code_llvm +(1,1)
```

```
; Function Attrs: sspreq
define i64 @"julia_+_130862"(i64, i64) #0 {
top:
    %2 = add i64 %1, %0, !dbg !8
    ret i64 %2, !dbg !8
}
```

(likewise @code\_typed, @code\_warntype, @code\_lowered, and @code\_native)



# Chapter 78

## Documentation of Julia's Internals

### 78.1 Initialization of the Julia runtime

How does the Julia runtime execute `julia -e 'println("Hello World!")'`?

`main()`

Execution starts at `main()` in `ui/repl.c`.

`main()` calls `libsupport_init()` to set the C library locale and to initialize the "ios" library (see `ios_init_stdstreams()` and [Legacy ios.c library](#)).

Next `parse_opts()` is called to process command line options. Note that `parse_opts()` only deals with options that affect code generation or early initialization. Other options are handled later by `process_options()` in `base/client.jl`.

`parse_opts()` stores command line options in the `global jl_options struct`.

`julia_init()`

`julia_init()` in `task.c` is called by `main()` and calls `_julia_init()` in `init.c`.

**160 `julia_init()` begins** CHAPTER 18 DOCUMENTATION (Flagged as INTERNALS)  
ing the second time).

`restore_signals()` is called to zero the signal handler mask.

`jl_resolve_sysimg_location()` searches configured paths for the base system image. See [Building the Julia system image](#).

`jl_gc_init()` sets up allocation pools and lists for weak refs, preserved values and finalization.

`jl_init_frontend()` loads and initializes a pre-compiled femtolisp image containing the scanner/parser.

`jl_init_types()` creates `jl_datatype_t` type description objects for the built-in types defined in `julia.h`. e.g.

```
jl_any_type = jl_new_abstracttype(jl_symbol("Any"), core, NULL,
    jl_emptyvec);
jl_any_type->super = jl_any_type;

jl_type_type = jl_new_abstracttype(jl_symbol("Type"), core,
    jl_any_type, jl_emptyvec);

jl_int32_type = jl_new_primitivetype(jl_symbol("Int32"), core,
    jl_any_type, jl_emptyvec,
    32);
```

`jl_init_tasks()` creates the `jl_datatype_t* jl_task_type` object; initializes the global `jl_root_task` struct; and sets `jl_current_task` to the root task.

`jl_init_codegen()` initializes the [LLVM library](#).

`jl_init_serializer()` initializes 8-bit serialization tags for builtin `jl_value_t` values.

If `_jl_init()` has already been run (e.g. if `jl_is_initialized()` returns true) then the `Core` and `Main` modules are created and `boot.jl` is evaluated:

`jl_core_module = jl_new_module(jl_symbol("Core"))` creates the Julia Core module.

`jl_init_intrinsic_functions()` creates a new Julia module `Intrinsics` containing constant `jl_intrinsic_type` symbols. These define an integer code for each intrinsic function. `emit_intrinsic()` translates these symbols into LLVM instructions during code generation.

`jl_init_primitives()` hooks C functions up to Julia function symbols. e.g. the symbol `Core.:(==)()` is bound to C function pointer `jl_f_is()` by calling `add_builtin_func("==", jl_f_is)`.

`jl_new_main_module()` creates the global "Main" module and sets `jl_current_task->current_module = jl_main_module`.

Note: `_julia_init()` then sets `jl_root_task->current_module = jl_core_module`. `jl_root_task` is an alias of `jl_current_task` at this point, so the `current_module` set by `jl_new_main_module()` above is overwritten.

`jl_load("boot.jl", sizeof("boot.jl"))` calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute `boot.jl`. <!-- TODO -->  
– drill down into eval? –>

`jl_get_builtin_hooks()` initializes global C pointers to Julia globals defined in `boot.jl`.

`jl_init_box_caches()` pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
| jl_value_t *jl_box_uint8(uint32_t x)
| {
|     return boxed_uint8_cache[(uint8_t)x];
| }
```

[1638](#) `ia_init()` iterates over the `jl_datatype_t` values and sets the type name's module prefix to `jl_core_module`.

`jl_add_standard_imports(jl_main_module)` does "using Base" in the "Main" module.

Note: `_julia_init()` now reverts to `jl_root_task->current_module = jl_main_module` as it was before being set to `jl_core_module` above.

Platform specific signal handlers are initialized for `SIGSEGV` (OSX, Linux), and `SIGFPE` (Windows).

Other signals (`SIGINFO`, `SIGBUS`, `SIGILL`, `SIGTERM`, `SIGABRT`, `SIGQUIT`, `SIGSYS` and `SIGPIPE`) are hooked up to `sigdie_handler()` which prints a backtrace.

`jl_init_restored_modules()` calls `jl_module_run_initializer()` for each deserialized module to run the `__init__()` function.

Finally `sigint_handler()` is hooked up to `SIGINT` and calls `jl_throw(jl_interrupt_exception)`.

`_julia_init()` then returns back to `main()` in `ui/repl.c` and `main()` calls `true_main(argc, (char**)argv)`.

## sysimg

If there is a sysimg file, it contains a pre-cooked image of the `Core` and `Main` modules (and whatever else is created by `boot.jl`). See [Building the Julia system image](#).

`jl_restore_system_image()` deserializes the saved sysimg into the current Julia runtime environment and initialization continues after `jl_init_box_cach` below...

Note: `jl_restore_system_image()` (and `staticdata.c` in general) uses the [Legacy ios.c library](#).

`true_main()` loads the contents of `argv[ ]` into `Base.ARGS`.

If a `.jl` "program" file was supplied on the command line, then `exec_program()` calls `jl_load(program, len)` which calls `jl_parse_eval_all` which repeatedly calls `jl_toplevel_eval_flex()` to execute the program.

However, in our example (`julia -e 'println("Hello World!")'`), `jl_get_global(module, jl_symbol("_start"))` looks up `Base._start` and `jl_apply()` executes it.

`Base._start`

`Base._start` calls `Base.process_options` which calls `jl_parse_input_line("println World!")` to create an expression object and `Base.eval()` to execute it.

`Base.eval`

`Base.eval()` was mapped to `jl_f_top_eval` by `jl_init_primitives()`.

`jl_f_top_eval()` calls `jl_toplevel_eval_in(jl_main_module, ex)`, where `ex` is the parsed expression `println("Hello World!")`.

`jl_toplevel_eval_in()` calls `jl_toplevel_eval_flex()` which calls `eval()` in `interpreter.c`.

The stack dump below shows how the interpreter works its way through various methods of `Base.println()` and `Base.print()` before arriving at `write(s::IO, a::Array{T})` where `T` which does `ccall(jl_uv_write())`.

`jl_uv_write()` calls `uv_write()` to write "Hello World!" to `JL_STDOUT`. See [Libuv wrappers for stdio](#):

| Hello World!

Since our example has just one function call, which has done its job of printing "Hello World!", the stack now rapidly unwinds back to `main()`.

## `jl_640exit_hook()` CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

`main()` calls `jl_atexit_hook()`. This calls `_atexit` for each module, then calls `jl_gc_run_all_finalizers()` and cleans up libuv handles.

### `julia_save()`

Finally, `main()` calls `julia_save()`, which if requested on the command line, saves the runtime state to a new system image. See `jl_compile_all()` and `jl_save_system_image()`.

## 78.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the `parse` function), and manipulated by macros. It is a structured representation of code as it is written, constructed by `julia-parser.scm` from a character stream. Next there is a lowered form, or IR (intermediate representation), which is used by type inference and code generation. In the lowered form there are fewer types of nodes, all macros are expanded, and all control flow is converted to explicit branches and sequences of statements. The lowered form is constructed by `julia-syntax.scm`.

First we will focus on the lowered form, since it is more important to the compiler. It is also less obvious to the human, since it results from a significant rearrangement of the input syntax.

### Lowered form

The following data types exist in lowered form:

#### `Expr`

Has a node type indicated by the `head` field, and an `args` field which is a `Vector{Any}` of subexpressions.

Identifies arguments and local variables by consecutive numbering. **Slot** is an abstract type with subtypes **SlotNumber** and **TypedSlot**. Both types have an integer-valued **id** field giving the slot index. Most slots have the same type at all uses, and so are represented with **SlotNumber**. The types of these slots are found in the **slottypes** field of their **MethodInstance** object. Slots that require per-use type annotations are represented with **TypedSlot**, which has a **typ** field.

### **CodeInfo**

Wraps the IR of a method.

### **LineNumberNode**

Contains a single number, specifying the line number the next statement came from.

### **LabelNode**

Branch target, a consecutively-numbered integer starting at 0.

### **GotoNode**

Unconditional branch.

### **QuoteNode**

Wraps an arbitrary value to reference as data. For example, the function `f() = :a` contains a **QuoteNode** whose **value** field is the symbol `a`, in order to return the symbol itself instead of evaluating it.

### **GlobalRef**

Refers to global variable `name` in module `mod`.

### **SSAValue**

Refers to a consecutively-numbered (starting at 0) static single assignment (SSA) variable inserted by the compiler.

Marks a point where a variable is created. This has the effect of resetting a variable to undefined.

## Expr types

These symbols appear in the `head` field of `Exprs` in lowered form.

### `call`

Function call (dynamic dispatch). `args[1]` is the function to call, `args[2:end]` are the arguments.

### `invoke`

Function call (static dispatch). `args[1]` is the `MethodInstance` to call, `args[2:end]` are the arguments (including the function that is being called, at `args[2]`).

### `static_parameter`

Reference a static parameter by index.

### `gotoifnot`

Conditional branch. If `args[1]` is false, goes to label identified in `args[2]`.

=

Assignment.

### `method`

Adds a method to a generic function and assigns the result if necessary.

Has a 1-argument form and a 4-argument form. The 1-argument form arises from the syntax `function foo end`. In the 1-argument form, the argument is a symbol. If this symbol already names a function in the current scope, nothing happens. If the symbol is undefined, a new function

78.2 is ~~the AST~~ assigned to the identifier specified by the symbol. If ~~the~~ symbol is defined but names a non-function, an error is raised. The definition of "names a function" is that the binding is constant, and refers to an object of singleton type. The rationale for this is that an instance of a singleton type uniquely identifies the type to add the method to. When the type has fields, it wouldn't be clear whether the method was being added to the instance or its type.

The 4-argument form has the following arguments:

- **args[1]**

A function name, or **false** if unknown. If a symbol, then the expression first behaves like the 1-argument form above. This argument is ignored from then on. When this is **false**, it means a method is being added strictly by type,  $(::T)(x) = x$ .

- **args[2]**

A **SimpleVector** of argument type data. **args[2][1]** is a **SimpleVector** of the argument types, and **args[2][2]** is a **SimpleVector** of type variables corresponding to the method's static parameters.

- **args[3]**

A **CodeInfo** of the method itself. For "out of scope" method definitions (adding a method to a function that also has methods defined in different scopes) this is an expression that evaluates to a **:lambda** expression.

- **args[4]**

**true** or **false**, identifying whether the method is staged (**@generated function**).

**const**

`null`

Has no arguments; simply yields the value `nothing`.

`new`

Allocates a new struct-like object. First argument is the type. The `new` pseudo-function is lowered to this, and the type is always inserted by the compiler. This is very much an internal-only feature, and does no checking. Evaluating arbitrary `new` expressions can easily segfault.

`return`

Returns its argument as the value of the enclosing function.

`the_exception`

Yields the caught exception inside a `catch` block. This is the value of the run time system variable `jl_exception_in_transit`.

`enter`

Enters an exception handler (`setjmp`). `args[1]` is the label of the catch block to jump to on error.

`leave`

Pop exception handlers. `args[1]` is the number of handlers to pop.

`inbounds`

Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (`true` means bounds checks are disabled), it is pushed onto the stack. If the first argument is `:pop`, the stack is popped.

`boundscheck`

Has the value `false` if inlined into a section of code marked with `@in-bounds`, otherwise has the value `true`.

Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.

### meta

Metadata. `args[1]` is typically a symbol specifying the kind of metadata, and the rest of the arguments are free-form. The following kinds of metadata are commonly used:

- `:inline` and `:noinline`: Inlining hints.
- `:push_loc`: enters a sequence of statements from a specified source location.
  - \* `args[2]` specifies a filename, as a symbol.
  - \* `args[3]` optionally specifies the name of an (inlined) function that originally contained the code.
- `:pop_loc`: returns to the source location before the matching `:push_loc`.
  - \* `args[2]::Int` (optional) specifies the number of `push_loc` to pop

### Method

A unique'd container describing the shared metadata for a single method.

#### `name, module, file, line, sig`

Metadata to uniquely identify the method for the computer and the human.

#### `ambig`

Cache of other methods that may be ambiguous with this one.

#### `specializations`

uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.

#### `source`

The original source code (usually compressed).

#### `roots`

Pointers to non-AST things that have been interpolated into the AST, required by compression of the AST, type-inference, or the generation of native code.

#### `nargs, isva, called, isstaged, pure`

Descriptive bit-fields for the source code of this Method.

#### `min_world / max_world`

The range of world ages for which this method is visible to dispatch.

### MethodInstance

A unique'd container describing a single callable signature for a Method. See especially [Proper maintenance and care of multi-threading locks](#) for important details on how to modify these fields safely.

#### `specTypes`

The primary key for this MethodInstance. Uniqueness is guaranteed through a `def.specializations` lookup.

#### `def`

The **Method** that this function describes a specialization of. Or a **Module**, if this is a top-level Lambda expanded in Module, and which is not part of a Method.

The values of the static parameters in `specTypes` indexed by `def.sparam_syms`.

For the `MethodInstance` at `Method.unspecialized`, this is the empty `SimpleVector`. But for a runtime `MethodInstance` from the `MethodTable` cache, this will always be defined and indexable.

### `rettype`

The inferred return type for the `specFunctionObject` field, which (in most cases) is also the computed return type for the function in general.

### `inferred`

May contain a cache of the inferred source for this function, or other information about the inference result such as a constant return value may be put here (if `jlcall_api == 2`), or it could be set to `nothing` to just indicate `rettype` is inferred.

### `ftptr`

The generic `jlcall` entry point.

### `jlcall_api`

The ABI to use when calling `ftptr`. Some significant ones include:

- 0 - Not compiled yet
- 1 - `JL_CALLABLE jl_value_t *(*)(jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
- 2 - Constant (value stored in `inferred`)
- 3 - With Static-parameters forwarded `jl_value_t *(*) (jl_svec_t *sparams, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`
- 4 - Run in interpreter `jl_value_t *(*) (jl_method_instance_t *meth, jl_function_t *f, jl_value_t *args[nargs], uint32_t nargs)`

The range of world ages for which this method instance is valid to be called.

### CodeInfo

A temporary container for holding lowered source code.

#### code

An Any array of statements

#### slotnames

An array of symbols giving the name of each slot (argument or local variable).

#### slottypes

An array of types for the slots.

#### slotflags

A UInt8 array of slot properties, represented as bit flags:

- 2 – assigned (only false if there are no assignment statements with this var on the left)
- 8 – const (currently unused for local variables)
- 16 – statically assigned once
- 32 – might be used before assigned. This flag is only valid after type inference.

#### ssavaluetypes

Either an array or an Int.

If an Int, it gives the number of compiler-inserted temporary locations in the function. If an array, specifies a type for each location.

**inferred**

Whether this has been produced by type inference.

**inlineable**

Whether this should be inlined.

**propagate\_inbounds**

Whether this should propagate `@inbounds` when inlined for the purpose of eliding `@boundscheck` blocks.

**pure**

Whether this is known to be a pure function of its arguments, without respect to the state of the method caches or other mutable global state.

## Surface syntax AST

Front end ASTs consist almost entirely of `Exprs` and atoms (e.g. symbols, numbers). There is generally a different expression head for each visually distinct syntactic form. Examples will be given in s-expression syntax. Each parenthesized list corresponds to an `Expr`, where the first element is the head. For example `(call f x)` corresponds to `Expr(:call, :f, :x)` in Julia.

### Calls

do syntax:

```
f(x) do a,b  
    body  
end
```

parses as `(call f (-> (tuple a b) (block body)) x)`.

Most uses of operators are just function calls, so they are parsed with the head `call`. However some operators are special forms (not necessarily function calls), and in those cases the operator itself is the expression head. In `julia-parser.scm` these are referred to as "syntactic operators". Some operators (+ and \*) use N-ary parsing; chained calls are parsed as a single N-argument call. Finally, chains of comparisons have their own special expression structure.

Bracketed forms

Macros

Strings

Doc string syntax:

```
"some docs"  
f(x) = x
```

parses as `(macrocall (|.| Core '@doc) (line) "some docs" (= (call f x) (block x)))`.

Imports and such

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Block forms

A block of statements is parsed as `(block stmt1 stmt2 ...)`.

If statement:

```
if a  
  b
```

```

elseif c
    d
else
    e
end

```

parses as:

```

(if a (block (line 2) b)
    (elseif (block (line 3) c) (block (line 4) d)
        (block (line 5 e))))

```

A **while** loop parses as (**while** condition body).

A **for** loop parses as (**for** (= var iter) body). If there is more than one iteration specification, they are parsed as a block: (**for** (block (= v1 iter1) (= v2 iter2)) body).

**break** and **continue** are parsed as 0-argument expressions (**break**) and (**continue**).

**let** is parsed as (**let** (= var val) body) or (**let** (block (= var1 val1) (= var2 val2) ...) body), like **for** loops.

A basic function definition is parsed as (**function** (call f x) body). A more complex example:

```

function f(x::T; k = 1) where T
    return x+1
end

```

parses as:

```

(function (where (call f (parameters (kw k 1))
                           (>:: x T)))
                           T)

```

Type definition:

```
mutable struct Foo{T<:S}
    x::T
end
```

parses as:

```
(struct true (curly Foo (<: T S))
    (block (line 2) (:: x T)))
```

The first argument is a boolean telling whether the type is mutable.

`try` blocks parse as (`try try_block var catch_block finally_block`). If no variable is present after `catch`, `var` is `#f`. If there is no `finally` clause, then the last argument is not present.

## Quote expressions

Julia source syntax forms for code quoting (`quote` and `:()`) support interpolation with `$`. In Lisp terminology, this means they are actually "backquote" or "quasiquote" forms. Internally, there is also a need for code quoting without interpolation. In Julia's scheme code, non-interpolating quote is represented with the expression head `inert`.

`inert` expressions are converted to Julia `QuoteNode` objects. These objects wrap a single value of any type, and when evaluated simply return that value.

A `quote` expression whose argument is an atom also gets converted to a `QuoteNode`.

Source location information is represented as `(line line_num file_name)` where the third component is optional (and omitted when the current line number, but not file name, changes).

These expressions are represented as `LineNumberNodes` in Julia.

## 78.3 More about types

If you've used Julia for a while, you understand the fundamental role that types play. Here we try to get under the hood, focusing particularly on [Parametric Types](#).

### Types and sets (and **Any** and **Union{ }/Bottom**)

It's perhaps easiest to conceive of Julia's type system in terms of sets. While programs manipulate individual values, a type refers to a set of values. This is not the same thing as a collection; for example a `Set` of values is itself a single `Set` value. Rather, a type describes a set of possible values, expressing uncertainty about which value we have.

A concrete type `T` describes the set of values whose direct tag, as returned by the `typeof` function, is `T`. An abstract type describes some possibly-larger set of values.

`Any` describes the entire universe of possible values. `Integer` is a subset of `Any` that includes `Int`, `Int8`, and other concrete types. Internally, Julia also makes heavy use of another type known as `Bottom`, which can also be written as `Union{ }`. This corresponds to the empty set.

Julia's types support the standard operations of set theory: you can ask whether `T1` is a "subset" (subtype) of `T2` with `T1 <: T2`. Likewise, you intersect two types using `typeintersect`, take their union with `Union`, and compute a type that contains their union with `typejoin`:

1654 CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

```
julia> typeintersect(Int, Float64)
```

```
Union{}
```

```
julia> Union{Int, Float64}
```

```
Union{Float64, Int64}
```

```
julia> typejoin(Int, Float64)
```

```
Real
```

```
julia> typeintersect(Signed, Union{UInt8, Int8})
```

```
Int8
```

```
julia> Union{Signed, Union{UInt8, Int8}}
```

```
Union{UInt8, Signed}
```

```
julia> typejoin(Signed, Union{UInt8, Int8})
```

```
Integer
```

```
julia> typeintersect(Tuple{Integer, Float64}, Tuple{Int, Real})
```

```
Tuple{Int64, Float64}
```

```
julia> Union{Tuple{Integer, Float64}, Tuple{Int, Real}}
```

```
Union{Tuple{Int64, Real}, Tuple{Integer, Float64}}
```

```
julia> typejoin(Tuple{Integer, Float64}, Tuple{Int, Real})
```

```
Tuple{Integer, Real}
```

While these operations may seem abstract, they lie at the heart of Julia. For example, method dispatch is implemented by stepping through the items in a method list until reaching one for which the type of the argument tuple is a subtype of the method signature. For this algorithm to work, it's important that

7.8.3. MORE ABOUT TYPES  
methods are sorted by their specificity, and that the search begins with the most specific methods. Consequently, Julia also implements a partial order on types; this is achieved by functionality that is similar to `<:`, but with differences that will be discussed below.

## UnionAll types

Julia's type system can also express an iterated union of types: a union of types over all values of some variable. This is needed to describe parametric types where the values of some parameters are not known.

For example, `:obj:Array` has two parameters as in `Array{Int,2}`. If we did not know the element type, we could write `Array{T,2}` where `T`, which is the union of `Array{T,2}` for all values of `T: Union{Array{Int8,2}, Array{Int16,2}, ...}`.

Such a type is represented by a `UnionAll` object, which contains a variable (`T` in this example, of type `TypeVar`), and a wrapped type (`Array{T,2}` in this example).

Consider the following methods:

```
f1(A::Array) = 1
f2(A::Array{Int}) = 2
f3(A::Array{T}) where {T<:Any} = 3
f4(A::Array{Any}) = 4
```

The signature of `f3` is a `UnionAll` type wrapping a tuple type. All but `f4` can be called with `a = [1,2]`; all but `f2` can be called with `b = Any[1,2]`.

Let's look at these types a little more closely:

```
julia> dump(Array)
UnionAll
  var: TypeVar
```

```
1656 name: Symbol TCHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS
```

```
    lb: Core.TypeofBottom Union{}
    ub: Any
  body: UnionAll
    var: TypeVar
  body: Array{T,N} <: DenseArray{T,N}
```

This indicates that `Array` actually names a `UnionAll` type. There is one `UnionAll` type for each parameter, nested. The syntax `Array{Int,2}` is equivalent to `Array{Int}{2}`; internally each `UnionAll` is instantiated with a particular variable value, one at a time, outermost-first. This gives a natural meaning to the omission of trailing type parameters: `Array{Int}` gives a type equivalent to `Array{Int,N}` where `N`.

A `TypeVar` is not itself a type, but rather should be considered part of the structure of a `UnionAll` type. Type variables have lower and upper bounds on their values (in the fields `lb` and `ub`). The symbol `name` is purely cosmetic. Internally, `TypeVars` are compared by address, so they are defined as mutable types to ensure that "different" type variables can be distinguished. However, by convention they should not be mutated.

One can construct `TypeVars` manually:

```
julia> TypeVar(:V, Signed, Real)
Signed<:V<:Real
```

There are convenience versions that allow you to omit any of these arguments except the `name` symbol.

The syntax `Array{T}` where `T<:Integer` is lowered to

```
let T = TypeVar(:T, Integer)
    UnionAll(T, Array{T})
end
```

38.3. MORE ABOUT TYPES To construct a `TypeVar` manually (indeed, this is ~~not~~ to be avoided).

## Free variables

The concept of a free type variable is extremely important in the type system. We say that a variable `V` is free in type `T` if `T` does not contain the `UnionAll` that introduces variable `V`. For example, the type `Array{Array{V}}` where `V<:Integer` has no free variables, but the `Array{V}` part inside of it does have a free variable, `V`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T}}` where `T`, which refers to all homogeneous arrays of arrays. The inner type `Array{T}`, seen by itself, might seem to refer to any kind of array. However, every element of the outer array must have the same array type, so `Array{T}` cannot refer to just any old array. One could say that `Array{T}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `jl_has_free_typevars` in the C API is very important. Types for which it returns true will not give meaningful answers in subtyping and other type functions.

## TypeNames

The following two `Array` types are functionally equivalent, yet print differently:

```
julia> TV, NV = TypeVar(:T), TypeVar(:N)
(T, N)
```

```
julia> Array
Array
```

1658

CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

Julia> Array{TV,NV}

Array{T,N}

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```
julia> dump(Array{Int,1}.name)
TypeName
  name: Symbol Array
  module: Module Core
  names: empty SimpleVector
  wrapper: UnionAll
    var: TypeVar
    body: UnionAll
  cache: SimpleVector
  ...
  linearcache: SimpleVector
  ...
hash: Int64 -7900426068641098781
mt: MethodTable
  name: Symbol Array
  defs: Void nothing
  cache: Void nothing
  max_args: Int64 0
  kwsorter: #undef
  module: Module Core
  : Int64 0
  : Int64 0
```

In this case, the relevant field is `wrapper`, which holds a reference to the

```
julia> pointer_from_objref(Array)
Ptr{Void} @0x00007fcc7de64850

julia> pointer_from_objref(Array.body.body.name.wrapper)
Ptr{Void} @0x00007fcc7de64850

julia> pointer_from_objref(Array{TV,NV})
Ptr{Void} @0x00007fcc80c4d930

julia> pointer_from_objref(Array{TV,NV}.name.wrapper)
Ptr{Void} @0x00007fcc7de64850
```

The `wrapper` field of `Array` points to itself, but for `Array{TV,NV}` it points back to the original definition of the type.

What about the other fields? `hash` assigns an integer to each type. To examine the `cache` field, it's helpful to pick a type that is less heavily used than `Array`. Let's first create our own type:

```
julia> struct MyType{T,N} end

julia> MyType{Int,2}
MyType{Int64,2}

julia> MyType{Float32, 5}
MyType{Float32,5}

julia> MyType.body.body.name.cache
svec(MyType{Int64,2}, MyType{Float32,5}, #undef, #undef, #undef,
→ #undef, #undef, #undef)
```

CHAPTER 7 HAVE DOCUMENTATION OF JULIA'S INTERNALS

The cache is pre-allocated when type parameters are populated.) Consequently, when you instantiate a parametric type, each concrete type gets saved in a type cache. However, instances containing free type variables are not cached.

## Tuple types

Tuple types constitute an interesting special case. For dispatch to work on declarations like `x::Tuple`, the type has to be able to accommodate any tuple. Let's check the parameters:

```
julia> Tuple
Tuple

julia> Tuple.parameters
svec(Vararg{Any,N} where N)
```

Unlike other types, tuple types are covariant in their parameters, so this definition permits `Tuple` to match any type of tuple:

```
julia> typeintersect(Tuple, Tuple{Int,Float64})
Tuple{Int64,Float64}

julia> typeintersect(Tuple{Vararg{Any}}, Tuple{Int,Float64})
Tuple{Int64,Float64}
```

However, if a variadic (`Vararg`) tuple type has free variables it can describe different kinds of tuples:

```
julia> typeintersect(Tuple{Vararg{T} where T}, Tuple{Int,Float64})
Tuple{Int64,Float64}
```

78.3 MORE ABOUT TYPES 1661  
**julia>** typeintersect(Tuple{Vararg{T}}) where T, Tuple{Int, Float64})  
Union{}

Notice that when `T` is free with respect to the `Tuple` type (i.e. its binding `UnionAll` type is outside the `Tuple` type), only one `T` value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `Tuple{()}` is distinct:

```
julia> Tuple{}  
Tuple{()  
  
julia> Tuple{}.parameters  
svec()  
  
julia> typeintersect(Tuple{}, Tuple{Int})  
Union{()}
```

What is the "primary" tuple-type?

```
julia> pointer_from_objref(Tuple)  
Ptr{Void} @0x00007f5998a04370  
  
julia> pointer_from_objref(Tuple{})  
Ptr{Void} @0x00007f5998a570d0  
  
julia> pointer_from_objref(Tuple.name.wrapper)  
Ptr{Void} @0x00007f5998a04370  
  
julia> pointer_from_objref(Tuple{}.name.wrapper)  
Ptr{Void} @0x00007f5998a04370
```

so `Tuple == Tuple{Vararg{Any}}` is indeed the primary type.

Consider the type `Tuple{T, T}` where `T`. A method with this signature would look like:

```
| f(x::T, y::T) where {T} = ...
```

According to the usual interpretation of a `UnionAll` type, this `T` ranges over all types, including `Any`, so this type should be equivalent to `Tuple{Any, Any}`. However, this interpretation causes some practical problems.

First, a value of `T` needs to be available inside the method definition. For a call like `f(1, 1.0)`, it's not clear what `T` should be. It could be `Union{Int, Float64}`, or perhaps `Real`. Intuitively, we expect the declaration `x::T` to mean `T === typeof(x)`. To make sure that invariant holds, we need `typeof(x) === typeof(y) === T` in this method. That implies the method should only be called for arguments of the exact same type.

It turns out that being able to dispatch on whether two values have the same type is very useful (this is used by the promotion system for example), so we have multiple reasons to want a different interpretation of `Tuple{T, T}` where `T`. To make this work we add the following rule to subtyping: if a variable occurs more than once in covariant position, it is restricted to ranging over only concrete types. ("Covariant position" means that only `Tuple` and `Union` types occur between an occurrence of a variable and the `UnionAll` type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `Tuple{T, T}` where `T` can be seen as `Union{Tuple{Int8, Int8}, Tuple{Int16, Int16}, ...}`, where `T` ranges over all concrete types. This gives rise to some interesting subtyping results. For example `Tuple{Real, Real}` is not a subtype of `Tuple{T, T}` where `T`, because it includes some types like `Tuple{Int8, Int16}` where the two elements have different types. Tu-

§8.2 { Real, Read } and  $\text{Tuple}\{T, T\}$  where  $T$  have the non-trivial intersection  $\text{Tuple}\{T, T\}$  where  $T < : \text{Real}$ . However,  $\text{Tuple}\{\text{Real}\}$  is a subtype of  $\text{Tuple}\{T\}$  where  $T$ , because in that case  $T$  occurs only once and so is not diagonal.

Next consider a signature like the following:

```
| f(a::Array{T}, x::T, y::T) where {T} = ...
```

In this case,  $T$  occurs in invariant position inside  $\text{Array}\{T\}$ . That means whatever type of array is passed unambiguously determines the value of  $T$  -- we say  $T$  has an equality constraint on it. Therefore in this case the diagonal rule is not really necessary, since the array determines  $T$  and we can then allow  $x$  and  $y$  to be of any subtypes of  $T$ . So variables that occur in invariant position are never considered diagonal. This choice of behavior is slightly controversial -- some feel this definition should be written as

```
| f(a::Array{T}, x::S, y::S) where {T, S<:T} = ...
```

to clarify whether  $x$  and  $y$  need to have the same type. In this version of the signature they would, or we could introduce a third variable for the type of  $y$  if  $x$  and  $y$  can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
| f(x::Union, y::T) where {T} = ...
```

Consider what this declaration means.  $y$  has type  $T$ .  $x$  then can have either the same type  $T$ , or else be of type `Void`. So all of the following calls should match:

```
| f(1, 1)
| f("", "")
```

| 1664.0, 2.0)

CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

| f(nothing, 1)

| f(nothing, "")

| f(nothing, 2.0)

These examples are telling us something: when `x` is `nothing::Void`, there are no extra constraints on `y`. It is as if the method signature had `y::Any`. This means that whether a variable is diagonal is not a static property based on where it appears in a type. Rather, it depends on where a variable appears when the subtyping algorithm uses it. When `x` has type `Void`, we don't need to use the `T` in `Union{Void, T}`, so `T` does not "occur". Indeed, we have the following type equivalence:

| (Tuple{Union{Void, T}, T} where T) == Union{Tuple{Void, Any},  
| ↳ Tuple{T, T} where T}

### Subtyping diagonal variables

The subtyping algorithm for diagonal variables has two components: (1) identifying variable occurrences, and (2) ensuring that diagonal variables range over concrete types only.

The first task is accomplished by keeping counters `occurs_inv` and `occurs_cov` (in `src/subtype.c`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `occurs_inv == 0 && occurs_cov > 1`.

The second task is accomplished by imposing a condition on a variable's lower bound. As the subtyping algorithm runs, it narrows the bounds of each variable (raising lower bounds and lowering upper bounds) to keep track of the range of variable values for which the subtype relation would hold. When we are done evaluating the body of a `UnionAll` type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a

783 more about types 1665

lower bound could not be a subtype of a concrete type. For example, an abstract type like `AbstractArray` cannot be a subtype of a concrete type, but a concrete type like `Int` can be, and the empty type `Bottom` can be as well. If a lower bound fails this test the algorithm stops with the answer `false`.

For example, in the problem `Tuple{Int, String} <: Tuple{T, T}` where `T`, we derive that this would be true if `T` were a supertype of `Union{Int, String}`. However, `Union{Int, String}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `is_leaf_bound`. Note that this test is slightly different from `jl_is_leaf_type`, since it also returns `true` for `Bottom`. Currently this function is heuristic, and does not catch all possible concrete types. The difficulty is that whether a lower bound is concrete might depend on the values of other type variable bounds. For example, `Vector{T}` is equivalent to the concrete type `Vector{Int}` only if both the upper and lower bounds of `T` equal `Int`. We have not yet worked out a complete algorithm for this.

## Introduction to the internal machinery

Most operations for dealing with types are found in the files `jltypes.c` and `subtype.c`. A good way to start is to watch subtyping in action. Build Julia with `make debug` and fire up Julia within a debugger. [gdb debugging tips](#) has some tips which may be useful.

Because the subtyping code is used heavily in the REPL itself – and hence breakpoints in this code get triggered often – it will be easiest if you make the following definition:

```
julia> function mysubtype(a, b)
```

1666

CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

```
ccall(:jl_breakpoint, Void, (Any,), nothing)

    a <: b

end
```

and then set a breakpoint in `jl_breakpoint`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
|mysubtype(Tuple{Int,Float64}, Tuple{Integer,Real})
```

We can make it more interesting by trying a more complex case:

```
|mysubtype(Tuple{Array{Int,2}, Int8}, Tuple{Array{T}, T} where T)
```

## Subtyping and method sorting

The `type_morespecific` functions are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if `a` is more specific than `b`, then `a` does not equal `b` and `b` is not more specific than `a`.

If `a` is a strict subtype of `b`, then it is automatically considered more specific. From there, `type_morespecific` employs some less formal rules. For example, `subtype` is sensitive to the number of arguments, but `type_morespecific` may not be. In particular, `Tuple{Int,AbstractFloat}` is more specific than `Tuple{Integer}`, even though it is not a subtype. (Of `Tuple{Int,AbstractFloat}` and `Tuple{Integer,Float64}`, neither is more specific than the other.) Likewise, `Tuple{Int,Vararg{Int}}` is not a subtype of `Tuple{Integer}`, but it is considered more specific. However, `more-specific` does get a bonus for length: in particular, `Tuple{Int,Int}` is more specific than `Tuple{Int,Vararg{Int}}`.

In 78.4.1 MEMORY LAYOUT OF JULIA OBJECTS, it can be convenient to ~~define~~ define the function:

```
type_morespecific(a, b) = ccall(:jl_type_morespecific, Cint,  
→ (Any, Any), a, b)
```

which allows you to test whether tuple type **a** is more specific than tuple type **b**.

## 78.4 Memory layout of Julia Objects

### Object layout (jl\_value\_t)

The **jl\_value\_t** struct is the name for a block of memory owned by the Julia Garbage Collector, representing the data associated with a Julia object in memory. Absent any type information, it is simply an opaque pointer:

```
typedef struct jl_value_t* jl_pvalue_t;
```

Each **jl\_value\_t** struct is contained in a **jl\_typetag\_t** struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

```
typedef struct {  
    opaque metadata;  
    jl_value_t value;  
} jl_typetag_t;
```

The type of any Julia object is an instance of a leaf **jl\_datatype\_t** object. The **jl\_typeof()** function can be used to query for it:

```
jl_value_t *jl_typeof(jl_value_t *v);
```

The layout of the object depends on its type. Reflection methods can be used to inspect that layout. A field can be accessed by calling one of the get-field methods:

```
| }668 jl_value_t *jl_get_nth_field_checked(jl_value_t *v, size_t i);  
| jl_value_t *jl_get_field(jl_value_t *o, char *fld);
```

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

```
| jl_value_t *v = value->fieldptr[n];
```

As an example, a "boxed" `uint16_t` is stored as follows:

```
| struct {  
|     opaque metadata;  
|     struct {  
|         uint16_t data;           // -- 2 bytes  
|     } jl_value_t;  
| };
```

This object is created by `jl_box_uint16()`. Note that the `jl_value_t` pointer references the data portion, not the metadata at the top of the struct.

A value may be stored "unboxed" in many circumstances (just the data, without the metadata, and possibly not even stored but just kept in registers), so it is unsafe to assume that the address of a box is a unique identifier. The "egal" test (corresponding to the `==` function in Julia), should instead be used to compare two unknown objects for equivalence:

```
| int jl_egal(jl_value_t *a, jl_value_t *b);
```

This optimization should be relatively transparent to the API, since the object will be "boxed" on-demand, whenever a `jl_value_t` pointer is needed.

Note that modification of a `jl_value_t` pointer in memory is permitted only if the object is mutable. Otherwise, modification of the value may corrupt the program and the result will be undefined. The mutability property of a value can be queried for with:

If the object being stored is a `jl_value_t`, the Julia garbage collector must be notified also:

```
| void jl_gc_wb(jl_value_t *parent, jl_value_t *ptr);
```

However, the [Embedding Julia](#) section of the manual is also required reading at this point, for covering other details of boxing and unboxing various types, and understanding the gc interactions.

Mirror structs for some of the built-in types are [defined in julia.h](#). The corresponding global `jl_datatype_t` objects are created by [jl\\_init\\_types](#) in [jltypes.c](#).

### Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `jl_typetag_t` to track each object in the system. Further details about this algorithm can be found in the comments of the [garbage collector implementation in gc.c](#).

### Object allocation

Most new objects are allocated by `jl_new_struct()`:

```
| jl_value_t *jl_new_struct(jl_datatype_t *type, ...);  
| jl_value_t *jl_new_structv(jl_datatype_t *type, jl_value_t **args,  
|   uint32_t na);
```

Although, `isbits` objects can be also constructed directly from memory:

```
| jl_value_t *jl_new_bits(jl_value_t *bt, void *data)
```

And some objects have special constructors that must be used instead of the above functions:

```

jl_datatype_t *jl_apply_type(jl_datatype_t *tc, jl_tuple_t *params
    );
jl_datatype_t *jl_apply_array_type(jl_datatype_t *type, size_t dim
    );
jl_uniontype_t *jl_new_uniontype(jl_tuple_t *types);

```

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in [julia.h](#). These are used in `jl_init_types()` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

```

jl_tuple_t *jl_tuple(size_t n, ...);
jl_tuple_t *jl_tuplev(size_t n, jl_value_t **v);
jl_tuple_t *jl_alloc_tuple(size_t n);

```

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a [Base.tuple\(\)](#) object may be an array of pointers to the objects contained by the tuple equivalent to:

```

typedef struct {
    size_t length;
    jl_value_t *data[length];
} jl_tuple_t;

```

However, in other cases, the tuple may be converted to an anonymous [isbits](#) type and stored unboxed, or it may not be stored at all (if it is not being used in a generic context as a `jl_value_t*`).

Symbols:

```

jl_sym_t *jl_symbol(const char *str);

```

Functions and MethodInstance:

```
78.4 MEMORY LAYOUT OF JULIA OBJECTS 1671
| jl_function_t *jl_new_generic_function(jl_sym_t *name);
| jl_method_instance_t *jl_new_method_instance(jl_value_t *ast,
|     jl_tuple_t *sparams);
```

Arrays:

```
jl_array_t *jl_new_array(jl_value_t *atype, jl_tuple_t *dims);
jl_array_t *jl_new_arrayv(jl_value_t *atype, ...);
jl_array_t *jl_alloc_array_1d(jl_value_t *atype, size_t nr);
jl_array_t *jl_alloc_array_2d(jl_value_t *atype, size_t nr, size_t
    nc);
jl_array_t *jl_alloc_array_3d(jl_value_t *atype, size_t nr, size_t
    nc, size_t z);
jl_array_t *jl_alloc_vec_any(size_t n);
```

Note that many of these have alternative allocation functions for various special-purposes. The list here reflects the more common usages, but a more complete list can be found by reading the [julia.h header file](#).

Internal to Julia, storage is typically allocated by `newstruct()` (or `newobj()` for the special types):

```
jl_value_t *newstruct(jl_value_t *type);
jl_value_t *newobj(jl_value_t *type, size_t nfields);
```

And at the lowest level, memory is getting allocated by a call to the garbage collector (in `gc.c`), then tagged with its type:

```
jl_value_t *jl_gc_allocobj(size_t nbytes);
void jl_set_typeof(jl_value_t *v, jl_datatype_t *type);
```

Note that all objects are allocated in multiples of 4 bytes and aligned to the platform pointer size. Memory is allocated from a pool for smaller objects, or directly with `malloc()` for large objects.

Singleton Types

instances have a size of 0 bytes, and consist only of their metadata.

e.g. `nothing::Void`.

See [Singleton Types](#) and [Nothingness and missing values](#)

## 78.5 Eval of Julia code

One of the hardest parts about learning how the Julia Language runs code is learning how all of the pieces work together to execute a block of code.

Each chunk of code typically makes a trip through many steps with potentially unfamiliar names, such as (in no particular order): `flisp`, `AST`, `C++`, `LLVM`, `eval`, `typeinf`, `macroexpand`, `sysimg` (or system image), bootstrapping, `compile`, `parse`, `execute`, `JIT`, `interpret`, `box`, `unbox`, intrinsic function, and primitive function, before turning into the desired result (hopefully).

### Definitions

#### REPL

REPL stands for Read-Eval-Print Loop. It's just what we call the command line environment for short.

#### AST

Abstract Syntax Tree The AST is the digital representation of the code structure. In this form the code has been tokenized for meaning so that it is more suitable for manipulation and execution.

### Julia Execution

The 10,000 foot view of the whole process is as follows:

1. The user starts `julia`.

78.5. The `EVALF@EtidHIAQD` from `ui/repl.c` gets called. This function processes the command line arguments, filling in the `jl_options` struct and setting the variable `ARGS`. It then initializes Julia (by calling `julia_init` in `task.c`, which may load a previously compiled `sysimg`). Finally, it passes off control to Julia by calling `Base._start()`.

3. When `_start()` takes over control, the subsequent sequence of commands depends on the command line arguments given. For example, if a filename was supplied, it will proceed to execute that file. Otherwise, it will start an interactive REPL.
4. Skipping the details about how the REPL interacts with the user, let's just say the program ends up with a block of code that it wants to run.
5. If the block of code to run is in a file, `jl_load(char *filename)` gets invoked to load the file and `parse` it. Each fragment of code is then passed to `eval` to execute.
6. Each fragment of code (or AST), is handed off to `eval()` to turn into results.
7. `eval()` takes each code fragment and tries to run it in `jl_toplevel_eval_flex()`.
8. `jl_toplevel_eval_flex()` decides whether the code is a "toplevel" action (such as `using` or `module`), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.
9. `jl_toplevel_eval_flex()` then `expands` the code to eliminate any macros and to "lower" the AST to make it simpler to execute.
10. `jl_toplevel_eval_flex()` then uses some simple heuristics to decide whether to JIT compiler the AST or to interpret it directly.
11. The bulk of the work to interpret code is handled by `eval in interpreter.c`.

gen.cpp. Whenever a Julia function is called for the first time with a given set of argument types, [type inference](#) will be run on that function. This information is used by the [codegen](#) step to generate faster code.

13. Eventually, the user quits the REPL, or the end of the program is reached, and the `_start()` method returns.
14. Just before exiting, `main()` calls `jl_atexit_hook(exit_code)`. This calls `Base._atexit()` (which calls any functions registered to `atexit()` inside Julia). Then it calls `jl_gc_run_all_finalizers()`. Finally, it gracefully cleans up all libuv handles and waits for them to flush and close.

## Parsing

The Julia parser is a small lisp program written in femtolisp, the source-code for which is distributed inside Julia in [src/flisp](#).

The interface functions for this are primarily defined in [jlfrontend.scm](#). The code in [ast.c](#) handles this handoff on the Julia side.

The other relevant files at this stage are [julia-parser.scm](#), which handles tokenizing Julia code and turning it into an AST, and [julia-syntax.scm](#), which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

## Macro Expansion

When `eval()` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `eval()` (in Julia), to the parser function `jl_macroexpand()` (written in flisp) to the Julia macro itself (written in - what else - Julia) via `f1_invoke_julia_macro()`, and back.

Typically, `jl_expander()` is invoked as a first step during a call to `Meta.lower()` or `jl_expand()`, although it can also be invoked directly by a call to `macroexpand()` or `jl_macroexpand()`.

## Type Inference

Type inference is implemented in Julia by `typeinf()` in `inference.jl`. Type inference is the process of examining a Julia function and determining bounds for the types of each of its variables, as well as bounds on the type of the return value from the function. This enables many future optimizations, such as unboxing of known immutable values, and compile-time hoisting of various run-time operations such as computing field offsets and function pointers. Type inference may also include other steps such as constant propagation and inlining.

## More Definitions

### JIT

Just-In-Time Compilation The process of generating native-machine code into memory right when it is needed.

### LLVM

Low-Level Virtual Machine (a compiler) The Julia JIT compiler is a program/library called libLLVM. Codegen in Julia refers both to the process of taking a Julia AST and turning it into LLVM instructions, and the process of LLVM optimizing that and turning it into native assembly instructions.

### C++

The programming language that LLVM is implemented in, which means that codegen is also implemented in this language. The rest of Julia's library is implemented in C, in part because its smaller feature set makes it more usable as a cross-language interface layer.

This term is used to describe the process of taking a value and allocating a wrapper around the data that is tracked by the garbage collector (gc) and is tagged with the object's type.

unbox

The reverse of boxing a value. This operation enables more efficient manipulation of data when the type of that data is fully known at compile-time (through type inference).

generic function

A Julia function composed of multiple "methods" that are selected for dynamic dispatch based on the argument type-signature  
anonymous function or "method"

A Julia function without a name and without type-dispatch capabilities

primitive function

A function implemented in C but exposed in Julia as a named function "method" (albeit without generic function dispatch capabilities, similar to a anonymous function)

intrinsic function

A low-level operation exposed as a function in Julia. These pseudo-functions implement operations on raw bits such as add and sign extend that cannot be expressed directly in any other way. Since they operate on bits directly, they must be compiled into a function and surrounded by a call to `Core.Intrinsics.box(T, ...)` to reassign type information to the value.

## JIT Code Generation

Codegen is the process of turning a Julia AST into native machine code.

The JIT environment is initialized by an early call to `jl_init_codegen` in `codegen.cpp`.

On demand, a Julia method is converted into a native function by the function `emit_function(jl_method_instance_t*)`. (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `emit_expr()` until the entire function has been emitted.

Much of the remaining bulk of this file is devoted to various manual optimizations of specific code patterns. For example, `emit_known_call()` knows how to inline many of the primitive functions (defined in `builtins.c`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

### `debuginfo.cpp`

Handles backtraces for JIT functions

### `ccall.cpp`

Handles the ccall and llvmpcall FFI, along with various `abi_*.cpp` files

### `intrinsics.cpp`

Handles the emission of various low-level intrinsic functions

## Bootstrapping

The process of creating a new system image is called "bootstrapping".

The etymology of this word comes from the phrase "pulling oneself up by the bootstraps", and refers to the idea of starting from a very limited set of available functions and definitions and ending with the creation of a full-featured environment.

The system image is a precompiled archive of a set of Julia files. The `sys.ji` file distributed with Julia is one such system image, generated by executing the file `sysimg.jl`, and serializing the resulting environment (including Types, Functions, Modules, and all other defined values) into a file. Therefore, it contains a frozen version of the `Main`, `Core`, and `Base` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `jl_save_system_image/jl_restore_system_image` in `staticdata.c`.

If there is no sysimg file (`jl_options.image_file == NULL`), this also implies that `--build` was given on the command line, so the final result should be a new sysimg file. During Julia initialization, minimal `Core` and `Main` modules are created. Then a file named `boot.jl` is evaluated from the current directory. Julia then evaluates any file given as a command line argument until it reaches the end. Finally, it saves the resulting environment to a "sysimg" file for use as a starting point for a future Julia run.

## 78.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

### Julia Native Calling Convention

The native calling convention is designed for fast non-generic calls. It usually uses a specialized signature.

LLVM ghosts (zero-length types) are omitted.

LLVM scalars and vectors are passed by value.

LLVM aggregates (arrays and structs) are passed by reference.

## A & High-level Overview of the Native Code Generation Process

returned via the "structure return" (`sret`) convention, where the caller provides a pointer to a return slot.

An argument or return values that is a homogeneous tuple is sometimes represented as an LLVM vector instead of an LLVM array.

### JL Call Convention

The JL Call convention is for builtins and generic dispatch. Hand-written functions using this convention are declared via the macro `JL_CALLABLE`. The convention uses exactly 3 parameters:

`F` – Julia representation of function that is being applied

`args` – pointer to array of pointers to boxes

`nargs` – length of the array

The return value is a pointer to a box.

### C ABI

C ABI wrappers enable calling Julia from C. The wrapper calls a function using the native calling convention.

Tuples are always represented as C arrays.

## 78.7 High-level Overview of the Native-Code Generation Process

### Representation of Pointers

When emitting code to an object file, pointers will be emitted as relocations. The deserialization code will ensure any object that pointed to one of these constants gets recreated and contains the right runtime pointer.

To emit one of these objects, call `literal_pointer_val`. It'll handle tracking the Julia value and the LLVM global, ensuring they are valid both for the current runtime and after deserialization.

When emitted into the object file, these globals are stored as references in a large `gvals` table. This allows the deserializer to reference them by index, and implement a custom manual mechanism similar to a Global Offset Table (GOT) to restore them.

Function pointers are handled similarly. They are stored as values in a large `fvals` table. Like globals, this allows the deserializer to reference them by index.

Note that `extern` functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that `ccall` functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

## Representation of Intermediate Values

Values are passed around in a `jl_cgval_t` struct. This represents an R-value, and includes enough information to determine how to assign or pass it somewhere.

They are created via one of the helper constructors, usually: `mark_julia_type` (for immediate values) and `mark_julia_slot` (for pointers to values).

The function `convert_julia_type` can transform between any two types. It returns an R-value with `cgval.typ` set to `typ`. It'll cast the object to the requested representation, making heap boxes, allocating stack copies, and computing tagged unions as needed to change the representation.

~~BEZONTHIGHLEVELOVERVIEWTYPEWHENATIVECODEGENERATIONHAPPENS~~

be done at zero-cost (i.e. without emitting any code).

## Union representation

Inferred union types may be stack allocated via a tagged type representation.

The primitive routines that need to be able to handle tagged unions are:

mark-type

load-local

store-local

isa

is

emit\_typeof

emit\_sizeof

boxed

unbox

specialized cc-ret

Everything else should be possible to handle in inference by using these primitives to implement union-splitting.

The representation of the tagged-union is as a pair of `< void* union, byte selector >`. The selector is fixed-size as `byte & 0x7f`, and will union-tag the first 126 isbits. It records the one-based depth-first count into the type-union of the isbits objects inside. An index of zero indicates that the `union*` is actually a tagged heap-allocated `jl_value_t*`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector `byte` is actually a heap-allocated (`jl_value_t*`) box, thus avoiding the cost of re-allocating a box, while maintaining the ability to efficiently handle union-splitting based on the low bits.

It is guaranteed that `byte & 0x7f` is an exact test for the type, if the value can be represented by a tag – it will never be marked `byte = 0x80`. It is not necessary to also test the type-tag when testing `isa`.

The `union*` memory region may be allocated at any size. The only constraint is that it is big enough to contain the data currently specified by `selector`. It might not be big enough to contain the union of all types that could be stored there according to the associated Union type field. Use appropriate care when copying.

## Specialized Calling Convention Signature Representation

A `jl_returninfo_t` object describes the calling convention details of any callable.

If any of the arguments or return type of a method can be represented unboxed, and the method is not varargs, it'll be given an optimized calling convention signature based on its `specTypes` and `rettype` fields.

The general principles are that:

Primitive types get passed in int/float registers.

Tuples of VecElement types get passed in vector registers.

Structs get passed on the stack.

Return values are handled similarly to arguments, with a size-cutoff at which they will instead be returned via a hidden `sret` argument.

~~78.8.10 JULIA FUNCTIONS~~ implemented by `get_specsig_function` and ~~1683~~ serves \_sret.

Additionally, if the return type is a union, it may be returned as a pair of values (a pointer and a tag). If the union values can be stack-allocated, then sufficient space to store them will also be passed as a hidden first argument. It is up to the callee whether the returned pointer will point to this space, a boxed object, or even other constant memory.

## 78.8 Julia Functions

This document will explain how functions, method definitions, and method tables work.

### Method Tables

Every function in Julia is a generic function. A generic function is conceptually a single function, but consists of many definitions, or methods. The methods of a generic function are stored in a method table. Method tables (type `MethodTable`) are associated with `TypeNames`. A `TypeName` describes a family of parameterized types. For example `Complex{Float32}` and `Complex{Float64}` share the same `Complex` type name object.

All objects in Julia are potentially callable, because every object has a type, which in turn has a `TypeName`.

### Function calls

Given the call `f(x, y)`, the following steps are performed: first, the method table to use is accessed as `typeof(f).name.mt`. Second, an argument tuple type is formed, `Tuple{typeof(f), typeof(x), typeof(y)}`. Note that the type of the function itself is the first element. This is because the type

1684 have parameters, CHAPTER 7. DOCUMENTATION IS PATHLESS INTERNALS  
is looked up in the method table.

This dispatch process is performed by `jl_apply_generic`, which takes two arguments: a pointer to an array of the values `f`, `x`, and `y`, and the number of values (in this case 3).

Throughout the system, there are two kinds of APIs that handle functions and argument lists: those that accept the function and arguments separately, and those that accept a single argument structure. In the first kind of API, the "arguments" part does not contain information about the function, since that is passed separately. In the second kind of API, the function is the first element of the argument structure.

For example, the following function for performing a call accepts just an `args` pointer, so the first element of the `args` array will be the function to call:

```
| jl_value_t *jl_apply(jl_value_t **args, uint32_t nargs)
```

This entry point for the same functionality accepts the function separately, so the `args` array does not contain the function:

```
| jl_value_t *jl_call(jl_function_t *f, jl_value_t **args, int32_t  
|     nargs);
```

## Adding methods

Given the above dispatch process, conceptually all that is needed to add a new method is (1) a tuple type, and (2) code for the body of the method. `jl_method_def` implements this operation. `jl_first_argument_datatype` is called to extract the relevant method table from what would be the type of the first argument. This is much more complicated than the corresponding procedure during dispatch, since the argument tuple type might be abstract. For example, we can define:

which works since all possible matching methods would belong to the same method table.

## Creating generic functions

Since every object is callable, nothing special is needed to create a generic function. Therefore `jl_new_generic_function` simply creates a new singleton (0 size) subtype of `Function` and returns its instance. A function can have a mnemonic "display name" which is used in debug info and when printing objects. For example the name of `Base.sin` is `sin`. By convention, the name of the created type is the same as the function name, with a `#` prepended. So `typeof(sin)` is `Base.#sin`.

## Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

```
function adder(x)
    return y->x+y
end
```

is lowered to (roughly):

```
struct ##1{T}
    x::T
end

(_::##1)(y) = _.x + y

function adder(x)
```

```
|1686 return ##1(x) CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS  
|  
|end
```

## Constructors

A constructor call is just a call to a type. The type of most types is **DataType**, so the method table for **DataType** contains most constructor definitions. One wrinkle is the fallback definition that makes all types callable via **convert**:

```
| ( ::Type{T})(args...) where {T} = convert(T, args...)::T
```

In this definition the function type is abstract, which is not normally supported. To make this work, all subtypes of **Type** (**Type**, **UnionAll**, **Union**, and **DataType**) currently share a method table via special arrangement.

## Builtins

The "builtin" functions, defined in the **Core** module, are:

```
=====  
typeof sizeof <: isa typeassert throw tuple getfield setfield!  
fieldtype  
nfields isdefined arrayref arrayset arraysizes applicable invoke  
apply_type _apply  
_expr svec
```

These are all singleton objects whose types are subtypes of **Builtin**, which is a subtype of **Function**. Their purpose is to expose entry points in the runtime that use the "jlcall" calling convention:

```
| jl_value_t *(jl_value_t*, jl_value_t**, uint32_t)
```

The method tables of builtins are empty. Instead, they have a single catch-all method cache entry (**Tuple{Vararg{Any}}**) whose jlcall fptr points to the correct function. This is kind of a hack but works reasonably well.

Keyword arguments work by associating a special, hidden function object with each method table that has definitions with keyword arguments. This function is called the "keyword argument sorter" or "keyword sorter", or "kwsorter", and is stored in the `kwsorter` field of `MethodTable` objects. Every definition in the kwsorter function has the same arguments as some definition in the normal method table, except with a single `Array` argument prepended. This array contains alternating symbols and values that represent the passed keyword arguments. The kwsorter's job is to move keyword arguments into their canonical positions based on name, plus evaluate and substite any needed default value expressions. The result is a normal positional argument list, which is then passed to yet another function.

The easiest way to understand the process is to look at how a keyword argument method definition is lowered. The code:

```
function circle(center, radius; color = black, fill::Bool = true,
    → options...)
    # draw
end
```

actually produces three method definitions. The first is a function that accepts all arguments (including keywords) as positional arguments, and includes the code for the method body. It has an auto-generated name:

```
function #circle#1(color, fill::Bool, options, circle, center,
    → radius)
    # draw
end
```

The second method is an ordinary definition for the original `circle` function, which handles the case where no keyword arguments are passed:

```
1688 CHAPTER 78 DOCUMENTATION OF JULIA'S INTERNALS
function circle(center, radius)
    #circle#1(black, true, Any[], circle, center, radius)
end
```

This simply dispatches to the first method, passing along default values. Finally there is the kwsorter definition:

```
function (::Core.kwftype(typeof(circle)))(kw::Array, circle,
    center, radius)
    options = Any[]
    color = arg associated with :color, or black if not found
    fill = arg associated with :fill, or true if not found
    # push remaining elements of kw into options array
    #circle#1(color, fill, options, circle, center, radius)
end
```

The front end generates code to loop over the `kw` array and pick out arguments in the right order, evaluating default expressions when an argument is not found.

The function `Core.kwftype(t)` fetches (and creates, if necessary) the field `t.name.mt.kwsorter`.

This design has the feature that call sites that don't use keyword arguments require no special handling; everything works as if they were not part of the language at all. Call sites that do use keyword arguments are dispatched directly to the called function's kwsorter. For example the call:

```
| circle((0,0), 1.0, color = red; other...)
```

is lowered to:

```
| kfunc(circle)(Any[:color, red, other...], circle, (0,0), 1.0)
```

~~THESE UNPACKING FUNCTIONS~~ are represented here as `other...` actually function<sup>1689</sup> unpacks each element of `other`, expecting each one to contain two values (a symbol and a value). `kwfunc` (also in `Core`) fetches the `kwsorter` for the called function. Notice that the original `circle` function is passed through, to handle closures.

## Compiler efficiency issues

Generating a new type for every function has potentially serious consequences for compiler resource use when combined with Julia's "specialize on all arguments by default" design. Indeed, the initial implementation of this design suffered from much longer build and test times, higher memory use, and a system image nearly 2x larger than the baseline. In a naive implementation, the problem is bad enough to make the system nearly unusable. Several significant optimizations were needed to make the design practical.

The first issue is excessive specialization of functions for different values of function-valued arguments. Many functions simply "pass through" an argument to somewhere else, e.g. to another function or to a storage location. Such functions do not need to be specialized for every closure that might be passed in. Fortunately this case is easy to distinguish by simply considering whether a function calls one of its arguments (i.e. the argument appears in "head position" somewhere). Performance-critical higher-order functions like `map` certainly call their argument function and so will still be specialized as expected. This optimization is implemented by recording which arguments are called during the `analyze-variables` pass in the front end. When `cache_method` sees an argument in the `Function` type hierarchy passed to a slot declared as `Any` or `Function`, it behaves as if the `@nospecialize` annotation were applied. This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method cache hash tables. Empirical

§ 16.00 shows that the ~~CHAPTER 7.8 DOCUMENTATION IS PART OF THE INTERNALS~~ one or two arguments. In turn, many of these cases can be resolved by considering only the first argument. (Aside: proponents of single dispatch would not be surprised by this at all. However, this argument means "multiple dispatch is easy to optimize in practice", and that we should therefore use it, not "we should use single dispatch"!) So the method cache uses the type of the first argument as its primary key. Note, however, that this corresponds to the second element of the tuple type for a function call (the first element being the type of the function itself). Typically, type variation in head position is extremely low – indeed, the majority of functions belong to singleton types with no parameters. However, this is not the case for constructors, where a single method table holds constructors for every type. Therefore the `Type` method table is special-cased to use the first tuple type element instead of the second.

The front end generates type declarations for all closures. Initially, this was implemented by generating normal type declarations. However, this produced an extremely large number of constructors, all of which were trivial (simply passing all arguments through to `new`). Since methods are partially ordered, inserting all of these methods is  $O(n^2)$ , plus there are just too many of them to keep around. This was optimized by generating `struct_type` expressions directly (bypassing default constructor generation), and using `new` directly to create closure instances. Not the prettiest thing ever, but you do what you gotta do.

The next problem was the `@test` macro, which generated a 0-argument closure for each test case. This is not really necessary, since each test case is simply run once in place. Therefore I modified `@test` to expand to a try-catch block that records the test result (true, false, or exception raised) and calls the test suite handler on it.

The (non-exported) Cartesian module provides macros that facilitate writing multidimensional algorithms. It is hoped that Cartesian will not, in the long term, be necessary; however, at present it is one of the few ways to write compact and performant multidimensional code.

### Principles of usage

A simple example of usage is:

```
@nloops 3 i A begin
    s += @nref 3 A i
end
```

which generates the following code:

```
for i_3 = 1:size(A,3)
    for i_2 = 1:size(A,2)
        for i_1 = 1:size(A,1)
            s += A[i_1,i_2,i_3]
    end
end
end
```

In general, Cartesian allows you to write generic code that contains repetitive elements, like the nested loops in this example. Other applications include repeated expressions (e.g., loop unwinding) or creating function calls with variable numbers of arguments without using the "splat" construct (*i...*).

### Basic syntax

The (basic) syntax of `@nloops` is as follows:

169 The first argument is the number of loops.

## CHAPTER 78. DOCUMENTATION & INTERNALS

The second argument is the symbol-prefix used for the iterator variable. Here we used `i`, and variables `i_1`, `i_2`, `i_3` were generated.

The third argument specifies the range for each iterator variable. If you use a variable (symbol) here, it's taken as `1:size(A, dim)`. More flexibly, you can use the anonymous-function expression syntax described below.

The last argument is the body of the loop. Here, that's what appears between the `begin...end`.

There are some additional features of `@nloops` described in the [reference section](#).

`@nref` follows a similar pattern, generating `A[i_1, i_2, i_3]` from `@nref 3 A i`. The general practice is to read from left to right, which is why `@nloops` is `@nloops 3 i A expr` (as in `for i_2 = 1:size(A, 2)`, where `i_2` is to the left and the range is to the right) whereas `@nref` is `@nref 3 A i` (as in `A[i_1, i_2, i_3]`, where the array comes first).

If you're developing code with Cartesian, you may find that debugging is easier when you examine the generated code, using `@macroexpand`:

```
julia> @macroexpand @nref 2 A i
:(A[i_1, i_2])
```

Supplying the number of expressions

The first argument to both of these macros is the number of expressions, which must be an integer. When you're writing a function that you intend to work in multiple dimensions, this may not be something you want to hard-code. If you're writing code that you need to work with older Julia versions, currently

y80shBASE.CARTESIAN generate macro described in an older version of 0.93 documentation.

Starting in Julia 0.4-pre, the recommended approach is to use a `@generated function`. Here's an example:

```
@generated function mysum(A::Array{T,N}) where {T,N}
    quote
        s = zero(T)
        @nloops $N i A begin
            s += @nref $N A i
        end
        s
    end
end
```

Naturally, you can also prepare expressions or perform calculations before the `quote` block.

Anonymous-function expressions as macro arguments

Perhaps the single most powerful feature in `Cartesian` is the ability to supply anonymous-function expressions that get evaluated at parsing time. Let's consider a simple example:

```
|@nexprs 2 j->(i_j = 1)
```

`@nexprs` generates `n` expressions that follow a pattern. This code would generate the following statements:

```
|i_1 = 1
|i_2 = 1
```

1684 CHAPTER 78 DOCUMENTATION INTERNALS  
Each generated `rangeexpr` (the `rangeexpr` in the previous code block) gets replaced by values in the range `1:2`. Generally speaking, Cartesian employs a LaTeX-like syntax. This allows you to do math on the index `j`. Here's an example computing the strides of an array:

```
s_1 = 1  
@nexprs 3 j->(s_{j+1} = s_j * size(A, j))
```

would generate expressions

```
s_1 = 1  
s_2 = s_1 * size(A, 1)  
s_3 = s_2 * size(A, 2)  
s_4 = s_3 * size(A, 3)
```

Anonymous-function expressions have many uses in practice.

Macro reference [Base.Cartesian.@nloops](#) – Macro.

```
@nloops N itersym rangeexpr bodyexpr  
@nloops N itersym rangeexpr preexpr bodyexpr  
@nloops N itersym rangeexpr preexpr postexpr bodyexpr
```

Generate `N` nested loops, using `itersym` as the prefix for the iteration variables. `rangeexpr` may be an anonymous-function expression, or a simple symbol `var` in which case the range is `indices(var, d)` for dimension `d`.

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
@nloops 2 i A d -> j_d = min(i_d, 5) begin  
    s += @nref 2 A j  
end
```

```
| for i_2 = indices(A, 2)
|   j_2 = min(i_2, 5)
|   for i_1 = indices(A, 1)
|     j_1 = min(i_1, 5)
|     s += A[j_1, j_2]
|   end
end
```

If you want just a post-expression, supply `nothing` for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

`source`

`Base.Cartesian.@nref` – Macro.

```
| @nref N A indexexpr
```

Generate expressions like `A[i_1, i_2, ...]`. `indexexpr` can either be an iteration-symbol prefix, or an anonymous-function expression.

Examples

```
| julia> @macroexpand Base.Cartesian.@nref 3 A i
| :(A[i_1, i_2, i_3])
```

`source`

`Base.Cartesian.@nextract` – Macro.

```
| @nextract N esym isym
```

Generate `N` variables `esym_1`, `esym_2`, ..., `esym_N` to extract values from `isym`. `isym` can be either a `Symbol` or anonymous-function expression.

`@nextract 2 x y` would generate

1696 | `x_1 = y[1]`      CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS  
| `x_2 = y[2]`

while @nextract 3 x d->y[2d-1] yields

| `x_1 = y[1]`  
| `x_2 = y[3]`  
| `x_3 = y[5]`

`source`

[Base.Cartesian.@nexprs](#) – Macro.

| `@nexprs N expr`

Generate N expressions. `expr` should be an anonymous-function expression.

Examples

**julia>** `@macroexpand Base.Cartesian.@nexprs 4 i -> y[i] =`  
  `↪ A[i+j]`  
  `quote`  
    `y[1] = A[1 + j]`  
    `y[2] = A[2 + j]`  
    `y[3] = A[3 + j]`  
    `y[4] = A[4 + j]`  
  `end`

`source`

[Base.Cartesian.@ncall](#) – Macro.

| `@ncall N f sym...`

Generate a function call expression. `sym` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into N arguments.

78.9 For example, `@nall 3 func a` generates

```
| func(a_1, a_2, a_3)
```

while `@ncall 2 func a b i->c[i]` yields

```
| func(a, b, c[1], c[2])
```

`source`

[Base.Cartesian.@ntuple](#) – Macro.

```
| @ntuple N expr
```

Generates an `N`-tuple. `@ntuple 2 i` would generate `(i_1, i_2)`, and `@ntuple 2 k->k+1` would generate `(2, 3)`.

`source`

[Base.Cartesian.@nall](#) – Macro.

```
| @nall N expr
```

Check whether all of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nall 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 && i_2 > 1 && i_3 > 1)`. This can be convenient for bounds-checking.

`source`

[Base.Cartesian.@nany](#) – Macro.

```
| @nany N expr
```

Check whether any of the expressions generated by the anonymous-function expression `expr` evaluate to `true`.

`@nany 3 d->(i_d > 1)` would generate the expression `(i_1 > 1 || i_2 > 1 || i_3 > 1)`.

`source`

1697

```
|@nif N conditionexpr expr
|@nif N conditionexpr expr elseexpr
```

Generates a sequence of `if ... elseif ... else ... end` statements. For example:

```
|@nif 3 d->(i_d >= size(A,d)) d->(error("Dimension ", d, " too
|    big")) d->println("All OK")
```

would generate:

```
if i_1 > size(A, 1)
    error("Dimension ", 1, " too big")
elseif i_2 > size(A, 2)
    error("Dimension ", 2, " too big")
else
    println("All OK")
end
```

[source](#)

## 78.10 Talking to the compiler (the `:meta` mechanism)

In some circumstances, one might wish to provide hints or instructions that a given block of code has special properties: you might always want to inline it, or you might want to turn on special compiler optimization passes. Starting with version 0.4, Julia has a convention that these instructions can be placed inside a `:meta` expression, which is typically (but not necessarily) the first expression in the body of a function.

`:meta` expressions are created with macros. As an example, consider the implementation of the `@inline` macro:

```
macro inline(ex)
    esc(isa(ex, Expr) ? pushmeta!(ex, :inline) : ex)
end
```

Here, `ex` is expected to be an expression defining a function. A statement like this:

```
@inline function myfunction(x)
    x*(x+3)
end
```

gets turned into an expression like this:

```
quote
    function myfunction(x)
        Expr(:meta, :inline)
        x*(x+3)
    end
end
```

`Base.pushmeta!(ex, :symbol, args...)` appends `:symbol` to the end of the `:meta` expression, creating a new `:meta` expression if necessary. If `args` is specified, a nested expression containing `:symbol` and these arguments is appended instead, which can be used to specify additional information.

To use the metadata, you have to parse these `:meta` expressions. If your implementation can be performed within Julia, `Base.popmeta!` is very handy: `Base.popmeta!(body, :symbol)` will scan a function body expression (one without the function signature) for the first `:meta` expression containing `:symbol`, extract any arguments, and return a tuple (`found::Bool, args::Array{Any}`). If the metadata did not have any arguments, or `:symbol` was not found, the `args` array will be empty.

Not yet provided is a [CHAPTER 7 DOCUMENTATION](#) on [Julia's AbstractArrays](#) from C++.

## 78.11 SubArrays

Julia's `SubArray` type is a container encoding a "view" of a parent [AbstractArray](#). This page documents some of the design principles and implementation of `SubArrays`.

### Indexing: cartesian vs. linear indexing

Broadly speaking, there are two main ways to access data in an array. The first, often called cartesian indexing, uses  $N$  indexes for an  $N$ -dimensional `AbstractArray`. For example, a matrix `A` (2-dimensional) can be indexed in cartesian style as `A[i, j]`. The second indexing method, referred to as linear indexing, uses a single index even for higher-dimensional objects. For example, if `A = reshape(1:12, 3, 4)`, then the expression `A[5]` returns the value 5. Julia allows you to combine these styles of indexing: for example, a 3d array `A3` can be indexed as `A3[i, j]`, in which case `i` is interpreted as a cartesian index for the first dimension, and `j` is a linear index over dimensions 2 and 3.

For `Arrays`, linear indexing appeals to the underlying storage format: an array is laid out as a contiguous block of memory, and hence the linear index is just the offset (+1) of the corresponding entry relative to the beginning of the array. However, this is not true for many other `AbstractArray` types: examples include [SparseMatrixCSC](#), arrays that require some kind of computation (such as interpolation), and the type under discussion here, `SubArray`. For these types, the underlying information is more naturally described in terms of cartesian indexes.

You can manually convert from a cartesian index to a linear index with `sub2ind`,

~~78d 1icSUBARRAYS~~ ~~ind2sub~~. `getindex` and `setindex!` functions for ~~Abstract~~ ~~Abstract~~ types may include similar operations.

While converting from a cartesian index to a linear index is fast (it's just multiplication and addition), converting from a linear index to a cartesian index is very slow: it relies on the `div` operation, which is one of the slowest low-level operations you can perform with a CPU. For this reason, any code that deals with `AbstractArray` types is best designed in terms of cartesian, rather than linear, indexing.

## Index replacement

Consider making 2d slices of a 3d array:

```
julia> A = rand(2,3,4);  
  
julia> S1 = view(A, :, 1, 2:3)  
2×2 view(::Array{Float64,3}, :, 1, 2:3) with eltype Float64:  
 0.200586  0.066423  
 0.298614  0.956753  
  
julia> S2 = view(A, 1, :, 2:3)  
3×2 view(::Array{Float64,3}, 1, :, 2:3) with eltype Float64:  
 0.200586  0.066423  
 0.246837  0.646691  
 0.648882  0.276021
```

`view` drops "singleton" dimensions (ones that are specified by an `Int`), so both `S1` and `S2` are two-dimensional `SubArrays`. Consequently, the natural way to index these is with `S1[i, j]`. To extract the value from the parent array `A`, the natural approach is to replace `S1[i, j]` with `A[i, 1, (2:3)[j]]` and `S2[i, j]` with `A[1, i, (2:3)[j]]`.

The key feature of the [CHAPTER 7 Subarray Documentation](#) is that all `get!` and `set!` operations can be performed without any runtime overhead.

## SubArray design

### Type parameters and fields

The strategy adopted is first and foremost expressed in the definition of the type:

```
struct SubArray{T,N,P,I,L} <: AbstractArray{T,N}
    parent::P
    indexes::I
    offset1::Int      # for linear indexing and pointer, only
    ↳ valid when L==true
    stride1::Int      # used only for linear indexing
    ...
end
```

`SubArray` has 5 type parameters. The first two are the standard element type and dimensionality. The next is the type of the parent `AbstractArray`. The most heavily-used is the fourth parameter, a `Tuple` of the types of the indices for each dimension. The final one, `L`, is only provided as a convenience for dispatch; it's a boolean that represents whether the index types support fast linear indexing. More on that later.

If in our example above `A` is a `AbstractArray{Float64, 3}`, our `S1` case above would be a `SubArray{Float64, 2, AbstractArray{Float64, 3}, Tuple{Base.Slice{Base.OneTo{Int64}}, Range{Int64}}}, false`. Note in particular the tuple parameter, which stores the types of the indices used to create `S1`. Likewise,

```
julia> S1.indexes
(Base.Slice(Base.OneTo(2)), 1, 2:3)
```

~~Slicing SubArrays~~ allows index replacement, and having the types encoded as parameters allows one to dispatch to efficient algorithms.

## Index translation

Performing index translation requires that you do different things for different concrete `SubArray` types. For example, for `S1`, one needs to apply the `i, j` indices to the first and third dimensions of the parent array, whereas for `S2` one needs to apply them to the second and third. The simplest approach to indexing would be to do the type-analysis at runtime:

```
parentindexes = Array{Any}(0)
for thisindex in S.indexes
    ...
    if isa(thisindex, Int)
        # Don't consume one of the input indexes
        push!(parentindexes, thisindex)
    elseif isa(thisindex, AbstractVector)
        # Consume an input index
        push!(parentindexes, thisindex[inputindex[j]])
        j += 1
    elseif isa(thisindex, AbstractMatrix)
        # Consume two input indices
        push!(parentindexes, thisindex[inputindex[j],
                                       inputindex[j+1]])
        j += 2
    elseif ...
end
S.parent[parentindexes...]
```

Unfortunately, this would be disastrous in terms of performance: each element access would allocate memory, and involves the running of a lot of poorly-typed code.

The better approach (CHAPTER 78 DOCUMENTATION FOUNDATION INTERNALS) is to use `reindex` to handle the internal details of stored index. That's what `reindex` does: it dispatches on the type of the first stored index and consumes the appropriate number of input indices, and then it recurses on the remaining indices. In the case of `S1`, this expands to

```
Base.reindex(S1, S1.indexes, (i, j)) == (i, S1.indexes[2],  
→ S1.indexes[3][j])
```

for any pair of indices `(i, j)` (except `CartesianIndex`s and arrays thereof, see below).

This is the core of a `SubArray`; indexing methods depend upon `reindex` to do this index translation. Sometimes, though, we can avoid the indirection and make it even faster.

### Linear indexing

Linear indexing can be implemented efficiently when the entire array has a single stride that separates successive elements, starting from some offset. This means that we can pre-compute these values and represent linear indexing simply as an addition and multiplication, avoiding the indirection of `reindex` and (more importantly) the slow computation of the cartesian coordinates entirely.

For `SubArray` types, the availability of efficient linear indexing is based purely on the types of the indices, and does not depend on values like the size of the parent array. You can ask whether a given set of indices supports fast linear indexing with the internal `Base.viewindexing` function:

```
julia> Base.viewindexing(S1.indexes)  
IndexCartesian()  
  
julia> Base.viewindexing(S2.indexes)  
IndexLinear()
```

This is compared during construction of the `SubArray` and stored in the `Linear` parameter as a boolean that encodes fast linear indexing support. While not strictly necessary, it means that we can define dispatch directly on `SubArray{T,N,A,I,true}` without any intermediaries.

Since this computation doesn't depend on runtime values, it can miss some cases in which the stride happens to be uniform:

```
julia> A = reshape(1:4*2, 4, 2)
4×2 reshape(::UnitRange{Int64}, 4, 2) with eltype Int64:
1 5
2 6
3 7
4 8
```

```
julia> diff(A[2:2:4, :][:])
3-element Array{Int64,1}:
2
2
2
```

A view constructed as `view(A, 2:2:4, :)` happens to have uniform stride, and therefore linear indexing indeed could be performed efficiently. However, success in this case depends on the size of the array: if the first dimension instead were odd,

```
julia> A = reshape(1:5*2, 5, 2)
5×2 reshape(::UnitRange{Int64}, 5, 2) with eltype Int64:
1 6
2 7
3 8
4 9
```

```
julia> diff(A[2:2:4, :][:])
3-element Array{Int64,1}:
 2
 3
 2
```

then `A[2:2:4, :]` does not have uniform stride, so we cannot guarantee efficient linear indexing. Since we have to base this decision based purely on types encoded in the parameters of the `SubArray`, `S = view(A, 2:2:4, :)` cannot implement efficient linear indexing.

A few details

Note that the `Base.reindex` function is agnostic to the types of the input indices; it simply determines how and where the stored indices should be reindexed. It not only supports integer indices, but it supports non-scalar indexing, too. This means that views of views don't need two levels of indirection; they can simply re-compute the indices into the original parent array!

Hopefully by now it's fairly clear that supporting slices means that the dimensionality, given by the parameter `N`, is not necessarily equal to the dimensionality of the parent array or the length of the `indexes` tuple. Neither do user-supplied indices necessarily line up with entries in the `indexes` tuple (e.g., the second user-supplied index might correspond to the third dimension of the parent array, and the third element in the `indexes` tuple).

What might be less obvious is that the dimensionality of the stored parent array must be equal to the number of effective indices in the `indexes`

```
A = reshape(1:35, 5, 7) # A 2d parent Array
S = view(A, 2:7)          # A 1d view created by linear indexing
S = view(A, :, :, 1:1)    # Appending extra indices is supported
```

Naively, you'd think you could just set `S.parent = A` and `S.indexes = ([:, :, 1:1])`, but supporting this dramatically complicates the reindexing process, especially for views of views. Not only do you need to dispatch on the types of the stored indices, but you need to examine whether a given index is the final one and "merge" any remaining stored indices together. This is not an easy task, and even worse: it's slow since it implicitly depends upon linear indexing.

Fortunately, this is precisely the computation that `ReshapedArray` performs, and it does so linearly if possible. Consequently, `view` ensures that the parent array is the appropriate dimensionality for the given indices by reshaping it if needed. The inner `SubArray` constructor ensures that this invariant is satisfied.

`CartesianIndex` and arrays thereof throw a nasty wrench into the `reindex` scheme. Recall that `reindex` simply dispatches on the type of the stored indices in order to determine how many passed indices should be used and where they should go. But with `CartesianIndex`, there's no longer a one-to-one correspondence between the number of passed arguments and the number of dimensions that they index into. If we return to the above example of `Base.reindex(S1, S1.indexes, (i, j))`, you can see that the expansion is incorrect for `i, j = CartesianIndex(), CartesianIndex(2, 1)`. It should skip the `CartesianIndex()` entirely and return:

```
(CartesianIndex(2,1)[1], S1.indexes[2],
 → S1.indexes[3][CartesianIndex(2,1)[2]])
```

```
(CartesianIndex(), S1.indexes[2],
    → S1.indexes[3][CartesianIndex(2,1)])
```

Doing this correctly would require combined dispatch on both the stored and passed indices across all combinations of dimensionalities in an intractable manner. As such, `reindex` must never be called with `CartesianIndex` indices. Fortunately, the scalar case is easily handled by first flattening the `CartesianIndex` arguments to plain integers. Arrays of `CartesianIndex`, however, cannot be split apart into orthogonal pieces so easily. Before attempting to use `reindex`, `view` must ensure that there are no arrays of `CartesianIndex` in the argument list. If there are, it can simply "punt" by avoiding the `reindex` calculation entirely, constructing a nested `SubArray` with two levels of indirection instead.

## 78.12 System Image Building

### Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `Base` module, named `sys.ji`. This file is also precompiled into a shared library called `sys.{so,dll,dylib}` on as many platforms as possible, so as to give vastly improved startup times. On systems that do not ship with a precompiled system image file, one can be generated from the source files shipped in Julia's `DATAROOTDIR/julia/base` folder.

This operation is useful for multiple reasons. A user may:

- Build a precompiled shared library system image on a platform that did not ship with one, thereby improving startup times.

- Modify `Base`, rebuild the system image and use the new `Base` next time Julia is started.

78.1 **Inclusion of SYSTEM IMAGE BUILDING** at includes packages into the system image<sup>1709</sup>, thereby creating a system image that has packages embedded into the startup environment.

Julia now ships with a script that automates the tasks of building the system image, wittingly named `build_sysimg.jl` that lives in `DATAROOTDIR/julia/`. That is, to include it into a current Julia session, type:

```
| include(joinpath(JULIA_HOME, Base.DATAROOTDIR, "julia",
|   ↵ "build_sysimg.jl"))
```

This will include a `build_sysimg` function:

[Main.BuildSysImg.build\\_sysimg](#) – Function.

```
| build_sysimg(sysimg_path=default_sysimg_path(), cpu_target=
|   ↵ "native", userimg_path=nothing; force=false)
```

Rebuild the system image. Store it in `sysimg_path`, which defaults to a file named `sys.ji` that sits in the same folder as `libjulia.{so,dylib}`, except on Windows where it defaults to `JULIA_HOME/..../lib/julia/sys.ji`. Use the cpu instruction set given by `cpu_target`. Valid CPU targets are the same as for the `-C` option to `julia`, or the `-march` option to `gcc`. Defaults to `native`, which means to use all CPU instructions available on the current processor. Include the user image file given by `userimg_path`, which should contain directives such as `using MyPackage` to include that package in the new system image. New system image will not replace an older image unless `force` is set to true.

[source](#)

Note that this file can also be run as a script itself, with command line arguments taking the place of arguments passed to the `build_sysimg` function. For example, to build a system image in `/tmp/sys.{so,dll,dylib}`, with

the Core2 CPU instruction set. If `JULIA_CPU_TARGET` is set to `core2` and `JULIA_DISABLE_THREADS` is set to `true`, one would execute:

```
| julia build_sysimg.jl /tmp/sys core2 ~/userimg.jl --force
```

## System image optimized for multiple microarchitectures

The system image can be compiled simultaneously for multiple CPU microarchitectures under the same instruction set architecture (ISA). Multiple versions of the same function may be created with minimum dispatch point inserted into shared functions in order to take advantage of different ISA extensions or other microarchitecture features. The version that offers the best performance will be selected automatically at runtime based on available features.

## Specifying multiple system image targets

Multi-microarch system image can be enabled by passing multiple targets during system image compilation. This can be done either with the `JULIA_CPU_TARGET` make option or with the `-C` command line option when running the compilation command manually. Multiple targets are separated by ; in the option. The syntax for each target is a CPU name followed by multiple features separated by ,. All features supported by LLVM is supported and a feature can be disabled with a - prefix. (+ prefix is also allowed and ignored to be consistent with LLVM syntax). Additionally, a few special features are supported to control the function cloning behavior.

### 1. `clone_all`

By default, only functions that are the most likely to benefit from the microarchitecture features will be cloned. When `clone_all` is specified for a target, however, all functions in the system image will be cloned for the target. The negative form `-clone_all` can be used to prevent the built-in heuristic from cloning all functions.

Where `<n>` is a placeholder for a non-negative number (e.g. `base(0)`, `base(1)`). By default, a partially cloned (i.e. not `clone_all`) target will use functions from the default target (first one specified) if a function is not cloned. This behavior can be changed by specifying a different base with the `base(<n>)` option. The `n`th target (0-based) will be used as the base target instead of the default (0th) one. The base target has to be either `0` or another `clone_all` target. Specifying a non default `clone_all` target as the base target will cause an error.

### 3. `opt_size`

This cause the function for the target to be optimize for size when there isn't a significant runtime performance impact. This corresponds to `-Os` GCC and Clang option.

### 4. `min_size`

This cause the function for the target to be optimize for size that might have a significant runtime performance impact. This corresponds to `-Oz` Clang option.

## Implementation overview

This is a brief overview of different part involved in the implementation. See code comments for each components for more implementation details.

### 1. System image compilation

The parsing and cloning decision are done in `src/processor*`. We currently support cloning of function based on the present of loops, simd instructions, or other math operations (e.g. fastmath, fma, muladd). This information is passed on to `src/llvm-multiversioning.cpp` which does the actual cloning. In addition to doing the cloning and insert dispatch

171 slots (see comment `CHAPTER 78 DOCUMENTATION MODULE INTERNALS`). After this is done), the pass also generates metadata so that the runtime can load and initialize the system image correctly. A detail description of the metadata is available in `src/processor.h`.

## 2. System image loading

The loading and initialization of the system image is done in `src/processor*` by parsing the metadata saved during system image generation. Host feature detection and selection decision are done in `src/processor_*.cpp` depending on the ISA. The target selection will prefer exact CPU name match, larger vector register size, and largest number of features. An overview of this process is in `src/processor.cpp`.

## 78.13 Working with LLVM

This is not a replacement for the LLVM documentation, but a collection of tips for working on LLVM for Julia.

### Overview of Julia to LLVM Interface

Julia statically links in LLVM by default. Build with `USE_LLVM_SHLIB=1` to link dynamically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/`.

Some of the `.cpp` files form a group that compile to a single object.

The difference between an intrinsic and a builtin is that a builtin is a first class function that can be used like any other Julia function. An intrinsic can operate only on unboxed data, and therefore its arguments must be statically typed.

Julia currently uses LLVM's [Type Based Alias Analysis](#). To find the comments that document the inclusion relationships, look for `static MDNode*` in `src/-codegen.cpp`.

The `-O` option enables LLVM's [Basic Alias Analysis](#).

## Building Julia with a different version of LLVM

The default version of LLVM is specified in `deps/Versions.make`. You can override it by creating a file called `Make.user` in the top-level directory and adding a line to it such as:

```
| LLVM_VER = 3.5.0
```

Besides the LLVM release numerals, you can also use `LLVM_VER = svn` to build against the latest development version of LLVM.

## Passing options to LLVM

You can pass options to LLVM using debug builds of Julia. To create a debug build, run `make debug`. The resulting executable is `usr/bin/julia-debug`. You can pass LLVM options to this executable via the environment variable `JULIA_LLVM_ARGS`. Here are example settings using `bash` syntax:

```
export JULIA_LLVM_ARGS = -print-after-all dumps IR after each  
pass.
```

```
export JULIA_LLVM_ARGS = -debug-only=loop-vectorize dumps  
LLVM DEBUG(...) diagnostics for loop vectorizer if you built Julia with  
LLVM_ASSERTIONS=1. Otherwise you will get warnings about "Unknown  
command line argument". Counter-intuitively, building Julia with LLVM_DE-  
BUG=1 is not enough to dump DEBUG diagnostics from a pass.
```

## Debugging LLVM transformations | DOCUMENTATION OF JULIA'S INTERNALS

On occasion, it can be useful to debug LLVM's transformations in isolation from the rest of the Julia system, e.g. because reproducing the issue inside `julia` would take too long, or because one wants to take advantage of LLVM's tooling (e.g. `bugpoint`). To get unoptimized IR for the entire system image, pass the `--output-unopt-bc unopt.bc` option to the system image build process, which will output the unoptimized IR to an `unopt.bc` file. This file can then be passed to LLVM tools as usual. `libjulia` can function as an LLVM pass plugin and can be loaded into LLVM tools, to make `julia`-specific passes available in this environment. In addition, it exposes the `-julia` meta-pass, which runs the entire Julia pass-pipeline over the IR. As an example, to generate a system image, one could do:

```
opt -load libjulia.so -julia -o opt.bc unopt.bc
llc -o sys.o opt.bc
cc -shared -o sys.so sys.o
```

This system image can then be loaded by `julia` as usual.

Alternatively, you can use `--output-jit-bc jit.bc` to obtain a trace of all IR passed to the JIT. This is useful for code that cannot be run as part of the sysimg generation process (e.g. because it creates unserializable state). However, the resulting `jit.bc` does not include sysimage data, and can thus not be used as such.

It is also possible to dump an LLVM IR module for just one Julia function, using:

```
f, T = +, Tuple{Int,Int} # Substitute your function of interest
→ here
optimize = false
open("plus.ll", "w") do f
    println(f, Base._dump_function(f, T, false, false, false,
→ true, :att, optimize))
```

These files can be processed the same way as the unoptimized sysimg IR shown above.

### Improving LLVM optimizations for Julia

Improving LLVM code generation usually involves either changing Julia lowering to be more friendly to LLVM's passes, or improving a pass.

If you are planning to improve a pass, be sure to read the [LLVM developer policy](#). The best strategy is to create a code example in a form where you can use LLVM's `opt` tool to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `JULIA_LLVM_ARGS = -print-after-all` to dump the IR.
3. Pick out the IR at the point just before the pass of interest runs.
4. Strip the debug metadata and fix up the TBAA metadata by hand.

The last step is labor intensive. Suggestions on a better way would be appreciated.

### The jlcall calling convention

Julia has a generic calling convention for unoptimized code, which looks somewhat as follows:

```
| jl_value_t *any_unoptimized_call(jl_value_t *, jl_value_t **, int)
| ;
```

where the first argument is the boxed function object, the second argument is an on-stack array of arguments and the third is the number of arguments. Now, we could perform a straightforward lowering and emit an alloca for the

argument array. However, determining the SSA form of pointers at the call site, making optimizations (including GC root placement), significantly harder. Instead, we emit it as follows:

```
%bitcast = bitcast @any_unoptimized_call to %jl_value_t *(*)(%  
    jl_value_t *, %jl_value_t *)  
call cc 37 %jl_value_t *%bitcast(%jl_value_t *%arg1, %jl_value_t  
    *%arg2)
```

The special `cc 37` annotation marks the fact that this call site is really using the `jlcall` calling convention. This allows us to retain the SSA-ness of the uses throughout the optimizer. GC root placement will later lower this call to the original C ABI. In the code the calling convention number is represented by the `JLCALL_F_CC` constant. In addition, there is the `JLCALL_CC` calling convention which functions similarly, but omits the first argument.

## GC root placement

GC root placement is done by an LLVM pass late in the pass pipeline. Doing GC root placement this late enables LLVM to make more aggressive optimizations around code that requires GC roots, as well as allowing us to reduce the number of required GC roots and GC root store operations (since LLVM doesn't understand our GC, it wouldn't otherwise know what it is and is not allowed to do with values stored to the GC frame, so it'll conservatively do very little). As an example, consider an error path

```
if some_condition()  
    #= Use some variables maybe =#  
    error("An error occurred")  
end
```

During constant folding, LLVM may discover that the condition is always false, and can remove the basic block. However, if GC root lowering is done early,

~~18.13. C WORKING WITH LLVM~~ deleted block, as well as any values kept ~~alive~~ in those slots only because they were used in the error path, would be kept alive by LLVM. By doing GC root lowering late, we give LLVM the license to do any of its usual optimizations (constant folding, dead code elimination, etc.), without having to worry (too much) about which values may or may not be GC tracked.

However, in order to be able to do late GC root placement, we need to be able to identify a) which pointers are gc tracked and b) all uses of such pointers. The goal of the GC placement pass is thus simple:

Minimize the number of needed GC roots/stores to them subject to the constraint that at every safepoint, any live GC-tracked pointer (i.e. for which there is a path after this point that contains a use of this pointer) is in some GC slot.

## Representation

The primary difficulty is thus choosing an IR representation that allows us to identify GC-tracked pointers and their uses, even after the program has been run through the optimizer. Our design makes use of three LLVM features to achieve this:

- Custom address spaces

- Operand Bundles

- Non-integral pointers

Custom address spaces allow us to tag every point with an integer that needs to be preserved through optimizations. The compiler may not insert casts between address spaces that did not exist in the original program and it must never change the address space of a pointer on a load/store/etc operation.

This allows us to annotate pointers to GC tracked values in a resistant way. Note that metadata would not be able to achieve the same purpose. Metadata is supposed to always be discardable without altering the semantics of the program. However, failing to identify a GC-tracked pointer alters the resulting program behavior dramatically – it'll probably crash or return wrong results. We currently use three different address spaces (their numbers are defined in `src/codegen_shared.cpp`):

GC Tracked Pointers (currently 10): These are pointers to boxed values that may be put into a GC frame. It is loosely equivalent to a `jl_value_t*` pointer on the C side. N.B. It is illegal to ever have a pointer in this address space that may not be stored to a GC slot.

Derived Pointers (currently 11): These are pointers that are derived from some GC tracked pointer. Uses of these pointers generate uses of the original pointer. However, they need not themselves be known to the GC. The GC root placement pass MUST always find the GC tracked pointer from which this pointer is derived and use that as the pointer to root.

Callee Rooted Pointers (currently 12): This is a utility address space to express the notion of a callee rooted value. All values of this address space MUST be storable to a GC root (though it is possible to relax this condition in the future), but unlike the other pointers need not be rooted if passed to a call (they do still need to be rooted if they are live across another safepoint between the definition and the call).

## Invariants

The GC root placement pass makes use of several invariants, which need to be observed by the frontend and are preserved by the optimizer.

First, only the following address space casts are allowed:

78.10.->WORKING-Derived, CalleeRooted}: It is allowable to decay an untracked pointer to any of the others. However, do note that the optimizer has broad license to not root such a value. It is never safe to have a value in address space 0 in any part of the program if it is (or is derived from) a value that requires a GC root.

Tracked->Derived: This is the standard decay route for interior values. The placement pass will look for these to identify the base pointer for any use.

Tracked->CalleeRooted: Addrspace CalleeRooted serves merely as a hint that a GC root is not required. However, do note that the Derived->CalleeRooted decay is prohibited, since pointers should generally be storable to a GC slot, even in this address space.

Now let us consider what constitutes a use:

Loads whose loaded values is in one of the address spaces

Stores of a value in one of the address spaces to a location

Stores to a pointer in one of the address spaces

Calls for which a value in one of the address spaces is an operand

Calls in jlcall ABI, for which the argument array contains a value

Return instructions.

We explicitly allow load/stores and simple calls in address spaces Tracked/Derived. Elements of jlcall argument arrays must always be in address space Tracked (it is required by the ABI that they are valid `jl_value_t*` pointers). The same is true for return instructions (though note that struct return arguments are allowed to have any of the address spaces). The only allowable use of an

Address space Called **Tracked**.  
an appropriately typed operand).

Further, we disallow `getelementptr` in `addrspace Tracked`. This is because unless the operation is a noop, the resulting pointer will not be validly storable to a GC slot and may thus not be in this address space. If such a pointer is required, it should be decayed to `addrspace Derived` first.

Lastly, we disallow `inttoptr/ptrtoint` instructions in these address spaces. Having these instructions would mean that some `i64` values are really GC tracked. This is problematic, because it breaks that stated requirement that we're able to identify GC-relevant pointers. This invariant is accomplished using the LLVM "non-integral pointers" feature, which is new in LLVM 5.0. It prohibits the optimizer from making optimizations that would introduce these operations. Note we can still insert static constants at JIT time by using `inttoptr` in address space 0 and then decaying to the appropriate address space afterwards.

## Supporting `ccall`

One important aspect missing from the discussion so far is the handling of `ccall`. `ccall` has the peculiar feature that the location and scope of a use do not coincide. As an example consider:

```
A = randn(1024)
ccall(:foo, Void, (Ptr{Float64},), A)
```

In lowering, the compiler will insert a conversion from the array to the pointer which drops the reference to the array value. However, we of course need to make sure that the array does stay alive while we're doing the `ccall`. To understand how this is done, first recall the lowering of the above code:

```
78.13 WORKING WITH LLVM 1721
return $Expr(:foreigncall, (:foo), Void, svec(Ptr{Float64}),
    ↳ (:ccall), 1, $($Expr(:foreigncall, (:jl_array_ptr),
    ↳ Ptr{Float64}, svec(Any), (:ccall), 1, :(A))), :(A)))
```

The last `:(A)`, is an extra argument list inserted during lowering that informs the code generator which Julia level values need to be kept alive for the duration of this `ccall`. We then take this information and represent it in an "operand bundle" at the IR level. An operand bundle is essentially a fake use that is attached to the call site. At the IR level, this looks like so:

```
call void inttoptr (i64 ... to void (double*)*)(double* %5) [ "
    jl_roots"(%jl_value_t addrspace(10)* %A) ]
```

The GC root placement pass will treat the `jl_roots` operand bundle as if it were a regular operand. However, as a final step, after the GC roots are inserted, it will drop the operand bundle to avoid confusing instruction selection.

### Supporting `pointer_from_objref`

`pointer_from_objref` is special because it requires the user to take explicit control of GC rooting. By our above invariants, this function is illegal, because it performs an address space cast from 10 to 0. However, it can be useful, in certain situations, so we provide a special intrinsic:

```
declared %jl_value_t *julia.pointer_from_objref(%jl_value_t
    addrspace(10)*)
```

which is lowered to the corresponding address space cast after GC root lowering. Do note however that by using this intrinsic, the caller assumes all responsibility for making sure that the value in question is rooted. Further this intrinsic is not considered a use, so the GC root placement pass will not provide a GC root for the function. As a result, the external rooting must be arranged while the value is still tracked by the system. I.e. it is not valid to attempt to

The result of this [CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS](#)  
have already dropped the value.

Keeping values alive in the absence of uses

In certain cases it is necessary to keep an object alive, even though there is no compiler-visible use of said object. This may be case for low level code that operates on the memory-representation of an object directly or code that needs to interface with C code. In order to allow this, we provide the following intrinsics at the LLVM level:

```
token @llvm.julia.gc_preserve_begin(...)  
void @llvm.julia.gc_preserve_end(token)
```

(The `llvm.` in the name is required in order to be able to use the `token` type). The semantics of these intrinsics are as follows: At any safepoint that is dominated by a `gc_preserve_begin` call, but that is not dominated by a corresponding `gc_preserve_end` call (i.e. a call whose argument is the token returned by a `gc_preserve_begin` call), the values passed as arguments to that `gc_preserve_begin` will be kept live. Note that the `gc_preserve_begin` still counts as a regular use of those values, so the standard lifetime semantics will ensure that the values will be kept alive before entering the preserve region.

## 78.14 printf() and stdio in the Julia runtime

Libuv wrappers for stdio

`julia.h` defines `libuv` wrappers for the `stdio.h` streams:

```
uv_stream_t *JL_STDIN;  
uv_stream_t *JL_STDOUT;  
uv_stream_t *JL_STDERR;
```

... and corresponding output functions:

```
| 7814_PRINTE() AND STDIO IN THE JULIA RUNTIME           1723
| int jl_printf(uv_stream_t *s, const char *format, ...);
| int jl_vprintf(uv_stream_t *s, const char *format, va_list args);
```

These `printf` functions are used by the `.c` files in the `src/` and `ui/` directories wherever stdio is needed to ensure that output buffering is handled in a unified way.

In special cases, like signal handlers, where the full libuv infrastructure is too heavy, `jl_safe_printf()` can be used to [write\(2\)](#) directly to `STDERR_FILENO`:

```
| void jl_safe_printf(const char *str, ...);
```

Interface between `JL_STD*` and Julia code

`Base.STDIN`, `Base.STDOUT` and `Base.STDERR` are bound to the `JL_STD*` libuv streams defined in the runtime.

Julia's `__init__()` function (in `base/sysimg.jl`) calls `reinit_stdio()` (in `base/stream.jl`) to create Julia objects for `Base.STDIN`, `Base.STDOUT` and `Base.STDERR`.

`reinit_stdio()` uses `ccall` to retrieve pointers to `JL_STD*` and calls `jl_uv_handle_type()` to inspect the type of each stream. It then creates a Julia `Base.IOStream`, `Base.TTY` or `Base.PipeEndpoint` object to represent each stream, e.g.:

```
$ julia -e 'println(typeof((STDIN, STDOUT, STDERR)))'
Tuple{Base.TTY,Base.TTY,Base.TTY}

$ julia -e 'println(typeof((STDIN, STDOUT, STDERR)))' < /dev/null
2>/dev/null
Tuple{IOStream,Base.TTY,IOStream}

$ echo hello | julia -e 'println(typeof((STDIN, STDOUT, STDERR)))'
| cat
```

The `Base.read` and `Base.write` methods for these streams use `ccall` to call libuv wrappers in `src/jl_uv.c`, e.g.:

```
stream.jl: function write(s::IO, p::Ptr, nb::Integer)
           -> ccall(:jl_uv_write, ...)
jl_uv.c:      -> int jl_uv_write(uv_stream_t *stream, ...)
           -> uv_write(uvw, stream, buf, ...)
```

`printf()` during initialization

The libuv streams relied upon by `jl_printf()` etc., are not available until midway through initialization of the runtime (see `init.c`, `init_stdio()`). Error messages or warnings that need to be printed before this are routed to the standard C library `fwrite()` function by the following mechanism:

In `sys.c`, the `JL_STD*` stream pointers are statically initialized to integer constants: `STD*_FILENO` (`0`, `1` and `2`). In `jl_uv.c` the `jl_uv_puts()` function checks its `uv_stream_t*` `stream` argument and calls `fwrite()` if `stream` is set to `STDOUT_FILENO` or `STDERR_FILENO`.

This allows for uniform use of `jl_printf()` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

## Legacy `ios.c` library

The `src/support/ios.c` library is inherited from `femtolisp`. It provides cross-platform buffered file IO and in-memory temporary buffers.

`ios.c` is still used by:

`src/flisp/*.c`

`src/dump.c` – for serialization file IO and for memory buffers.

`base/iostream.jl` – for file IO (see `base/fs.jl` for libuv equivalent).

Use of `ios.c` in these modules is mostly self-contained and separated from the libuv I/O system. However, there is [one place](#) where femtolisp calls through to `jl_printf()` with a legacy `ios_t` stream.

There is a hack in `ios.h` that makes the `ios_t.bm` field line up with the `uv_stream_t.type` and ensures that the values used for `ios_t.bm` do not overlap with valid UV\_HANDLE\_TYPE values. This allows `uv_stream_t` pointers to point to `ios_t` streams.

This is needed because `jl_printf()` caller `jl_static_show()` is passed an `ios_t` stream by femtolisp's `f1_print()` function. Julia's `jl_uv_puts()` function has special handling for this:

```
|if (stream->type > UV_HANDLE_TYPE_MAX) {
    return ios_write((ios_t*)stream, str, n);
}
```

## 78.15 Bounds checking

Like many modern programming languages, Julia uses bounds checking to ensure program safety when accessing arrays. In tight inner loops or other performance critical situations, you may wish to skip these bounds checks to improve runtime performance. For instance, in order to emit vectorized (SIMD) instructions, your loop body cannot contain branches, and thus cannot contain bounds checks. Consequently, Julia includes an `@inbounds(...)` macro to tell the compiler to skip such bounds checks within the given block. User-defined array types can use the `@boundscheck(...)` macro to achieve context-sensitive code selection.

## ~~Eliding~~ bounds checks

## CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

The `@boundscheck(...)` macro marks blocks of code that perform bounds checking. When such blocks are inlined into an `@inbounds(...)` block, the compiler may remove these blocks. The compiler removes the `@boundscheck` block only if it is inlined into the calling function. For example, you might write the method `sum` as:

```
function sum(A::AbstractArray)
    r = zero(eltype(A))
    for i = 1:length(A)
        @inbounds r += A[i]
    end
    return r
end
```

With a custom array-like type `MyArray` having:

```
@inline getindex(A::MyArray, i::Real) = (@boundscheck
    ↳ checkbounds(A, i); A.data[to_index(i)])
```

Then when `getindex` is inlined into `sum`, the call to `checkbounds(A, i)` will be elided. If your function contains multiple layers of inlining, only `@boundscheck` blocks at most one level of inlining deeper are eliminated. The rule prevents unintended changes in program behavior from code further up the stack.

### Propagating inbounds

There may be certain scenarios where for code-organization reasons you want more than one layer between the `@inbounds` and `@boundscheck` declarations. For instance, the default `getindex` methods have the chain `getindex(A::AbstractArray, i::Real)` calls `getindex(IndexStyle(A), A, i)` calls `_getindex(:dexLinear, A, i)`.

78.15. ~~OVERBOUNDS CHECKING~~ of inlining" rule, a function may be marked with ~~with~~ `@propagate_inbounds` to propagate an inbounds context (or out of bounds context) through one additional layer of inlining.

The bounds checking call hierarchy

The overall hierarchy is:

`checkbounds(A, I...)` which calls

- `checkbounds(Bool, A, I...)` which calls
  - \* `checkbounds_indices(Bool, indices(A), I)` which recursively calls
    - `checkindex` for each dimension

Here `A` is the array, and `I` contains the "requested" indices. `indices(A)` returns a tuple of "permitted" indices of `A`.

`checkbounds(A, I...)` throws an error if the indices are invalid, whereas `checkbounds(Bool, A, I...)` returns `false` in that circumstance. `checkbounds_indices` discards any information about the array other than its `indices` tuple, and performs a pure indices-vs-indices comparison: this allows relatively few compiled methods to serve a huge variety of array types. Indices are specified as tuples, and are usually compared in a 1-1 fashion with individual dimensions handled by calling another important function, `checkindex`: typically,

```
checkbounds_indices(Bool, (IA1, IA...), (I1, I...)) =  
  ↳ checkindex(Bool, IA1, I1) &  
    ↳ check-  
      ↳ bounds_in-  
      ↳ dices(Bool,  
      ↳ IA, I)
```

**checkindex** checks if indices are valid. The **checkbounds** function is documented in the [internals](#) module. The unexported **checkbounds\_indices** have docstrings accessible with `? .`

If you have to customize bounds checking for a specific array type, you should specialize **checkbounds(Bool, A, I...)**. However, in most cases you should be able to rely on **checkbounds\_indices** as long as you supply useful **indices** for your array type.

If you have novel index types, first consider specializing **checkindex**, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to **CartesianIndex**), then you may have to consider specializing **checkbounds\_indices**.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make **checkbounds** the place to specialize on array type, and try to avoid specializations on index types; conversely, **checkindex** is intended to be specialized only on index type (especially, the last argument).

## 78.16 Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

1. structure code such that only one lock will need to be acquired at a time
2. always acquire shared locks in the same order, as given by the table below
3. avoid constructs that expect to need unrestricted recursion

### Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed

## REF ID: PROPER MAINTENANCE AND CARE OF MULTI-THREADING LOCKS

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

safepoint

Note that this lock is acquired implicitly by JL\_LOCK and JL\_UNLOCK. use the \_NOCG variants to avoid that for level 1 locks.

While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.

shared\_map

finalizers

pagealloc

gc\_perm\_lock

flisp

flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

typecache

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

Method->writelock

The following is a level 3 lock, which documentation of each Julia's `STRUCTS`  
3 locks:

`MethodTable->writelock`

No Julia code may be called while holding a lock above this point.

The following is a level 6 lock, which can only recurse to acquire locks at lower levels:

`codegen`

The following is an almost root lock (level end-1), meaning only the root lock may be held when trying to acquire it:

`typeinf`

this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points  
currently the lock is merged with the `codegen` lock, since they call each other recursively

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

`toplevel`

this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!

additionally, it's unclear if any code can safely run in parallel with an arbitrary `toplevel` expression, so it may require all threads to get to a safepoint first

The following locks are broken:

toplevel

doesn't exist right now

fix: create it

## Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache, kwsorter type) : MethodTable->write-lock

Type declarations : toplevel lock

Type application : typecache lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance updates : codegen lock

These fields are generally lazy initialized, using the test-and-test-and-set pattern.

These are set at construction and immutable:

- specTypes
- sparam\_vals
- def

These are set by `jl_type_infer` (while holding codegen lock):

- rettype

1732 – inferred CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS  
– these can also be reset, see `jl_set_lambda_retttype` for that logic as it needs to keep `functionObjectsDecls` in sync `inInference` flag:

- optimization to quickly avoid recurring into `jl_type_infer` while it is already running
- actual state (of setting `inferred`, then `fptr`) is protected by codegen lock

Function pointers (`jlcall_api` and `fptr, unspecialized_ductape`):

- these transition once, from `NULL` to a value, while the codegen lock is held

Code-generator cache (the contents of `functionObjectsDecls`):

- these can transition multiple times, but only while the codegen lock is held
- it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as `retttype`) and assume it is coordinated, unless also holding the codegen lock

`compile_traced` flag:

- unknown

LLVMContext : codegen lock

Method : Method -> writelock

roots array (serializer and codegen)

invoke / specializations / tfunc modifications

Julia 0.5 adds experimental support for arrays with arbitrary indices. Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A, d)` (and not just `0:size(A, d)-1`, either). Such array types are expected to be supplied through packages.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia.

## Generalizing existing code

As an overview, the steps are:

replace many uses of `size` with `indices`

replace `1:length(A)` with `eachindex(A)`, or in some cases `linearindices(A)`

replace `length(A)` with `length(linearindices(A))`

replace explicit allocations like `Array{Int}(size(B))` with `similar(Array{Int}, indices(B))`

These are described in more detail below.

## Background

Because unconventional indexing breaks deeply-held assumptions throughout the Julia ecosystem, early adopters running code that has not been updated

likely to experience results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) ||
        throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1. Previously that was a safe assumption, so this code was fine, but (depending on what types the user passes to this function) it may no longer be safe. If this code continued to work when passed a vector with non-1 indices, it would either produce an incorrect answer or it would segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

To ensure that such errors are caught, in Julia 0.5 both `length` and `size` should throw an error when passed an array with non-1 indexing. This is designed to force users of such arrays to check the code, and inspect it for whether it needs to be generalized.

Using `indices` for bounds checks and loop iteration

`indices(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange` objects, specifying the range of valid indices along each dimension of A. When A has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension d, there is `indices(A, d)`.

~~Base7 implements WITH CUSTOM INDICES~~, ~~OneTo~~, where ~~OneTo(n)~~ means ~~the~~ same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new [AbstractArray](#) type, this is the default returned by `indices`, and it indicates that this array type uses "conventional" 1-based indexing. Note that if you don't want to be bothered supporting arrays with non-1 indexing, you can add the following line:

```
|@assert all(x->isa(x, Base.OneTo), indices(A))
```

at the top of any function.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

## Linear indexing (`linearindices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., [AbstractVector](#)): does `v[i]` mean linear indexing , or Cartesian indexing with the array's native indices?

For this reason, your best option may be to iterate over the array with `eachindex(A)`, or, if you require the indices to be sequential integers, to get the index range by calling `linearindices(A)`. This will return `indices(A, 1)` if `A` is an `AbstractVector`, and the equivalent of `1:length(A)` otherwise.

By this definition, 1-dimensional arrays always use Cartesian indexing with the array's native indices. To help enforce this, it's worth noting that `sub2ind(shape, i...)` and `ind2sub(shape, ind)` will throw an error if `shape` indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than

4736le of OneTo). For CHAPTER 18 DOCUMENTATION OF THE API AND INTERNALS  
tinue to work the same as always.

Using `indices` and `linearindices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    indices(dest) == indices(src) ||
        → throw(DimensionMismatch("vectors must match"))
    for i in linearindices(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

Allocating storage using generalizations of `similar`

Storage is often allocated with `Array{Int}(dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use. Note that a convenient way of producing an all-zeros array that matches the indices of `A` is simply `zeros(A)`.

Let's walk through a couple of explicit examples. First, if `A` has conventional indices, then `similar(Array{Int}, indices(A))` would end up calling `Array{Int}(size(A))`, and thus return an array. If `A` is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, indices(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches `A`. (The most obvious implementation is to al-

~~CREATE ARRAYS WITH CUSTOM INDEXES~~ then "wrap" it in a type that shifts ~~THE~~ indices.)

Note also that `similar(Array{Int}, (indices(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of A.

## Deprecations

In generalizing Julia's code base, at least one deprecation was unavoidable: earlier versions of Julia defined `first(::Colon) = 1`, meaning that the first index along a dimension indexed by `:` is 1. This definition can no longer be justified, so it was deprecated. There is no provided replacement, because the proper replacement depends on what you are doing and might need to know more about the array. However, it appears that many uses of `first(::Colon)` are really about computing an index offset; when that is the case, a candidate replacement is:

```
indexoffset(r::AbstractVector) = first(r) - 1  
indexoffset(::Colon) = 0
```

In other words, while `first(:)` does not itself make sense, in general you can say that the offset associated with a colon-index is zero.

## Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see [Abstract Arrays](#). This page focuses on the steps needed to define unconventional indexing.

### Do not implement `size` or `length`

Perhaps the majority of pre-existing code that uses `size` will not work properly for arrays with non-1 indices. For that reason, it is much better to avoid

Implementing these methods is a good way to implement one of the internal code that needs to be audited and perhaps generalized.

Do not annotate bounds checks

Julia 0.5 includes `@boundscheck` to annotate code that can be removed for callers that exploit `@inbounds`. Initially, it seems far preferable to run with bounds checking always enabled (i.e., omit the `@boundscheck` annotation so the check always runs).

Custom `AbstractUnitRange` types

If you're writing a non-1 indexed array type, you will want to specialize `indices` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably not export `ZeroRange` from your package: there may be other packages that implement their own `ZeroRange`, and having multiple distinct `ZeroRange` types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package [CustomUnitRanges.jl](#) can sometimes be used to avoid the need to write your own `ZeroRange` type.

Specializing `indices`

Once you have your `AbstractUnitRange` type, then specialize `indices`:

where here we imagine that `ZeroArray` has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `indices(A, d)`:

```
indices(A::AbstractArray{T,N}, d) where {T,N} = d <= N ?  
    indices(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in `Base` there is a dedicated function `indices1` which is equivalent to `indices(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
indices1(A::AbstractArray{T,0}) where {T} = OneTo(1)  
indices1(A::AbstractArray) = indices(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

## Specializing `similar`

Given your custom `ZeroRange` type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type,  
    shape::Tuple{ZeroRange,Vararg{ZeroRange}})  
    # body  
end
```

```
function Base.similar(f::Union{Function, DataType},  
    shape::Tuple{ZeroRange,Vararg{ZeroRange}})
```

```
|1740 # body  
|  
|end
```

## CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

Both of these should allocate your custom array type.

### Specializing `reshape`

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange, Vararg{  
    ZeroRange}}) = ...
```

and you can `reshape` an array so that the result has custom indices.

### Summary

Writing code that doesn't make assumptions about indexing requires a few extra abstractions, but hopefully the necessary changes are relatively straightforward.

As a reminder, this support is still experimental. While much of Julia's base code has been updated to support unconventional indexing, without a doubt there are many omissions that will be discovered only through usage. Moreover, at the time of this writing, most packages do not support unconventional indexing. As a consequence, early adopters should be prepared to identify and/or fix bugs. On the other hand, only through practical usage will it become clear whether this experimental feature should be retained in future versions of Julia; consequently, interested parties are encouraged to accept some ownership for putting it through its paces.

## 78.18 Base.LibGit2

The LibGit2 module provides bindings to [libgit2](#), a portable C library that implements core functionality for the [Git](#) version control system. These bindings

~~7818~~ `Base.LibGit2` currently used to power Julia's package manager. It is expected that ~~741\$~~ module will eventually be moved into a separate package.

## Functionality

Some of this documentation assumes some prior knowledge of the libgit2 API. For more information on some of the objects and methods referenced here, consult the upstream [libgit2 API reference](#).

`Base.LibGit2.Buffer` – Type.

```
| LibGit2.Buffer
```

A data buffer for exporting data from libgit2. Matches the `git_buf` struct.

When fetching data from LibGit2, a typical usage would look like:

```
| buf_ref = Ref(Buffer())
| @check ccall(..., (Ptr{Buffer},), buf_ref)
| # operation on buf_ref
| free(buf_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

[source](#)

`Base.LibGit2.CheckoutOptions` – Type.

```
| LibGit2.CheckoutOptions
```

Matches the `git_checkout_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

1742

## checkout\_strategy

to force the checkout/recreate missing files.

**disable\_filters**: if nonzero, do not apply filters like CLRF (to convert file newlines between UNIX and DOS).

**dir\_mode**: read/write/access mode for any directories involved in the checkout. Default is **0755**.

**file\_mode**: read/write/access mode for any files involved in the checkout. Default is **0755** or **0644**, depending on the blob.

**file\_open\_flags**: bitflags used to open any files during the checkout.

**notify\_flags**: Flags for what sort of conflicts the user should be notified about.

**notify\_cb**: An optional callback function to notify the user if a checkout conflict occurs. If this function returns a non-zero value, the checkout will be cancelled.

**notify\_payload**: Payload for the notify callback function.

**progress\_cb**: An optional callback function to display checkout progress.

**progress\_payload**: Payload for the progress callback.

**paths**: If not empty, describes which paths to search during the checkout. If empty, the checkout will occur over all files in the repository.

**baseline**: Expected content of the [workdir](#), captured in a (pointer to a) [GitTree](#). Defaults to the state of the tree at HEAD.

**baseline\_index**: Expected content of the [workdir](#), captured in a (pointer to a) [GitIndex](#). Defaults to the state of the index at HEAD.

**target\_directory**: If not empty, checkout to this directory instead of the [workdir](#).

78.18. `ancestor_label`: In case of conflicts, the name of the common ancestor side.<sup>743</sup>

`our_label`: In case of conflicts, the name of "our" side.

`their_label`: In case of conflicts, the name of "their" side.

`perfdata_cb`: An optional callback function to display performance data.

`perfdata_payload`: Payload for the performance callback.

## source

[Base.LibGit2.CloneOptions](#) – Type.

### | [LibGit2.CloneOptions](#)

Matches the `git_clone_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

`checkout_opts`: The options for performing the checkout of the remote as part of the clone.

`fetch_opts`: The options for performing the pre-checkout fetch of the remote as part of the clone.

`bare`: If 0, clone the full remote repository. If non-zero, perform a bare clone, in which there is no local copy of the source files in the repository and the `gitdir` and `workdir` are the same.

`localclone`: Flag whether to clone a local object database or do a fetch. The default is to let git decide. It will not use the git-aware transport for a local clone, but will use it for URLs which begin with `file://`.

1744 **checkout\_branch**: A string containing the name of the branch to check out. If this is a string, the default branch of the remote will be checked out.

**repository\_cb**: An optional callback which will be used to create the new repository into which the clone is made.

**repository\_cb\_payload**: The payload for the repository callback.

**remote\_cb**: An optional callback used to create the [GitRemote](#) before making the clone from it.

**remote\_cb\_payload**: The payload for the remote callback.

## source

[Base.LibGit2.DescribeOptions](#) – Type.

### [LibGit2.DescribeOptions](#)

Matches the [git\\_describe\\_options](#) struct.

The fields represent:

**version**: version of the struct in use, in case this changes later. For now, always 1.

**max\_candidates\_tags**: consider this many most recent tags in **refs/tags** to describe a commit. Defaults to 10 (so that the 10 most recent tags would be examined to see if they describe a commit).

**describe\_strategy**: whether to consider all entries in **refs/tags** (equivalent to `git-describe --tags`) or all entries in **refs/** (equivalent to `git-describe --all`). The default is to only show annotated tags. If `Consts.DESCRIBE_TAGS` is passed, all tags, annotated or not, will be considered. If `Consts.DESCRIBE_ALL` is passed, any ref in **refs/** will be considered.

**pattern**: only consider tags which match **pattern**. Supports glob expansion.

78.18. `Base.LibGit2.ShowFirstParent`: when finding the distance from a matching reference to the described object, only consider the distance from the first parent.

`show_commit_oid_as_fallback`: if no matching reference can be found which describes a commit, show the commit's `GitHash` instead of throwing an error (the default behavior).

`source`

`Base.LibGit2.DescribeFormatOptions` – Type.

| `LibGit2.DescribeFormatOptions`

Matches the `git_describe_format_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

`abbreviated_size`: lower bound on the size of the abbreviated `GitHash` to use, defaulting to 7.

`always_use_long_format`: set to 1 to use the long format for strings even if a short format can be used.

`dirty_suffix`: if set, this will be appended to the end of the description string if the `workdir` is dirty.

`source`

`Base.LibGit2.DiffDelta` – Type.

| `LibGit2.DiffDelta`

Description of changes to one entry. Matches the `git_diff_delta` struct.

The fields represent:

**1746    status:** One of `ADDED`, `CHANGED`, `DELETED`, `MISSING`, `UNKNOWN`, or `UNTRACKED`. A file has been added/modified/deleted.

**flags:** Flags for the delta and the objects on each side. Determines whether to treat the file(s) as binary/text, whether they exist on each side of the diff, and whether the object ids are known to be correct.

**similarity:** Used to indicate if a file has been renamed or copied.

**nfiles:** The number of files in the delta (for instance, if the delta was run on a submodule commit id, it may contain more than one file).

**old\_file:** A [DiffFile](#) containing information about the file(s) before the changes.

**new\_file:** A [DiffFile](#) containing information about the file(s) after the changes.

## source

[Base.LibGit2.DiffFile](#) – Type.

### [LibGit2.DiffFile](#)

Description of one side of a delta. Matches the [git\\_diff\\_file](#) struct.

The fields represent:

**id:** the [GitHash](#) of the item in the diff. If the item is empty on this side of the diff (for instance, if the diff is of the removal of a file), this will be `GitHash(0)`.

**path:** a NULL terminated path to the item relative to the working directory of the repository.

**size:** the size of the item in bytes.

**flags:** a combination of the [git\\_diff\\_flag\\_t](#) flags. The `i`th bit of this integer sets the `i`th flag.

`id_abbrev`: only present in LibGit2 versions newer than or equal to 0.25.0. The length of the `id` field when converted using `hex`. Usually equal to `OID_HEXSZ` (40).

`source`

`Base.LibGit2.DiffOptionsStruct` – Type.

| `LibGit2.DiffOptionsStruct`

Matches the `git_diff_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

`flags`: flags controlling which files will appear in the diff. Defaults to `DIFF_NORMAL`.

`ignore_submodules`: whether to look at files in submodules or not. Defaults to `SUBMODULE_IGNORE_UNSPECIFIED`, which means the submodule's configuration will control whether it appears in the diff or not.

`pathspec`: path to files to include in the diff. Default is to use all files in the repository.

`notify_cb`: optional callback which will notify the user of changes to the diff as file deltas are added to it.

`progress_cb`: optional callback which will display diff progress. Only relevant on libgit2 versions at least as new as 0.24.0.

`payload`: the payload to pass to `notify_cb` and `progress_cb`.

`context_lines`: the number of unchanged lines used to define the edges of a hunk. This is also the number of lines which will be shown before/after a hunk to provide context. Default is 3.

1748 **interhunk\_max\_size**: the maximum size in bytes between two separate hunks allowed before the hunks will be combined. Default is 0.

**id\_abbrev**: sets the length of the abbreviated [GitHash](#) to print. Default is 7.

**max\_size**: the maximum file size of a blob. Above this size, it will be treated as a binary blob. The default is 512 MB.

**old\_prefix**: the virtual file directory in which to place old files on one side of the diff. Default is "a".

**new\_prefix**: the virtual file directory in which to place new files on one side of the diff. Default is "b".

## source

[Base.LibGit2.FetchHead](#) – Type.

### [LibGit2.FetchHead](#)

Contains the information about HEAD during a fetch, including the name and URL of the branch fetched from, the oid of the HEAD, and whether the fetched HEAD has been merged locally.

The fields represent:

**name**: The name in the local reference database of the fetch head, for example, "refs/heads/master".

**url**: The URL of the fetch head.

**oid**: The [GitHash](#) of the tip of the fetch head.

**ismerge**: Boolean flag indicating whether the changes at the remote have been merged into the local copy yet or not. If **true**, the local copy is up to date with the remote fetch head.

[Base.LibGit2.FetchOptions](#) – Type.

| [LibGit2.FetchOptions](#)

Matches the [git\\_fetch\\_options](#) struct.

The fields represent:

**version**: version of the struct in use, in case this changes later. For now, always 1.

**callbacks**: remote callbacks to use during the fetch.

**prune**: whether to perform a prune after the fetch or not. The default is to use the setting from the [GitConfig](#).

**update\_fetchhead**: whether to update the [FetchHead](#) after the fetch. The default is to perform the update, which is the normal git behavior.

**download\_tags**: whether to download tags present at the remote or not. The default is to request the tags for objects which are being downloaded anyway from the server.

**proxy\_opts**: options for connecting to the remote through a proxy. See [ProxyOptions](#). Only present on libgit2 versions newer than or equal to 0.25.0.

**custom\_headers**: any extra headers needed for the fetch. Only present on libgit2 versions newer than or equal to 0.24.0.

**source**

[Base.LibGit2.GitAnnotated](#) – Type.

| [GitAnnotated\(repo::GitRepo, commit\\_id::GitHash\)](#)  
| [GitAnnotated\(repo::GitRepo, ref::GitReference\)](#)

```
1750 |     GitAnnotated(repo::GitRepo, th::FetchHead)
|     GitAnnotated(repo::GitRepo, comittish::AbstractString)
```

An annotated git commit carries with it information about how it was looked up and why, so that rebase or merge operations have more information about the context of the commit. Conflict files contain information about the source/target branches in the merge which are conflicting, for instance. An annotated commit can refer to the tip of a remote branch, for instance when a [FetchHead](#) is passed, or to a branch head described using [GitReference](#).

[source](#)

[Base.LibGit2.GitBlame](#) – Type.

```
|     GitBlame(repo::GitRepo, path::AbstractString; options::
|               BlameOptions=BlameOptions())
```

Construct a [GitBlame](#) object for the file at `path`, using change information gleaned from the history of `repo`. The [GitBlame](#) object records who changed which chunks of the file when, and how. `options` controls how to separate the contents of the file and which commits to probe – see [BlameOptions](#) for more information.

[source](#)

[Base.LibGit2.GitBlob](#) – Type.

```
|     GitBlob(repo::GitRepo, hash::AbstractGitHash)
|     GitBlob(repo::GitRepo, spec::AbstractString)
```

Return a [GitBlob](#) object from `repo` specified by `hash/spec`.

`hash` is a full ([GitHash](#)) or partial ([GitShortHash](#)) hash.

`spec` is a textual specification: see [the git docs](#) for a full list.

**Base.LibGit2.GitCommit** – Type.

```
| GitCommit(repo::GitRepo, hash::AbstractGitHash)
| GitCommit(repo::GitRepo, spec::AbstractString)
```

Return a **GitCommit** object from **repo** specified by **hash/spec**.

**hash** is a full (**GitHash**) or partial (**GitShortHash**) hash.

**spec** is a textual specification: see [the git docs](#) for a full list.

**source**

**Base.LibGit2.GitHash** – Type.

```
| GitHash
```

A git object identifier, based on the sha-1 hash. It is a 20 byte string (40 hex digits) used to identify a **GitObject** in a repository.

**source**

**Base.LibGit2.GitObject** – Type.

```
| GitObject(repo::GitRepo, hash::AbstractGitHash)
| GitObject(repo::GitRepo, spec::AbstractString)
```

Return the specified object ([GitCommit](#), [GitBlob](#), [GitTree](#) or [GitTag](#)) from **repo** specified by **hash/spec**.

**hash** is a full (**GitHash**) or partial (**GitShortHash**) hash.

**spec** is a textual specification: see [the git docs](#) for a full list.

**source**

**Base.LibGit2.GitRemote** – Type.

1752 | GitRemote(repo::GitRepo, rmt\_name::AbstractString, rmt\_url::AbstractString) -> GitRemote

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemote(repo, "upstream", repo_url)

source
GitRemote(repo::GitRepo, rmt_name::AbstractString, rmt_url::AbstractString, fetch_spec::AbstractString) -> GitRemote
```

Look up a remote git repository using the repository's name and URL, as well as specifications for how to fetch from the remote (e.g. which remote branch to fetch from).

Examples

```
repo = LibGit2.init(repo_path)
refspec = "+refs/heads/mybranch:refs/remotes/origin/mybranch"
remote = LibGit2.GitRemote(repo, "upstream", repo_url,
                           ↳ refspect)

source
```

[Base.LibGit2.GitRemoteAnon](#) – Function.

```
GitRemoteAnon(repo::GitRepo, url::AbstractString) -> GitRemote
```

Look up a remote git repository using only its URL, not its name.

Examples

```
repo = LibGit2.init(repo_path)
remote = LibGit2.GitRemoteAnon(repo, repo_url)
```

[Base.LibGit2.GitRepo](#) – Type.

```
| LibGit2.GitRepo(path::AbstractString)
```

Open a git repository at `path`.

[source](#)

[Base.LibGit2.GitRepoExt](#) – Function.

```
| LibGit2.GitRepoExt(path::AbstractString, flags::Cuint = Cuint(
```

```
|   Consts.REPOSITORY_OPEN_DEFAULT))
```

Open a git repository at `path` with extended controls (for instance, if the current user must be a member of a special access group to read `path`).

[source](#)

[Base.LibGit2.GitRevWalker](#) – Type.

```
| GitRevWalker(repo::GitRepo)
```

A `GitRevWalker` walks through the revisions (i.e. commits) of a git repository `repo`. It is a collection of the commits in the repository, and supports iteration and calls to `map` and `count` (for instance, `count` could be used to determine what percentage of commits in a repository were made by a certain author).

```
| cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
```

```
|   count((oid, repo)->(oid == commit_oid1), walker,
```

```
|     → oid=commit_oid1, by=LibGit2.Consts.SORT_TIME)
```

```
| end
```

Here, `count` finds the number of commits along the walk with a certain `GitHash`. Since the `GitHash` is unique to a commit, `cnt` will be 1.

[source](#)

```
| GitShortHash(hash::GitHash, len::Integer)
```

A shortened git object identifier, which can be used to identify a git object when it is unique, consisting of the initial `len` hexadecimal digits of `hash` (the remaining digits are ignored).

`source`

[Base.LibGit2.GitSignature](#) – Type.

```
| LibGit2.GitSignature
```

This is a Julia wrapper around a pointer to a `git_signature` object.

`source`

[Base.LibGit2.GitStatus](#) – Type.

```
| LibGit2.GitStatus(repo::GitRepo; status_opts=StatusOptions())
```

Collect information about the status of each file in the git repository `repo` (e.g. is the file modified, staged, etc.). `status_opts` can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not. See [StatusOptions](#) for more information.

`source`

[Base.LibGit2.GitTag](#) – Type.

```
| GitTag(repo::GitRepo, hash::AbstractGitHash)
```

```
| GitTag(repo::GitRepo, spec::AbstractString)
```

Return a `GitTag` object from `repo` specified by `hash/spec`.

`hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.

`spec` is a textual specification: see the [git docs](#) for a full list.

[Base.LibGit2.GitTree](#) – Type.

```
| GitTree(repo::GitRepo, hash::AbstractGitHash)  
| GitTree(repo::GitRepo, spec::AbstractString)
```

Return a `GitTree` object from `repo` specified by `hash/spec`.

`hash` is a full (`GitHash`) or partial (`GitShortHash`) hash.

`spec` is a textual specification: see [the git docs](#) for a full list.

[source](#)

[Base.LibGit2.IndexEntry](#) – Type.

```
| LibGit2.IndexEntry
```

In-memory representation of a file entry in the index. Matches the `git_index_entry` struct.

[source](#)

[Base.LibGit2.IndexTime](#) – Type.

```
| LibGit2.IndexTime
```

Matches the `git_index_time` struct.

[source](#)

[Base.LibGit2.BlameOptions](#) – Type.

```
| LibGit2.BlameOptions
```

Matches the `git_blame_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

1756    **flags**: one of ~~CHAPTER.BLAMENORMALIZATIONCONSTS.BLAMESFIRSTNAPARENTPARENT~~  
ENT (the other blame flags are not yet implemented by libgit2).

**min\_match\_characters**: the minimum number of alphanumeric characters which much change in a commit in order for the change to be associated with that commit. The default is 20. Only takes effect if one of the `Consts.BLAME_*_COPIES` flags are used, which libgit2 does not implement yet.

**newest\_commit**: the [GitHash](#) of the newest commit from which to look at changes.

**oldest\_commit**: the [GitHash](#) of the oldest commit from which to look at changes.

**min\_line**: the first line of the file from which to starting blaming. The default is 1.

**max\_line**: the last line of the file to which to blame. The default is 0, meaning the last line of the file.

## source

[Base.LibGit2.MergeOptions](#) – Type.

### | [LibGit2.MergeOptions](#)

Matches the [git\\_merge\\_options](#) struct.

The fields represent:

**version**: version of the struct in use, in case this changes later. For now, always 1.

**flags**: an `enum` for flags describing merge behavior. Defined in [git\\_merge\\_flag\\_t](#). The corresponding Julia enum is `GIT_MERGE` and has values:

78.18. ~~BASIC\_MERGE~~<sup>MERGE\_IT2</sup><sub>FIND\_RENAMES</sub>: detect if a file has been renamed between the common ancestor and the "ours" or "theirs" side of the merge.  
Allows merges where a file has been renamed.

- `MERGE_FAIL_ON_CONFLICT`: exit immediately if a conflict is found rather than trying to resolve it.
- `MERGE_SKIP_REUC`: do not write the REUC extension on the index resulting from the merge.
- `MERGE_NO_RECURSIVE`: if the commits being merged have multiple merge bases, use the first one, rather than trying to recursively merge the bases.

`rename_threshold`: how similar two files must be to consider one a rename of the other. This is an integer that sets the percentage similarity. The default is 50.

`target_limit`: the maximum number of files to compare with to look for renames. The default is 200.

`metric`: optional custom function to use to determine the similarity between two files for rename detection.

`recursion_limit`: the upper limit on the number of merges of common ancestors to perform to try to build a new virtual merge base for the merge. The default is no limit. This field is only present on libgit2 versions newer than 0.24.0.

`default_driver`: the merge driver to use if both sides have changed. This field is only present on libgit2 versions newer than 0.25.0.

`file_favor`: how to handle conflicting file contents for the `text` driver.

- `MERGE_FILE_FAVOR_NORMAL`: if both sides of the merge have changes to a section, make a note of the conflict in the index which git

- 1758 checkout  
reference to resolve the conflicts. This is the default.
- **MERGE\_FILE\_FAVOR\_OURS**: if both sides of the merge have changes to a section, use the version in the "ours" side of the merge in the index.
  - **MERGE\_FILE\_FAVOR\_THEIRS**: if both sides of the merge have changes to a section, use the version in the "theirs" side of the merge in the index.
  - **MERGE\_FILE\_FAVOR\_UNION**: if both sides of the merge have changes to a section, include each unique line from both sides in the file which is put into the index.

**file\_flags**: guidelines for merging files.

## source

[Base.LibGit2.ProxyOptions](#) – Type.

### | [LibGit2.ProxyOptions](#)

Options for connecting through a proxy.

Matches the [git\\_proxy\\_options](#) struct.

The fields represent:

**version**: version of the struct in use, in case this changes later. For now, always 1.

**proxytype**: an enum for the type of proxy to use. Defined in [git\\_proxy\\_t](#).

The corresponding Julia enum is `GIT_PROXY` and has values:

- **PROXY\_NONE**: do not attempt the connection through a proxy.
- **PROXY\_AUTO**: attempt to figure out the proxy configuration from the git configuration.

78.18. ~~BASED PROXY IS SPECIFIED~~: connect using the URL given in the `url` field of this struct.

Default is to auto-detect the proxy type.

`url`: the URL of the proxy.

`credential_cb`: a pointer to a callback function which will be called if the remote requires authentication to connect.

`certificate_cb`: a pointer to a callback function which will be called if certificate verification fails. This lets the user decide whether or not to keep connecting. If the function returns 1, connecting will be allowed. If it returns 0, the connection will not be allowed. A negative value can be used to return errors.

`payload`: the payload to be provided to the two callback functions.

Examples

```
julia> fo = LibGit2.FetchOptions(  
  
           proxy_opts = LibGit2.ProxyOptions(url =  
           ← Cstring("https://my_proxy_url.com")))  
  
julia> fetch(remote, "master", options=fo)
```

`source`

`Base.LibGit2.PushOptions` – Type.

| `LibGit2.PushOptions`

Matches the `git_push_options` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

1760 parallelism CHAPTER 18. FILE DOCUMENTATION OF THIS MODULE'S INTERNALS  
number of worker threads which will be spawned by the packbuilder.  
If 0, the packbuilder will auto-set the number of threads to use. The  
default is 1.

**callbacks**: the callbacks (e.g. for authentication with the remote) to  
use for the push.

**proxy\_opts**: only relevant if the LibGit2 version is greater than or  
equal to 0.25.0. Sets options for using a proxy to communicate with  
a remote. See [ProxyOptions](#) for more information.

**custom\_headers**: only relevant if the LibGit2 version is greater than  
or equal to 0.24.0. Extra headers needed for the push operation.

## source

[Base.LibGit2.RebaseOperation](#) – Type.

### [LibGit2.RebaseOperation](#)

Describes a single instruction/operation to be performed during the re-base. Matches the [git\\_rebase\\_operation](#) struct.

The fields represent:

**otype**: the type of rebase operation currently being performed. The  
options are:

- REBASE\_OPERATION\_PICK: cherry-pick the commit in question.
- REBASE\_OPERATION\_REWORD: cherry-pick the commit in question,  
but rewrite its message using the prompt.
- REBASE\_OPERATION\_EDIT: cherry-pick the commit in question, but  
allow the user to edit the commit's contents and its message.
- REBASE\_OPERATION\_SQUASH: squash the commit in question into  
the previous commit. The commit messages of the two commits will  
be merged.

78.18. ~~REBASE\_OPERATION\_FIXUP~~: squash the commit in question<sup>1761</sup>  
the previous commit. Only the commit message of the previous  
commit will be used.

- **REBASE\_OPERATION\_EXEC**: do not cherry-pick a commit. Run a  
command and continue if the command exits successfully.

**id**: the [GitHash](#) of the commit being worked on during this rebase  
step.

**exec**: in case **REBASE\_OPERATION\_EXEC** is used, the command to run  
during this step (for instance, running the test suite after each commit).

## source

[Base.LibGit2.RebaseOptions](#) – Type.

### [LibGit2.RebaseOptions](#)

Matches the `git_rebase_options` struct.

The fields represent:

**version**: version of the struct in use, in case this changes later. For  
now, always 1.

**quiet**: inform other git clients helping with/working on the rebase  
that the rebase should be done "quietly". Used for interoperability.  
The default is 1.

**inmemory**: start an in-memory rebase. Callers working on the rebase  
can go through its steps and commit any changes, but cannot rewind  
HEAD or update the repository. The [workdir](#) will not be modified.  
Only present on libgit2 versions newer than or equal to 0.24.0.

**rewrite\_notes\_ref**: name of the reference to notes to use to rewrite  
the commit notes as the rebase is finished.

1762 **merge\_opts**: CHAPTER 78 DOCUMENTATION AND THE FIELDS IN INTERNALS  
at each rebase step. Only present on libgit2 versions newer than or equal to 0.24.0.

**checkout\_opts**: checkout options for writing files when initializing the rebase, stepping through it, and aborting it. See [CheckoutOptions](#) for more information.

**source**

[Base.LibGit2.RemoteCallbacks](#) – Type.

| **LibGit2.RemoteCallbacks**

Callback settings. Matches the [git\\_remote\\_callbacks](#) struct.

**source**

[Base.LibGit2.SignatureStruct](#) – Type.

| **LibGit2.SignatureStruct**

An action signature (e.g. for committers, taggers, etc). Matches the [git\\_signature](#) struct.

The fields represent:

**name**: The full name of the committer or author of the commit.

**email**: The email at which the committer/author can be contacted.

**when**: a [TimeStruct](#) indicating when the commit was authored/committed into the repository.

**source**

[Base.LibGit2.StatusEntry](#) – Type.

| **LibGit2.StatusEntry**

78.1 [Base.LibGit2](#) Differences between the file as it exists in HEAD and ~~the~~<sup>763</sup> index, and providing the differences between the index and the working directory. Matches the `git_status_entry` struct.

The fields represent:

`status`: contains the status flags for the file, indicating if it is current, or has been changed in some way in the index or work tree.

`head_to_index`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in HEAD and in the index.

`index_to_workdir`: a pointer to a `DiffDelta` which encapsulates the difference(s) between the file as it exists in the index and in the `workdir`.

`source`

[Base.LibGit2.StatusOptions](#) – Type.

| `LibGit2.StatusOptions`

Options to control how `git_status_foreach_ext()` will issue callbacks.

Matches the `git_status_opt_t` struct.

The fields represent:

`version`: version of the struct in use, in case this changes later. For now, always 1.

`show`: a flag for which files to examine and in which order. The default is `Consts.STATUS_SHOW_INDEX_AND_WORKDIR`.

`flags`: flags for controlling any callbacks used in a status call.

`pathspec`: an array of paths to use for path-matching. The behavior of the path-matching will vary depending on the values of `show` and `flags`.

`Base.LibGit2.StrArrayStruct` – Type.

```
| LibGit2.StrArrayStruct
```

A LibGit2 representation of an array of strings. Matches the `git_strarray` struct.

When fetching data from LibGit2, a typical usage would look like:

```
sa_ref = Ref(StrArrayStruct())
@check ccall(..., (Ptr{StrArrayStruct},), sa_ref)
res = convert(Vector{String}, sa_ref[])
free(sa_ref)
```

In particular, note that `LibGit2.free` should be called afterward on the `Ref` object.

Conversely, when passing a vector of strings to LibGit2, it is generally simplest to rely on implicit conversion:

```
strs = String[...]
@check ccall(..., (Ptr{StrArrayStruct},), strs)
```

Note that no call to `free` is required as the data is allocated by Julia.

`source`

`Base.LibGit2.TimeStruct` – Type.

```
| LibGit2.TimeStruct
```

Time in a signature. Matches the `git_time` struct.

`source`

`Base.LibGit2.add!` – Function.

```
78.18 BASE.LibGit2.add!(Repo::GitRepo, files::AbstractString...; flags::Cuint = 1765
    Consts.INDEX_ADD_DEFAULT)
add!(idx::GitIndex, files::AbstractString...; flags::Cuint =
    Consts.INDEX_ADD_DEFAULT)
```

Add all the files with paths specified by `files` to the index `idx` (or the index of the `repo`). If the file already exists, the index entry will be updated. If the file does not exist already, it will be newly added into the index. `files` may contain glob patterns which will be expanded and any matching files will be added (unless `INDEX_ADD_DISABLE_PATHSPEC_MATCH` is set, see below). If a file has been ignored (in `.gitignore` or in the config), it will not be added, unless it is already being tracked in the index, in which case it will be updated. The keyword argument `flags` is a set of bit-flags which control the behavior with respect to ignored files:

`Consts.INDEX_ADD_DEFAULT` – default, described above.

`Consts.INDEX_ADD_FORCE` – disregard the existing ignore rules and force addition of the file to the index even if it is already ignored.

`Consts.INDEX_ADD_CHECK_PATHSPEC` – cannot be used at the same time as `INDEX_ADD_FORCE`. Check that each file in `files` which exists on disk is not in the ignore list. If one of the files is ignored, the function will return `EINVALIDSPEC`.

`Consts.INDEX_ADD_DISABLE_PATHSPEC_MATCH` – turn off glob matching, and only add files to the index which exactly match the paths specified in `files`.

`source`

[Base.LibGit2.add\\_fetch!](#) – Function.

```
| add_fetch!(repo::GitRepo, rmt::GitRemote, fetch_spec::String)
```

mation about which branch(es) to fetch from.

### Examples

```
julia> LibGit2.add_fetch!(repo, remote, "upstream");
```

```
julia> LibGit2.fetch_refspecs(remote)
```

```
String["+refs/heads/*:refs/remotes/upstream/*"]
```

### source

[Base.LibGit2.add\\_push!](#) – Function.

```
add_push!(repo::GitRepo, rmt::GitRemote, push_spec::String)
```

Add a push refspec for the specified `rmt`. This refspec will contain information about which branch(es) to push to.

### Examples

```
julia> LibGit2.add_push!(repo, remote, "refs/heads/master");
```

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo, branch);
```

```
julia> LibGit2.push_refspecs(remote)
```

```
String["refs/heads/master"]
```

### Note

You may need to `close` and reopen the `GitRemote` in question after updating its push refs in order for the change to take effect and for calls to `push` to work.

### source

[Base.LibGit2.addblob!](#) – Function.

78.18 `LibGit2.addblob!(repo::GitRepo, path::AbstractString)`

1767

Reads the file at `path` and adds it to the object database of `repo` as a loose blob. Returns the [GitHash](#) of the resulting blob.

Examples

```
hash_str = hex(commit_oid)
blob_file = joinpath(repo_path, ".git", "objects",
    hash_str[1:2], hash_str[3:end])
id = LibGit2.addblob!(repo, blob_file)
```

[source](#)

[Base.LibGit2.author](#) – Function.

```
author(c::GitCommit)
```

Return the [Signature](#) of the author of the commit `c`. The author is the person who made changes to the relevant file(s). See also [committer](#).

[source](#)

[Base.LibGit2.authors](#) – Function.

```
authors(repo::GitRepo) -> Vector{Signature}
```

Returns all authors of commits to the `repo` repository.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")

println(repo_file, commit_msg)
flush(repo_file)
LibGit2.add!(repo, test_file)
```

```
1768 sig = LibGit2.Signature(TEST, TESTTEST.COM, roundtime),
    ↵  0), 0)
commit_oid1 = LibGit2.commit(repo, "commit1"; author=sig,
    ↵  committer=sig)
println(repo_file, randstring(10))
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit2"; author=sig,
    ↵  committer=sig)

# will be a Vector of [sig, sig]
auths = LibGit2.authors(repo)
```

**source**

[Base.LibGit2.branch](#) – Function.

```
| branch(repo::GitRepo)
```

Equivalent to `git branch`. Create a new branch from the current HEAD.

**source**

[Base.LibGit2.branch!](#) – Function.

```
| branch!(repo::GitRepo, branch_name::AbstractString, commit::
    ↵ AbstractString=""; kwargs...)
```

Checkout a new git branch in the `repo` repository. `commit` is the [GitHash](#), in string form, which will be the start of the new branch. If `commit` is an empty string, the current HEAD will be used.

The keyword arguments are:

`track::AbstractString=""`: the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.

78.18. `force_branch=false`: if `true`, branch creation will be forced. 1769

`set_head::Bool=true`: if `true`, after the branch creation finishes the branch head will be set as the HEAD of `repo`.

Equivalent to `git checkout [-b|-B] <branch_name> [<commit>] [--track <track>]`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.branch!(repo, "new_branch", set_head=false)
```

`source`

[Base.LibGit2.checkout!](#) – Function.

```
checkout!(repo::GitRepo, commit::AbstractString="" ; force::Bool
          =true)
```

Equivalent to `git checkout [-f] --detach <commit>`. Checkout the git commit `commit` (a [GitHash](#) in string form) in `repo`. If `force` is `true`, force the checkout and discard any current changes. Note that this detaches the current HEAD.

Examples

```
repo = LibGit2.init(repo_path)
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "111")
end
LibGit2.add!(repo, "file1")
commit_oid = LibGit2.commit(repo, "add file1")
open(joinpath(LibGit2.path(repo), "file1"), "w") do f
    write(f, "112")
```

1770")

CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

```
end

# would fail without the force=true
# since there are modifications to the file
LibGit2.checkout!(repo, string(commit_oid), force=true)
```

source

[Base.LibGit2.clone](#) – Function.

```
clone(repo_url::AbstractString, repo_path::AbstractString,
      clone_opts::CloneOptions)
```

Clone the remote repository at `repo_url` (which can be a remote URL or a path on the local filesystem) to `repo_path` (which must be a path on the local filesystem). Options for the clone, such as whether to perform a bare clone or not, are set by [CloneOptions](#).

Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo = LibGit2.clone(repo_url, "/home/me/projects/Example")
```

source

```
clone(repo_url::AbstractString, repo_path::AbstractString;
      kwargs...)
```

Clone a remote repository located at `repo_url` to the local filesystem location `repo_path`.

The keyword arguments are:

`branch::AbstractString=""`: which branch of the remote to clone, if not the default repository branch (usually `master`).

78.18. `isbare::Bool=false`: if `true`, clone the remote as a bare repository, which will make `repo_path` itself the git directory instead of `repo_path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.

`remote_cb::Ptr{Void}=C_NULL`: a callback which will be used to create the remote before it is cloned. If `C_NULL` (the default), no attempt will be made to create the remote – it will be assumed to already exist.

`payload::CredentialPayload=CredentialPayload()`: provides credentials and/or settings when authenticating against a private repository.

Equivalent to `git clone [-b <branch>] [--bare] <repo_url> <repo_path>`.

## Examples

```
repo_url = "https://github.com/JuliaLang/Example.jl"
repo1 = LibGit2.clone(repo_url, "test_path")
repo2 = LibGit2.clone(repo_url, "test_path", isbare=true)
julia_url = "https://github.com/JuliaLang/julia"
julia_repo = LibGit2.clone(julia_url, "julia_path",
    → branch="release-0.6")
```

`source`

`Base.LibGit2.commit` – Function.

```
commit(repo::GitRepo, msg::AbstractString; kwargs...) ->
    GitHash
```

Wrapper around `git_commit_create`. Create a commit in the repository `repo`. `msg` is the commit message. Return the OID of the new commit.

The keyword arguments are:

1772 ~~refname::AbstractString DOCUMENTATION FOR INTERNAL USE~~  
name of the reference to update to point to the new commit. For example, "HEAD" will update the HEAD of the current branch. If the reference does not yet exist, it will be created.

`author::Signature = Signature(repo)` is a `Signature` containing information about the person who authored the commit.

`committer::Signature = Signature(repo)` is a `Signature` containing information about the person who committed the commit to the repository. Not necessarily the same as `author`, for instance if `author` emailed a patch to `committer` who committed it.

`tree_id::GitHash = GitHash()` is a git tree to use to create the commit, showing its ancestry and relationship with any other history. `tree` must belong to `repo`.

`parent_ids::Vector<GitHash>=GitHash[ ]` is a list of commits by `GitHash` to use as parent commits for the new one, and may be empty. A commit might have multiple parents if it is a merge commit, for example.

`source`

```
| LibGit2.commit(rb::GitRebase, sig::GitSignature)
```

Commit the current patch to the rebase `rb`, using `sig` as the committer. Is silent if the commit has already been applied.

`source`

[Base.LibGit2.committer](#) – Function.

```
| committer(c::GitCommit)
```

Return the `Signature` of the committer of the commit `c`. The committer is the person who committed the changes originally authored by the `author`,

78.1.8 But `Base.LibGit2` be the same as the `author`, for example, if the `author` emailed a patch to a `committer` who committed it.

`source`

`Base.count` – Method.

```
LibGit2.count(f::Function, walker::GitRevWalker; oid::GitHash=GitHash(), by::Cint=Consts.SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` `walker` to “walk” over every commit in the repository’s history, find the number of commits which return `true` when `f` is applied to them. The keyword arguments are:

- \* `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors.
- \* `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`) or most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first).
- \* `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
cnt = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    count((oid, repo)->(oid == commit_oid1), walker,
          ↵ oid=commit_oid1, by=LibGit2.Consts.SORT_TIME)
end
```

`count` finds the number of commits along the walk with a certain `GitHash` `commit_oid1`, starting the walk from that commit and moving forwards in time from it. Since the `GitHash` is unique to a commit, `cnt` will be 1.

`source`

`Base.LibGit2.counthunks` – Function.

Return the number of distinct "hunks" with a file. A hunk may contain multiple lines. A hunk is usually a piece of a file that was added/changed/removed together, for example, a function added to a source file or an inner loop that was optimized out of that function later.

[source](#)

[Base.LibGit2.create\\_branch](#) – Function.

```
| LibGit2.create_branch(repo::GitRepo, bname::AbstractString,
    commit_obj::GitCommit; force::Bool=false)
```

Create a new branch in the repository `repo` with name `bname`, which points to commit `commit_obj` (which has to be part of `repo`). If `force` is `true`, overwrite an existing branch named `bname` if it exists. If `force` is `false` and a branch already exists named `bname`, this function will throw an error.

[source](#)

[Base.LibGit2.credentials\\_callback](#) – Function.

```
| credential_callback(...) -> Cint
```

A LibGit2 credential callback function which provides different credential acquisition functionality w.r.t. a connection protocol. The `payload_ptr` is required to contain a `LibGit2.CredentialPayload` object which will keep track of state and settings.

The `allowed_types` contains a bitmask of `LibGit2.Consts.GIT_CRED_TYPE` values specifying which authentication methods should be attempted.

Credential authentication is done in the following order (if supported):

- SSH agent

- SSH private/public key pair

If a user is presented with a credential prompt they can abort the prompt by typing ^D (pressing the control key together with the d key).

Note: Due to the specifics of the `libgit2` authentication procedure, when authentication fails, this function is called again without any indication whether authentication was successful or not. To avoid an infinite loop from repeatedly using the same faulty credentials, we will keep track of state using the payload.

For addition details see the LibGit2 guide on [authenticating against a server](#).

`source`

`Base.LibGit2.credentials_cb` – Function.

C function pointer for `credentials_callback`

`source`

`Base.LibGit2.default_signature` – Function.

Return signature object. Free it after use.

`source`

`Base.LibGit2.delete_branch` – Function.

`| LibGit2.delete_branch(branch::GitReference)`

Delete the branch pointed to by `branch`.

`source`

`Base.LibGit2.diff_files` – Function.

`| diff_files(repo::GitRepo, branch1::AbstractString, branch2::AbstractString; kwarg...) -> Vector{AbstractString}`

1776 Show which files have changed in the implementation repository's internodes  
branch1 and branch2.

The keyword argument is:

`filter::Set{Consts.DELTA_STATUS}=Set([Consts.DELTA_ADDED, Consts.DELTA_MODIFIED, Consts.DELTA_DELETED])`, and it sets options for the diff. The default is to show files added, modified, or deleted.

Returns only the names of the files which have changed, not their contents.

Examples

```
LibGit2.branch!(repo, "branch/a")
LibGit2.branch!(repo, "branch/b")
# add a file to repo
open(joinpath(LibGit2.path(repo), "file"), "w") do f
    write(f, "hello repo
")
end
LibGit2.add!(repo, "file")
LibGit2.commit(repo, "add file")
# returns ["file"]
filt = Set([LibGit2.Consts.DELTA_ADDED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b",
    ↳ filter=filt)
# returns [] because existing files weren't modified
filt = Set([LibGit2.Consts.DELTA_MODIFIED])
files = LibGit2.diff_files(repo, "branch/a", "branch/b",
    ↳ filter=filt)
```

Equivalent to `git diff --name-only --diff-filter=<filter> <branch1> <branch2>`.

[Base.LibGit2.entryid](#) – Function.

```
| entryid(te::GitTreeEntry)
```

Return the [GitHash](#) of the object to which `te` refers.

**source**

[Base.LibGit2.entrytype](#) – Function.

```
| entrytype(te::GitTreeEntry)
```

Return the type of the object to which `te` refers. The result will be one of the types which [objtype](#) returns, e.g. a [GitTree](#) or [GitBlob](#).

**source**

[Base.LibGit2.fetch](#) – Function.

```
| fetch(rmt::GitRemote, refspecs; options::FetchOptions=
|   FetchOptions(), msg="")
```

Fetch from the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to fetch. The keyword arguments are:

`options`: determines the options for the fetch, e.g. whether to prune afterwards. See [FetchOptions](#) for more information.

`msg`: a message to insert into the reflogs.

**source**

```
| fetch(repo::GitRepo; kwargs...)
```

Fetches updates from an upstream of the repository `repo`.

The keyword arguments are:

1778 **remote**::AbstractString DOCUMENTATION

The name, of **repo** to fetch from. If this is empty, the URL will be used to construct an anonymous remote.

**remoteurl**::AbstractString="" : the URL of **remote**. If not specified, will be assumed based on the given name of **remote**.

**refsspecs**=AbstractString[] : determines properties of the fetch.

**payload**=CredentialPayload() : provides credentials and/or settings when authenticating against a private **remote**.

Equivalent to `git fetch [<remoteurl>|<repo>] [<refsspecs>]`.

#### source

[Base.LibGit2.fetchheads](#) – Function.

```
fetchheads(repo::GitRepo) -> Vector{FetchHead}
```

Return the list of all the fetch heads for **repo**, each represented as a [FetchHead](#), including their names, URLs, and merge statuses.

#### Examples

```
julia> fetch_heads = LibGit2.fetchheads(repo);
```

```
julia> fetch_heads[1].name
```

```
"refs/heads/master"
```

```
julia> fetch_heads[1].ismerge
```

```
true
```

```
julia> fetch_heads[2].name
```

```
"refs/heads/test_branch"
```

```
julia> fetch_heads[2].ismerge
```

```
false
```

[Base.LibGit2.fetch\\_refspecs](#) – Function.

```
| fetch_refspecs(rmt::GitRemote) -> Vector{String}
```

Get the fetch refsspecs for the specified `rmt`. These refsspecs contain information about which branch(es) to fetch from.

Examples

```
| julia> remote = LibGit2.get(LibGit2.GitRemote, repo,  
|   ↵ "upstream");
```

```
| julia> LibGit2.add_fetch!(repo, remote, "upstream");
```

```
| julia> LibGit2.fetch_refspecs(remote)  
String["+refs/heads/*:refs/remotes/upstream/*"]
```

[source](#)

[Base.LibGit2.fetchhead\\_FOREACH\\_CB](#) – Function.

C function pointer for `fetchhead_FOREACH_CALLBACK`

[source](#)

[Base.LibGit2.merge\\_base](#) – Function.

```
| merge_base(repo::GitRepo, one::AbstractString, two::  
| AbstractString) -> GitHash
```

Find a merge base (a common ancestor) between the commits `one` and `two`. `one` and `two` may both be in string form. Return the `GitHash` of the merge base.

[source](#)

[Base.merge!](#) – Method.

Perform a git merge on the repository `repo`, merging commits with diverging history into the current branch. Returns `true` if the merge succeeded, `false` if not.

The keyword arguments are:

`committish::AbstractString=""`: Merge the named commit(s) in `committish`.

`branch::AbstractString=""`: Merge the branch `branch` and all its commits since it diverged from the current branch.

`fastforward::Bool=false`: If `fastforward` is `true`, only merge if the merge is a fast-forward (the current branch head is an ancestor of the commits to be merged), otherwise refuse to merge and return `false`. This is equivalent to the git CLI option `--ff-only`.

`merge_opts::MergeOptions=MergeOptions()`: `merge_opts` specifies options for the merge, such as merge strategy in case of conflicts.

`checkout_opts::CheckoutOptions=CheckoutOptions()`: `checkout_opts` specifies options for the checkout step.

Equivalent to `git merge [--ff-only] [<committish> | <branch>]`.

### Note

If you specify a `branch`, this must be done in reference format, since the string will be turned into a `GitReference`. For example, if you wanted to merge branch `branch_a`, you would call `merge!(repo, branch="refs/heads/branch_a")`.

### source

[Base.merge!](#) – Method.

```
78.18 BASE LIBGIT2
| merge!(repo::GitRepo, anns::Vector{GitAnnotated}; kwargs...)781
|     Bool
```

Merge changes from the annotated commits (captured as [GitAnnotated](#) objects) `anns` into the HEAD of the repository `repo`. The keyword arguments are:

`merge_opts::MergeOptions = MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See [MergeOptions](#) for more information.

`checkout_opts::CheckoutOptions = CheckoutOptions()`: options for how to perform the checkout. See [CheckoutOptions](#) for more information.

`anns` may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in
LibGit2.merge!(repo, [upst_ann])
```

[source](#)

[Base.merge!](#) – Method.

```
merge!(repo::GitRepo, anns::Vector{GitAnnotated}, fastforward::
|     Bool; kwargs...) -> Bool
```

Merge changes from the annotated commits (captured as [GitAnnotated](#) objects) `anns` into the HEAD of the repository `repo`. If `fastforward` is

1782 true, only a fastforward merge will succeed; if fastforward is false, the merge will fail. Otherwise, if fastforward is false, the merge may produce a conflict file which the user will need to resolve.

The keyword arguments are:

`merge_opts::MergeOptions` = `MergeOptions()`: options for how to perform the merge, including whether fastforwarding is allowed. See [MergeOptions](#) for more information.

`checkout_opts::CheckoutOptions` = `ChecoutOptions()`: options for how to perform the checkout. See [CheckoutOptions](#) for more information.

`anns` may refer to remote or local branch heads. Return `true` if the merge is successful, otherwise return `false` (for instance, if no merge is possible because the branches have no common ancestor).

Examples

```
upst_ann_1 = LibGit2.GitAnnotated(repo, "branch/a")

# merge the branch in, fastforward
LibGit2.merge!(repo, [upst_ann_1], true)

# merge conflicts!
upst_ann_2 = LibGit2.GitAnnotated(repo, "branch/b")
# merge the branch in, try to fastforward
LibGit2.merge!(repo, [upst_ann_2], true) # will return false
LibGit2.merge!(repo, [upst_ann_2], false) # will return true

source
```

[Base.LibGit2.ffmerge!](#) – Function.

```
| ffmerge!(repo::GitRepo, ann::GitAnnotated)
```

78.1 **Base.LibGit2::BaseLibGit2** Merge changes into current HEAD. This is only possible if the commit referred to by `ann` is descended from the current HEAD (e.g. if pulling changes from a remote branch which is simply ahead of the local branch tip).

**source**

[Base.LibGit2.fullname](#) – Function.

```
| LibGit2.fullname(ref::GitReference)
```

Return the name of the reference pointed to by the symbolic reference `ref`. If `ref` is not a symbolic reference, returns an empty string.

**source**

[Base.LibGit2.features](#) – Function.

```
| features()
```

Return a list of git features the current version of libgit2 supports, such as threading or using HTTPS or SSH.

**source**

[Base.LibGit2.filename](#) – Function.

```
| filename(te::GitTreeEntry)
```

Return the filename of the object on disk to which `te` refers.

**source**

[Base.LibGit2.filemode](#) – Function.

```
| filemode(te::GitTreeEntry) -> Cint
```

Return the UNIX filemode of the object on disk to which `te` refers as an integer.

**source**

```
| LibGit2.gitdir(repo::GitRepo)
```

Return the location of the "git" files of `repo`:

for normal repositories, this is the location of the `.git` folder.

for bare repositories, this is the location of the repository itself.

See also [workdir](#), [path](#).

`source`

[Base.LibGit2.git\\_url](#) – Function.

```
| LibGit2.git_url(; kwargs...) -> String
```

Create a string based upon the URL components provided. When the `scheme` keyword is not provided the URL produced will use the alternative [scp-like syntax](#).

Keywords

`scheme::AbstractString=""`: the URL scheme which identifies the protocol to be used. For HTTP use "http", SSH use "ssh", etc. When `scheme` is not provided the output format will be "ssh" but using the [scp-like syntax](#).

`username::AbstractString=""`: the username to use in the output if provided.

`password::AbstractString=""`: the password to use in the output if provided.

`host::AbstractString=""`: the hostname to use in the output. A hostname is required to be specified.

78.18. `path::AbstractString`: the port number<sup>1785</sup> to use in the output if provided. Cannot be specified when using the scp-like syntax.

`path::AbstractString`: the path to use in the output if provided.

## Examples

```
julia> LibGit2.git_url(username="git", host="github.com",
   ↪   path="JuliaLang/julia.git")
"git@github.com:JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="https", host="github.com",
   ↪   path="/JuliaLang/julia.git")
"https://github.com/JuliaLang/julia.git"

julia> LibGit2.git_url(scheme="ssh", username="git",
   ↪   host="github.com", port=2222, path="JuliaLang/julia.git")
"ssh://git@github.com:2222/JuliaLang/julia.git"
```

## source

`Base.LibGit2.@githash_str` – Macro.

```
@githash_str -> AbstractGitHash
```

Construct a git hash object from the given string, returning a `GitShortHash` if the string is shorter than 40 hexadecimal digits, otherwise a `GitHash`.

## Examples

```
julia> LibGit2.githash"d114feb74ce633"
GitShortHash("d114feb74ce633")
```

1786 **julia>**

## CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

```
↳ LibGit2.githash" d114feb74ce63307afe878a5228ad014e0289a85"  
GitHash("d114feb74ce63307afe878a5228ad014e0289a85")
```

**source**

[Base.LibGit2.head](#) – Function.

```
| LibGit2.head(repo::GitRepo) -> GitReference
```

Returns a `GitReference` to the current HEAD of `repo`.

**source**

```
| head(pkg::AbstractString) -> String
```

Return current HEAD `GitHash` of the `pkg` repo as a string.

**source**

[Base.LibGit2.head!](#) – Function.

```
| LibGit2.head!(repo::GitRepo, ref::GitReference) -> GitReference
```

Set the HEAD of `repo` to the object pointed to by `ref`.

**source**

[Base.LibGit2.head\\_oid](#) – Function.

```
| LibGit2.head_oid(repo::GitRepo) -> GitHash
```

Lookup the object id of the current HEAD of git repository `repo`.

**source**

[Base.LibGit2.headname](#) – Function.

```
| LibGit2.headname(repo::GitRepo)
```

Lookup the name of the current HEAD of git repository `repo`. If `repo` is currently detached, returns the name of the HEAD it's detached from.

[Base.LibGit2.init](#) – Function.

```
| LibGit2.init(path::AbstractString, bare::Bool=false) -> GitRepo
```

Open a new git repository at `path`. If `bare` is `false`, the working tree will be created in `path/.git`. If `bare` is `true`, no working directory will be created.

[source](#)

[Base.LibGit2.is\\_ancestor\\_of](#) – Function.

```
| is_ancestor_of(a::AbstractString, b::AbstractString, repo::  
|   GitRepo) -> Bool
```

Returns `true` if `a`, a [GitHash](#) in string form, is an ancestor of `b`, a [GitHash](#) in string form.

Examples

```
julia> repo = LibGit2.GitRepo(repo_path);  
  
julia> LibGit2.add!(repo, test_file1);  
  
julia> commit_oid1 = LibGit2.commit(repo, "commit1");  
  
julia> LibGit2.add!(repo, test_file2);  
  
julia> commit_oid2 = LibGit2.commit(repo, "commit2");  
  
julia> LibGit2.is_ancestor_of(string(commit_oid1),  
|   ↵ string(commit_oid2), repo)  
true
```

[source](#)

```
| isbinary(blob::GitBlob) -> Bool
```

Use a heuristic to guess if a file is binary: searching for NULL bytes and looking for a reasonable ratio of printable to non-printable characters among the first 8000 bytes.

`source`

[Base.LibGit2.iscommit](#) – Function.

```
| iscommit(id::AbstractString, repo::GitRepo) -> Bool
```

Checks if commit `id` (which is a [GitHash](#) in string form) is in the repository.

Examples

```
julia> repo = LibGit2.GitRepo(repo_path);

julia> LibGit2.add!(repo, test_file);

julia> commit_oid = LibGit2.commit(repo, "add test_file");

julia> LibGit2.iscommit(string(commit_oid), repo)
true
```

`source`

[Base.LibGit2.isdiff](#) – Function.

```
| LibGit2.isdiff(repo::GitRepo, treeish::AbstractString,
    pathspecs::AbstractString="" ; cached::Bool=false)
```

Checks if there are any differences between the tree specified by `treeish` and the tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdiff(repo, "HEAD") # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdiff(repo, "HEAD") # now true
```

Equivalent to `git diff-index <treeish> [-- <pathspecs>]`.

**source**

[Base.LibGit2.isdirty](#) – Function.

```
LibGit2.isdirty(repo::GitRepo, pathspecs::AbstractString="";
cached::Bool=false) -> Bool
```

Checks if there have been any changes to tracked files in the working tree (if `cached=false`) or the index (if `cached=true`). `pathspecs` are the specifications for options for the diff.

**Examples**

```
repo = LibGit2.GitRepo(repo_path)
LibGit2.isdirty(repo) # should be false
open(joinpath(repo_path, new_file), "a") do f
    println(f, "here's my cool new file")
end
LibGit2.isdirty(repo) # now true
LibGit2.isdirty(repo, new_file) # now true
```

Equivalent to `git diff-index HEAD [-- <pathspecs>]`.

**source**

[Base.LibGit2.isorphan](#) – Function.

## 1790 `CHAPTER 78 DOCUMENTATION OF JULIA'S INTERNALS` `LibGit2.isorphan(repo::GitRepo)`

Checks if the current branch is an "orphan" branch, i.e. has no commits.  
The first commit to this branch will have no parents.

`source`

`Base.LibGit2.isset` – Function.

```
| isset(val::Integer, flag::Integer)
```

Test whether the bits of `val` indexed by `flag` are set (1) or unset (0).

`source`

`Base.LibGit2.iszero` – Function.

```
| iszero(id::GitHash) -> Bool
```

Determine whether all hexadecimal digits of the given `GitHash` are zero.

`source`

`Base.LibGit2.lookup_branch` – Function.

```
| lookup_branch(repo::GitRepo, branch_name::AbstractString,  
| remote::Bool=false) -> Nullable{GitReference}
```

Determine if the branch specified by `branch_name` exists in the repository `repo`. If `remote` is `true`, `repo` is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

`lookup_branch` returns a `Nullable`, which will be null if the requested branch does not exist yet. If the branch does exist, the `Nullable` contains a `GitReference` to the branch.

`source`

`Base.map` – Method.

```
78.18 BASE LIBGIT2 LibGit2::map(f::Function, walker::GitRevWalker; oid::GitHash=1791
|   GitHash(), range::AbstractString="", by::Cint=Consts.
|   SORT_NONE, rev::Bool=false)
```

Using the `GitRevWalker` walker to "walk" over every commit in the repository's history, apply `f` to each commit in the walk. The keyword arguments are:

- \* `oid`: The `GitHash` of the commit to begin the walk from. The default is to use `push_head!` and therefore the HEAD commit and all its ancestors.
- \* `range`: A range of `GitHashes` in the format `oid1..oid2`. `f` will be applied to all commits between the two.
- \* `by`: The sorting method. The default is not to sort. Other options are to sort by topology (`LibGit2.Consts.SORT_TOPLOGICAL`), to sort forwards in time (`LibGit2.Consts.SORT_TIME`, most ancient first) or to sort backwards in time (`LibGit2.Consts.SORT_REVERSE`, most recent first).
- \* `rev`: Whether to reverse the sorted order (for instance, if topological sorting is used).

Examples

```
oids = LibGit2.with(LibGit2.GitRevWalker(repo)) do walker
    LibGit2.map((oid, repo)->string(oid), walker,
    ↳   by=LibGit2.Consts.SORT_TIME)
end
```

Here, `map` visits each commit using the `GitRevWalker` and finds its `GitHash`.

`source`

`Base.LibGit2.mirror_callback` – Function.

Mirror callback function

Function sets `+refs/*:refs/*` refsspecs and `mirror` flag for remote reference.

`source`

C function pointer for `mirror_callback`

`source`

[Base.LibGit2.message](#) – Function.

```
| message(c::GitCommit, raw::Bool=false)
```

Return the commit message describing the changes made in commit `c`. If `raw` is `false`, return a slightly “cleaned up” message (which has any leading newlines removed). If `raw` is `true`, the message is not stripped of any such newlines.

`source`

[Base.LibGit2.merge\\_analysis](#) – Function.

```
| merge_analysis(repo::GitRepo, anns::Vector{GitAnnotated}) ->
|   analysis, preference
```

Run analysis on the branches pointed to by the annotated branch tips `anns` and determine under what circumstances they can be merged. For instance, if `anns[1]` is simply an ancestor of `anns[2]`, then `merge_analysis` will report that a fast-forward merge is possible.

`merge_analysis` returns two outputs. `analysis` has several possible values:

- \* `MERGE_ANALYSIS_NONE`: it is not possible to merge the elements of `anns`.
- \* `MERGE_ANALYSIS_NORMAL`: a regular merge, when HEAD and the commits that the user wishes to merge have all diverged from a common ancestor. In this case the changes have to be resolved and conflicts may occur.
- \* `MERGE_ANALYSIS_UP_TO_DATE`: all the input commits the user wishes to merge can be reached from HEAD, so no merge needs to be performed.
- \* `MERGE_ANALYSIS_FASTFORWARD`: the input commit is a descendant of HEAD and so no merge needs to

78.1.8 Be performed. Instead, the user can simply checkout the input commit(s). \* **MERGE\_ANALYSIS\_UNBORN**: the HEAD of the repository refers to a commit which does not exist. It is not possible to merge, but it may be possible to checkout the input commits. **preference** also has several possible values:

- \* **MERGE\_PREFERENCE\_NONE**: the user has no preference.
- \* **MERGE\_PREFERENCE\_NO\_FASTFORWARD**: do not allow any fast-forward merges.
- \* **MERGE\_PREFERENCE\_FASTFORWARD\_ONLY**: allow only fast-forward merges and no other type (which may introduce conflicts).

**preference** can be controlled through the repository or global git configuration.

**source**

[Base.LibGit2.name](#) – Function.

```
| LibGit2.name(ref::GitReference)
```

Return the full name of **ref**.

**source**

```
| name(rmt::GitRemote)
```

Get the name of a remote repository, for instance "**origin**". If the remote is anonymous (see [GitRemoteAnon](#)) the name will be an empty string "".

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";
```

```
julia> repo = LibGit2.clone(cache_repo, "test_directory");
```

```
julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);
```

```
julia> name(remote)
```

```
"origin"
```

1794 **source**

CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

| `LibGit2.name(tag::GitTag)`

The name of `tag` (e.g. "v0.5").

**source**

`Base.LibGit2.need_update` – Function.

| `need_update(repo::GitRepo)`

Equivalent to `git update-index`. Returns `true` if `repo` needs updating.

**source**

`Base.LibGit2.objtype` – Function.

| `objtype(obj_type::Consts.OBJECT)`

Returns the type corresponding to the enum value.

**source**

`Base.LibGit2.path` – Function.

| `LibGit2.path(repo::GitRepo)`

Return the base file path of the repository `repo`.

for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see `workdir` for more details).

for bare repositories, this is the location of the "git" files.

See also `gitdir`, `workdir`.

**source**

`Base.LibGit2.peel` – Function.

78.1.8 `Base.LibGit2.peel([T,] Ref::GitReference)`

1795

Recursively peel `ref` until an object of type `T` is obtained. If no `T` is provided, then `ref` will be peeled until an object other than a [GitTag](#) is obtained.

A [GitTag](#) will be peeled to the object it references.

A [GitCommit](#) will be peeled to a [GitTree](#).

Note

Only annotated tags can be peeled to [GitTag](#) objects. Lightweight tags (the default) are references under `refs/tags/` which point directly to [GitCommit](#) objects.

`source`

| `peel([T,] obj::GitObject)`

Recursively peel `obj` until an object of type `T` is obtained. If no `T` is provided, then `obj` will be peeled until the type changes.

A [GitTag](#) will be peeled to the object it references.

A [GitCommit](#) will be peeled to a [GitTree](#).

`source`

[Base.LibGit2 posixpath](#) – Function.

| `LibGit2.posixpath(path)`

Standardise the path string `path` to use POSIX separators.

`source`

[Base.LibGit2 push](#) – Function.

1796 push(rmt::GitRemote, refspecs, force::Bool=false, options::  
CHAPTER 78 DOCUMENTATION OF JULIA'S INTERNALS

```
| PushOptions=PushOptions())
```

Push to the specified `rmt` remote git repository, using `refspecs` to determine which remote branch(es) to push to. The keyword arguments are:

- `force`: if `true`, a force-push will occur, disregarding conflicts.
- `options`: determines the options for the push, e.g. which proxy headers to use. See [PushOptions](#) for more information.

#### Note

You can add information about the push `refspecs` in two other ways: by setting an option in the repository's `GitConfig` (with `push.default` as the key) or by calling [add\\_push!](#). Otherwise you will need to explicitly specify a push `refspec` in the call to `push` for it to have any effect, like so: `LibGit2.push(repo, refspecs=["refs/heads/master"])`.

#### source

```
| push(repo::GitRepo; kwargs...)
```

Pushes updates to an upstream of `repo`.

The keyword arguments are:

`remote::AbstractString="origin"`: the name of the upstream remote to push to.

`remoteurl::AbstractString=""`: the URL of `remote`.

`refspecs=AbstractString[]`: determines properties of the push.

`force::Bool=false`: determines if the push will be a force push, overwriting the remote branch.

78.18. `Base.LibGit2.CredentialPayload()`: provides credentials and/or settings when authenticating against a private `remote`.<sup>1797</sup>

Equivalent to `git push [<remoteurl>|<repo>] [<refspecs>]`.

`source`

`Base.push!` – Method.

```
| LibGit2.push!(w::GitRevWalker, cid::GitHash)
```

Start the `GitRevWalker` walker at commit `cid`. This function can be used to apply a function to all commits since a certain year, by passing the first commit of that year as `cid` and then passing the resulting `w` to `map`.

`source`

`Base.LibGit2.push_head!` – Function.

```
| LibGit2.push_head!(w::GitRevWalker)
```

Push the HEAD commit and its ancestors onto the `GitRevWalker` `w`. This ensures that HEAD and all its ancestor commits will be encountered during the walk.

`source`

`Base.LibGit2.push.refsspecs` – Function.

```
| push_refsspecs(rmt::GitRemote) -> Vector{String}
```

Get the push refsspecs for the specified `rmt`. These refsspecs contain information about which branch(es) to push to.

Examples

```
julia> remote = LibGit2.get(LibGit2.GitRemote, repo,
    "upstream");
```

1798 **julia>** LibGit2.add\_push!(repo, Remote("refs/heads/master"),

**julia>** close(remote);

**julia>** remote = LibGit2.get(LibGit2.GitRemote, repo,  
→ "upstream");

**julia>** LibGit2.push\_refspecs(remote)  
String["refs/heads/master"]

**source**

[Base.LibGit2.raw](#) – Function.

| raw(id::GitHash) -> Vector{UInt8}

Obtain the raw bytes of the [GitHash](#) as a vector of length 20.

**source**

[Base.LibGit2.read\\_tree!](#) – Function.

| LibGit2.read\_tree!(idx::GitIndex, tree::GitTree)

| LibGit2.read\_tree!(idx::GitIndex, treehash::AbstractGitHash)

Read the tree `tree` (or the tree pointed to by `treehash` in the repository owned by `idx`) into the index `idx`. The current index contents will be replaced.

**source**

[Base.LibGit2.rebase!](#) – Function.

| LibGit2.rebase!(repo::GitRepo, upstream::AbstractString="",

| newbase::AbstractString="")

78.1. **attempt\_rebase**<sup>1780</sup>  
Automatic merge rebase of the current branch, from `upstream` if provided, or otherwise from the upstream tracking branch. `newbase` is the branch to rebase onto. By default this is `upstream`.

If any conflicts arise which cannot be automatically resolved, the rebase will abort, leaving the repository and working tree in its original state, and the function will throw a `GitError`. This is roughly equivalent to the following command line statement:

```
git rebase --merge [<upstream>]
if [ -d ".git/rebase-merge" ]; then
    git rebase --abort
fi
```

`source`

[Base.LibGit2.ref\\_list](#) – Function.

```
| LibGit2.ref_list(repo::GitRepo) -> Vector{String}
```

Get a list of all reference names in the `repo` repository.

`source`

[Base.LibGit2.reftype](#) – Function.

```
| LibGit2.reftype(ref::GitReference) -> Cint
```

Returns a `Cint` corresponding to the type of `ref`:

- 0 if the reference is invalid
- 1 if the reference is an object id
- 2 if the reference is symbolic

`source`

[Base.LibGit2.remotes](#) – Function.

## 1800 LibGit2.remotes(repo::GitRepo)

CHAPTER 78 DOCUMENTATION OF JULIA'S INTERNALS

Return a vector of the names of the remotes of `repo`.

[source](#)

`Base.LibGit2.remove!` – Function.

```
| remove!(repo::GitRepo, files::AbstractString...)
| remove!(idx::GitIndex, files::AbstractString...)
```

Remove all the files with paths specified by `files` in the index `idx` (or the index of the `repo`).

[source](#)

`Base.LibGit2.reset` – Function.

```
| reset(val::Integer, flag::Integer)
```

Unset the bits of `val` indexed by `flag`, returning them to 0.

[source](#)

`Base.LibGit2.reset!` – Function.

```
| reset!(payload, [config]) -> CredentialPayload
```

Reset the `payload` state back to the initial values so that it can be used again within the credential callback. If a `config` is provided the configuration will also be updated.

[source](#)

Updates some entries, determined by the `pathspecs`, in the index from the target commit tree.

[source](#)

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

```
git reset [<committish>] [- ] <paths...>
```

### source

```
| reset!(repo::GitRepo, id::GitHash, mode::Cint=Consts.  
|   RESET_MIXED)
```

Reset the repository `repo` to its state at `id`, using one of three modes set by `mode`:

1. `Consts.RESET_SOFT` - move HEAD to `id`.
2. `Consts.RESET_MIXED` - default, move HEAD to `id` and reset the index to `id`.
3. `Consts.RESET_HARD` - move HEAD to `id`, reset the index to `id`, and discard all working changes.

### Examples

```
# fetch changes  
LibGit2.fetch(repo)  
isfile(joinpath(repo_path, our_file)) # will be false  
  
# fastforward merge the changes  
LibGit2.merge!(repo, fastforward=true)  
  
# because there was not any file locally, but there is  
# a file remotely, we need to reset the branch  
head_oid = LibGit2.head_oid(repo)  
new_head = LibGit2.reset!(repo, head_oid,  
  ↳ LibGit2.Consts.RESET_HARD)
```

In this example, the remote which is being fetched from does have a file called `our_file` in its index, which is why we must reset.

## Examples

```

repo = LibGit2.GitRepo(repo_path)
head_oid = LibGit2.head_oid(repo)
open(joinpath(repo_path, "file1"), "w") do f
    write(f, "111")
end
LibGit2.add!(repo, "file1")
mode = LibGit2.Consts.RESET_HARD
# will discard the changes to file1
# and unstage it
new_head = LibGit2.reset!(repo, head_oid, mode)

```

### source

[Base.LibGit2.restore](#) – Function.

```
| restore(s::State, repo::GitRepo)
```

Return a repository `repo` to a previous `State s`, for example the HEAD of a branch before a merge attempt. `s` can be generated using the [snapshot](#) function.

### source

[Base.LibGit2.revcount](#) – Function.

```
| LibGit2.revcount(repo::GitRepo, commit1::AbstractString,
                  commit2::AbstractString)
```

List the number of revisions between `commit1` and `commit2` (committish OIDs in string form). Since `commit1` and `commit2` may be on different branches, `revcount` performs a "left-right" revision list (and count), returning a tuple of `Ints` – the number of left and right commits, respectively.

78.1 A `LibGit2` commit refers to which side of a symmetric difference tree the commit is reachable from.

Equivalent to `git rev-list --left-right --count <commit1> <commit2>`.

Examples

```
repo = LibGit2.GitRepo(repo_path)
repo_file = open(joinpath(repo_path, test_file), "a")
println(repo_file, "hello world")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid1 = LibGit2.commit(repo, "commit 1")
println(repo_file, "hello world again")
flush(repo_file)
LibGit2.add!(repo, test_file)
commit_oid2 = LibGit2.commit(repo, "commit 2")
LibGit2.revcount(repo, string(commit_oid1),
    → string(commit_oid2))
```

This will return `(-1, 0)`.

`source`

`Base.LibGit2.set_remote_url` – Function.

```
set_remote_url(repo::GitRepo, remote_name, url)
set_remote_url(repo::String, remote_name, url)
```

Set both the fetch and push `url` for `remote_name` for the `GitRepo` or the git repository located at `path`. Typically git repos use "origin" as the remote name.

Examples

```
repo_path = joinpath(tempdir(), "Example")
repo = LibGit2.init(repo_path)
```

1804 CHAPTER 78. DOCUMENTATION OF JULIA'S INTERNALS

```
| LibGit2.set_remote_url(repo, "upstream",
|   ↳ "https://github.com/JuliaLang/Example.jl")
| LibGit2.set_remote_url(repo_path, "upstream2",
|   ↳ "https://github.com/JuliaLang/Example2.jl")
```

source

Base.LibGit2.shortname – Function.

```
| LibGit2.shortname(ref::GitReference)
```

Returns a shortened version of the name of `ref` that's "human-readable".

```
| julia> repo = LibGit2.GitRepo(path_to_repo);
```

```
| julia> branch_ref = LibGit2.head(repo);
```

```
| julia> LibGit2.name(branch_ref)
```

```
"refs/heads/master"
```

```
| julia> LibGit2.shortname(branch_ref)
```

```
"master"
```

source

Base.LibGit2.snapshot – Function.

```
| snapshot(repo::GitRepo) -> State
```

Take a snapshot of the current state of the repository `repo`, storing the current HEAD, index, and any uncommitted work. The output `State` can be used later during a call to `restore` to return the repository to the snapshotted state.

source

Base.LibGit2.status – Function.

Lookup the status of the file at `path` in the git repository `repo`. For instance, this can be used to check if the file at `path` has been modified and needs to be staged and committed.

`source`

`Base.LibGit2.stage` – Function.

`| stage(ie::IndexEntry) -> Cint`

Get the stage number of `ie`. The stage number `0` represents the current state of the working tree, but other numbers can be used in the case of a merge conflict. In such a case, the various stage numbers on an `IndexEntry` describe which side(s) of the conflict the current state of the file belongs to. Stage `0` is the state before the attempted merge, stage `1` is the changes which have been made locally, stages `2` and larger are for changes from other branches (for instance, in the case of a multi-branch "octopus" merge, stages `2`, `3`, and `4` might be used).

`source`

`Base.LibGit2.tag_create` – Function.

`| LibGit2.tag_create(repo::GitRepo, tag::AbstractString, commit; kwargs...)`

Create a new git tag `tag` (e.g. "`v0.5`") in the repository `repo`, at the commit `commit`.

The keyword arguments are:

`msg::AbstractString=""`: the message for the tag.

`force::Bool=false`: if `true`, existing references will be overwritten.

`sig::Signature=Signature(repo)`: the tagger's signature.

[Base.LibGit2.tag\\_delete](#) – Function.

```
| LibGit2.tag_delete(repo::GitRepo, tag::AbstractString)
```

Remove the git tag `tag` from the repository `repo`.

[source](#)

[Base.LibGit2.tag\\_list](#) – Function.

```
| LibGit2.tag_list(repo::GitRepo) -> Vector{String}
```

Get a list of all tags in the git repository `repo`.

[source](#)

[Base.LibGit2.target](#) – Function.

```
| LibGit2.target(tag::GitTag)
```

The `GitHash` of the target object of `tag`.

[source](#)

[Base.LibGit2.toggle](#) – Function.

```
| toggle(val::Integer, flag::Integer)
```

Flip the bits of `val` indexed by `flag`, so that if a bit is 0 it will be 1 after the toggle, and vice-versa.

[source](#)

[Base.LibGit2.transact](#) – Function.

```
| transact(f::Function, repo::GitRepo)
```

Apply function `f` to the git repository `repo`, taking a `snapshot` before applying `f`. If an error occurs within `f`, `repo` will be returned to its snapshot

78.18 **STATE** Using `Restore`. The error which occurred will be rethrown, but state of `repo` will not be corrupted.

`source`

[Base.LibGit2.treewalk](#) – Function.

```
| treewalk(f::Function, tree::GitTree, payload=Any[], post::Bool=
|   false)
```

Traverse the entries in `tree` and its subtrees in post or pre order. Preorder means beginning at the root and then traversing the leftmost subtree (and recursively on down through that subtree's leftmost subtrees) and moving right through the subtrees. Postorder means beginning at the bottom of the leftmost subtree, traversing upwards through it, then traversing the next right subtree (again beginning at the bottom) and finally visiting the tree root last of all.

The function parameter `f` should have following signature:

```
| (Cstring, Ptr{Void}, Ptr{Void}) -> Cint
```

A negative value returned from `f` stops the tree walk. A positive value means that the entry will be skipped if `post` is `false`.

`source`

[Base.LibGit2.upstream](#) – Function.

```
| upstream(ref::GitReference) -> Nullable{GitReference}
```

Determine if the branch containing `ref` has a specified upstream branch.

`upstream` returns a `Nullable`, which will be null if the requested branch does not have an upstream counterpart. If the upstream branch does exist, the `Nullable` contains a `GitReference` to the upstream branch.

`source`

```
| update!(repo::GitRepo, files::AbstractString...)
| update!(idx::GitIndex, files::AbstractString...)
```

Update all the files with paths specified by `files` in the index `idx` (or the index of the `repo`). Match the state of each file in the index with the current state on disk, removing it if it has been removed on disk, or updating its entry in the object database.

`source`

[Base.LibGit2.url](#) – Function.

```
| url(rmt::GitRemote)
```

Get the fetch URL of a remote git repository.

Examples

```
julia> repo_url = "https://github.com/JuliaLang/Example.jl";  
  
julia> repo = LibGit2.init(mktempdir());  
  
julia> remote = LibGit2.GitRemote(repo, "origin", repo_url);  
  
julia> LibGit2.url(remote)  
"https://github.com/JuliaLang/Example.jl"
```

`source`

[Base.LibGit2.version](#) – Function.

```
| version() -> VersionNumber
```

Return the version of libgit2 in use, as a [VersionNumber](#).

`source`

```
|with(f::Function, obj)
```

Resource management helper function. Applies **f** to **obj**, making sure to call **close** on **obj** after **f** successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed.

**source**

```
|with_warn(f::Function, ::Type{T}, args...)
```

Resource management helper function. Apply **f** to **args**, first constructing an instance of type **T** from **args**. Makes sure to call **close** on the resulting object after **f** successfully returns or throws an error. Ensures that allocated git resources are finalized as soon as they are no longer needed. If an error is thrown by **f**, a warning is shown containing the error.

**source**

```
|LibGit2.workdir(repo::GitRepo)
```

Return the location of the working directory of **repo**. This will throw an error for bare repositories.

#### Note

This will typically be the parent directory of **gitdir(repo)**, but can be different in some cases: e.g. if either the **core.work-tree** configuration variable or the **GIT\_WORK\_TREE** environment variable is set.

181 See also [gitdir](#), [path](#).

[source](#)

[Base.LibGit2.GitObject](#) – Method.

```
| (>::Type{T})(te)::GitTreeEntry) where T<:GitObject
```

Get the git object to which `te` refers and return it as its actual type (the type `entrytype` would show), for instance a `GitBlob` or `GitTag`.

[Examples](#)

```
| tree = LibGit2.GitTree(repo, "HEAD^{tree}")
| tree_entry = tree[1]
| blob = LibGit2.GitBlob(tree_entry)
```

[source](#)

[Base.LibGit2.AbstractCredentials](#) – Type.

Abstract credentials payload

[source](#)

[Base.LibGit2.UserPasswordCredentials](#) – Type.

Credentials that support only `user` and `password` parameters

[source](#)

[Base.LibGit2.SSHCredentials](#) – Type.

SSH credentials type

[source](#)

[Base.LibGit2.isfilled](#) – Function.

```
| isfilled(cred)::AbstractCredentials) -> Bool
```

[source](#)

[Base.LibGit2.CachedCredentials](#) – Type.

Credentials that support caching

[source](#)

[Base.LibGit2.CredentialPayload](#) – Type.

[| LibGit2.CredentialPayload](#)

Retains the state between multiple calls to the credential callback for the same URL. A `CredentialPayload` instance is expected to be `reset!` whenever it will be used with a different URL.

[source](#)

[Base.LibGit2.approve](#) – Function.

[| approve\(payload::CredentialPayload\) -> Void](#)

Store the `payload` credential for re-use in a future authentication. Should only be called when authentication was successful.

[source](#)

[Base.LibGit2.reject](#) – Function.

[| reject\(payload::CredentialPayload\) -> Void](#)

Discard the `payload` credential from begin re-used in future authentication. Should only be called when authentication was unsuccessful.

[source](#)

## 78.19 Module loading

`Base.require` is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the `import` statement.

The features below are experimental and not part of the stable Julia API. Before building upon them inform yourself about the current thinking and whether they might change soon.

## Module loading callbacks

It is possible to listen to the modules loaded by `Base.require`, by registering a callback.

```
loaded_packages = Channel{Symbol}()  
callback = (mod::Symbol) -> put!(loaded_packages, mod)  
push!(Base.package_callbacks, callback)
```

Please note that the symbol given to the callback is a non-unique identifier and it is the responsibility of the callback provider to walk the module chain to determine the fully qualified name of the loaded binding.

The callback below is an example of how to do that:

```
# Get the fully-qualified name of a module.  
function module_fqn(name::Symbol)  
    fqn = fullname(Base.root_module(name))  
    return join(fqn, '.')  
end
```

## 78.20 Inference

### How inference works

Type inference refers to the process of deducing the types of later values from the types of input values. Julia's approach to inference has been described in blog posts (1, 2).

You can start a Julia session, edit `inference.jl` (for example to insert `print` statements), and then replace `Core.Inference` in your running session by navigating to `base/` and executing `include("coreimg.jl")`. This trick typically leads to much faster development than if you rebuild Julia for each change.

A convenient entry point into inference is `typeinf_code`. Here's a demo running inference on `convert(Int, UInt(1))`:

```
# Get the method
atypes = Tuple{Type{Int}, UInt} # argument types
mths = methods(convert, atypes) # worth checking that there is
    ↳ only one
m = first(mths)

# Create variables needed to call `typeinf_code`
params = Core.Inference.InferenceParams(typemax(UInt)) #
    ↳ parameter is the world age,
                                                # type-
                                                ↳ max(UInt)
                                                ↳ ->
                                                ↳ most
                                                ↳ recent
sparms = Core.svec()          # this particular method doesn't have
    ↳ type-parameters
optimize = true                # run all inference optimizations
cached = false                 # force inference to happen (do not use
    ↳ cached results)
Core.Inference.typeinf_code(m, atypes, sparms, optimize, cached,
    ↳ params)
```

For debugging and documentation purposes, Julia's internals by calling `Core.Inference.code_for_method` using many of the variables above. A `CodeInfo` object may be obtained with

```
# Returns the CodeInfo object for `convert(Int, ::UInt)`:  
ci = (@code_typed convert(Int, UInt(1)))[1]
```

The inlining algorithm (`inline_worthy`)

Much of the hardest work for inlining runs in `inlining_pass`. However, if your question is "why didn't my function inline?" then you will most likely be interested in `isinlineable` and its primary callee, `inline_worthy`. `isinlineable` handles a number of special cases (e.g., critical functions like `next` and `done`, incorporating a bonus for functions that return tuples, etc.). The main decision-making happens in `inline_worthy`, which returns `true` if the function should be inlined.

`inline_worthy` implements a cost-model, where "cheap" functions get inlined; more specifically, we inline functions if their anticipated run-time is not large compared to the time it would take to `issue a call` to them if they were not inlined. The cost-model is extremely simple and ignores many important details: for example, all `for` loops are analyzed as if they will be executed once, and the cost of an `if...else...end` includes the summed cost of all branches. It's also worth acknowledging that we currently lack a suite of functions suitable for testing how well the cost model predicts the actual run-time cost, although `BaseBenchmarks` provides a great deal of indirect information about the successes and failures of any modification to the inlining algorithm.

The foundation of the cost-model is a lookup table, implemented in `add_tfunc` and its callers, that assigns an estimated cost (measured in CPU cycles) to each of Julia's intrinsic functions. These costs are based on `standard ranges for common architectures` (see [Agner Fog's analysis](#) for more detail).

~~W20\_INFERENCE~~ This low-level lookup table with a number of special cases<sup>1815</sup>

For example, an `:invoke` expression (a call for which all input and output types were inferred in advance) is assigned a fixed cost (currently 20 cycles). In contrast, a `:call` expression, for functions other than intrinsics/builtins, indicates that the call will require dynamic dispatch, in which case we assign a cost set by `InferenceParams.inline_nonleaf_penalty` (currently set at 1000). Note that this is not a "first-principles" estimate of the raw cost of dynamic dispatch, but a mere heuristic indicating that dynamic dispatch is extremely expensive.

Each statement gets analyzed for its total cost in a function called `statement_cost`. You can run this yourself by following this example:

```
params = Core.Inference.InferenceParams(typemax(UInt))  
# Get the CodeInfo object  
ci = (@code_typed fill(3, (5, 5)))[1] # we'll try this on the  
# code for `fill(3, (5, 5))`  
# Calculate cost of each statement  
cost(stmt) = Core.Inference.statement_cost(stmt, ci, Base, params)  
cst = map(cost, ci.code)
```

The output is a `Vector{Int}` holding the estimated cost of each statement in `ci.code`. Note that `ci` includes the consequences of inlining callees, and consequently the costs do too.

Stack frame	Source code	Notes
jl_uv_write()	jl_uv.c	called though <code>ccall</code>
ju- lia_write_282942	stream.jl	function <code>write!(s::IO, a::Array{T})</code> where <code>T</code>
ju- lia_print_284639	ascii.jl	<code>print(io::IO, s::String) = (write(io, s); nothing)</code>
jl- call_print_284639		
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_ap- ply_generic()	gf.c	<code>Base.print(Base.TTY, String)</code>
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_ap- ply_generic()	gf.c	<code>Base.print(Base.TTY, String, Char, Char...)</code>
jl_apply()	julia.h	
jl_f_apply()	builtins.c	
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_ap- ply_generic()	gf.c	<code>Base.println(Base.TTY, String, String...)</code>
jl_apply()	julia.h	
jl_trampoline()	builtins.c	
jl_apply()	julia.h	
jl_ap- ply_generic()	gf.c	<code>Base.println(String, )</code>
jl_apply()	julia.h	
do_call()	inter- preter.c	
eval()	inter- preter.c	
jl_inter- pret_toplevel_expr	inter- preter.c	
jl_toplevel_eval_f	toplevel.c	
jl_toplevel_eval()	toplevel.c	
jl_toplevel_eval_i	builtins.c	
jl_f_top_eval()	builtins.c	

Input	AST
f(x)	(call f x)
f(x, y=1, z=2)	(call f x (kw y 1) (kw z 2))
f(x; y=1)	(call f (parameters (kw y 1)) x)
f(x...)	(call f (... x))

Input	AST
x+y	(call + x y)
a+b+c+d	(call + a b c d)
2x	(call * 2 x)
a&&b	(&& a b)
x += 1	(+= x 1)
a ? 1 : 2	(if a 1 2)
a:b	(: a b)
a:b:c	(: a b c)
a,b	(tuple a b)
a==b	(call == a b)
1 < i <= n	(comparison 1 < i <= n)
a.b	(. a (quote b))
a.(b)	(. a b)

Input	AST
a[i]	(ref a i)
t[i;j]	(typed_vcat t i j)
t[i j]	(typed_hcat t i j)
t[a b; c d]	(typed_vcat t (row a b) (row c d))
a{b}	(curly a b)
a{b;c}	(curly a (parameters c) b)
[x]	(vect x)
[x,y]	(vect x y)
[x;y]	(vcat x y)
[x y]	(hcat x y)
[x y; z t]	(vcat (row x y) (row z t))
[x for y in z, a in b]	(comprehension x (= y z) (= a b))
T[x for y in z]	(typed_comprehension T x (= y z))
(a, b, c)	(tuple a b c)
(a; b; c)	(block a (block b c))

Input	AST
@m x y	(macrocall @m (line) x y)
Base.@m x y	(macrocall (. Base (quote @m)) (line) x y)
@Base.m x y	(macrocall (. Base (quote @m)) (line) x y)

Input	AST
"a"	"a"
x"y"	(macrocall @x_str (line) "y")
x"y"z	(macrocall @x_str (line) "y" "z")
"x = \$x"	(string "x = " x)
`a b c`	(macrocall @cmd (line) "a b c")

Input	AST
import a	(import a)
import a.b.c	(import a b c)
import ...a	(import . . . a)
import a.b, c.d	(toplevel (import a b) (import c d))
import Base: x	(import Base x)
import Base: x, y	(toplevel (import Base x) (import Base y))
export a, b	(export a b)

Input	AST
1111111111111111	1(macrocall @int128_str (null) "1111111111111111")
0xffffffffffffffffffff	ff(macrocall @uint128_str (null) "0xffffffffffffffffffff")
1111...many digits...	(macrocall @big_str (null) "1111....")

Name	Prefix	Purpose
Native	julia_	Speed via specialized signatures
JL Call	jlcall_	Wrapper for generic calls
JL Call	jl_	Builtins
C ABI	jlapi_	Wrapper callable from C

File	Description
builtins.c	Builtin functions
ccall.cpp	Lowering <code>ccall</code>
cgutils.cpp	Lowering utilities, notably for array and tuple accesses
codegen.cpp	Top-level of code generation, pass list, lowering builtins
debuginfo.cpp	Tracks debug information for JIT code
disasm.cpp	Handles native object file and JIT code disassembly
gf.c	Generic functions
intrinsics.cpp	Lowering intrinsics
llvm-simdloop.cpp	Custom LLVM pass for <code>@simd</code>
sys.c	I/O and operating system utility functions

# Chapter 79

## Developing/debugging Julia's C code

### 79.1 Reporting and analyzing crashes (segfaults)

So you managed to break Julia. Congratulations! Collected here are some general procedures you can undergo for common symptoms encountered when something goes awry. Including the information from these debugging steps can greatly help the maintainers when tracking down a segfault or trying to figure out why your script is running slower than expected.

If you've been directed to this page, find the symptom that best matches what you're experiencing and follow the instructions to generate the debugging information requested. Table of symptoms:

[Segfaults during bootstrap \(sysimg.jl\)](#)

[Segfaults when running a script](#)

[Errors during Julia startup](#)

No matter the error, we will always need to know what version of Julia you are running. When Julia first starts up, a header is printed out with a version number and date. Please also include the output of `versioninfo()` in any report you create:

```
julia> versioninfo()
Julia Version 0.7.0-DEV.2588
Commit fb2ba438c6* (2017-11-23 08:37 UTC)
Platform Info:
  OS: macOS (x86_64-apple-darwin16.7.0)
  CPU: Intel(R) Core(TM) i5-5287U CPU @ 2.90GHz
  WORD_SIZE: 64
  BLAS: libopenblas (USE64BITINT DYNAMIC_ARCH NO_AFFINITY Haswell)
  LAPACK: libopenblas64_
  LIBM: libopenlibm
  LLVM: libLLVM-5.0.0-git-6e4f2fa226 (ORCJIT, broadwell)

Environment:
  JULIA_NUM_THREADS = 2
  JULIA_EDITOR = vim
  JULIA = /Users/wookyoung/head/julia/usr/bin/julia
  JULIA_EXE = /Applications/Julia-
  ↳ 0.6.0.app/Contents/Resources/julia/bin/julia
  JULIA_PKGDIR =
  ↳ /Users/wookyoung/ipap/juliakorea/translate-docdeps
```

## Segfaults during bootstrap (`sysimg.jl`)

Segfaults toward the end of the `make` process of building Julia are a common symptom of something going wrong while Julia is preparing the corpus of code in the `base/` folder. Many factors can contribute toward this process dying unexpectedly, however it is as often as not due to an error in the C-

REPORTING AND ANALYZING CRASHES (SEGFAULTS)

Code reporting and analysis on crashes (segfaults) with a debug build inside of **gdb**. Explicitly:

Create a debug build of Julia:

```
$ cd <julia_root>
$ make debug
```

Note that this process will likely fail with the same error as a normal `make` incantation, however this will create a debug executable that will offer `gdb` the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of `gdb`:

```
$ cd base/
$ gdb -x ../contrib/debug_bootstrap.gdb
```

This will start `gdb`, attempt to run the bootstrap process using the debug build of Julia, and print out a backtrace if (when) it segfaults. You may need to hit `<enter>` a few times to get the full backtrace. Create a [gist](#) with the backtrace, the [version info](#), and any other pertinent information you can think of and open a new [issue](#) on Github with a link to the gist.

## Segfaults when running a script

The procedure is very similar to [Segfaults during bootstrap \(sysimg.jl\)](#). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
$ cd <julia_root>
$ make debug
$ gdb --args usr/bin/julia-debug <path_to_your_script>
```

Note that `gdb` will sit there, waiting for instructions. Type `r` to run the process, and `bt` to generate a backtrace once it segfaults:

```
(gdb) r
```

```
| 1822 Starting program: /home/sabae/src/Julia/usr/bin/julia-debug ./test
| .jl
| ...
| (gdb) bt
```

Create a [gist](#) with the backtrace, the [version info](#), and any other pertinent information you can think of and open a new [issue](#) on Github with a link to the gist.

## Errors during Julia startup

Occasionally errors occur during Julia's startup process (especially when using binary distributions, as opposed to compiling from source) such as the following:

```
$ julia
exec: error -5
```

These errors typically indicate something is not getting loaded properly very early on in the bootup phase, and our best bet in determining what's going wrong is to use external tools to audit the disk activity of the `julia` process:

On Linux, use `strace`:

```
$ strace julia
```

On OSX, use `dtruss`:

```
$ dtruss -f julia
```

Create a [gist](#) with the `strace`/ `dtruss` ouput, the [version info](#), and any other pertinent information and open a new [issue](#) on Github with a link to the gist.

A few terms have been used as shorthand in this guide:

<julia\_root> refers to the root directory of the Julia source tree; e.g. it should contain folders such as `base`, `deps`, `src`, `test`, etc.....

## 79.2 gdb debugging tips

### Displaying Julia variables

Within `gdb`, any `jl_value_t*` object `obj` can be displayed using

```
| (gdb) call jl_(obj)
```

The object will be displayed in the `julia` session, not in the `gdb` session. This is a useful way to discover the types and values of objects being manipulated by Julia's C code.

Similarly, if you're debugging some of Julia's internals (e.g., `inference.jl`), you can print `obj` using

```
| ccall(:jl_, Void, (Any,), obj)
```

This is a good way to circumvent problems that arise from the order in which julia's output streams are initialized.

Julia's flisp interpreter uses `value_t` objects; these can be displayed with `call fl_print(fl_ctx, ios_stdout, obj)`.

### Useful Julia variables for Inspecting

While the addresses of many variables, like singletons, can be useful to print for many failures, there are a number of additional variables (see `julia.h` for a complete list) that are even more useful.

1824when in jl\_applyCHAPTER 7: Debugging Julia's Core Code  
>def, mfunc->code) :: for figuring out a bit about the call-stack

jl\_lineno and jl\_filename :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)

\$1 :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as `print`)

jl\_options :: sometimes useful, since it lists all of the command line options that were successfully parsed

jl\_uv\_stderr :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

`jl_gdblookup($rip)` :: For looking up the current function and line. (use `$eip` on i686 platforms)

`jlbacktrace()` :: For dumping the current Julia backtrace stack to stderr. Only usable after `record_backtrace()` has been called.

`jl_dump_llvm_value(Value*)` :: For invoking `Value->dump()` in gdb, where it doesn't work natively. For example, `f->linfo->functionObject`, `f->linfo->specFunctionObject`, and `to_function(f->linfo)`.

`Type->dump()` :: only works in lldb. Note: add something like ;1 to prevent lldb from printing its prompt over the output

`jl_eval_string("expr")` :: for invoking side-effects to modify the current state or to lookup symbols

`jl_typeof(jl_value_t*)` :: for extracting the type tag of a Julia value (in gdb, call `macro define jl_typeof jl_typeof first`, or pick something short like `ty` for the first arg to define a shorthand)

In your `gdb` session, set a breakpoint in `jl_breakpoint` like so:

```
| (gdb) break jl_breakpoint
```

Then within your Julia code, insert a call to `jl_breakpoint` by adding

```
| ccall(:jl_breakpoint, Void, (Any,), obj)
```

where `obj` can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the `jl_apply` frame, from which you can display the arguments to a function using, e.g.,

```
| (gdb) call jl_(args[0])
```

Another useful frame is `to_function(jl_method_instance_t *li, bool cstyle)`. The `jl_method_instance_t*` argument is a struct with a reference to the final AST sent into the compiler. However, the AST at this point will usually be compressed; to view the AST, call `jl_uncompress_ast` and then pass the result to `jl_`:

```
#2 0x00007ffff7928bf7 in to_function (li=0x2812060, cstyle=false)
    at codegen.cpp:584
584         abort();
(gdb) p jl_(jl_uncompress_ast(li, li->ast))
```

## Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is `sysimg.jl`:

```
| (gdb) break jl_load if strcmp(fname, "sysimg.jl")==0
```

```
(gdb) break jl_apply_generic if strcmp((char*)(jl_symbol_name)(  
    jl_gf_mtable(F)->name), "method_to_break")==0
```

Since this function is used for every call, you will make everything 1000x slower if you do this.

## Dealing with signals

Julia requires a few signal to function property. The profiler uses **SIGUSR2** for sampling and the garbage collector uses **SIGSEGV** for threads synchronization. If you are debugging some code that uses the profiler or multiple threads, you may want to let the debugger ignore these signals since they can be triggered very often during normal operations. The command to do this in GDB is (replace **SIGSEGV** with **SIGUSRS** or other signals you want to ignore):

```
(gdb) handle SIGSEGV noprint nostop pass
```

The corresponding LLDB command is (after the process is started):

```
(lldb) pro hand -p true -s false -n false SIGSEGV
```

If you are debugging a segfault with threaded code, you can set a breakpoint on **jl\_critical\_error** (**sigdie\_handler** should also work on Linux and BSD) in order to only catch the actual segfault rather than the GC synchronization points.

## Debugging during Julia's build process (bootstrap)

Errors that occur during **make** need special handling. Julia is built in two stages, constructing **sys0** and **sys.ji**. To see what commands are running at the time of failure, use **make VERBOSE=1**.

At the time of this writing, you can debug build errors during the **sys0** phase from the **base** directory using:

```
| 79.2 GDB DEBUGGING TIPS | 1827
| Julia/Base$ gdb --args ./usr/bin/julia-debug -C native --build
|   ..../usr/lib/julia/sys0 sysimg.jl
```

You might need to delete all the files in `usr/lib/julia/` to get this to work.

You can debug the `sys.ji` phase using:

```
| julia/base$ gdb --args ..../usr/bin/julia-debug -C native --build
|   ..../usr/lib/julia/sys -J ..../usr/lib/julia/sys0.ji sysimg.jl
```

By default, any errors will cause Julia to exit, even under gdb. To catch an error "in the act", set a breakpoint in `jl_error` (there are several other useful spots, for specific kinds of failures, including: `jl_too_few_args`, `jl_too_many_args`, and `jl_throw`).

Once an error is caught, a useful technique is to walk up the stack and examine the function by inspecting the related call to `jl_apply`. To take a real-world example:

```
Breakpoint 1, jl_throw (e=0x7ffd42de400) at task.c:802
802 {
(gdb) p jl_(e)
ErrorException("auto_unbox: unable to determine argument type")
$2 = void
(gdb) bt 10
#0  jl_throw (e=0x7ffd42de400) at task.c:802
#1  0x00007ffff65412fe in jl_error (str=0x7ffde56be000 <_j_str267>
    "auto_unbox:
    unable to determine argument type")
    at builtins.c:39
#2  0x00007ffde56bd01a in julia_convert_16886 ()
#3  0x00007ffff6541154 in jl_apply (f=0x7ffd4367f630, args=0
    x7fffffff2b0, nargs=2) at julia.h:1281
...
...
```

The most recent `jl_FUNCTIONs` frame from the debugger has code at the AST for the function `julia_convert_16886`. This is the unique name for some method of `convert`. `f` in this frame is a `jl_function_t*`, so we can look at the type signature, if any, from the `specTypes` field:

```
(gdb) f 3
#3 0x00007ffff6541154 in jl_apply (f=0x7ffdf367f630, args=0
    x7fffffff2b0, nargs=2) at julia.h:1281
1281          return f->fptr((jl_value_t*)f, args, nargs);
(gdb) p f->linfo->specTypes
$4 = (jl_tupletype_t *) 0x7ffdf39b1030
(gdb) p jl_( f->linfo->specTypes )
Tuple{Type{Float32}, Float64}           # <-- type signature for
julia_convert_16886
```

Then, we can look at the AST for this function:

```
(gdb) p jl_( jl_uncompress_ast(f->linfo, f->linfo->ast) )
Expr(:lambda, Array{Any, 1}[:#s29, :x], Array{Any, 1}[Array{Any,
    1}[], Array{Any, 1}[Array{Any, 1}[:#s29, :Any, 0], Array{Any,
    1][:x, :Any, 0]], Array{Any, 1}[], 0], Expr(:body,
Expr(:line, 90, :float.jl)::Any,
Expr(:return, Expr(:call, :box, :Float32, Expr(:call, :fptrunc, :
    Float32, :x)::Any)::Any)::Any)::Any
```

Finally, and perhaps most usefully, we can force the function to be recompiled in order to step through the codegen process. To do this, clear the cached `functionObject` from the `jl_lamdba_info_t*`:

```
(gdb) p f->linfo->functionObject
$8 = (void *) 0x1289d070
(gdb) set f->linfo->functionObject = NULL
```

Then, set a breakpoint somewhere useful (e.g. `emit_function`, `emit_expr`, `emit_call`, etc.), and run codegen:

| 79.3 USING VALGRIND WITH JULIA  
| (gdb) p JI\_compile(f)  
| ... # your breakpoint here

1829

## Debugging precompilation errors

Module precompilation spawns a separate Julia process to precompile each module. Setting a breakpoint or catching failures in a precompile worker requires attaching a debugger to the worker. The easiest approach is to set the debugger watch for new process launches matching a given name. For example:

| (gdb) attach -w -n julia-debug

or:

| (lldb) process attach -w -n julia-debug

Then run a script/command to start precompilation. As described earlier, use conditional breakpoints in the parent process to catch specific file-loading events and narrow the debugging window. (some operating systems may require alternative approaches, such as following each `fork` from the parent process)

## Mozilla's Record and Replay Framework (rr)

Julia now works out of the box with `rr`, the lightweight recording and deterministic debugging framework from Mozilla. This allows you to replay the trace of an execution deterministically. The replayed execution's address spaces, register contents, syscall data etc are exactly the same in every run.

A recent version of `rr` (3.1.0 or higher) is required.

## 79.3 Using Valgrind with Julia

[Valgrind](#) is a tool for memory debugging, memory leak detection, and profiling.

This section describes Chapter 70, **DEVELOPING WITH DEBUGGING IN JULIA'S CODE**, memory issues with Julia.

## General considerations

By default, Valgrind assumes that there is no self modifying code in the programs it runs. This assumption works fine in most instances but fails miserably for a just-in-time compiler like `julia`. For this reason it is crucial to pass `--smc-check=all-non-file` to `valgrind`, else code may crash or behave unexpectedly (often in subtle ways).

In some cases, to better detect memory errors using Valgrind it can help to compile `julia` with memory pools disabled. The compile-time flag `MEMDEBUG` disables memory pools in Julia, and `MEMDEBUG2` disables memory pools in FemtoLisp. To build `julia` with both flags, add the following line to `Make.user`:

```
|CFLAGS = -DMEMDEBUG -DMEMDEBUG2
```

Another thing to note: if your program uses multiple workers processes, it is likely that you want all such worker processes to run under Valgrind, not just the parent process. To do this, pass `--trace-children=yes` to `valgrind`.

## Suppressions

Valgrind will typically display spurious warnings as it runs. To reduce the number of such warnings, it helps to provide a [suppressions file](#) to Valgrind. A sample suppressions file is included in the Julia source distribution at `contrib/valgrind-julia.supp`.

The suppressions file can be used from the `julia/` source directory as follows:

```
$ valgrind --smc-check=all-non-file --suppressions=contrib/
    valgrind-julia.supp ./julia progname.jl
```

~~Any memory errors~~ displayed should either be reported as bug #~~1831~~ contributed as additional suppressions. Note that some versions of Valgrind are shipped with insufficient default suppressions, so that may be one thing to consider before submitting any bugs.

## Running the Julia test suite under Valgrind

It is possible to run the entire Julia test suite under Valgrind, but it does take quite some time (typically several hours). To do so, run the following command from the `julia/test/` directory:

```
valgrind --smc-check=all-non-file --trace-children=yes --  
    suppressions=$PWD/./contrib/valgrind-julia.supp ./julia  
    runtests.jl all
```

If you would like to see a report of "definite" memory leaks, pass the flags `--leak-check=full --show-leak-kinds=definite` to `valgrind` as well.

## Caveats

Valgrind currently does not support multiple rounding modes, so code that adjusts the rounding mode will behave differently when run under Valgrind.

In general, if after setting `--smc-check=all-non-file` you find that your program behaves differently when run under Valgrind, it may help to pass `--tool=none` to `valgrind` as you investigate further. This will enable the minimal Valgrind machinery but will also run much faster than when the full memory checker is enabled.

## 79.4 Sanitizer support

### General considerations

Using Clang's sanitizers obviously require you to use Clang (`USECLANG=1`), but there's another catch: most sanitizers require a run-time library, provided by

the host compiler, **CHAPTER 79 DEVELOPING/DEBUGGING JULIA'S JIT CODE** on functionality from that library. This implies that the LLVM version of your host compiler matches that of the LLVM library used within Julia.

An easy solution is to have an dedicated build folder for providing a matching toolchain, by building with `BUILD_LLVM_CLANG=1`. You can then refer to this toolchain from another build folder by specifying `USECLANG=1` while overriding the `CC` and `CXX` variables.

## Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's [address sanitizer \(ASAN\)](#). By compiling with `SANITIZE=1` you enable ASAN for the Julia compiler and its generated code. In addition, you can specify `LLVM_SANITIZE=1` to sanitize the LLVM library as well. Note that these options incur a high performance and memory cost. For example, using ASAN for Julia and LLVM makes `testall11` takes 8–10 times as long while using 20 times as much memory (this can be reduced to respectively a factor of 3 and 4 by using the options described below).

By default, Julia sets the `allow_user_segv_handler=1` ASAN flag, which is required for signal delivery to work properly. You can define other options using the `ASAN_OPTIONS` environment flag, in which case you'll need to repeat the default option mentioned before. For example, memory usage can be reduced by specifying `fast_unwind_on_malloc=0` and `malloc_context_size=2`, at the cost of backtrace accuracy. For now, Julia also sets `detect_leaks=0`, but this should be removed in the future.

## Memory Sanitizer (MSAN)

For detecting use of uninitialized memory, you can use Clang's [memory sanitizer \(MSAN\)](#) by compiling with `SANITIZE_MEMORY=1`.