

The Julia Language

The Julia Project

December 29, 2017

Contents

Contents	i
I Home	1
II Julia Documentation	3
1 Manual	5
2 Standard Library	7
3 Developer Documentation	9
III Manual	11
4 Introducción	13
5 Empezando	15
5.1 Resources	17
6 Variables	19
6.1 Nombres de Variables Permitidos	20
6.2 Convenciones de Estilo	21
7 Números enteros y en punto flotante	23
7.1 Enteros	24
Comportamiento ante el Desbordamiento	26
Errores de división	26

7.2	Números en Punto Flotante	26
	Cero en punto flotante	28
	Valores especiales en punto flotante	28
	Epsilon de máquina	29
	Modos de Redondeo	30
	Antecedentes y referencias	31
7.3	Aritmética de Precisión Arbitraria	31
7.4	Coeficientes Literales Numéricos	33
	Conflictos de Sintaxis	34
7.5	Literales cero and uno	34
8	Mathematical Operations and Elementary Functions	35
8.1	Operadores Aritméticos	35
8.2	Operadores bit a bit	36
8.3	Operaciones de actualización	36
8.4	Operadores vectorizados con "punto"	37
8.5	Comparaciones Numéricas	38
	Comparaciones Encadenadas	40
	Funciones Elementales	40
8.6	Precedencia de Operadores	41
8.7	Conversiones Numéricas	41
	Funciones de Redondeo	42
	Funciones de División	42
	Funciones de signo y valor absoluto	42
	Potencias, logaritmos y raíces	42
	Funciones Trigonométricas e Hiperbólicas	43
	Funciones Especiales	44
9	Números Racionales y Complejos	45
9.1	Números Complejos	45
9.2	Números Racionales	48
10	Strings	51
10.1	Caracteres	52
10.2	Fundamentos de Cadenas	53
10.3	Unicode y UTF-8	54
10.4	Concatenación	56
10.5	Interpolación	56
10.6	Literales cadena con triples comillas	57
10.7	Operaciones Comunes	58
10.8	Literales cadena no estándar	60
10.9	Expresiones Regulares	60
10.10	Byte Array Literals	63
10.11	Literales Número de Versión	65
10.12	Raw String Literals	65
11	Functions	67
11.1	Argument Passing Behavior	68
11.2	The Keyword	68
11.3	Operators Are Functions	69
11.4	Operators With Special Names	70
11.5	Anonymous Functions	70

11.6 Multiple Return Values	71
11.7 Varargs Functions	71
11.8 Optional Arguments	73
11.9 Keyword Arguments	74
11.10 Evaluation Scope of Default Values	75
11.11 Do-Block Syntax for Function Arguments	75
11.12 Dot Syntax for Vectorizing Functions	76
11.13 Further Reading	77
12 Control Flow	79
12.1 Compound Expressions	79
12.2 Conditional Evaluation	80
12.3 Short-Circuit Evaluation	83
12.4 Repeated Evaluation: Loops	86
12.5 Exception Handling	89
Built-in s	89
The function	90
Errors	91
Warnings and informational messages	91
The statement	92
Clauses	93
12.6 Tasks (aka Coroutines)	94
Core task operations	95
Tasks and events	96
Task states	96
13 Scope of Variables	97
13.1 Global Scope	98
13.2 Local Scope	98
Soft Local Scope	100
Hard Local Scope	100
Hard vs. Soft Local Scope	103
Let Blocks	104
For Loops and Comprehensions	105
13.3 Constants	106
14 Types	107
14.1 Type Declarations	108
14.2 Abstract Types	109
14.3 Primitive Types	110
14.4 Composite Types	111
14.5 Mutable Composite Types	113
14.6 Declared Types	114
14.7 Type Unions	114
14.8 Parametric Types	115
Parametric Composite Types	115
Parametric Abstract Types	117
Tuple Types	119
Vararg Tuple Types	120
Parametric Primitive Types	121
14.9 UnionAll Types	122
14.10 Type Aliases	123

14.11 Operations on Types	123
14.12 Custom pretty-printing	124
14.13 "Value types"	125
14.14 Nullable Types: Representing Missing Values	126
Constructing objects	127
Checking if a object has a value	127
Safely accessing the value of a object	127
Performing operations on objects	128
15 Methods	131
15.1 Defining Methods	131
15.2 Method Ambiguities	134
15.3 Parametric Methods	135
15.4 Redefining Methods	137
15.5 Parametrically-constrained Varargs methods	139
15.6 Note on Optional and keyword Arguments	139
15.7 Function-like objects	140
15.8 Empty generic functions	140
15.9 Method design and the avoidance of ambiguities	140
Tuple and NTuple arguments	141
Orthogonalize your design	141
Dispatch on one argument at a time	142
Abstract containers and element types	142
Complex method "cascades" with default arguments	143
16 Constructors	145
16.1 Outer Constructor Methods	145
16.2 Inner Constructor Methods	146
16.3 Incomplete Initialization	148
16.4 Parametric Constructors	149
16.5 Case Study: Rational	151
16.6 Constructors and Conversion	153
16.7 Outer-only constructors	153
17 Conversion and Promotion	155
17.1 Conversion	156
Defining New Conversions	156
Case Study: Rational Conversions	157
17.2 Promotion	158
Defining Promotion Rules	159
Case Study: Rational Promotions	160
18 Interfaces	161
18.1 Iteration	161
18.2 Indexing	163
18.3 Abstract Arrays	164
19 Modules	167
19.1 Summary of module usage	168
Modules and files	168
Standard modules	169
Default top-level definitions and bare modules	169

Relative and absolute module paths	169
Module file paths	170
Namespace miscellanea	170
Module initialization and precompilation	170
20 Documentation	175
20.1 Accessing Documentation	177
20.2 Functions & Methods	178
20.3 Advanced Usage	178
Dynamic documentation	179
20.4 Syntax Guide	179
Functions and Methods	180
Macros	180
Types	181
Modules	181
Global Variables	182
Multiple Objects	182
Macro-generated code	183
20.5 Markdown syntax	183
Inline elements	183
Toplevel elements	185
20.6 Markdown Syntax Extensions	189
21 Metaprogramming	191
21.1 Program representation	191
Symbols	192
21.2 Expressions and evaluation	193
Quoting	193
Interpolation	194
and effects	194
Functions on sessions	196
21.3 Macros	196
Basics	196
Hold up: why macros?	197
Macro invocation	198
Building an advanced macro	198
Hygiene	200
21.4 Code Generation	202
21.5 Non-Standard String Literals	202
21.6 Generated functions	204
An advanced example	207
22 Arrays Multi-dimensionales	211
22.1 Arrays	211
Funciones Básicas	211
Construcción e Inicialización	212
Concatenación	212
Inicializadores de Array Tipados	213
Comprensiones	213
Expresiones Generador	214
Indexación	215
Asignación	215

Tipos de Índices Soportados	216
Iteración	218
Rasgos de Array	219
Arrays, Funciones y Operadores Vectorizados	219
Retransmisión	220
Implementation	221
22.2 Vectores y Matrices <i>Sparse</i>	222
Columna Comprimida <i>Sparse</i> (CSC) Para Almacenamiento de Matrices <i>Sparse</i>	222
Almacenamiento de Vectores <i>Sparse</i>	223
Constructores de Vectores y Matrices <i>Sparse</i>	223
Operaciones con matrices <i>sparse</i>	225
Correspondence of dense and sparse methods	225
23 Linear algebra	227
23.1 Special matrices	229
Elementary operations	229
Matrix factorizations	229
The uniform scaling operator	230
23.2 Matrix factorizations	230
24 Networking and Streams	233
24.1 Basic Stream I/O	233
24.2 Text I/O	234
24.3 IO Output Contextual Properties	234
24.4 Working with Files	235
24.5 A simple TCP example	236
24.6 Resolving IP Addresses	237
25 Parallel Computing	239
25.1 Code Availability and Loading Packages	240
25.2 Data Movement	242
26 Global variables	245
26.1 Parallel Map and Loops	246
26.2 Synchronization With Remote References	248
26.3 Scheduling	248
26.4 Channels	249
26.5 Remote References and AbstractChannels	252
26.6 Channels and RemoteChannels	252
26.7 Remote References and Distributed Garbage Collection	254
26.8 Shared Arrays	255
26.9 Shared Arrays and Distributed Garbage Collection	258
26.10 ClusterManagers	258
26.11 Cluster Managers with Custom Transports	261
26.12 Network Requirements for LocalManager and SSHManager	262
26.13 Cluster Cookie	263
26.14 Specifying Network Topology (Experimental)	263
26.15 Multi-Threading (Experimental)	263
Setup	263
The Macro	264
26.16 @threadcall (Experimental)	265

27	Date and DateTime	267
27.1	Constructors	267
27.2	Durations/Comparisons	269
27.3	Accessor Functions	270
27.4	Query Functions	271
27.5	TimeType-Period Arithmetic	272
27.6	Adjuster Functions	273
27.7	Period Types	275
27.8	Rounding	275
	Rounding Epoch	276
28	Interacting With Julia	277
28.1	The different prompt modes	277
	The Julian mode	277
	Help mode	278
	Shell mode	278
	Search modes	279
28.2	Key bindings	279
	Customizing keybindings	279
28.3	Tab completion	279
28.4	Customizing Colors	282
29	Running External Programs	283
29.1	Interpolation	284
29.2	Quoting	285
29.3	Pipelines	286
	Avoiding Deadlock in Pipelines	287
	Complex Example	288
30	Llamando a código C y Fortran	289
30.1	Creating C-Compatible Julia Function Pointers	291
30.2	Mapping C Types to Julia	292
	Auto-conversion:	293
	Type Correspondences:	293
	Bits Types:	293
	Struct Type correspondences	296
	Type Parameters	297
	SIMD Values	297
	Memory Ownership	298
	When to use T, Ptr{T} and Ref{T}	298
30.3	Mapping C Functions to Julia	298
	/ argument translation guide	298
	/ return type translation guide	299
	Passing Pointers for Modifying Inputs	300
	Special Reference Syntax for ccall (deprecated):	300
30.4	Some Examples of C Wrappers	301
30.5	Garbage Collection Safety	302
30.6	Non-constant Function Specifications	302
30.7	Indirect Calls	303
30.8	Calling Convention	303
30.9	Accessing Global Variables	304
30.10	Accessing Data through a Pointer	304

30.11 Thread-safety	304
30.12 More About Callbacks	305
30.13 C++	305
31 Handling Operating System Variation	307
32 Environment Variables	309
32.1 File locations	309
.	309
.	310
.	310
.	310
.	310
32.2 External applications	311
.	311
.	311
32.3 Parallelization	311
.	311
.	311
.	311
.	311
.	311
32.4 REPL formatting	311
.	312
.	312
.	312
.	312
.	312
.	312
.	312
32.5 Debugging and profiling	312
, ,	312
.	313
.	313
.	313
.	313
.	313
33 Embedding Julia	315
33.1 High-Level Embedding	315
Using julia-config to automatically determine build parameters	316
33.2 Converting Types	317
33.3 Calling Julia Functions	317
33.4 Memory Management	318
Manipulating the Garbage Collector	319
33.5 Working with Arrays	319
Accessing Returned Arrays	320
Multidimensional Arrays	320
33.6 Exceptions	320
Throwing Julia Exceptions	320
34 Packages	323

34.1	Package Status	323
34.2	Adding and Removing Packages	324
34.3	Offline Installation of Packages	326
34.4	Installing Unregistered Packages	326
34.5	Updating Packages	327
34.6	Checkout, Pin and Free	327
34.7	Custom METADATA Repository	329
35	Package Development	331
35.1	Initial Setup	331
35.2	Making changes to an existing package	331
	Documentation changes	331
	Code changes	332
	Dirty packages	333
	Making a branch <i>post hoc</i>	333
	Squashing and rebasing	334
35.3	Creating a new Package	334
	REQUIRE speaks for itself	334
	Guidelines for naming a package	335
	Generating the package	336
	Loading Static Non-Julia Files	337
	Making Your Package Available	337
	Tagging and Publishing Your Package	337
35.4	Fixing Package Requirements	340
35.5	Requirements Specification	340
36	Profiling	343
36.1	Basic usage	343
36.2	Accumulation and clearing	346
36.3	Options for controlling the display of profile results	346
36.4	Configuration	347
37	Memory allocation analysis	349
38	Stack Traces	351
38.1	Viewing a stack trace	351
38.2	Extracting useful information	352
38.3	Error handling	352
38.4	Comparison with	354
39	Performance Tips	357
39.1	Avoid global variables	357
39.2	Measure performance with <code>@time</code> and pay attention to memory allocation	358
39.3	Tools	359
39.4	Avoid containers with abstract type parameters	359
39.5	Type declarations	359
	Avoid fields with abstract type	359
	Avoid fields with abstract containers	361
	Annotate values taken from untyped locations	364
	Declare types of keyword arguments	364
39.6	Break functions into multiple definitions	365
39.7	Write "type-stable" functions	365

39.8	Avoid changing the type of a variable	365
39.9	Separate kernel functions (aka, function barriers)	366
39.10	Types with values-as-parameters	367
39.11	The dangers of abusing multiple dispatch (aka, more on types with values-as-parameters)	368
39.12	Access arrays in memory order, along columns	369
39.13	Pre-allocating outputs	371
39.14	More dots: Fuse vectorized operations	372
39.15	Consider using views for slices	372
39.16	Avoid string interpolation for I/O	373
39.17	Optimize network I/O during parallel execution	373
39.18	Fix deprecation warnings	374
39.19	Tweaks	374
39.20	Performance Annotations	374
39.21	Treat Subnormal Numbers as Zeros	377
39.22		378
40	Workflow Tips	381
40.1	REPL-based workflow	381
	A basic editor/REPL workflow	381
	Simplify initialization	382
40.2	Browser-based workflow	382
41	Style Guide	383
41.1	Write functions, not just scripts	383
41.2	Avoid writing overly-specific types	383
41.3	Handle excess argument diversity in the caller	384
41.4	Append to names of functions that modify their arguments	384
41.5	Avoid strange type s	385
41.6	Avoid type Unions in fields	385
41.7	Avoid elaborate container types	385
41.8	Use naming conventions consistent with Julia's	385
41.9	Don't overuse try-catch	386
41.10	Don't parenthesize conditions	386
41.11	Don't overuse	386
41.12	Don't use unnecessary static parameters	386
41.13	Avoid confusion about whether something is an instance or a type	386
41.14	Don't overuse macros	387
41.15	Don't expose unsafe operations at the interface level	387
41.16	Don't overload methods of base container types	387
41.17	Avoid type piracy	387
41.18	Be careful with type equality	388
41.19	Do not write	388
41.20	Avoid using floats for numeric literals in generic code when possible	388
42	Frequently Asked Questions	391
42.1	Sessions and the REPL	391
	How do I delete an object in memory?	391
	How can I modify the declaration of a type in my session?	391
42.2	Functions	391
	I passed an argument to a function, modified it inside that function, but on the outside,	391
	Can I use or inside a function?	392
	What does the operator do?	393

The two uses of the <code>...</code> operator: slurping and splatting	393
combines many arguments into one argument in function definitions	393
splits one argument into many different arguments in function calls	394
42.3 Types, type declarations, and constructors	394
What does "type-stable" mean?	394
Why does Julia give a warning for certain seemingly-sensible operations?	395
Why does Julia use native machine integer arithmetic?	395
What are the possible causes of an error during remote execution?	399
42.4 Packages and Modules	400
What is the difference between "using" and "importall"?	400
42.5 Nothingness and missing values	401
How does "null" or "nothingness" work in Julia?	401
42.6 Memory	401
Why does Julia allocate memory when <code>Array{<T>}</code> and <code>Array{<T>{<S>}}</code> are arrays?	401
42.7 Asynchronous IO and concurrent synchronous writes	402
Why do concurrent writes to the same stream result in inter-mixed output?	402
42.8 Julia Releases	403
Do I want to use a release, beta, or nightly version of Julia?	403
When are deprecated functions removed?	403
43 Noteworthy Differences from other Languages	405
43.1 Noteworthy differences from MATLAB	405
43.2 Noteworthy differences from R	406
43.3 Noteworthy differences from Python	409
43.4 Noteworthy differences from C/C++	409
44 Unicode Input	413
 IV Standard Library	 415
45 Essentials	417
45.1 Introducción	417
45.2 Moviéndose	417
45.3 Todos los Objetos	423
45.4 Tipos	429
45.5 Funciones Genéricas	436
45.6 Sintaxis	438
45.7 <i>Nullables</i>	441
45.8 Sistema	443
45.9 Errores	451
45.10 Eventos	455
45.11 Reflexión	456
45.12 Interioridades	459
46 Colecciones y Estructuras de Datos	463
46.1 Iteración	463
46.2 Colecciones generales	465
46.3 Colecciones Iterables	467
46.4 Colecciones Indexables	488
46.5 Colecciones asociativas	489
46.6 Colecciones tipo Conjunto	496

46.7 Acciones relacionadas con colas	499
47 Mathematics	505
47.1 Operadores Matemáticos	505
47.2 Funciones Matemáticas	520
47.3 Estadística	544
47.4 Procesamiento de Señales	549
48 Números	557
48.1 Tipos Numéricos Estándar	557
Tipos Numéricos Abstractos	557
Tipos Numéricos Concretos	558
48.2 Formatos de Datos	560
48.3 Constantes y Funciones de Números Generales	566
Enteros	573
48.4 BigFloats	575
48.5 Números Aleatorios	576
49 Cadenas	581
50 Arrays	597
50.1 Constructores y Tipos	597
50.2 Funciones básicas	605
50.3 Retransmisión y Vectorización	610
50.4 Indexación y Asignación	613
50.5 Vistas (SubArrays y otros tipos de vistas)	617
50.6 Concatenación and permutación	621
50.7 Funciones de Arrays	633
50.8 Combinatoria	640
50.9 BitArrays	645
50.10 Matrices y Vectores <i>Sparse</i>	647
51 Tareas y Computación Paralela	657
51.1 Tareas	657
51.2 Soporte General a la Computación Paralela	664
51.3 Arrays Compartidos	674
51.4 Multi-Threading	676
51.5 ccall using a threadpool (Experimental)	681
51.6 Primitivas de Sincronización	681
51.7 Interfaz de Administración de Cluster	683
52 Álgebra Lineal	685
52.1 Funciones Estándar	685
52.2 Operaciones matriciales de bajo nivel	740
52.3 Funciones BLAS	743
BLAS Character Arguments	744
52.4 LAPACK Functions	751
53 Constantes	767
54 Sistema de Ficheros	771
55 E/S y Redes	781

55.1 E/S General	781
55.2 E/S Texto	791
55.3 E/S Multimedia	798
55.4 E/S Mapeada en Memoria	801
55.5 E/S por Red	802
56 Puntuación	809
57 Ordenación y Funciones Relacionadas	811
57.1 Funciones de Ordenación	813
57.2 Funciones relacionadas con Orden	816
57.3 Algoritmos de Ordenación	819
58 Funciones del Administrador de Paquetes	821
59 Fechas y Tiempo	825
59.1 Tipos para Fechas y Tiempo	825
59.2 Funciones para Fechas	826
Funciones Accesoras	831
Funciones de Consulta	834
Funciones de Ajuste	836
Períodos	838
Funciones de Redondeo	839
Funciones de Conversión	841
Constantes	841
60 Utilidades para Iteración	843
61 Haciendo Pruebas Unitarias	849
61.1 Probando Julia Base	849
61.2 Pruebas Unitarias Básicas	849
61.3 Trabajando con Conjuntos de Test	851
61.4 Otras Macros para Tests	852
61.5 Tests Rotos	854
61.6 Creando Tipos Personalizados	854
62 Interfaz C	857
63 Interfaz LLVM	865
64 Librería Estándar C	867
65 Enlazador Dinámico	869
66 Profiling	871
67 StackTraces	873
68 Soporte SIMD	875
V Developer Documentation	877
69 Reflection and introspection	879

69.1	Module bindings	879
69.2	DataType fields	879
69.3	Subtypes	879
69.4	DataType layout	880
69.5	Function methods	880
69.6	Expansion and lowering	880
69.7	Intermediate and compiled representations	880
70	Documentation of Julia's Internals	883
70.1	Initialization of the Julia runtime	883
	main()	883
	julia_init()	883
	true_main()	884
	Base.start	884
	Base.eval	885
	jl_atexit_hook()	885
	julia_save()	886
70.2	Julia ASTs	886
	Lowered form	886
	Surface syntax AST	891
70.3	Más sobre tipos	894
	Tipos y conjuntos (y y /)	894
	UnionAll types	895
	Free variables	896
	TypeNames	896
	Tuple types	897
	Diagonal types	898
	Subtyping diagonal variables	899
	Introduction to the internal machinery	900
	Subtyping and method sorting	900
70.4	Memory layout of Julia Objects	901
	Object layout (jl_value_t)	901
	Garbage collector mark bits	902
	Object allocation	902
70.5	Eval of Julia code	903
	Julia Execution	904
	Parsing	904
	Macro Expansion	904
	Type Inference	905
	JIT Code Generation	905
	System Image	906
70.6	Calling Conventions	906
	Julia Native Calling Convention	906
	JL Call Convention	907
	C ABI	907
70.7	High-level Overview of the Native-Code Generation Process	907
	Representation of Pointers	907
	Representation of Intermediate Values	907
	Union representation	908
	Specialized Calling Convention Signature Representation	908
70.8	Julia Functions	909
	Method Tables	909

Function calls	909
Adding methods	909
Creating generic functions	910
Closures	910
Constructors	910
Builtins	910
Keyword arguments	911
Compiler efficiency issues	912
70.9 Base.Cartesian	913
Principios de uso	913
Sintaxis Básica	914
70.10 Talking to the compiler (the mechanism)	918
70.11 SubArrays	918
Indexación: indexación cartesiana vs. lineal	918
Reemplazo de Índices	919
Diseño de SubArray's	919
70.12 System Image Building	922
Building the Julia system image	922
70.13 Working with LLVM	923
Overview of Julia to LLVM Interface	923
Building Julia with a different version of LLVM	924
Passing options to LLVM	924
Improving LLVM optimizations for Julia	924
70.14 printf() and stdio in the Julia runtime	924
Libuv wrappers for stdio	924
Interface between JL_STD* and Julia code	925
printf() during initialization	925
Legacy library	925
70.15 Bounds checking	926
Eliding bounds checks	926
Propagating inbounds	926
The bounds checking call hierarchy	927
70.16 Proper maintenance and care of multi-threading locks	927
Locks	927
Broken Locks	929
Shared Global Data Structures	929
70.17 Arrays with custom indices	930
Generalizing existing code	930
Writing custom array types with non-1 indexing	932
Summary	933
70.18 Base.LibGit2	933
70.19 Module loading	956
Experimental features	956
71 Developing/debugging Julia's C code	957
71.1 Reporting and analyzing crashes (segfaults)	957
Version/Environment info	957
Segfaults during bootstrap ()	957
Segfaults when running a script	958
Errors during Julia startup	958
Glossary	959
71.2 gdb debugging tips	959

Displaying Julia variables	959
Useful Julia variables for Inspecting	959
Useful Julia functions for Inspecting those variables	959
Inserting breakpoints for inspection from gdb	960
Inserting breakpoints upon certain conditions	960
Dealing with signals	960
Debugging during Julia's build process (bootstrap)	961
Debugging precompilation errors	962
Mozilla's Record and Replay Framework (rr)	962
71.3 Using Valgrind with Julia	962
Consideraciones Generales	962
Supresiones	963
Running the Julia test suite under Valgrind	963
Caveats	963
71.4 Sanitizer support	963
General considerations	963
Address Sanitizer (ASAN)	963
Memory Sanitizer (MSAN)	964

Part I

Home

Part II

Julia Documentation

Chapter 1

Manual

- [Introducción](#)
- [Empezando](#)
- [Variables](#)
- [Números Enteros y en Punto Flotante](#)
- [Operaciones Matemáticas y Funciones Elementales](#)
- [Números Complejos y Racionales](#)
- [Cadenas](#)
- [Functions](#)
- [Control Flow](#)
- [Scope of Variables](#)
- [Types](#)
- [Methods](#)
- [Constructors](#)
- [Conversion and Promotion](#)
- [Interfaces](#)
- [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](#)
- [Noteworthy Differences from other Languages](#)
- [Unicode Input](#)

Chapter 2

Standard Library

- [Essentials](#)
- [Collections and Data Structures](#)
- [Mathematics](#)
- [Numbers](#)
- [Strings](#)
- [Arrays](#)
- [Tasks and Parallel Computing](#)
- [Linear Algebra](#)
- [Constants](#)
- [Filesystem](#)
- [I/O and Network](#)
- [Punctuation](#)
- [Sorting and Related Functions](#)
- [Package Manager Functions](#)
- [Dates and Time](#)
- [Iteration utilities](#)
- [Unit Testing](#)
- [C Interface](#)
- [C Standard Library](#)
- [Dynamic Linker](#)
- [Profiling](#)
- [StackTraces](#)
- [SIMD Support](#)

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 \(the mechanism\)](#)
 - [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](#)
- 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

Introducción

La computación científica ha requerido tradicionalmente el máximo rendimiento, aunque los expertos de los distintos dominios se hayan movido en gran parte a los lenguajes dinámicos, más lentos, para el trabajo diario. Creemos que hay muchas buenas razones para preferir lenguajes dinámicos para estas aplicaciones, y no esperamos que su uso disminuya. Afortunadamente, el diseño de lenguajes y las técnicas de compilación modernos permiten casi eliminar el compromiso del rendimiento y proporcionar un solo entorno suficientemente productivo para la creación de prototipos y suficientemente eficiente para implementar aplicaciones de alto rendimiento. El lenguaje de programación de Julia cumple este papel: es un lenguaje dinámico y flexible, apropiado para la computación científica y numérica, con un rendimiento comparable al de los lenguajes tradicionales de tipo estático.

Debido a que el compilador de Julia es diferente de los intérpretes utilizados para lenguajes como Python o R, podría parecer al principio que el rendimiento de Julia no es intuitivo. Si encuentra que algo es lento, le recomendamos que lea la sección [Consejos de Rendimiento](#) antes de intentar otra cosa. Una vez que entienda cómo funciona Julia, será fácil escribir código casi tan rápido como el código C.

Julia ofrece tipado opcional, despacho múltiple, y buen desempeño, logrado usando inferencia de tipos y [compilación just-in-time \(JIT\)](#), implementada usando [LLVM](#). Es multi-paradigma, combinando características de programación imperativa, funcional y orientada a objetos. Julia proporciona facilidad y expresividad para la computación numérica de alto nivel, de la misma manera que idiomas como R, MATLAB y Python, pero también soporta la programación general. Para conseguirlo, Julia se basa en el linaje de los lenguajes de programación matemáticos, pero también toma prestado mucho de los lenguajes dinámicos populares, incluyendo [Lisp](#), [Perl](#), [Python](#), [Lua](#), y [Ruby](#).

Las diferencias más significativas de Julia de los lenguajes dinámicos típicos son:

- El lenguaje básico impone muy poco; La biblioteca estándar se ha escrito en el propio Julia, incluyendo operaciones primitivas como la aritmética entera.
- Un lenguaje enriquecido de tipos para construir y describir objetos, que también se puede utilizar opcionalmente para hacer declaraciones de tipo.
- La capacidad de definir el comportamiento de la función a través de muchas combinaciones de tipos de argumentos mediante el [despacho múltiple](#).
- Generación automática de código eficiente y especializado para diferentes tipos de argumentos.
- Buen rendimiento, aproximándose al de los lenguajes compilados estáticamente como C.

Aunque a veces se dice de los lenguajes dinámicos que son lenguajes "sin tipo", esto no es cierto en absoluto: cada objeto, ya sea primitivo o definido por el usuario, tiene un tipo. La falta de declaraciones de tipos en la mayoría de los lenguajes dinámicos, sin embargo, significa que uno no puede instruir al compilador acerca de los tipos de valores

y, a menudo, no puede hablar explícitamente de tipos en absoluto. En lenguajes estáticos, por otro lado, aunque uno puede -y normalmente debe- anotar tipos para el compilador, los tipos sólo existen en tiempo de compilación y no pueden ser manipulados o expresados en tiempo de ejecución. En Julia, los tipos son objetos en tiempo de ejecución y también se pueden utilizar para transmitir información al compilador.

Aunque el programador casual no necesita usar explícitamente los tipos o el despacho múltiple, ellas son las características centrales unificadoras de Julia: las funciones se definen en diferentes combinaciones de tipos de argumentos y se aplican despachando a la definición concordante más específica. Este modelo se ajusta bien a la programación matemática, donde no es natural que el primer argumento "posea" una operación como en la programación orientada a objetos tradicional. Los operadores son sólo funciones con notación especial - para ampliar la adición a nuevos tipos de datos definidos por el usuario, se definen nuevos métodos para la función `+`. El código existente se aplica sin problemas a los nuevos tipos de datos.

En parte debido a la inferencia de tipo en tiempo de ejecución (aumentada por anotaciones de tipos opcionales), y en parte debido a enfoque muy basado en el rendimiento desde el inicio del proyecto, la eficiencia computacional de Julia supera la de otros lenguajes dinámicos e incluso rivaliza con la de lenguajes de compilación estática. Para los problemas numéricos a gran escala, la velocidad siempre ha sido, continúa siendo, y probablemente siempre será crucial: la cantidad de datos procesados se ha mantenido fácilmente al ritmo de la Ley de Moore durante las últimas décadas.

Julia tiene como objetivo crear una combinación sin precedentes de facilidad de uso, potencia y eficiencia en un solo lenguaje de programación. Además de lo anterior, algunas ventajas de Julia sobre sistemas comparables son:

- Libre y de código abierto ([con licencia MIT](#))
- Los tipos definidos por el usuario son tan rápidos y compactos como los predefinidos.
- No hay necesidad de vectorizar código para el rendimiento; el código devectorizado es rápido
- Diseñado para el paralelismo y la computación distribuida.
- Hilos "verdes" de peso ligero ([coroutines](#)).
- Sistema de tipos discreto pero potente.
- Conversiones y promociones elegantes y extensibles para números y otros tipos.
- Soporte eficiente para [Unicode](#), incluyendo pero no

limitado a [UTF-8](#)

- Llamada a las funciones C directamente (no se necesitan envolturas o API especiales).
- Poderosas capacidades tipo shell para administrar otros procesos.
- Macros similares a Lisp y otras instalaciones de metaprogramación.

Chapter 5

Empezando

La instalación de Julia es sencilla, ya sea utilizando binarios precompilados o compilando desde la fuente. Descargue e instale Julia siguiendo las instrucciones disponibles en <https://julialang.org/downloads/>.

La forma más fácil de aprender y experimentar con Julia es iniciando una sesión interactiva (también conocida como *read-eval-print loop* o "REPL") haciendo doble clic en el ejecutable de Julia o ejecutando desde la línea de mandatos:

```
|
```

Para salir de la sesión interactiva, escriba `(la tecla de control junto con la tecla)` o escriba `.`. Cuando se ejecuta en modo interactivo, Julia muestra un banner y solicita al usuario la entrada. Una vez que el usuario ha introducido una expresión completa, como `, y pulsa Enter, la sesión interactiva evalúa la expresión y muestra su valor. Si se introduce una expresión en una sesión interactiva con un punto y coma al final, no se muestra su valor. La variable est está enlazada al valor de la última expresión evaluada, sea mostrada o no. La variable sólo está enlazada a las sesiones interactivas, no cuando el código Julia se ejecuta de otras maneras.`

Para evaluar expresiones escritas en un archivo de origen `, escriba .`

Para ejecutar código en un archivo de forma no interactiva, puede darlo como el primer argumento al mandato Julia:

```
|
```

Como indica el ejemplo, los siguientes argumentos de línea de mandatos de Julia se toman como argumentos de línea de mandatos al programa `del programa, pasados a través de la constante global . El nombre del propio script se pasa como la variable global . Tenga en cuenta que también se establece cuando se da el código de script usando la opción en la línea de órdenes (vea la salida de ayuda de más abajo) pero estará vacío. Por ejemplo, para imprimir los argumentos que se le dan a un script, puede hacer esto:`

```
|
```

O puede poner ese código en un script y ejecutarlo:

El delimitador `|` puede usarse para separar argumentos en línea de mandatos al fichero del *script* a los argumentos de Julia:

Julia se puede iniciar en modo paralelo con las opciones `-p` o `-P`. pondrá en marcha un procesos *worker* adicionales, mientras que `-j` iniciará un *worker* para cada línea en el archivo de archivo. Las máquinas definidas en el archivo deben ser accesibles a través de un login ssh sin contraseña, con Julia instalado en la misma ubicación que el *host* actual. Cada definición de máquina toma la forma `host:port`. El valor por defecto de `host` es el usuario actual, y el de `port` el puerto ssh estándar. Las variables opcionales `ip` especifican la dirección IP y el puerto que otros *workers* deberían usar para conectarse a este *worker*.

Si tiene código que desea ejecutar cada vez que Julia se inicia, puede ponerlo en :

Hay varias formas de ejecutar el código Julia y proporcionar opciones, similares a las disponibles para los programas `python` y `ruby`:



5.1 Resources

Además de este manual, hay otros recursos que pueden ayudar a los usuarios nuevos cuando empiezan con Julia:

- [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

Una variable en Julia es un nombre asociado a un valor. Esto es útil cuando pretendemos almacenar un valor (como el que obtenemos después de un cálculo) para un uso posterior. Por ejemplo:

Julia proporciona un sistema muy flexible para nombrar las variables. Los nombres de variable son sensibles a las mayúsculas, y no tienen significado semántico (es decir, que el lenguaje no trata de modo distinto a las variables basándose en sus nombres).

Los nombres Unicode (usando codificación UTF-8) están permitidos:

En el REPL y otros entornos de edición Julia se pueden introducir símbolos matemáticos Unicode usando la notación de *Latex* precedido de backslash y seguido de un tabulador. Por ejemplo, podemos crear el nombre de variable tecleando `-tab`, o incluso el nombre `tecleando -tab-- tab--tab`. (Si encuentras un símbolo en algún sitio, como por ejemplo en el código de alguien, y no sabes como escribirlo, el REPL te ayudará: solamente teclea y luego pega el símbolo.)

Julia también permite redefinir constantes predefinidas si fuera necesario:

Sin embargo, esto no se recomienda para evitar una potencial confusión.

6.1 Nombres de Variables Permitidos

Los nombres de variable deben comenzar con una letra (`-` o `_`), símbolo de subrayado, o un subconjunto de puntos Unicode mayores que `.`. En particular, se permiten las [categorías de caracteres Unicode](#) `Lu/LI/Lt/Lm/Lo/Nl` (letras), `Sc/So` (monedas y otros símbolos), y otros pocos caracteres (por ejemplo, un subconjunto de los símbolos matemáticos `Sm`). Entre los caracteres subsecuentes se pueden también incluir `-` y los dígitos `0-9` y otros caracteres en las categorías `Nd/No`, así como otros puntos de código Unicode: diacríticas y otras marcas de modificación (categorías `Mn/Mc/Me/Sk`), algunos conectores de puntuación (category `Pc`), primos, y otros cuantos caracteres.

Los operadores como `+` son también identificadores válidos, pero son analizados sintácticamente de un modo especial. En algunos contextos, los operadores pueden ser usados justo como variables; por ejemplo `sum` se refiere a la función de suma, y `sum` la reasignará. La mayoría de los operadores infijos Unicode (en la categoría `Sm`), tal como `⋅`, son analizados como operadores infijos y están disponibles para métodos definidos por el usuario (por ejemplo, podemos usar `⋅` para definir `⋅` como un operador infijo producto de Kronecker).

Los únicos nombres específicamente prohibidos para nombres de variables son los nombres de las instrucciones predefinidas:

Algunos caracteres Unicode son considerados equivalentes en identificadores. Las distintas formas de introducir caracteres que combinan en Unicode (por ejemplo, acentos) son tratadas como equivalentes (específicamente los identificadores Julia son normalizados NFC). Los caracteres Unicode (U+025B: Latin small letter open e) y (U+00B5: micro sign) son tratados como las letras griegas correspondientes, debido que las primeras son más fácilmente accesibles via algunos métodos de entrada.

6.2 Convenciones de Estilo

Aunque Julia impone pocas restricciones a los nombres válidos, se ha vuelto útil adoptar las siguientes convenciones:

- Los nombres de variable van en minúsculas.
- La separación entre palabras puede indicarse mediante el símbolo de guión bajo, aunque se desaconseja su uso a menos que los símbolos sean difíciles de leer.
- Los nombres de tipos y módulos comienzan con mayúscula y la separación entre palabras se representa con el formato *camel case*.
- Los nombres de funciones y macros van en minúscula, sin símbolos de guión bajo.
- Las funciones que escriben en sus argumentos tienen nombres que finalizan con el símbolo de admiración . Estas suelen ser llamadas funciones "mutadoras" o funciones "*in-place*" debido a que pretenden producir

cambios en sus argumentos después de que la función sea invocada, no solo devolver un valor.

Para más información sobre convenciones de estilo, ver la [Guía de Estilo](#).

Chapter 7

Números enteros y en punto flotante

Los valores enteros y punto flotante son los bloques constructivos básicos de la aritmética y la computación. Las representaciones predefinidas para estos valores se denominan *tipos primitivos*, mientras que las representaciones de números enteros y en punto flotante como valores inmediatos en código se conocen como *literales numéricos*. Por ejemplo, `1` es un literal entero, mientras que `1.0` es un literal en punto flotante; sus representaciones binarias en memoria como objetos son los tipos primitivos.

Julia proporciona un amplio rango de tipos primitivos numéricos, y un complemento de operadores aritméticos y de bits así como funciones matemáticas estándar definidas sobre ellos. Los operadores establecen una correspondencia entre los tipos numéricos y las operaciones que son soportadas de forma nativa sobre los ordenadores modernos, permitiendo a Julia sacar plena ventaja de los recursos computacionales. Además, Julia proporciona soporte software para *aritmética de precisión arbitraria* que puede manejar operaciones sobre valores numéricos que no puede ser representada de forma efectiva en representaciones hardware nativas, pero al coste de un rendimiento relativamente menor.

Los tipos primitivos de Julia son los siguientes:

- **Tipos enteros:**

Tipo	Signo?	Número de bits	Valor más pequeño	Valor más grande
		8	-2^7	$2^7 - 1$
		8	0	$2^8 - 1$
		16	-2^{15}	$2^{15} - 1$
		16	0	$2^{16} - 1$
		32	-2^{31}	$2^{31} - 1$
		32	0	$2^{32} - 1$
		64	-2^{63}	$2^{63} - 1$
		64	0	$2^{64} - 1$
		128	-2^{127}	$2^{127} - 1$
		128	0	$2^{128} - 1$
	N/A	8	(0)	(1)

- **Tipos en punto flotante:**

Adicionalmente, se ha construido un soporte completo para [Números Complejos y Racionales](#) encima de estos tipos primitivos. Todos los tipos primitivos interoperan de forma natural sin tener que realizar conversiones específicas, gracias a un [sistema de promoción de tipos](#) flexible y extensible por el usuario.

Tipo	Precisión	Número de bits
	media	16
	sencilla	32
	doble	64

7.1 Enteros

Los literales enteros se representan del modo estándar:

|

El tipo por defecto para un literal entero depende de si el sistema de trabajo tiene una arquitectura de 32 o de 64 bits:

|

La variable interna de Julia indica si el sistema en el que trabajamos es de 32 bits o de 64 bits:

|

Julia también define los tipos `Int` y `UInt`, que son alias para los tipos enteros nativos del sistema con y sin signo:

|

Los literales enteros mayores que no pueden ser representados usando sólo 32 bits pero pueden ser representados en 64 bits se crean como enteros de 64 bits, independientemente del tipo que tenga el sistema por defecto:

Los enteros sin signo se introducen y se muestran usando el prefijo y los dígitos hexadecimales (los dígitos capitalizados también funcionan para la entrada). El tamaño de un valor sin signo está determinado por el número de dígitos hexadecimales usados:

Este comportamiento está basado en la observación de que cuando uno usa literales hexadecimales sin signo para valores enteros, se los suele utilizar para representar una secuencia de bytes numéricos fijos en lugar de un valor entero.

Recuerde que la variable `__` contiene el valor de la última expresión evaluada en una sesión interactiva. Esto no ocurre cuando el código Julia se ejecuta de otra forma.

Los literales binarios y octales también están soportados:

Los valores máximo y mínimo de tipos primitivos numéricos representables como enteros vienen dados por las funciones `isbitsize` y `isbitsize`:

Los valores devueltos por `div` y `rem` siempre son del tipo de argumento dado. (La expresión anterior utiliza varias características que todavía tenemos que introducir, incluyendo [bucles for](#), [Cadenas](#), e [Interpolación](#), pero debería ser lo suficientemente fácil de entender para los usuarios con cierta experiencia en programación.)

Comportamiento ante el Desbordamiento

En Julia, superar el valor máximo representable de un tipo dado da como resultado un comportamiento envolvente:

Así, la aritmética con enteros de Julia es en realidad una forma de [aritmética modular](#). Esto refleja las características de la aritmética subyacente de números enteros tal como se implementa en las computadoras modernas. En aplicaciones donde es posible el desbordamiento, es esencial comprobar explícitamente el envolvente producido por el desbordamiento. De lo contrario, se recomienda el tipo `BigInt` en [Aritmética de Precisión Arbitraria](#).

Errores de división

La división entera (la función `div`) tiene dos casos excepcionales: división por cero, y dividir el número negativo más bajo (`-1`) por `-1`. Ambos casos lanzan un `DivideError`. El resto y las funciones de módulo (`rem` y `%`) lanzan un `DivideError` cuando su segundo argumento es cero.

7.2 Números en Punto Flotante

Los literales de números en punto flotante son representados en las formas estándar:



Los resultados anteriores son todos valores . Los valores literales pueden introducirse escribiendo en lugar de :



Los valores pueden ser convertidos a fácilmente:



También son válidos los literales de punto flotante en formato hexadecimal, pero sólo como valores :



También esta soportados los números en punto flotante de media precisión (), pero ellos están implementados en software y usan para los cálculos.

El guión bajo (*underscore*) puede usarse como separador de dígitos:

Cero en punto flotante

Los números en punto flotante tienen **dos ceros**, positivo y negativo. Ellos son iguales entre sí, pero tienen distintas representaciones, como puede verse si usamos la función :

Valores especiales en punto flotante

Hay tres valores especificados en el estándar de punto flotante para valores que no se corresponden con ningún punto en la línea de números reales:

			Name	Description
			Infinito positivo	Un valor mayor que todos los valores finitos de punto flotante
			Infinito negativo	Un valor menos que todos los valores finitos de punto flotante
			<i>not a number</i>	Un valor no a otro valor en punto flotante (incluido el mismo)

Para más información sobre cómo estos valores de punto flotante no finitos están ordenados entre sí y otros flotantes, vea [Comparaciones Numéricas](#). Mediante el [estándar IEEE 754](#), estos valores de punto flotante son el resultado de ciertas operaciones aritméticas:

|

Las funciones `nextafter` y `nexttoward` también se aplican a los tipos en punto flotante:

|

Epsilon de máquina

La mayoría de los números reales no pueden representarse exactamente con números de coma flotante, por lo que para muchos propósitos es importante conocer la distancia entre dos números de punto flotante representables adyacentes, lo que a menudo se conoce como *epsilon de máquina*.

Julia proporciona `eps{T}`, que da la distancia entre 1.0 y el siguiente valor de punto flotante representable más grande:

|

Estos valores son `eps{Float64}` y `eps{Float32}` como valores `eps{Float64}` y `eps{Float32}`, respectivamente. La función `eps` también puede tomar un valor de punto flotante como un argumento y da la diferencia absoluta entre ese valor y el siguiente valor de punto flotante representable. Es decir, produce un valor del mismo tipo que `x` tal que `x + eps(x)` es el siguiente valor de punto flotante representable mayor que `x`.

La distancia entre dos números de punto flotante representables adyacentes no es constante, pero es menor para valores más pequeños y mayor para valores mayores. En otras palabras, los números de punto flotante representables son más densos en la línea de números reales cerca de cero, y crecen exponencialmente dispersos a medida que uno se aleja de cero. Por definición, `eps` es el mismo que `mach` ya que `eps` es un valor de coma flotante de 64 bits.

Julia también proporciona las funciones `nextfloat` y `prevfloat` que devuelven el siguiente número de punto flotante representable más grande o más pequeño al argumento, respectivamente:

Este ejemplo resalta el principio general de que los números de punto flotante representables adyacentes también tienen representaciones binarias enteras adyacentes.

Modos de Redondeo

Si un número no tiene una representación de punto flotante exacta, debe redondearse a un valor representable apropiado. Sin embargo, si se desea, la forma en que se realiza este redondeo puede cambiarse de acuerdo con los modos de redondeo presentados en el [estándar IEEE 754](#).

El modo predeterminado utilizado siempre es `RoundNearest`, que redondea al valor representable más cercano, con arcos redondeados hacia el valor más cercano con un bit menos significativo.

Warning

El redondeo generalmente sólo es correcto para las funciones aritméticas básicas (`+`, `-`, `*`, and `/`) y las operaciones de conversión de tipos. Muchas otras funciones asumen que el modo por defecto está establecido y pueden dar resultados erróneos al operar bajo otros modos de redondeo.

Antecedentes y referencias

La aritmética de punto flotante supone muchas sutilezas que pueden sorprender a los usuarios que no están familiarizados con los detalles de implementación de bajo nivel. Sin embargo, estas sutilezas se describen en detalle en la mayoría de los libros sobre computación científica, y también en las siguientes referencias:

- La guía definitiva para la aritmética de coma flotante es el estándar [IEEE 754-2008 (<http://standards.ieee.org/findstds/standard/754-2008.html>); Sin embargo, no está disponible

en línea gratis.

- Para una presentación breve pero lúcida de cómo los números de punto flotante están

representados, vea el [artículo de John D. Cook](#) sobre el tema, así como su [introducción](#) a algunas de las cuestiones que surgen de cómo esta representación difiere en el comportamiento de la abstracción idealizada de números reales.

* También se recomienda la serie de [publicaciones de Bruce Dawson sobre números en punto flotante](#).

- Para un excelente y profundo análisis de los números de punto flotante y los problemas de precisión numérica encontrados al calcular con ellos, vea el artículo de David Goldberg [What Every Computer Scientist Should Know About Floating-Point Arithmetic](#).
- Para una documentación aún más extensa de la historia de, la razón y las cuestiones con los números de punto flotante, así como la discusión de muchos otros temas en la computación numérica, ver los [escritos recolectados](#) de [William Kahan](#), comúnmente conocido como el "padre de punto flotante". De interés particular puede ser [An Interview with the Old Man of Floating-Point](#).

7.3 Aritmética de Precisión Arbitraria

Para permitir cálculos con enteros y números de coma flotante de precisión arbitraria, Julia envuelve la [Biblioteca Aritmética de Precisión Múltiple GNU \(GMP\)](#) y la [biblioteca GNU MPFR](#), respectivamente. Los tipos `BigInt` y `BigFloat` están disponibles en Julia para números enteros de precisión arbitraria y números de coma flotante, respectivamente.

Existen constructores para crear estos tipos de tipos numéricos primitivos, y podemos también utilizar `BigInt{<T>}` para construirlos a partir de `T`. Una vez creados, participan en la aritmética con todos los demás tipos numéricos gracias al [mecanismo de promotion y conversion de tipos](#) de Julia:

Sin embargo, la promoción de tipos entre los tipos primitivos ya vistos y / no es automática y debe ser establecida explícitamente:

La precisión predeterminada (en número de bits del significado) y el modo de redondeo de las operaciones de pueden cambiarse globalmente llamando `std::setprecision` y `std::setroundmode`, y todos los cálculos adicionales tomarán en cuenta estos cambios. Alternativamente, la precisión o el redondeo se puede cambiar dentro sólo de la ejecución de un bloque particular de código utilizando las mismas funciones dentro de un bloque :

7.4 Coeficientes Literales Numéricos

Para hacer más claras fórmulas numéricas y expresiones, Julia permite que las variables sean precedidas inmediatamente por un literal numérico, implicando la multiplicación. Esto hace que la escritura de las expresiones polinómicas sea mucho más limpias:

También hace que escribir funciones exponenciales sea más elegante:

La precedencia de los coeficientes literales numéricos es la misma que la de los operadores unarios como la negación. Así que `-x` se analiza como `-x`, y `-x^2` se analiza como `-(x^2)`.

Los literales numéricos también funcionan como coeficientes de las expresiones entre paréntesis:

Note

La precedencia de coeficientes literales numéricos usada para multiplicación implícita es mayor que otros operadores binarios tales como la multiplicación (`*`), y división (`/`, `÷`, and `÷`). Esto significa, por ejemplo, que `2x^2` es igual a `2*(x^2)` y `x^2` es igual a `(x^2)`.

Además, las expresiones entre paréntesis se pueden utilizar como coeficientes a las variables, lo que implica la multiplicación de la expresión por la variable:

Sin embargo, ni la yuxtaposición de dos expresiones entre paréntesis, ni la colocación de una variable antes de una expresión entre paréntesis puede ser usada para implicar multiplicación:

Ambas expresiones se interpretan como la aplicación de una función: cualquier expresión que no sea un literal numérico, inmediatamente seguida de una entre paréntesis, se interpreta como una función aplicada a los valores entre paréntesis (ver [Funciones](#) para más información sobre las funciones). Por lo tanto, en ambos casos, se produce un error, ya que el valor de la izquierda no es una función.

Las mejoras sintácticas anteriores reducen significativamente el ruido visual producido al escribir fórmulas matemáticas comunes. Obsérvese que ningún espacio en blanco puede encontrarse entre un coeficiente literal numérico y el identificador o la expresión entre paréntesis que multiplica.

Conflictos de Sintaxis

La sintaxis de los coeficientes literales yuxtapuestos puede entrar en conflicto con dos sintaxis numéricas literales: literales enteros hexadecimales y notación ingenieril para literales de punto flotante. Aquí hay algunas situaciones donde surgen conflictos sintácticos:

- La expresión literal de enteros hexadecimales podría interpretarse como el literal numérico multiplicado por la variable .
- La expresión literal de punto flotante podría interpretarse como el literal numérico multiplicado por la variable , e igualmente con la forma equivalente

En ambos casos, resolvemos la ambigüedad a favor de la interpretación como literales numéricos:

- Las expresiones que comienzan con siempre son literales hexadecimales.
- Las expresiones que empiezan con un literal numérico seguido por e o E siempre son literales de coma flotante.

7.5 Literales cero and uno

Julia proporciona funciones que devuelven los literales y correspondientes a un tipo especificado o al tipo de una variable dada.

Function	Description
	Literal cero del tipo o del tipo de la variable
	Literal uno del tipo o del tipo de la variable

Estas funciones son útiles en [comparaciones numéricas](#) para evitar la sobrecarga de una [conversión de tipo](#) innecesaria.

Ejemplos:

|

Chapter 8

Mathematical Operations and Elementary Functions

Julia proporciona una colección completa de operadores aritméticos básicos y de operadores de bits para todos sus tipos numéricos primitivos, así como implementaciones portables y eficientes de una colección comprensiva de funciones matemática estándar.

8.1 Operadores Aritméticos

Los siguientes **operadores aritméticos** están soportados sobre todos los tipos primitivos:

Expression	Name	Description
	más unario	Operación identidad
	menos unario	Inverso matemático de un número
	suma binaria	suma
	menos binario	resta
	producto	multiplicación
	división	división
	división inversa	Equivalente a
	potencia	eleva a la -ésima potencia
	resto	Equivalente a

así como la negación sobre tipos :

Expression	Name	Description
	negación	Cambia a y viceversa

El sistema de promoción de Julia hace que las operaciones aritméticas sobre mezclas de tipos de argumentos funcione de forma natural y automáticamente. Ver [Conversión y Promoción](#) para los detalles del sistema de promoción.

He aquí algunos ejemplos simples de usar operadores aritméticos:

|

(Por convención, tendemos a separar con menos distancia los operadores cuando se aplican antes de otros operadores cercanos. Por ejemplo, generalmente escribimos `~a` para reflejar que primero se niega `a`, y a continuación, se agrega a ese resultado.)

8.2 Operadores bit a bit

Los siguientes **operadores bit a bit** son soportados sobre todos los tipos enteros primitivos:

Expression	Name
<code>~a</code>	Negación bit a bit
<code>a & b</code>	Conjunción (<i>and</i>) bit a bit
<code>a b</code>	Disyunción (<i>or</i>) bit a bit
<code>a ^ b</code>	Or exclusivo bit a bit (<i>xor</i>)
<code>a >> n</code>	Desplazamiento lógico hacia la derecha
<code>a >>= n</code>	Desplazamiento aritmético hacia la derecha
<code>a << n</code>	Desplazamiento hacia la izquierda lógico/aritmético

He aquí algunos ejemplos de uso de operadores bit a bit:

8.3 Operaciones de actualización

Cada operador binario aritmético y bit a bit también tiene una versión de actualización que asigna el resultado de la operación de nuevo a su operando izquierdo. La versión de actualización del operador binario se forma colocando `=` inmediatamente después del operador. Por ejemplo, `a = a & b` es equivalente a escribir :

Las versiones de actualización de todos los operadores binarios, aritméticos de bits son:

Note

Un operador de actualización reasigna la variable sobre la parte izquierda de la ecuación. Como resultado, el tipo de la variable puede cambiar:

8.4 Operadores vectorizados con "punto"

Para cada operación binaria como `+` hay su correspondiente operación "con punto" que se define *automáticamente* para realizar la operación elemento a elemento sobre arrays. Por ejemplo, la operación `^` no está definida, porque no hay un significado matemático estándar para calcular el cubo de un array, pero `.^` si lo está como el cálculo de la operación cubo elemento a elemento (o vectorizada). Lo mismo puede decirse para operadores unarios tales como `abs` o `sqrt`, que existe el correspondiente operador vectorizado que aplica el operador elemento a elemento.

Más específicamente, `.*` es analizado como la llamada `punto`, que realiza una operación de *retransmisión (broadcast)*: ella puede combinar arrays y escalares, arrays del mismo tamaño (realizando la operación elemento a elemento), o incluso arrays de diferentes formas (por ejemplo, combinar vectores fila y columna para producir una matriz). Además, como todas las "llamadas punto", estos "operadores punto" están *fusionados*. Por ejemplo, si calculamos `sum(A.*B)` (o, equivalentemente `sum(A*B)`, usando la macro `sumprod`) para un array `A`, se realiza un *único* bucle sobre `A`, computando `sumprod(A,B)` para cada elemento de `A`. En particular, las llamadas vectorizadas anidadas como `sumprod(A,B,C)` están *fusionadas*, y los operadores binarios adyacentes como `A.*B.*C` son equivalentes a llamadas vectorizadas anidadas `sumprod(A,B,C)`.

Además, los operadores de actualización "vectorizados" como `.*=` (o `./=`) son transformados en `sumprod(A,B,C)`, donde `sumprod` es un operador de asignación *fusionado in-place* (ver la [documentación de la sintaxis vectorizada](#)).

Nótese que la sintaxis de punto es también aplicable a operadores definidos por el usuario. Por ejemplo, si definimos el operador `prod` para dar una sintaxis infija al producto de Kronecker (`prod(A,B)`), entonces `prod(A,B)` calculará `prod(A,B)` sin ninguna codificación adicional.

Operador	Nombre
	Igualdad
,	Desigualdad
	Menor que
,	Menor o igual que
	Mayor que
,	Mayor o igual que

8.5 Comparaciones Numéricas

Los operadores de comparación estándar están definidos para todos los tipos numéricos primitivos:

He aquí algunos ejemplos:

Los enteros se comparan de un modo estándar, mediante comparación de bits. Los números de punto flotante se comparan de acuerdo al [estándar IEEE 754](#):

- Los números finitos son ordenados del modo habitual.
- El cero positivo es igual pero no mayor que el cero negativo.

- es igual a si mismo y mayor que todo excepto
- es igual a si mismo y menor que todo excepto
- no es igual, mayor o menor a nadie, excepto a sí mismo.

Este último punto es potencialmente sorprendente y, por tanto, vale la pena señalar que:

|

y puede causar dolores de cabeza especiales con [Arrays](#):

|

Julia proporciona funciones adicionales para comprobar números para valores especiales, lo cuál pueden ser útil en situaciones como las comparaciones de claves hash:

Function	Tests if
	e son idénticos
	es un número finito
	es infinito
	no es un número

considera los s iguales entre sí:

|

también puede usarse para distinguir los ceros con signo:

|

Las comparaciones de tipos mezclados entre enteros con signo, enteros sin signo y valores en punto flotante pueden ser complicadas. Se ha tomado mucho cuidado para asegurarse de que Julia las realiza correctamente.

Para otros tipos, llama por defecto a `isless`, así que si uno quiere definir la igualdad para sus propios tipos, solo tiene que agregar un método. Si uno define su propia función de igualdad, probablemente deba definir un método correspondiente para asegurar de que implica.

Comparaciones Encadenadas

A diferencia de la mayoría de los idiomas, [con la notable excepción de Python](#), las comparaciones pueden encadenarse arbitrariamente:

```
|
```

El encadenamiento de comparaciones suele ser bastante conveniente en el código numérico. Las comparaciones encadenadas utilizan el operador `<` para comparaciones escalares y el operador `<.>` para comparaciones elemento a elemento, lo que les permite trabajar sobre arrays. Por ejemplo, `0 < A < 1` da un array booleano cuyas entradas son `true` en posiciones en las que los elementos correspondientes de `A` están entre 0 y 1.

Nótese el comportamiento de evaluación de las comparaciones encadenadas:

```
|
```

La expresión del medio sólo se evalúa una vez, en lugar de dos veces como lo sería si la expresión se escribiera como `0 < A && A < 1`. Sin embargo, el orden de las evaluaciones en una comparación encadenada no está definido. Se recomienda encarecidamente no utilizar expresiones que puedan tener efectos secundarios (como la impresión) en comparaciones encadenadas. Si se requieren efectos secundarios, se debe utilizar explícitamente el operador de cortocircuito `&&` ([ver Evaluación en cortocircuito](#)).

Funciones Elementales

Julia proporciona una colección completa de funciones matemáticas y operadores. Estas operaciones matemáticas se definen sobre una clase de valores numéricos suficientemente amplia como para permitir definiciones apropiadas para enteros, números de punto flotante, racionales y complejos, dondequiera que tales definiciones tengan sentido.

Además, estas funciones (como cualquier función de Julia) se pueden aplicar de manera "vectorizada" a matrices y otras colecciones con la [sintaxis vectorizada](#), por ejemplo, `sin(A)` calculará el seno de cada elemento de una matriz.

Categorí	Operadores
Syntax	seguido por
Exponentiation	
Fractions	
Multiplication	
Bitshifts	
Addition	
Syntax	seguido por
Comparisons	
Control flow	seguido por seguido por
Assignments	

8.6 Precedencia de Operadores

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

Para una lista completa de cada una de las precedencias de operadores de Julia, consultar el fichero

También puede encontrarse la precedencia numérica pra cualquier operación dada mediante la función intrínseca `precedence` donde el número mayor corresponde a la operación con mayor precedencia.

8.7 Conversiones Numéricas

Julia soporta tres formas de conversión numérica, que difieren en su manejo de las conversiones inexactas.

- La notación `Float{T}` o `Float64` convierte a un valor de tipo `T`.
 - Si `x` es un tipo en punto flotante, el resultado es el valor más cercano representable, que podría ser infinito positivo o negativo.
 - Si `x` es un tipo entero, se lanzará un `OverflowError` si no es representable por `T`.
- `round` convierte un entero `x` a un valor de un tipo entero congruente a `m` modulo `2^m`, donde `m` es el número de bits en `T`. En otras palabras, la representación binaria es truncada para ajustarse.
- Las [Funciones de Redondeo](#) toman un tipo `T` como argumento opcional. Por ejemplo, `round{Float64}` es una abreviatura de `round{Float64}(x)`.

Los siguientes ejemplos muestran las siguientes formas:

Ver [Conversión y Promoción](#) para ver cómo definir tus propias conversiones y promociones.

Funciones de Redondeo

Función	Descripción	Tipo devuelto
	Redondea al entero más cercano	
	Redondea al entero más cercano	
	Redondea hacia	
	Redondea hacia	
	Redondea hacia	
	Redondea hacia	
	Redondea hacia cero	
	Redondea hacia cero	

Funciones de División

Funciones de signo y valor absoluto

Potencias, logaritmos y raíces

Para una explicación de por qué son necesarias funciones como `expm1`, `log1p`, `erfc`, e `hypot`, véase el excelente par de artículos en el blog de John D. Cook's sobre el tema: [expm1](#), [log1p](#), [erfc](#), e [hypot](#).

Función	Descripción
	División truncada; cociente redondeado hacia cero
	División <i>floored</i> ; cociente redondeado hacia
	División <i>ceiling</i> ; cociente redondeado hacia
	Resto; satisface ; el signo se corresponde con el de
	Módulo; satisface ; el signo se corresponde con el de
	Módulo con un desplazamiento de 1; devuelve para o para , donde
	Módulo con respecto a 2pi;
	Devuelve
	Devuelve
	Máximo común divisor positivo de , ,...
	Mínimo común múltiplo positivo de , ,...

Función	Descripción
	Un valor positivo con la magnitud de
	El cuadrado de la magnitud de
	Indica el signo de , devolviendo -1, 0, o +1
	Indica que si el bit de signo está en on () o en off ()
	Indica un valor con la magnitud de y el signo de
	Indica un valor con la magnitud de y el signo de

Función	Descripción
,	Raíz cuadrada de
,	Raíz cúbica de
	Hipotenusa del triángulo rectángulo cuyos catetos son de longitudes e
	Función exponencial natural sobre
	Valor exacto de para cercano a zero
	calculado eficientemente para valores enteros de
	Logaritmo neperiano de
	Logaritmo en base de
	Logaritmo en base 2 de
	Logaritmo decimal de
	Valor exacto de para cercano a zero
	Exponente binario de
	Significando binario (alias <i>mantisa</i>) de un número en punto flotante

Funciones Trigonómicas e Hiperbólicas

Todas las funciones trigonométricas e hiperbólicas estándar están también definidas:

Son todas funciones de un solo argumento, con la excepción de `atan2`, que da el ángulo en **radians** entre el eje x y el punto especificado por sus argumentos, interpretado como sus coordenadas x e y.

Adicionalmente, se proporcionan e para cálculos más exactos de y respectivamente.

Para computar funciones trigonométricas con grados en lugar de con rdianes, añada al nombre de la función el sufijo

. Por ejemplo, calcula el seno de , donde se especifica en grados. La lista completa de funciones trigonométricas con variantes grados es:

|

Funciones Especiales

Función	Descripción
	Función gamma en
	Valor exacto de para valores grandes de
	Valor exacto de para valores grandes de ; igual que para , cero en otros caso
	Función beta en
	Valor exacto de para valores grandes de o

Chapter 9

Números Racionales y Complejos

Julia se distribuye con tipos predefinidos que representan números complejos y racionales, y soporta todas las [Operaciones Matemáticas y Funciones Elementales](#) estándar sobre ellos. Se han definido [conversiones y promociones](#) de modo que las operaciones con cualquier combinación de tipos numéricos predefinidos, primitivos o compuestos, se comporten como se esperaba.

como se esperaba.

9.1 Números Complejos

La constante global `im` está ligada al número complejo i , que representa la raíz cuadrada principal de -1 . Se consideró nocivo para co-optar el nombre `im` para una constante global, ya que es un nombre de variable de índice popular. Como Julia permite que los literales numéricos se [yuxtapongan con identificadores como coeficientes](#), esta unión es suficiente para proporcionar sintaxis conveniente para números complejos, similar a la notación matemática tradicional:

|

Podemos realizar todas las operaciones aritméticas estándar con los números complejos:

|

El mecanismo de promoción asegura que las combinaciones de operandos de distintos tipos funcionarán:

Nótese que , ya que un coeficiente literal se enlaza más fuerte que la división.

También se proporcionan las funciones estándar para manipular valores complejos:

Como de costumbre, el valor absoluto ($|z|$) de un número complejo es su distancia a cero. $|z|$ da el cuadrado del valor absoluto, y es de uso particular para los números complejos donde se evita tomar una raíz cuadrada. $\arg(z)$ devuelve el ángulo de fase en radianes (también conocido como *argumento* o función *arg*). La gama completa de otras [funciones elementales](#) está también definida para los números complejos:

Tenga en cuenta que las funciones matemáticas normalmente devuelven valores reales cuando se aplican a números reales y valores complejos cuando se aplican a números complejos. Por ejemplo, \sin se comporta de forma diferente cuando se aplica a π que cuando se aplica sobre i , aunque :

La [notación de coeficiente numérico literal](#) no funciona cuando se construye un número complejo a partir de variables. En su lugar, la multiplicación debe expresarse explícitamente:

Sin embargo, esto no es lo recomendable; En su lugar, utilice la función `complex` para construir un valor complejo directamente de sus partes real e imaginaria:

Esta construcción evita las operaciones de multiplicación y adición.

y se propagan a través de números complejos en las partes real e imaginaria de un número complejo como se describe en la sección [valores especiales en punto flotante](#) section:

9.2 Números Racionales

Julia tiene un tipo numérico racional para representar razones exactas de enteros. Los racionales se construyen usando el operador :

Si el numerador y el denominador de un racional tienen factores comunes, ellos son reducidos a los términos mínimos tales que el denominador sea no negativo:

Esta forma normalizada para una razón de enteros es única, por lo que la igualdad de valores racionales puede ser testada comprobando la igualdad del numerador y el denominador. El numerador estandarizado y el denominador de un valor racional pueden ser extraídos usando las funciones `num` y `den`:

La comparación directa de numerador y denominador no suele ser necesaria, ya que la aritmetica estándar y las operaciones de comparación están definidas para los valores racionales:

|

Los racionales pueden convertirse fácilmente en número en punto flotante:

|

La conversión de racional a punto flotante respeta la siguiente identidad para dos valores enteros cualesquiera y y x , con las excepciones de los casos `and` :

|

Construir valores racionales infinitos es aceptable:

|

Sin embargo, no lo es tratar de construir un valor NaN racional:

|

Como es natural, el sistema de promoción hace que las interacciones con otros tipos numéricos se hagan sin esfuerzo alguno:

|

Chapter 10

Strings

Las cadenas son secuencias finitas de caracteres. Por supuesto, el verdadero problema viene cuando uno se pregunta qué es un carácter. Los caracteres con los que están familiarizados con los hablantes de inglés son las letras , , , etc., junto con los números y los símbolos de puntuación comunes. Estos caracteres se estandarizan junto con una correspondencia a valores enteros entre 0 y 127 a través del estándar ASCII. Hay, por supuesto, muchos otros caracteres utilizados en lenguas no inglesas, incluyendo variantes de los caracteres ASCII con acentos y otras modificaciones, escrituras relacionadas como cirílico y griego, y escrituras no relacionadas en absoluto con ASCII o inglés, entre los que se incluyen árabe, chino, Hebreo, hindi, japonés y coreano. El estándar [Unicode](#) aborda las complejidades de lo que es exactamente un carácter, y es generalmente aceptado como el estándar definitivo que aborda este problema. Dependiendo de tus necesidades, puedes ignorar estas complejidades por completo y fingir que sólo existen caracteres ASCII, o puedes escribir código que pueda manejar cualquiera de los caracteres o codificaciones que se pueden encontrar al manejar texto no ASCII. Julia hace que el manejo de texto ASCII sencillo sea simple y eficiente, y el manejo de Unicode tan simple y eficiente como sea posible. En particular, puedes escribir código de cadenas con estilo C para procesar cadenas ASCII y funcionarán como se esperaba, tanto en términos de rendimiento como de semántica. Si dicho código encuentra texto no ASCII, fallará graciosamente con un mensaje de error claro, en lugar de introducir en silencio resultados corruptos. Cuando esto sucede, modificar el código para manejar datos no ASCII es sencillo.

Hay algunas características destacadas de alto nivel sobre las cadenas de caracteres en Julia:

- El tipo de concreto incorporado utilizado para cadenas (y literales de cadena) en Julia es `String`. Esto soporta el rango completo de caracteres [Unicode](#) a través de la codificación [UTF-8](#). (se proporciona una función `String{Char}` para convertir a/desde otras codificaciones Unicode).
- Todos los tipos de cadenas son subtipos del tipo abstracto `AbstractString` y los paquetes externos definen subtipos adicionales (por ejemplo, para otras codificaciones). Si define una función que espera un argumento de cadena, debe declarar el tipo como `AbstractString` para aceptar cualquier tipo de cadena.
- Como C y Java, pero a diferencia de la mayoría de los lenguajes dinámicos, Julia tiene un tipo de primera clase que representa un solo carácter, llamado `Char`. Esto es sólo un tipo especial de bits de 32 bits cuyo valor numérico representa un punto de código Unicode.
- Como en Java, las cadenas son inmutables: el valor de un objeto `String` no se puede cambiar. Para construir un valor de cadena diferente, se construye una nueva cadena de partes de otras cadenas.
- Conceptualmente, una cadena es una *función parcial* de índices a caracteres: para algunos valores de índice, no se devuelve ningún valor de carácter y, en su lugar, se genera una excepción. Esto permite una indexación eficiente en cadenas por el índice de bytes de una representación codificada en lugar de por un índice de caracteres, que no se puede implementar de manera eficiente y sencilla para encodificaciones de anchura variable de cadenas Unicode.

10.1 Caracteres

Un valor `char` representa un solo carácter: es sólo un *bitstype* de 32 bits con una representación literal especial y comportamientos aritméticos apropiados, cuyo valor numérico se interpreta como un *punto de código Unicode*. Aquí se muestra cómo se introducen y se muestran los valores :

```
|
```

Podemos convertir un `char` a su valor entero (su punto de código) fácilmente:

```
|
```

En arquitecturas de 32 bits, `char` será `int`. Puede convertir un valor entero de nuevo a un `char` fácilmente:

```
|
```

No todos los valores enteros son puntos de código Unicode válidos, pero por una cuestión de rendimiento, la conversión `int` no comprueba que cada valor de carácter sea válido. Si desea comprobar que cada valor convertido es un punto de código válido, utilice la función :

```
|
```

A partir de este momento, los puntos de código Unicode válidos son `0` a `0xFFFF` y `0x10000` a `0x10FFFF`. A estos no se les han asignado todavía significados inteligibles, ni son necesariamente interpretables por las aplicaciones, pero todos ellos se consideran caracteres Unicode válidos.

Puede introducir cualquier carácter Unicode entre comillas simples utilizando `u''` seguido de hasta cuatro dígitos hexadecimales o `U''` seguido de hasta ocho dígitos hexadecimales (el valor válido más largo sólo requiere seis):

```
|
```

Julia utiliza la configuración regional y de idioma de tu sistema para determinar qué caracteres se pueden imprimir tal cual y cuáles se deben imprimir utilizando las formas de entrada genéricas, escapadas con `\` o `u`. Además de estas formas de escape de Unicode, también se pueden usar todas las [formas de entrada de escape tradicionales de C](#):

|

Puedes hacer comparaciones y una cantidad limitada de aritmética con los valores :

|

10.2 Fundamentos de Cadenas

Los literales de cadenas están delimitados por comillas dobles o comillas dobles triples:

|

Si desea extraer un carácter de una cadena, indéxelo:

Toda la indexación en Julia está basada en 1: el primer elemento de cualquier objeto indexado mediante enteros se encuentra en el índice `1`, y el último elemento se encuentra en el índice `length(s)`, cuando la cadena tiene una longitud de `length(s)`.

En cualquier expresión de indexación, puede usarse la palabra clave `end` como una abreviatura para el último índice (calculado mediante `length(s)`). Puede realizar operaciones aritméticas y otras con `end`, como si de un valor normal se tratara:

Usar un índice menor que 1 o mayor que `length(s)` lanza un error:

También puedes extraer una subcadena usando indexación mediante un rango:

Nótese que las expresiones `s[1:2]` y `s[1:1]` no dan el mismo resultado:

La primera es un valor carácter de tipo `Char`, mientras que la segunda es un valor cadena que tiene un único carácter. En Julia se trata de cosas muy diferentes.

10.3 Unicode y UTF-8

Julia soporta totalmente caracteres y cadenas Unicode. Como se ha [comentado anteriormente](#), en literales de caracteres, los puntos de código Unicode se pueden representar usando las secuencias de escape Unicode `\uXXXX` y `\UXXXXXXXX`, así como todas las secuencias de escape C estándar. Éstos también se pueden utilizar para escribir literales de cadena:

Si estos caracteres Unicode se muestran como escapes o se muestran como caracteres especiales depende de la configuración regional de tu terminal y su compatibilidad con Unicode. Los literales de cadena se codifican utilizando la codificación UTF-8. UTF-8 es una codificación de ancho variable, lo que significa que no todos los caracteres están codificados en el mismo número de bytes. En UTF-8, los caracteres ASCII -es decir, aquellos con puntos de código inferiores a 0x80 (128) - están codificados como lo están en ASCII, usando un solo byte, mientras que los puntos de código 0x80 y superiores se codifican utilizando múltiples bytes (hasta cuatro por carácter). Esto significa que no todos los índices de bytes en una cadena UTF-8 es necesariamente un índice válido para un carácter. Si indexas una cadena en un índice de bytes no válido, se genera un error:

```

1  string s = "hola mundo";
2  for (int i = 0; i < s.Length; i++)
3  {
4      Console.WriteLine(s[i]);
5  }
6  Console.ReadLine();
7  }
```

En este caso, el carácter es un carácter de tres bytes, por lo que los índices 2 y 3 no son válidos y el índice del siguiente carácter es 4; este siguiente índice válido puede ser calculado con `s.IndexOf(s[i])`, y el siguiente índice después de éste con `s.IndexOf(s[s.IndexOf(s[i])])` y así sucesivamente.

Debido a las codificaciones de longitud variable, el número de caracteres de una cadena (dada por `s.Length`) no siempre lo mismo que el último índice. Si se itera a través de los índices 1 hasta `s.Length` y se indexa en `s[i]`, la secuencia de caracteres devueltos cuando no se lanzan errores es la secuencia de caracteres que contiene la cadena `s`. Por tanto, tenemos la identidad de que `s[s.IndexOf(s[i])] == s[i]`, ya que cada carácter en una cadena debe tener su propio índice. La siguiente es una forma ineficaz y verbosa de iterar a través de los caracteres de `s`:

```

1  string s = "hola mundo";
2  for (int i = 0; i < s.Length; i++)
3  {
4      Console.WriteLine(s[s.IndexOf(s[i])]);
5  }
6  Console.ReadLine();
7  }
```

Las líneas en blanco en realidad tienen espacios en ellos. Afortunadamente, el idioma anterior incómodo es innecesario para iterar a través de los caracteres de una cadena, ya que se puede utilizar la cadena como un objeto iterable, sin que se requiera el manejo de excepciones:

```

1  string s = "hola mundo";
2  foreach (char c in s)
3  {
4      Console.WriteLine(c);
5  }
6  Console.ReadLine();
7  }
```

Julia utiliza la codificación UTF-8 de forma predeterminada y el soporte para nuevas codificaciones puede agregarse mediante paquetes. Por ejemplo, el paquete [LegacyStrings.jl](#) implementa los tipos `LegacyString` y `LegacyChar`. Una mayor discusión sobre otras codificaciones y cómo implementar el soporte para ellas está más allá del alcance de este documento por el momento. Para más información sobre los problemas de codificación UTF-8, consulte la sección siguiente sobre [literales byte array](#). La función `encode_utf8` se proporciona para convertir datos entre las distintas codificaciones UTF-xx, principalmente para trabajar con datos y bibliotecas externas.

10.4 Concatenación

Una de las operaciones de cadena más comunes y útiles es la concatenación:

Julia también proporciona el operador `⋈` para concatenar cadenas:

Aunque `⋈` puede parecer una elección sorprendente a los usuarios de lenguajes que proporcionan `+` para concatenación de cadenas, este uso de `⋈` tiene precedentes en matemáticas, particularmente en álgebra abstracta.

En matemáticas, `⋈` suele denotar una operación *conmutativa*, donde el orden de los operandos no importa. Un ejemplo de esto es la suma de matrices, donde `⋈` para dos matrices cualesquiera `A` y `B` que tengan la misma forma. En contraste, `⋈` suele denotar una operación no conmutativa, donde el orden de los operandos *importa*. Un ejemplo de esto es la multiplicación de matrices donde, en general, `A ⋈ B` no es igual a `B ⋈ A`. Como con la multiplicación de matrices, la concatenación es no conmutativa: `"a" ⋈ "b"` no es igual a `"b" ⋈ "a"`. Por tanto, `⋈` es una elección más natural para el operador infijo de concatenación, consistente con el uso matemático común.

Más precisamente, el conjunto de todas las cadenas `S` de longitud finita junto con el operador de concatenación forman un [monoide libre](#) $(S, ⋈)$. El elemento identidad de este conjunto es la cadena vacía `""`. Siempre que un monoide libre es no conmutativo, la operación suele ser representada por `⋈`, `⋈`, o un símbolo similar, en lugar de `+` que implica conmutatividad.

10.5 Interpolación

Construir cadenas mediante concatenación puede llegar a ser un poco engorroso, sin embargo. Para reducir la necesidad de estas llamadas verbosas a `⋈` o multiplicaciones repetidas, Julia permite la interpolación en literales de cadena usando `@`, como en Perl:

Esto es más legible y conveniente, y equivalente a la concatenación de cadena anterior – el sistema rescribe este aparente literal de cadena única en una concatenación de literales de cadena con variables.

La expresión completa más corta después de `se` toma como la expresión cuyo valor debe ser interpolado en la cadena. Por lo tanto, puede interpolar cualquier expresión en una cadena usando paréntesis:

```
|
```

Tanto la concatenación como la interpolación de cadena llaman a `toString()` para convertir objetos al formato de cadena. La mayoría de los objetos que no son `String` se convierten en cadenas que se corresponden estrechamente con la forma en que se introducen como expresiones literales:

```
|
```

`String` es la identidad para los valores `String` y `values`, por lo que estos se interpolan en cadenas como ellos mismos, sin entrecomillar y sin escapar:

```
|
```

Para incluir un literal `String` en una cadena, lo escaparemos con un backslash:

```
|
```

10.6 Literales cadena con triples comillas

Cuando las cadenas se crean utilizando comillas triples (`"""`) tienen un comportamiento especial que puede ser útil para crear bloques de texto más largos. En primer lugar, si la apertura `"""` es seguida por una nueva línea, la nueva línea se quita de la cadena resultante:

```
|
```

es equivalente a

```
|
```

pero

contendrá un literal *new line* al principio. Los espacios en blanco no se modifican. Pueden contener símbolos sin escapar. Las cadenas de triple comilla también se dedican al nivel de la línea menos indentada. Esto es útil para definir cadenas dentro del código que está sangrado Por ejemplo:

En este caso la línea final (vacía) antes del cierre establece el nivel de indentación.

Tenga en cuenta que los saltos de línea en cadenas literales, sean de una sola o triple comilla, resultan en un carácter de línea nueva (LF) en la cadena, incluso si su editor usa una combinación de retorno de carro (CR) o CRLF para finalizar líneas. Para incluir un CR en una cadena, utilice un escape explícito ; Por ejemplo, puede introducir la cadena literal .

10.7 Operaciones Comunes

Podemos comparar cadenas lexicográficamente usando los operadores de comparación estándar:

La función `strchr` permite buscar el índice de un carácter en una cadena:

Y se puede arrancar la búsqueda de un carácter a partir de un desplazamiento proporcionado por un tercer argumento:

La función `in` se usa para comprobar si una subcadena está contenida en una cadena:

Este último error es debido a que `'o'` es un literal carácter, y `in` es una función genérica que busca subsecuencias. Para buscar un elemento en una secuencia, debemos usar la función `find` en lugar de la anterior.

y `findall` son otras dos funciones de cadena muy útiles:

Algunas otras funciones útiles son:

- el índice máximo (byte) que se puede utilizar para indexar en `str`.
- el número de caracteres en `str`.

* `find` da el primer índice válido en el que se puede encontrar un carácter en `'str'` (típicamente 1).

- devuelve el carácter siguiente en `str` o después del índice `index` y el siguiente índice de carácter válido que sigue a éste. Con `find` y `findall`, se puede utilizar para iterar a través de los caracteres en `str`.
- da el número de caracteres en `str` hasta e incluyendo cualquiera en el índice `index`.
- da el índice en el cual ocurre el carácter `char`-ésimo en `str`.

10.8 Literales cadena no estándar

Hay situaciones en las que se desea construir una cadena o utilizar semántica de cadenas, pero el comportamiento de la construcción de cadena estándar no es lo que se necesita. Para este tipo de situaciones, Julia proporciona [literales cadena no estándar](#). Un literal de cadena no estándar es como una cadena literal normal de doble comilla, pero va inmediatamente precedido de un identificador y no se comporta como un literal de cadena normal. El convenio es que los literales no estándar con prefijos en mayúsculas producen objetos cadena reales, mientras que aquellos con prefijos en minúsculas producen objetos que no cadena, como arrays de bytes o expresiones regulares compiladas. Las expresiones regulares, literales arrays de bytes y literales de números de versión, como se describe a continuación, son algunos ejemplos de literales de cadena no estándar. Otros ejemplos se dan en la sección [Metaprogramación](#).

10.9 Expresiones Regulares

Julia tiene expresiones regulares compatibles con Perl (expresiones regulares), tal y como las proporciona la biblioteca [PCRE](#). Las expresiones regulares se relacionan con las cadenas de dos maneras: la conexión obvia es que las expresiones regulares se utilizan para encontrar patrones regulares en cadenas; La otra conexión es que las expresiones regulares se introducen ellas mismas como cadenas, que se analizan en una máquina de estado que puede utilizarse para buscar patrones en cadenas de forma eficiente. En Julia, las expresiones regulares se introducen usando literales de cadena no estándar prefijados con varios identificadores comenzando por `r`. El literal de expresión regular más básico sin ninguna opción activada sólo utiliza :

```
|
```

Para comprobar si una *regex* se corresponde con una cadena, se utiliza :

```
|
```

Como puede verse aquí, simplemente devuelve `0` o `1`, indicando si la *regex* dada coincide o no con la cadena. Es común, sin embargo, que uno quiera saber no sólo si una cadena coincide, sino también *cómo* coincide. Para capturar esta información sobre una coincidencia, se utiliza la función :

```
|
```

Si la expresión regular no coincide con la cadena dada, devuelve `nothing` – un valor especial que no imprime nada en el indicador interactivo. Aparte de no imprimir, es un valor completamente normal, como podemos comprobar en el siguiente código:

```
|
```

Si la expresión regular coincide, el valor devuelto por es un objeto . Estos objetos registran cómo coincide la expresión, incluyendo la subcadena que coincide con el patrón y cualquier subcadena capturada, si la hay. Este ejemplo sólo captura la parte de la subcadena que coincide, pero tal vez quisiéramos capturar cualquier texto no en blanco después del carácter de comentario. Podríamos hacer lo siguiente:

Al invocar a , tenemos la opción de especificar un índice en el que iniciar la búsqueda. Por ejemplo:

Puede extraer la siguiente información de un objeto :

- La totalidad de la subcadena emparejada:
- Las subcadenas capturadas como una matriz de cadenas:
- El desplazamiento en el que comienza la coincidencia del patrón:
- Los desplazamientos de las subcadenas capturadas como un vector:

Para cuando una captura no coincide, en lugar de una subcadena, no contiene nada en esa posición, y tiene un desplazamiento de cero (recuerde que los índices en Julia son *1-based*, por lo que un desplazamiento de cero en una cadena es inválido). Aquí hay un par de ejemplos algo artificiales:

Es conveniente que las capturas sean retornadas como un array para que uno pueda usar la sintaxis de desestructurante para enlazarlas a variables locales:

Las capturas también están accesibles indexando el objeto con el número o nombre del grupo captura:

Las capturas pueden referenciarse en una cadena de sustitución cuando se utiliza `$` utilizando `$<groupname>` para referirse al grupo de captura `-ésimo` y prefijando la cadena de sustitución con `<groupname>`. El grupo de captura `0` se refiere a todo el objeto de coincidencia. Los grupos de captura nombrados se pueden hacer referencia en la sustitución con `<groupname>`. Por ejemplo:

Los grupos de captura numerados pueden también ser referenciados como `$<number>` para evitar ambigüedad, como en:

Puedes modificar el comportamiento de las expresiones regulares mediante una combinación de los flags `.`, `s`, y `m` después de la marca de comillas dobles de cierre. Estas banderas tienen el mismo significado que en Perl, tal y como se describe en este fragmento de la [página de manual del referencia de Perl](http://perldoc.perl.org/perlre.html#Modifiers):



Por ejemplo, la siguiente regex tiene activados los tres *flags*:



Las cadenas *regex* con triples comillas, de la forma `'''...'''` están también soportadas (y puede ser conveniente para expresiones regulares que contengan comillas o caracteres de salto de línea).

10.10 Byte Array Literals

Otro literal de cadena no estándar útil es el literal de cadena de bytes: `b'...'`. Esta forma nos permite usar la notación de cadena para expresar arrays de bytes literales, es decir, arrays de valores `0..255`. Las reglas para los literales de arrays de bytes son las siguientes:

- Los caracteres ASCII y los escapes ASCII producen un solo byte.
- y las secuencias de escape octales producen el *byte* correspondiente al valor de escape.
- Las secuencias de escape Unicode producen una secuencia de bytes que codifican ese punto de código en UTF-8.

Hay una cierta superposición entre estas reglas ya que el comportamiento de `\x` y escapes octales menores de 128 están cubiertos por las dos primeras reglas, pero aquí estas reglas están de acuerdo. Juntas, estas reglas permiten usar fácilmente caracteres ASCII, valores arbitrarios de bytes y secuencias UTF-8 para producir matrices de bytes. Aquí hay un ejemplo usando los tres:

```

// Producing a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences

```

La secuencia ASCII "DATA" corresponde a los bytes 68, 65, 84, 65. `\x55` produce el byte simple 255. El escape Unicode está codificado en UTF-8 como los tres bytes 226, 136, 128. Nótese que la matriz de bytes resultante no corresponde a una cadena UTF-8 válida - si intenta utilizar esto como una cadena literal normal, obtendrá un error de sintaxis:

```

// Producing a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences

```

Observe también la distinción significativa entre `\x` y `\u`: la secuencia de escape anterior codifica el *byte* 255, mientras que la última secuencia de escape representa el *punto de código* 255, que se codifica como dos bytes en UTF-8:

```

// Producing a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences
// The following code produces a matrix of bytes
// using the three rules
// 1. ASCII characters
// 2. Arbitrary bytes
// 3. UTF-8 sequences

```

En los literales de caracteres, esta distinción se pasa por alto y `\u` está autorizado a representar el punto de código 255, porque los caracteres *siempre* representan puntos de código. En las cadenas, sin embargo, los escapes `\x` siempre representan bytes, no puntos de código, mientras que los escapes `\u` siempre representan puntos de código, que están codificados en uno o más bytes. Para los puntos de código inferiores a 256, ocurre que la codificación UTF-8 de cada punto de código es sólo el byte producido por el escape correspondiente, por lo que la distinción puede ignorarse con seguridad. Sin embargo, para los escapes `\u` en comparación con `\x`, existe una diferencia importante: el primero escapa a todos los bytes sencillos de codificación, los cuales -a menos que sean seguidos por bytes de continuación muy específicos- no forman UTF-8 válido, mientras que los últimos escapes representan puntos de código Unicode con codificaciones de dos bytes.

Si todo esto es muy confuso, intente leer ["The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets"](#). Es una excelente introducción a Unicode y UTF-8, y puede ayudar a aliviar cierta confusión sobre el asunto.

10.11 Literales Número de Versión

Los números de versión se pueden expresar fácilmente con literales de cadena no estándar del forma `\"`. Los literales de número de versión crean objetos `Version` que siguen las especificaciones del [control de versiones semánticas](#) y, por lo tanto, se componen de valores numéricos mayor, menor y de parche, seguidos de anotaciones alfanuméricas de pre-liberación y construcción. Por ejemplo, se divide en versión principal, versión secundaria, versión de revisión, de pre-lanzamiento y construcción. Al introducir una versión literal, todo excepto el número de versión principal es opcional, por ejemplo, `\"` es equivalente a `\"` (con anotaciones previas / de compilación vacías), equivale a `\"`, y así sucesivamente.

Los objetos `Version` son en su mayoría útiles para comparar fácilmente y correctamente dos (o más) versiones. Por ejemplo, la constante `\"` contiene el número de versión de Julia como un objeto `Version` y, por lo tanto, se puede definir algún comportamiento específico de la versión utilizando declaraciones simples como:

```
|
```

Obsérvese que en el ejemplo anterior se utiliza el número de versión no estándar `\"`, con un guión en cola: esta notación es una extensión Julia del estándar, y se usa para indicar una versión que es más baja que cualquier versión `\"`, incluyendo todas sus pre-lanzamientos. Por lo tanto, en el ejemplo anterior, el código sólo se ejecuta con versiones estable y excluye las versiones `\"`. Para permitir también versiones inestables (es decir, pre-liberación), la verificación del límite inferior debería modificarse de la siguiente manera: `\"`.

Otra extensión de especificación de versión no estándar permite usar un `\"` de cola para expresar un límite superior en las versiones de compilación, por ej. `\"` se puede utilizar para significar cualquier versión por encima de `\"` y cualquiera de sus compilaciones: devolverá `\"` para la versión `\"` y para `\"`.

Es una buena práctica utilizar estas versiones especiales en comparaciones (en particular, el valor de cola siempre debe utilizarse en los límites superiores a menos que haya una buena razón para no hacerlo), pero no deben utilizarse como el número de versión real de nada, ya que son inválidos en el esquema de versiones semánticas.

Además de ser utilizados por la constante `\"`, los objetos `Version` son ampliamente utilizados en el módulo `\"`, para especificar las versiones de paquetes y sus dependencias.

10.12 Raw String Literals

Las cadenas en bruto (*raw*) sin interpolación o *unescapeing* pueden ser expresadas con literales cadena no estándar de la forma `\"`. Los literales cadena en bruto crean objetos `RawString` ordinarios que contienen los contenidos encerrados exactamente como entrados sin interpolación ni separación. Esto es útil para cadenas que contiene código o marcado en otros idiomas que usan `\"` o como caracteres especiales. La excepción son las comillas que aún deben ser escapadas, por ejemplo, es equivalente a `\"`.

Chapter 11

Functions

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:

```
|
```

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":

```
|
```

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 short function syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

```
|
```

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

```
|
```

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

```
|
```

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 `s`) 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.

11.2 The Keyword

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`. 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:

```
|
```

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

```
|
```

Of course, in a purely linear function body like `f`, the usage of `return` is pointless since the expression `x` is never evaluated and we could simply make `x` 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 `a` and `b`, avoiding overflow:

```
|
```

```

function f(x, y)
    if x < 0
        return -x - y
    elseif x > 0
        return x - y
    else
        return x + y
end

f(-1, 2)
f(1, 2)
f(0, 2)

```

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:

```

x = 1; y = 2
x && y
(x && y)

```

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:

```

f = &&
f(1, 2)

```

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

Expression	Calls

11.4 Operators With Special Names

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

These functions are included in the `Base` module even though they do not have operator-like names.

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:

```
|
```

This creates a function taking one argument and returning the value of the polynomial 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:

```
|
```

This is fine if a named function effecting the transform one wants 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:

```
|
```

An anonymous function accepting multiple arguments can be written using the syntax `function()::T = ...`. A zero-argument anonymous function is written as `()::T = ...`. 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 `()`.

11.6 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:

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

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

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

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

11.7 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:

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

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

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 "splice" the values contained in an iterable collection into a function call as individual arguments. To do this, one also uses `*` but in the function call instead:

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:

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

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

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

11.8 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 `parseInt` interprets a string as a number in some base. The `base` argument defaults to `10`. This behavior can be expressed concisely as:

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

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](#)).

11.9 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 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, color='red')`, 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:

```
|
```

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

Keyword argument default values are evaluated only when necessary (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:

```
|
```

Extra keyword arguments can be collected using `**kwargs`, as in varargs functions:

```
|
```

Inside `kwargs`, will be a collection of tuples, where each `(key, value)` is a symbol. Such collections can be passed as keyword arguments using a semicolon in a call, e.g. `plot(x, y, width=2; kwargs)`. Dictionaries can also be used for this purpose.

One can also pass 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; color='red')` and `plot(x, y, width=2; ('color', 'red'))` are equivalent to `plot(x, y, width=2; color='red')`. 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, width=2, color='red', width=4)` it is possible that the `kwargs` 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 `4`.

11.10 Evaluation Scope of Default Values

Optional and keyword arguments differ slightly in how their default values are evaluated. When optional argument default expressions are evaluated, only *previous* arguments are in scope. In contrast, *all* the arguments are in scope when keyword arguments default expressions are evaluated. For example, given this definition:

```
|
```

the `in` refers to a `in` in an outer scope, not the subsequent argument `.`. However, if `and` were keyword arguments instead, then both would be created in the same scope and the `in` would refer to the subsequent argument (shadowing any `in` in an outer scope), which would result in an undefined variable error (since the default expressions are evaluated left-to-right, and `has` has not been assigned yet).

11.11 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 `on` a function with several cases:

```
|
```

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

```
|
```

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

How these arguments are initialized depends on the "outer" function; here, `will` sequentially set `to` `,` `,` `,` calling the anonymous function on each, just as would happen in the syntax `.`

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 `,` such as managing system state. For example, there is a version of `that` runs code ensuring that the opened file is eventually closed:

This is accomplished by the following definition:

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

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

11.12 Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have "vectorized" versions of functions, which simply apply a given function to each element of an array to yield a new array via `.`. 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 can be applied elementwise to any array (or other collection) with the syntax `array. f`. For example `sin.` can be applied to all elements in the vector `x`, like so:

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

More generally, `f.(x, y, z)` is actually equivalent to `f(x, y, z)`, 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 `x`, `y`, and `z`, then `f.(x, y, z)` will return a new array consisting of `f(x[i], y[i], z[i])` for each `i` in `1:length(x)`, and `f.(x, y, z)` will return a new vector consisting of `f(x[i], y[i], z[i])` for each index `i` (throwing an exception if the vectors have different length).

Moreover, *nested* calls are *fused* into a single loop. For example, `for i in 1:n, for j in 1:m, A[i,j] = B[i,j] + C[i,j]` is equivalent to `for i in 1:n, for j in 1:m, A[i,j] = B[i,j] + C[i,j]`, similar to `for i in 1:n, for j in 1:m, A[i,j] = B[i,j] + C[i,j]`: there is only a single loop over `j`, and a single array is allocated for the result. [In contrast, in a typical "vectorized" language would first allocate one temporary array for `B[i,j]`, and then compute `A[i,j]` 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 calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in the `for` and `while` loops cannot be merged because of the intervening `function` 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 ([Pre-allocating outputs](#)). A convenient syntax for this is `zeros(n,m)`, which is equivalent to `zeros{m}(n)` except that, as above, the `zeros` loop is fused with any nested "dot" calls. For example, `zeros(n,m)` is equivalent to `zeros{m}(n)`, overwriting `zeros` with `zeros` in-place. If the left-hand side is an array-indexing expression, e.g. `A[i,j]`, then it translates to `A[i,j]` on `A`, e.g. `A[i,j]`, so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can be tedious and lead to code that is difficult to read, the macro `@dot` is provided to convert every function call, operation, and assignment in an expression into the "dotted" version.

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

11.13 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

Julia provides a variety of control flow constructs:

- [Compound Expressions](#): `and` and `.`
- [Conditional Evaluation](#): `--` and `?:` (ternary operator).
- [Short-Circuit Evaluation](#): `&&`, `&`, and chained comparisons.
- [Repeated Evaluation](#): `Loops`: `while` and `for`.
- [Exception Handling](#): `try`, `catch`, and `finally`.
- [Tasks \(aka Coroutines\)](#): `task`.

The first five control flow mechanisms are standard to high-level programming languages. `tasks` are not so standard: they provide non-local control flow, making it possible to switch between temporarily-suspended computations. This is a powerful construct: both exception handling and cooperative multitasking are implemented in Julia using tasks. Everyday programming requires no direct usage of tasks, but certain problems can be solved much more easily by using tasks.

12.1 Compound Expressions

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: `blocks` and `chains`. The value of both compound expression constructs is that of the last subexpression. Here's an example of a `block`:

```
block{
    x = 1
    y = 2
    z = 3
    return x + y + z
}
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the `chain` syntax comes in handy:

This syntax is particularly useful with the terse single-line function definition form introduced in [Functions](#). Although it is typical, there is no requirement that blocks be multiline or that chains be single-line:

12.2 Conditional Evaluation

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the `--` conditional syntax:

If the condition expression `is` , then the corresponding block is evaluated; otherwise the condition expression `is` evaluated, and if it is , the corresponding block is evaluated; if neither expression is true, the block is evaluated. Here it is in action:

The `and` and `or` blocks are optional, and as many blocks as desired can be used. The condition expressions in the `--` construct are evaluated until the first one evaluates to `false`, after which the associated block is evaluated, and no further condition expressions or blocks are evaluated.

`do` blocks are "leaky", i.e. they do not introduce a local scope. This means that new variables defined inside the `do` clauses can be used after the `do` block, even if they weren't defined before. So, we could have defined the `is_prime` function above as

The variable `is_prime` is declared inside the `do` block, but used outside. However, when depending on this behavior, make sure all possible code paths define a value for the variable. The following change to the above function results in a runtime error

blocks also return a value, which may seem unintuitive to users coming from many other languages. This value is simply the return value of the last executed statement in the branch that was chosen, so

Note that very short conditional statements (one-liners) are frequently expressed using Short-Circuit Evaluation in Julia, as outlined in the next section.

Unlike C, MATLAB, Perl, Python, and Ruby – but like Java, and a few other stricter, typed languages – it is an error if the value of a conditional expression is anything but `true` or `false`:

This error indicates that the conditional was of the wrong type: `Bool` rather than the required `Boolean`.

The so-called "ternary operator", `?:`, is closely related to the `if` syntax, but is used where a conditional choice between single expression values is required, as opposed to conditional execution of longer blocks of code. It gets its name from being the only operator in most languages taking three operands:

The expression , before the , is a condition expression, and the ternary operation evaluates the expression , before the , if the condition is or the expression , after the , if it is .

The easiest way to understand this behavior is to see an example. In the previous example, the call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```
|
```

If the expression is true, the entire ternary operator expression evaluates to the string and otherwise it evaluates to the string . The original three-way example requires chaining multiple uses of the ternary operator together:

```
|
```

To facilitate chaining, the operator associates from right to left.

It is significant that like --, the expressions before and after the are only evaluated if the condition expression evaluates to or , respectively:

```
|
```

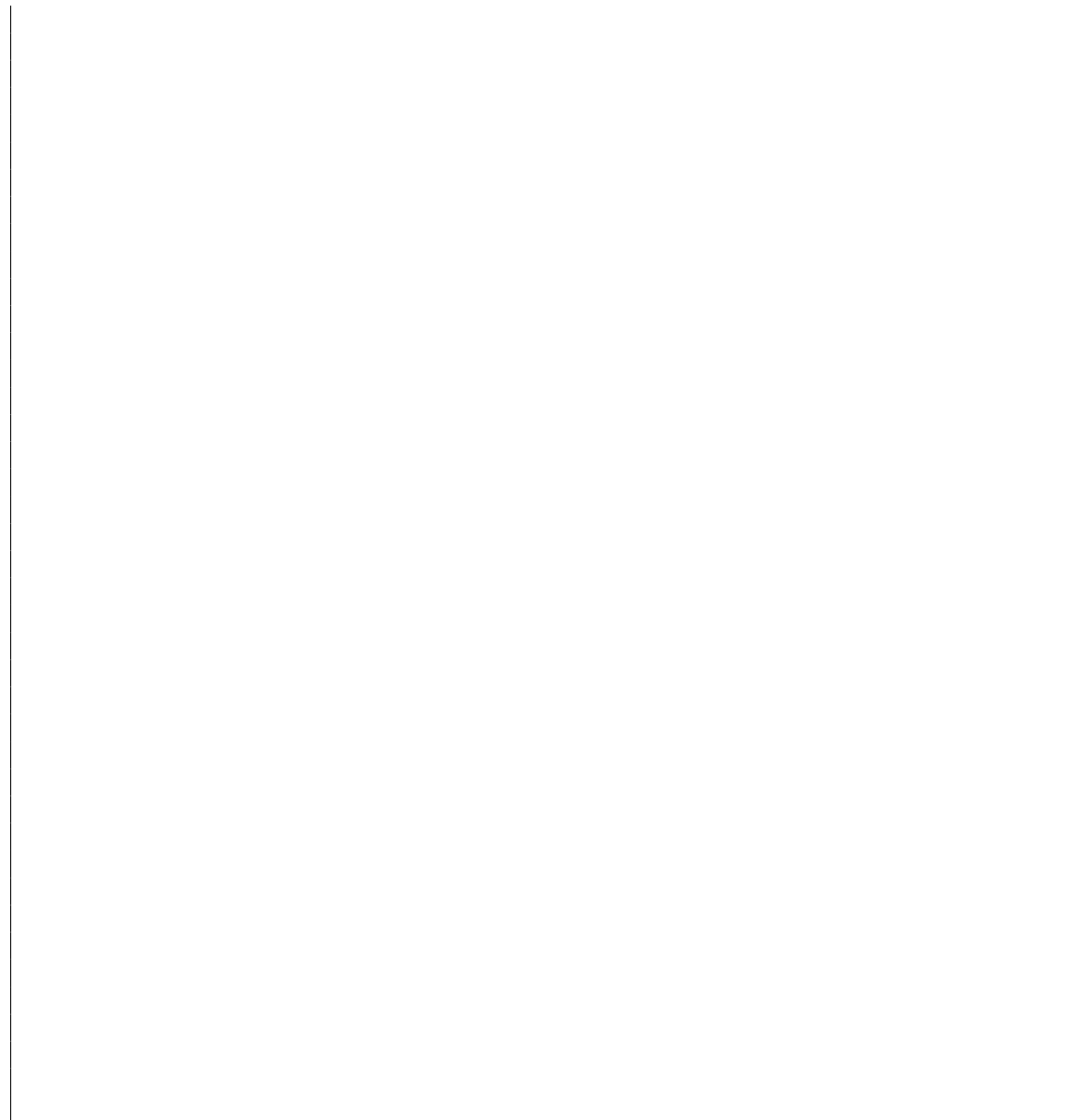
12.3 Short-Circuit Evaluation

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the and boolean operators: in a series of boolean expressions connected by these operators,

only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

- In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`.
- In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`.

The reasoning is that `a && b` must be `false` if `a` is `false`, regardless of the value of `b`, and likewise, the value of `a || b` must be `true` if `a` is `true`, regardless of the value of `b`. Both `&&` and `||` associate to the right, but `&&` has higher precedence than `||` does. It's easy to experiment with this behavior:



You can easily experiment in the same way with the associativity and precedence of various combinations of `&` and `||` operators.

This behavior is frequently used in Julia to form an alternative to very short `if` statements. Instead of `if <cond>`, one can write `<cond> && <statement>` (which could be read as: `<cond>` *and then* `<statement>`). Similarly, instead of `if <cond> ... else ... end`, one can write `<cond> || <statement>` (which could be read as: `<cond>` *or else* `<statement>`).

For example, a recursive factorial routine could be defined like this:

```
function factorial(n)
    n < 1 && return 1
    n * factorial(n - 1)
end
```

Boolean operations *without* short-circuit evaluation can be done with the bitwise boolean operators introduced in [Mathematical Operations and Elementary Functions](#): `&` and `|`. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

```
&(1 < 2, 2 < 3) # returns false
|(1 < 2, 2 < 3) # returns true
```

Just like condition expressions used in `if`, or the ternary operator, the operands of `&` or `|` must be boolean values (`true` or `false`). Using a non-boolean value anywhere except for the last entry in a conditional chain is an error:

```
&(1 < 2, 2 < 3, "error") # throws an error
```

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

```
if 1 < 2
    "true"
else
    "false"
end # returns "true"
```

12.4 Repeated Evaluation: Loops

There are two constructs for repeated evaluation of expressions: the `while` loop and the `do-while` loop. Here is an example of a `while` loop:

```
while (condition) {
    // body
}
```

The `while` loop evaluates the condition expression (in this case), and as long it remains `true`, keeps also evaluating the body of the `while` loop. If the condition expression is `false` when the `while` loop is first reached, the body is never evaluated.

The `do-while` loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above `while` loop does is so common, it can be expressed more concisely with a `do-while` loop:

```
do {
    // body
} while (condition);
```

Here the `range` is a `Range` object, representing the sequence of numbers 1, 2, 3, 4, 5. The `for` loop iterates through these values, assigning each one in turn to the variable `i`. One rather important distinction between the previous `while` loop form and the `for` loop form is the scope during which the variable is visible. If the variable `i` has not been introduced in an other scope, in the `for` loop form, it is visible only inside of the `for` loop, and not afterwards. You'll either need a new interactive session instance or a different variable name to test this:

```
for (int i = 1; i <= 5; i++) {
    // body
}
```

See [Scope of Variables](#) for a detailed explanation of variable scope and how it works in Julia.

In general, the `for` loop construct can iterate over any container. In these cases, the alternative (but fully equivalent) keyword `in` is typically used instead of `:`, since it makes the code read more clearly:

Various types of iterable containers will be introduced and discussed in later sections of the manual (see, e.g., [Multi-dimensional Arrays](#)).

It is sometimes convenient to terminate the repetition of a `for` loop before the test condition is falsified or stop iterating in a loop before the end of the iterable object is reached. This can be accomplished with the `break` keyword:

```
for i in range(1000):
    for j in range(1000):
        if j > 500:
            continue
        if i > 500:
            break
        # do something
```

Without the `break` keyword, the above loop would never terminate on its own, and the loop would iterate up to 1000. These loops are both exited early by using `break`.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The `continue` keyword accomplishes this:

```
for i in range(1000):
    for j in range(1000):
        if j > 500:
            continue
        # do something
```

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the `continue` call inside the `if` block. In realistic usage there is more code to be evaluated after the `if`, and often there are multiple points from which one calls `continue`.

Multiple nested loops can be combined into a single outer loop, forming the cartesian product of its iterables:

```
for i, j in zip(range(1000), range(1000)):
    # do something
```

A `break` statement inside such a loop exits the entire nest of loops, not just the inner one.

The function

Exceptions can be created explicitly with `ValueError`. For example, a function defined only for nonnegative numbers could be written to `raise ValueError` if the argument is negative:

```
def sqrt(x):
    if x < 0:
        raise ValueError('sqrt() argument must be non-negative')
```

Note that `ValueError` without parentheses is not an exception, but a type of exception. It needs to be called to obtain an exception object:

```
try:
    sqrt(-1)
except ValueError:
    pass
```

Additionally, some exception types take one or more arguments that are used for error reporting:

```
raise ValueError('sqrt() argument must be non-negative', x)
```

This mechanism can be implemented easily by custom exception types following the way `ValueError` is written:

```
class ValueError(Exception):
    """ValueError exception base class"""
    def __init__(self, message, arg=None):
        self.message = message
        self.arg = arg
    def __str__(self):
        return self.message
```

Note

When writing an error message, it is preferred to make the first word lowercase. For example, `ValueError: sqrt() argument must be non-negative` is preferred over `ValueError: Sqrt() argument must be non-negative`.

However, sometimes it makes sense to keep the uppercase first letter, for instance if an argument to a function is a capital letter: `ValueError: Invalid argument: A`.

Errors

The `error` function is used to produce an `ErrorException` that interrupts the normal flow of control.

Suppose we want to stop execution immediately if the square root of a negative number is taken. To do this, we can define a fussy version of the `sqrt` function that raises an error if its argument is negative:

```
function fussy_sqrt(x)
    if x < 0
        error("square root of negative number")
    end
    sqrt(x)
end
```

If `fussy_sqrt` is called with a negative value from another function, instead of trying to continue execution of the calling function, it returns immediately, displaying the error message in the interactive session:

```
julia> fussy_sqrt(-1)
ERROR: ArgumentError: square root of negative number
Stacktrace:
 [1] fussy_sqrt(x::Float64) at ./REPL[1]:1
 [2] <unknown> at ./REPL[1]:1
```

Warnings and informational messages

Julia also provides other functions that write messages to the standard error I/O, but do not throw any `Exceptions` and hence do not interrupt execution:

```
function warn(msg)
    @warn msg
end

function info(msg)
    @info msg
end
```

The statement

The statement allows for `s` to be tested for. For example, a customized square root function can be written to automatically call either the real or complex square root method on demand using `s` :

```
def sqrt(s):
    """Return the square root of s.
    If s is a string, return the complex square root.
    If s is a number, return the real square root.
    """
    if isinstance(s, str):
        return cmath.sqrt(s)
    else:
        return math.sqrt(s)
```

It is important to note that in real code computing this function, one would compare `s` to zero instead of catching an exception. The exception is much slower than simply comparing and branching.

statements also allow the result to be saved in a variable. In this contrived example, the following example calculates the square root of the second element of `l` if `l` is indexable, otherwise assumes `s` is a real number and returns its square root:

```
def sqrt2(l):
    """Return the square root of the second element of l.
    If l is a list, return the square root of the second element.
    If l is a number, return the real square root.
    """
    if isinstance(l, list):
        return sqrt(l[2])
    else:
        return sqrt(l)
```

Note that the symbol following `try` will always be interpreted as a name for the exception, so care is needed when writing expressions on a single line. The following code will *not* work to return the value of `x` in case of an error:

Instead, use a semicolon or insert a line break after `try`:

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. There are situations where no error has occurred, but the ability to unwind the stack and pass a value to a higher level is desirable. Julia provides the `try-catch`, `try-maybe`, and `try-error` functions for more advanced error handling.

Clauses

In code that performs state changes or uses resources like files, there is typically clean-up work (such as closing files) that needs to be done when the code is finished. Exceptions potentially complicate this task, since they can cause a block of code to exit before reaching its normal end. The `finally` keyword 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:

When control leaves the `try` block (for example due to a `return`, or just finishing normally), `finally` will be executed. If the `try` block exits due to an exception, the exception will continue propagating. A `try-catch` block may be combined with `finally` and `try-maybe` as well. In this case the `finally` block will run after `catch` has handled the error.

12.6 Tasks (aka Coroutines)

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. This feature is sometimes called by other names, such as symmetric coroutines, lightweight threads, cooperative multitasking, or one-shot continuations.

When a piece of computing work (in practice, executing a particular function) is designated as a `task`, it becomes possible to interrupt it by switching to another `task`. The original `task` can later be resumed, at which point it will pick up right where it left off. At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function.

This kind of control flow can make it much easier to solve certain problems. In some problems, the various pieces of required work are not naturally related by function calls; there is no obvious "caller" or "callee" among the jobs that need to be done. An example is the producer-consumer problem, where one complex procedure is generating values and another complex procedure is consuming them. The consumer cannot simply call a producer function to get a value, because the producer may have more values to generate and so might not yet be ready to return. With tasks, the producer and consumer can both run as long as they need to, passing values back and forth as necessary.

Julia provides a `Task` mechanism for solving this problem. A `Task` is a waitable first-in first-out queue which can have multiple tasks reading from and writing to it.

Let's define a producer task, which produces values via the `put!` call. To consume values, we need to schedule the producer to run in a new task. A special `Task` constructor which accepts a 1-arg function as an argument can be used to run a task bound to a channel. We can then `take` values repeatedly from the channel object:

One way to think of this behavior is that `put!` was able to return multiple times. Between calls to `take`, the producer's execution is suspended and the consumer has control.

The returned `Channel` can be used as an iterable object in a `for` loop, in which case the loop variable takes on all the produced values. The loop is terminated when the channel is closed.

Note that we did not have to explicitly close the channel in the producer. This is because the act of binding a `Task` to a `Channel` associates the open lifetime of a channel with that of the bound task. The channel object is closed automatically when the task terminates. Multiple channels can be bound to a task, and vice-versa.

While the `Channel` constructor expects a 0-argument function, the `bind` method which creates a channel bound task expects a function that accepts a single argument of type `Channel`. A common pattern is for the producer to be parameterized, in which case a partial function application is needed to create a 0 or 1 argument [anonymous function](#).

For `Channel` objects this can be done either directly or by use of a convenience macro:

To orchestrate more advanced work distribution patterns, `Channel` can be used in conjunction with `Task` and `Channel` constructors to explicitly link a set of channels with a set of producer/consumer tasks.

Note that currently Julia tasks are not scheduled to run on separate CPU cores. True kernel threads are discussed under the topic of [Parallel Computing](#).

Core task operations

Let us explore the low level construct `yield` to understand how task switching works. `yield` suspends the current task, switches to the specified `Channel`, and causes that task's last `yield` call to return the specified `Channel`. Notice that `yield` is the only operation required to use task-style control flow; instead of calling and returning we are always just switching to a different task. This is why this feature is also called "symmetric coroutines"; each task is switched to and from using the same mechanism.

`yield` is powerful, but most uses of tasks do not invoke it directly. Consider why this might be. If you switch away from the current task, you will probably want to switch back to it at some point, but knowing when to switch back, and knowing which task has the responsibility of switching back, can require considerable coordination. For example, `wait` and `close` are blocking operations, which, when used in the context of channels maintain state to remember who the consumers are. Not needing to manually keep track of the consuming task is what makes `Task` easier to use than the low-level `yield`.

In addition to `yield`, a few other basic functions are needed to use tasks effectively.

- `current_task()` gets a reference to the currently-running task.
- `isdone(task)` queries whether a task has exited.
- `isrun(task)` queries whether a task has run yet.
- `setkeyvalue(task, key, value)` manipulates a key-value store specific to the current task.

Tasks and events

Most task switches occur as a result of waiting for events such as I/O requests, and are performed by a scheduler included in the standard library. The scheduler maintains a queue of runnable tasks, and executes an event loop that restarts tasks based on external events such as message arrival.

The basic function for waiting for an event is `wait`. Several objects implement `wait`; for example, given a `condition` object, `wait` will wait for it to exit. `wait` is often implicit; for example, a `wait` can happen inside a call to `wait` to wait for data to be available.

In all of these cases, `wait` ultimately operates on a `condition` object, which is in charge of queueing and restarting tasks. When a task calls `wait` on a `condition`, the task is marked as non-runnable, added to the condition's queue, and switches to the scheduler. The scheduler will then pick another task to run, or block waiting for external events. If all goes well, eventually an event handler will call `wait` on the condition, which causes tasks waiting for that condition to become runnable again.

A task created explicitly by calling `task_create` is initially not known to the scheduler. This allows you to manage tasks manually using `task_wait` if you wish. However, when such a task waits for an event, it still gets restarted automatically when the event happens, as you would expect. It is also possible to make the scheduler run a task whenever it can, without necessarily waiting for any events. This is done by calling `task_run`, or using the `task_run` or `task_wait` macros (see [Parallel Computing](#) for more details).

Task states

Tasks have a `state` field that describes their execution status. A `state` is one of the following symbols:

Symbol	Meaning
<code>task_running</code>	Currently running, or available to be switched to
<code>task_blocked</code>	Blocked waiting for a specific event
<code>task_waiting</code>	In the scheduler's run queue about to be restarted
<code>task_finished</code>	Successfully finished executing
<code>task_exception</code>	Finished with an uncaught exception

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 `without` without the two `'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:

Scope name	block/construct introducing this kind of scope
Global Scope	<code>,</code> <code>,</code> at interactive prompt (REPL)
Local Scope	Soft Local Scope: <code>,</code> <code>,</code> comprehensions, try-catch-finally,
Local Scope	Hard Local Scope: functions (either syntax, anonymous & do-blocks), <code>,</code>

Notably missing from this table are [begin blocks](#) and [if blocks](#), which do *not* introduce new scope blocks. All three types of scopes follow somewhat different rules which will be explained below as well as some extra rules for certain blocks.

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 `inside` refers to the `in` in the global scope of its module :

```
|
```

and not a `in` in the scope where `is` used:

```
|
```

Thus *lexical scope* means that the scope of variables can be inferred from the source code alone.

13.1 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.

Note that the interactive prompt (aka REPL) is in the global scope of the module .

13.2 Local Scope

A new local scope is introduced by most code-blocks, see above table for a complete list. A local scope *usually* inherits all the variables from its parent scope, both for reading and writing. There are two subtypes of local scopes, hard and soft, with slightly different rules concerning what variables are inherited. 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 both hard and soft local scopes. A newly introduced variable in a local scope does not back-propagate to its parent scope. For example, here the `is` is not introduced into the top-level scope:

```
|
```

(Note, in this and all following examples it is assumed that their top-level is a global scope with a clean workspace, for instance a newly started REPL.)

Inside a local scope a variable can be forced to be a local variable using the `local` keyword:

```
|
```

Inside a local scope a new global variable can be defined using the keyword `global`:

```
|
```

The location of both the `local` and `global` keywords within the scope block is irrelevant. The following is equivalent to the last example (although stylistically worse):

```
|
```

Soft Local Scope

In a soft local scope, all variables are inherited from its parent scope unless a variable is specifically marked with the keyword `global`.

Soft local scopes are introduced by for-loops, while-loops, comprehensions, try-catch-finally-blocks, and let-blocks. There are some extra rules for [Let Blocks](#) and for [For Loops and Comprehensions](#).

In the following example the `read` and `write` refer always to the same variables as the soft local scope inherits both read and write variables:

Within soft scopes, the *global* keyword is never necessary, although allowed. The only case when it would change the semantics is (currently) a syntax error:

Hard Local Scope

Hard local scopes are introduced by function definitions (in all their forms), struct type definition blocks, and macro-definitions.

In a hard local scope, all variables are inherited from its parent scope unless:

- an assignment would result in a modified *global* variable, or

- a variable is specifically marked with the keyword `global`.

Thus global variables are only inherited for reading but not for writing:

|

An explicit `global` is needed to assign to a global variable:

|

Note that *nested functions* can behave differently to functions defined in the global scope as they can modify their parent scope's *local* variables:

|

The distinction between inheriting global and local variables for assignment can lead to some slight differences between functions defined in local vs. global scopes. Consider the modification of the last example by moving to the global scope:

Note that above subtlety does 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:

This behavior may seem 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 provides built-in, efficient functions to test for oddness and evenness called `isodd` and `iseven` so the above definitions should only be taken as examples.

Hard vs. Soft Local Scope

Blocks which introduce a soft local scope, such as loops, are generally used to manipulate the variables in their parent scope. Thus their default is to fully access all variables in their parent scope.

Conversely, the code inside blocks which introduce a hard local scope (function, type, and macro definitions) can be executed at any place in a program. Remotely changing the state of global variables in other modules should be done with care and thus this is an opt-in feature requiring the `global` keyword.

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 `count` variable in the following example:

See also the closures in the examples in the next two sections.

Let Blocks

Unlike assignments to local variables, statements allocate new variable bindings each time they run. An assignment modifies an existing value location, and 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 syntax accepts a comma-separated series of assignments and variable names:

```
(let [x 1 y 2]
  (+ x y))
```

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 1] (let [x 2] ...))` since the two variables are distinct and have separate storage. Here is an example where the behavior of `let` is needed:

```
(let [x 1]
  (let [x 2]
    (+ x x)))
```

Here we create and store two closures that return variable `x`. However, it is always the same variable, so the two closures behave identically. We can use `let*` to create a new binding for `x`:

```
(let* [x 1]
  (let* [x 2]
    (+ x x)))
```


Since the `with` construct does not introduce a new scope, it can be useful to use a zero-argument `with` to just introduce a new scope block without creating any new bindings:

Since `with` introduces a new scope block, the inner `local` is a different variable than the outer `local`.

For Loops and Comprehensions

`for` loops and [Comprehensions](#) have the following behavior: any new variables introduced in their body scopes are freshly allocated for each loop iteration. This is in contrast to `for` loops which reuse the variables for all iterations. Therefore these constructs are similar to `for` loops with `with` blocks inside:

`for` loops will reuse existing variables for its iteration variable:

However, comprehensions do not do this, and always freshly allocate their iteration variables:

13.3 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 `final` keyword:

The `final` declaration is allowed on both global and local variables, but is especially useful for 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 `final` 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 for performance purposes.

Special top-level assignments, such as those performed by the `static` and `final` keywords, are constant by default.

Note that `final` 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 14

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 using types. When additional expressiveness is needed, however, it is easy to 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 which `isvalue` returns true (essentially, things like numbers and bools that 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.

14.1 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:

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 `let` 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 :

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:

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:

Returning from this function behaves just like an assignment to a variable with a declared type: the value is always converted to .

14.2 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: `Int`, `Float64`, `Float32`, `Complex{T}`, and `BigInt`. Although they have different representation sizes, `Int`, `Float64`, and `Complex{T}` all have in common that they are signed integer types. Likewise, `Float32`, `Float64`, and `Complex{T}` are all unsigned integer types, while `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:

The `abstract type` keyword introduces a new abstract type, whose name is given by `T`. This name can be optionally followed by `where S` 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 `Nothing`. It is the exact opposite of `Any`: no object is an instance of `Nothing` and all types are supertypes of `Nothing`.

Let's consider some of the abstract types that make up Julia's numerical hierarchy:

The `Real` type is a direct child type of `Number`, and `Integer` 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): `Float` and `Rational`, 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, `Float` is a proper subtype of `Real`, including only floating-point representations of real numbers. Integers are further subdivided into `Int` and `BigInt` 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:

```
|
```

An important use of abstract types is to provide default implementations for concrete types. To give a simple example, consider:

```
|
```

The first thing to note is that the above argument declarations are equivalent to `Int` and `Float`. When this function is invoked, say as `f(1, 2.0)`, the dispatcher chooses the most specific method named `f` 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 `f{Int, Float}` specifically for two arguments based on the generic function given above, i.e., it implicitly defines and compiles:

```
|
```

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](#).)

14.3 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 only a fixed set of built-in ones. In fact, the standard primitive types are all defined in the language itself:

The general syntaxes for declaring a primitive type are:

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{8}` above therefore means that a boolean value takes eight bits to store, and has `Any` as its immediate supertype. Currently, only sizes that are multiples of 8 bits are supported. Therefore, boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

The types `Bool{8}`, `Bool{16}`, and `Bool{32}` 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{8}`'s direct supertype is `Bool{16}`, `Bool{16}`'s is `Bool{32}`, and `Bool{32}`'s is `Any`. All other differences between `Bool{8}`, `Bool{16}`, and `Bool{32}` 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{8}` behave any differently than `Bool{16}` or `Bool{32}`.

14.4 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 to use by multiple 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:

Fields with no type annotation default to `Any`, and can accordingly hold any type of value.

New objects of type `T` are created by applying the `T` type object like a function to values for its fields:

When a type is applied like a function it is called a *constructor*. Two constructors are generated automatically (these are called *default constructors*). One accepts any arguments and calls `to` 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 `field` is unconstrained in type, any value will do. However, the value for `field` must be convertible to `T`:

You may find a list of field names using the `fields` function.

You can access the field values of a composite object using the traditional `obj.field` notation:

Composite objects declared with `case class` 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`, to be discussed in the next section.

Composite types with no fields are singletons; there can be only one instance of such types:

The function confirms that the "two" constructed instances of `are` 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](#).

14.5 Mutable Composite Types

If a composite type is declared with `mutable` instead of `final`, then instances of it can be modified:

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 with specific field values -- 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 `f` is a function argument of an abstract type, and suppose that the function changes a field: `f.f = 1`. Depending on whether `f` 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.

14.6 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 have explicitly declared supertypes.
- 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:

```
|
```

A `DataType` may be abstract or concrete. If it is concrete, it has a specified size, storage layout, and (optionally) field names. Thus a `bits` 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`.

14.7 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:

```
|
```

The compilers for many languages have an internal union construct for reasoning about types; Julia simply exposes it to the programmer.

14.8 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 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 parametric bits types.

Parametric Composite Types

Type parameters are introduced immediately after the type name, surrounded by curly braces:

This declaration defines a new parametric type, `Point{T1, T2}`, holding two "coordinates" of type `T1`. What, one may ask, is `T1`? 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, String}` is a concrete type equivalent to the type defined by replacing `T1` in the definition of `Point{T1, T2}` with `Float64`. Thus, this single declaration actually declares an unlimited number of types: `Point{Int}`, `Point{String}`, etc. Each of these is now a usable concrete type:

The type `Point{Float64, String}` is a point whose coordinates are 64-bit floating-point values, while the type `Point{String, String}` is a "point" whose "coordinates" are string objects (see [Strings](#)).

`Point{T1, T2}` itself is also a valid type object, containing all instances `Point{T1, T2}`, etc. as subtypes:

Other types, of course, are not subtypes of it:

Concrete types with different values of `isimmutable` are never subtypes of each other:

Warning

This last point is very important: even though we **DO NOT** have `isimmutable`.

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 `T{A,B}` may conceptually be like an instance of `T{A',B'}` as well, the two types have different representations in memory:

- An instance of `T{A,B}` can be represented compactly and efficiently as an immediate pair of 64-bit values;
- An instance of `T{A',B'}` must be able to hold any pair of instances of `A` and `B`. Since objects that are instances of `A` can be of arbitrary size and structure, in practice an instance of `T{A',B'}` must be represented as a pair of pointers to individually allocated objects.

The efficiency gained by being able to store objects with immediate values is magnified enormously in the case of arrays: an `Array{T{A,B}}` can be stored as a contiguous memory block of 64-bit floating-point values, whereas an `Array{T{A',B'}}` must be an array of pointers to individually allocated 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 `AbstractArray` type.

Since `T{A,B}` is not a subtype of `T{A',B'}`, the following method can't be applied to arguments of type `T{A',B'}`:

A correct way to define a method that accepts all arguments of type `T{A,B}` where `A` is a subtype of `A'` is:

(Equivalently, one could define `isimmutable` or ; see [UnionAll Types](#).)

More examples will be discussed later in [Methods](#).

How does one construct a `T{A,B}` 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 `T{A,B}` is a concrete type equivalent to `T{A,B}::Type` declared with `immutable` in place of `mutable`, it can be applied as a constructor accordingly:

For the default constructor, exactly one argument must be supplied for each field:

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 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 itself as a constructor, provided that the implied value of the parameter type is unambiguous:

In the case of , the type of is unambiguously implied if and only if the two arguments to have the same type. When this isn't the case, the constructor will fail with a :

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:

With this declaration, is a distinct abstract type for each type or integer value of . As with parametric composite types, each such instance is a subtype of :

Parametric abstract types are invariant, much as parametric composite types are:

The notation `Union{A, B}` can be used to express the Julia analogue of a *covariant* type, while `Union{A, B}` the analogue of a *contravariant* type, but technically these represent *sets* of types (see [UnionAll Types](#)).

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 `AbstractPoint` to be a subtype of `Abstract{A, B}` as follows:

Given such a declaration, for each choice of `A`, we have `AbstractPoint{A}` as a subtype of `Abstract{A, B}`:

This relationship is also invariant:

What purpose do parametric abstract types like `AbstractPoint` serve? Consider if we create a point-like implementation that only requires a single coordinate because the point is on the diagonal line $x = y$:

Now both `AbstractPoint{A}` and `AbstractPoint{B}` are implementations of the `AbstractPoint` abstraction, and similarly for every other possible choice of type `A`. This allows programming to a common interface shared by all `AbstractPoint` objects, implemented for both `A` and `B`. 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 `A` like so:

With such a declaration, it is acceptable to use any type that is a subtype of `T` in place of `T`, but not types that are not subtypes of `T`:

```
|
```

Type parameters for parametric composite types can be restricted in the same manner:

```
|
```

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{T}` immutable type (except that we omit the constructor here for simplicity), representing an exact ratio of integers:

```
|
```

It only makes sense to take ratios of integer values, so the parameter type `T` is restricted to being a subtype of `Integer`, and a ratio of integers represents a value on the real number line, so any `Rational{T}` 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:

```
|
```

However, there are three key differences:

- Tuple types may have any number of parameters.
- Tuple types are *covariant* in their parameters: `T1 <: T2` is a subtype of `T1`. Therefore `T1` 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:

```
|
```

Note the implications of covariance:

```
|
```

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 `_`, which denotes any number of trailing elements:

```
|
```

Notice that `_` corresponds to zero or more elements of type `T`. Vararg tuple types are used to represent the arguments accepted by varargs methods (see [Varargs Functions](#)).

The type `T*` corresponds to exactly `n` elements of type `T`. `T*` is a convenient alias for `T*`, i.e. a tuple type containing exactly `n` elements of type `T`.

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" `T.type` is an abstract type whose only instance is the object `T.type`. Since the definition is a little difficult to parse, let's look at some examples:

In other words, `is` is true if and only if `is` and `is` are the same object and that object is a type. Without the parameter, `is` is simply an abstract type which has all type objects as its instances, including, of course, singleton types:

Any object that is not a type is not an instance of :

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:

The slightly odd feature of these declarations as compared to typical parametric composite types, is that the type parameter 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, `int` and `float` are distinct types, even though they have identical representations. And of course, all specific pointer types are subtypes of the umbrella `type`:

14.9 UnionAll Types

We have said that a parametric type like `Array{T}` acts as a supertype of all its instances (`Array{Int}`, etc.). How does this work? `Array{T}` 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 `Array{T}` (or other parametric types like `Vector{T}`) is a different kind of type called a *type variable*. Such a type expresses the *iterated union* of types for all values of some parameter.

Types are usually written using the keyword `UnionAll`. For example `Array{T}` could be more accurately written as `UnionAll{T, Array{T}}`, meaning all values whose type is `Array{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 `UnionAll` introduces a single type variable, so these expressions are nested for types with multiple parameters, for example `UnionAll{T, UnionAll{S, Array{T, S}}}`.

The type application syntax `Array{T, S}` requires `T` to be a *type variable*, and first substitutes `S` for the outermost type variable in `T`. The result is expected to be another *type variable*, into which `S` is then substituted. So `Array{T, S}` is equivalent to `UnionAll{T, Array{T, S}}`. This explains why it is possible to partially instantiate a type, as in `Array{Int, S}`: the first parameter value has been fixed, but the second still ranges over all possible values. Using explicit `UnionAll` syntax, any subset of parameters can be fixed. For example, the type of all 1-dimensional arrays can be written as `UnionAll{T, Array{T}}`.

Type variables can be restricted with subtype relations. `Array{T, S}` refers to all arrays whose element type is some kind of `S`. The syntax `Array{T, S} <: Array{T, S'}` is a convenient shorthand for `UnionAll{T, Array{T, S} <: Array{T, S'}}`. Type variables can have both lower and upper bounds. `Array{T, S} <: Array{T, S'}` refers to all arrays of `S` that are able to contain `S'` (since `S` must be at least as big as `S'`). The syntax `Array{T, S} <: Array{T, S'}` also works to specify only the lower bound of a type variable, and `Array{T, S} <: Array{T, S'}` is equivalent to `UnionAll{T, Array{T, S} <: Array{T, S'}}`.

Since `UnionAll` expressions nest, type variable bounds can refer to outer type variables. For example `Array{T, S} <: Array{T, S'}` refers to 2-tuples whose first element is some `S`, and whose second element is an `Array{T, S'}` of any kind of array whose element type contains the type of the first tuple element.

The `UnionAll` keyword itself can be nested inside a more complex declaration. For example, consider the two types created by the following declarations:

`Array{Array{T}}` 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, `Array{Array{T, S}}` defines a 1-dimensional array of 1-dimensional arrays all of whose inner arrays must have the same type. Note that `Array{Array{T}}` is an abstract type, e.g., `Array{Array{Int}}`, whereas `Array{Array{T, S}}` is a concrete type. As a consequence, `Array{Array{T}}` can be constructed with a zero-argument constructor `Array{Array{T}}` but `Array{Array{T, S}}` cannot.

There is a convenient syntax for naming such types, similar to the short form of function definition syntax:

This is equivalent to `Array{T,1}`. Writing `Array{T}` is equivalent to writing `Array{T,1}`, and the umbrella type `Array{T}` has as instances all `Array{T,1}` 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 `Array{T}` for the abstract type including all one-dimensional dense arrays of any element type.

14.10 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, `Ptr` is aliased to either `Ptr{Cint}` or `Ptr{Cvoid}` as is appropriate for the size of pointers on the system:

```
Ptr{Cint} = Ptr{Cint}
```

This is accomplished via the following code in `libjulia`:

```
Ptr{Cint} = Ptr{Cint}
```

Of course, this depends on what `Ptr` is aliased to – but that is predefined to be the correct type – either `Ptr{Cint}` or `Ptr{Cvoid}`.

(Note that unlike `Ptr`, `Ptr{Cint}` does not exist as a type alias for a specific sized `Ptr`. Unlike with integer registers, the floating point register sizes are specified by the IEEE-754 standard. Whereas the size of `Ptr{Cint}` reflects the size of a native pointer on that machine.)

14.11 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 `isa` operator, which indicates whether its left hand operand is a subtype of its right hand operand.

The `typeof` function tests if an object is of a given type and returns true or false:

```
typeof(1) == Int
```

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:

```
typeof(typeof(1))
```

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 :

is its own type.

Another operation that applies to some types is , which reveals a type's supertype. Only declared types () have unambiguous supertypes:

If you apply to other type objects (or non-type objects), a is raised:

14.12 Custom pretty-printing

Often, one wants to customize how instances of a type are displayed. This is accomplished by overloading the function. For example, suppose we define a type to represent complex numbers in polar form:

Here, we've added a custom constructor function so that it can take arguments of different types and promote them to a common type (see [Constructors](#) and [Conversion and Promotion](#)). (Of course, we would have to define lots of other methods, too, to make it act like a , e.g. , , , , 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. .

If we want it to display instead as , we would define the following method to print the object to a given output object (representing a file, terminal, buffer, etcetera; see [Networking and Streams](#)):

More fine-grained control over display of 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 or for displaying the object as part of another object (e.g. in an array). Although by default the 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 that takes the MIME type as its second argument (see [Multimedia I/O](#)), for example:

(Note that here will call the 2-argument method.) This results in:

where the single-line form is still used for an array of values. Technically, the REPL calls to display the result of executing a line, which defaults to , which in turn defaults to , but you should *not* define new methods unless you are defining a new multimedia display handler (see [Multimedia I/O](#)).

Moreover, you can also define methods for other MIME types in order to enable richer display (HTML, images, etcetera) of objects in environments that support this (e.g. Julia). For example, we can define formatted HTML display of objects, with superscripts and italics, via:

A object will then display automatically using HTML in an environment that supports HTML display, but you can call manually to get HTML output if you want:

14.13 "Value types"

In Julia, you can't dispatch on a *value* such as or . 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 , where is a type (e.g.,) but is just an .

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, , which serves as a customary way to exploit this technique for cases where you don't need a more elaborate hierarchy.

is defined as:

There is no more to the implementation of `Nullable{T}` than this. Some functions in Julia's standard library accept `Nullable{T}` types as arguments, and you can also use it to write your own functions. For example:

For consistency across Julia, the call site should always pass a *type* rather than creating an *instance*, i.e., use `Nullable{T}` rather than `Nullable{T}()`.

It's worth noting that it's extremely easy to mis-use parametric "value" types, including `Nullable{T}`; 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 `Nullable{T}`, please read the more extensive discussion in [the performance tips](#).

14.14 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{T}` object.
- Check if a `Nullable{T}` object has a missing value.
- Access the value of a `Nullable{T}` object with a guarantee that a `MissingException` will be thrown if the object's value is missing.
- Access the value of a `Nullable{T}` 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{T}` object, getting a `Nullable{Result}` result. The result will be missing if the original value was missing.
- Performing a test on the value (if it exists) of a `Nullable{T}` object, getting a result that is missing if either the `Nullable{T}` itself was missing, or the test failed.
- Perform general operations on single `Nullable{T}` objects, propagating the missing data.

Constructing objects

To construct an object representing a missing value of type `T`, use the `missing` function:

```
|
```

To construct an object representing a non-missing value of type `T`, use the `value` function:

```
|
```

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.

Checking if a `Nullable` object has a value

You can check if a `Nullable` object has any value using `isDefined`:

```
|
```

Safely accessing the value of a `Nullable` object

You can safely access the value of a `Nullable` object using `getOrElse`:

```
|
```

If the value is not present, as it would be for `missing`, a `NoSuchElementException` error will be thrown. The error-throwing nature of the `getOrElse` 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 `getOrElse`:

Tip

Make sure the type of the default value passed to `coalesce` and that of the `object` match to avoid type instability, which could hurt performance. Use `typeof` 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 `ismissing`, 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 `coalesce` function takes as arguments a function `f` and a value `v`. It produces a :

- If `v` is a missing value, then it produces a missing value;
- If `v` has a value, then it produces a `Nullable` containing `f(v)` 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 `ifelse` function takes as arguments a predicate function (that is, a function returning a boolean) and a value `v`. It produces a `Nullable` value:

- If `v` is a missing value, then it produces a missing value;
- If `predicate` is true, then it produces the original value ;
- If `predicate` is false, then it produces a missing value.

In this way, `ifelse` can be thought of as selecting only allowable values, and converting non-allowable values to missing values.

While `coalesce` and `ifelse` are useful in specific cases, by far the most useful higher-order function is `map`, which can handle a wide variety of cases, including making existing operations work and propagate `NaN`s. An example will motivate the need for `map`. Suppose we have a function that computes the greater of two real roots of a quadratic equation, using the quadratic formula:

We may verify that the result of `map` is `NaN`, as we expect, since `NaN` 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 `map` function makes this task easy; we can simply pass the `greater` function we wrote to :

|

If one or more of the inputs is missing, then the output of `ifnull` will be missing.

There exists special syntactic sugar for the `ifnull` function using a dot notation:

|

In particular, the regular arithmetic operators can be conveniently using `ifnull`-prefixed operators:

|

Chapter 15

Methods

Recall from [Functions](#) that a function is an object that maps a tuple of arguments to a return value, or throws an exception if no appropriate value can be returned. It is common for the same conceptual function or operation to be implemented quite differently for different types of arguments: adding two integers is very different from adding two floating-point numbers, both of which are distinct from adding an integer to a floating-point number. Despite their implementation differences, these operations all fall under the general concept of "addition". Accordingly, in Julia, these behaviors all belong to a single object: the `function`.

To facilitate using many different implementations of the same concept smoothly, functions need not be defined all at once, but can rather be defined piecewise by providing specific behaviors for certain combinations of argument types and counts. A definition of one possible behavior for a function is called a *method*. Thus far, we have presented only examples of functions defined with a single method, applicable to all types of arguments. However, the signatures of method definitions can be annotated to indicate the types of arguments in addition to their number, and more than a single method definition may be provided. When a function is applied to a particular tuple of arguments, the most specific method applicable to those arguments is applied. Thus, the overall behavior of a function is a patchwork of the behaviors of its various method definitions. If the patchwork is well designed, even though the implementations of the methods may be quite different, the outward behavior of the function will appear seamless and consistent.

The choice of which method to execute when a function is applied is called *dispatch*. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given, and on the types of all of the function's arguments. This is different than traditional object-oriented languages, where dispatch occurs based only on the first argument, which often has a special argument syntax, and is sometimes implied rather than explicitly written as an argument.¹ Using all of a function's arguments to choose which method should be invoked, rather than just the first, is known as *multiple dispatch*. Multiple dispatch is particularly useful for mathematical code, where it makes little sense to artificially deem the operations to "belong" to one argument more than any of the others: does the addition operation in `1 + 2` belong to `1` any more than it does to `2`? The implementation of a mathematical operator generally depends on the types of all of its arguments. Even beyond mathematical operations, however, multiple dispatch ends up being a powerful and convenient paradigm for structuring and organizing programs.

15.1 Defining Methods

Until now, we have, in our examples, defined only functions with a single method having unconstrained argument types. Such functions behave just like they would in traditional dynamically typed languages. Nevertheless, we have used multiple dispatch and methods almost continually without being aware of it: all of Julia's standard functions

¹In C++ or Java, for example, in a method call like `obj.method()`, the object `obj` "receives" the method call and is implicitly passed to the method via the `this` keyword, rather than as an explicit method argument. When the current object is the receiver of a method call, it can be omitted altogether, writing just `method()`, with `obj` implied as the receiving object.

and operators, like the aforementioned `+` function, have many methods defining their behavior over various possible combinations of argument type and count.

When defining a function, one can optionally constrain the types of parameters it is applicable to, using the `typeof` type-assertion operator, introduced in the section on [Composite Types](#):

This function definition applies only to calls where `x` and `y` are both values of type `T`:

Applying it to any other types of arguments will result in a `TypeError`:

As you can see, the arguments must be precisely of type `T`. Other numeric types, such as integers or 32-bit floating-point values, are not automatically converted to 64-bit floating-point, nor are strings parsed as numbers. Because `T` is a concrete type and concrete types cannot be subclassed in Julia, such a definition can only be applied to arguments that are exactly of type `T`. It may often be useful, however, to write more general methods where the declared parameter types are abstract:

This method definition applies to any pair of arguments that are instances of `T`. They need not be of the same type, so long as they are each numeric values. The problem of handling disparate numeric types is delegated to the arithmetic operations in the expression `x + y`.

To define a function with multiple methods, one simply defines the function multiple times, with different numbers and types of arguments. The first method definition for a function creates the function object, and subsequent method definitions add new methods to the existing function object. The most specific method definition matching the number and types of the arguments will be executed when the function is applied. Thus, the two method definitions above, taken together, define the behavior for `+` over all pairs of instances of the abstract type `T` – but with a different behavior specific to pairs of `T` values. If one of the arguments is a 64-bit float but the other one is not, then the `+` method cannot be called and the more general `+` method must be used:

The `typeof` definition is only used in the first case, while the `isa` definition is used in the others. No automatic casting or conversion of function arguments is ever performed: all conversion in Julia is non-magical and completely explicit. [Conversion and Promotion](#), however, shows how clever application of sufficiently advanced technology can be indistinguishable from magic.²

For non-numeric values, and for fewer or more than two arguments, the function `typeof` remains undefined, and applying it will still result in a `MethodError`:

You can easily see which methods exist for a function by entering the function object itself in an interactive session:

This output tells us that `typeof` is a function object with two methods. To find out what the signatures of those methods are, use the `methods` function:

which shows that `typeof` has two methods, one taking two arguments and one taking arguments of type `Any`. It also indicates the file and line number where the methods were defined: because these methods were defined at the REPL, we get the apparent line number 1.

In the absence of a type declaration with `::`, the type of a method parameter is `Any` by default, meaning that it is unconstrained since all values in Julia are instances of the abstract type `Any`. Thus, we can define a catch-all method for `typeof` like so:

This catch-all is less specific than any other possible method definition for a pair of parameter values, so it will only be called on pairs of arguments to which no other method definition applies.

Although it seems a simple concept, multiple dispatch on the types of values is perhaps the single most powerful and central feature of the Julia language. Core operations typically have dozens of methods:



Multiple dispatch together with the flexible parametric type system give Julia its ability to abstractly express high-level algorithms decoupled from implementation details, yet generate efficient, specialized code to handle each case at run time.

15.2 Method Ambiguities

It is possible to define a set of function methods such that there is no unique most specific method applicable to some combinations of arguments:



Here the call could be handled by either the or the method, and neither is more specific than the other. In such cases, Julia raises a rather than arbitrarily picking a method. You can avoid method ambiguities by specifying an appropriate method for the intersection case:

It is recommended that the disambiguating method be defined first, since otherwise the ambiguity exists, if transiently, until the more specific method is defined.

In more complex cases, resolving method ambiguities involves a certain element of design; this topic is explored further [below](#).

15.3 Parametric Methods

Method definitions can optionally have type parameters qualifying the signature:

The first method applies whenever both arguments are of the same concrete type, regardless of what type that is, while the second method acts as a catch-all, covering all other cases. Thus, overall, this defines a boolean function that checks whether its two arguments are of the same type:

Such definitions correspond to methods whose type signatures are `Ts` types (see [UnionAll Types](#)).

This kind of definition of function behavior by dispatch is quite common – idiomatic, even – in Julia. Method type parameters are not restricted to being used as the types of arguments: they can be used anywhere a value would be in the signature of the function or body of the function. Here's an example where the method type parameter `T` is used as the type parameter to the parametric type `Vector{T}` in the method signature:

As you can see, the type of the appended element must match the element type of the vector it is appended to, or else a `TypeError` is raised. In the following example, the method type parameter `T` is used as the return value:

Just as you can put subtype constraints on type parameters in type declarations (see [Parametric Types](#)), you can also constrain type parameters of methods:

The `pair` function behaves much like the `ispair` function defined above, but is only defined for pairs of numbers.

Parametric methods allow the same syntax as `ispair` expressions used to write types (see [UnionAll Types](#)). If there is only a single parameter, the enclosing curly braces (`{}`) can be omitted, but are often preferred for clarity. Multiple parameters can be separated with commas, e.g. `{T, S}`, or written using nested `{}`, e.g. `{T, S{}}`.

15.4 Redefining Methods

When redefining a method or adding new methods, it is important to realize that these changes don't take effect immediately. This is key to Julia's ability to statically infer and compile code to run fast, without the usual JIT tricks and overhead. Indeed, any new method definition won't be visible to the current runtime environment, including `Tasks` and `Threads` (and any previously defined `ispair` functions). Let's start with an example to see what this means:

In this example, observe that the new definition for `ispair` has been created, but can't be immediately called. The new global is immediately visible to the `ispair` function, so you could write `ispair{Int, Int}` (without parentheses). But neither you, nor any of your callers, nor the functions they call, or etc. can call this new method definition!

But there's an exception: future calls to `ispair` from the *REPL* work as expected, being able to both see and call the new definition of `ispair`.

However, future calls to `world` will continue to see the definition of `world` as it was *at the previous statement at the REPL*, and thus before that call to `world`.

You may want to try this for yourself to see how it works.

The implementation of this behavior is a "world age counter". This monotonically increasing value tracks each method definition operation. This allows describing "the set of method definitions visible to a given runtime environment" as a single number, or "world age". It also allows comparing the methods available in two worlds just by comparing their ordinal value. In the example above, we see that the "current world" (in which the method `world` exists), is one greater than the task-local "runtime world" that was fixed when the execution of `world` started.

Sometimes it is necessary to get around this (for example, if you are implementing the above REPL). Fortunately, there is an easy solution: call the function using :

```
|
```

Finally, let's take a look at some more complex examples where this rule comes into play. Define a function `world`, which initially has one method:

```
|
```

Start some other operations that use :

```
|
```

Now we add some new methods to :

```
|
```

Compare how these results differ:

```
|
```

15.5 Parametrically-constrained Varargs methods

Function parameters can also be used to constrain the number of arguments that may be supplied to a "varargs" function ([Varargs Functions](#)). The notation `is` is used to indicate such a constraint. For example:

More usefully, it is possible to constrain varargs methods by a parameter. For example:

would be called only when the number of `args` matches the dimensionality of the array.

15.6 Note on Optional and keyword Arguments

As mentioned briefly in [Functions](#), optional arguments are implemented as syntax for multiple method definitions. For example, this definition:

translates to the following three methods:

This means that calling `foo(1)` is equivalent to calling `foo(1, 1, 1)`. In this case the result is `1`, because `foo` invokes the first method of `foo` above. However, this need not always be the case. If you define a fourth method that is more specialized for integers:

then the result of both `and` and `is` is `.` In other words, optional arguments are tied to a function, not to any specific method of that function. It depends on the types of the optional arguments which method is invoked. When optional arguments are defined in terms of a global variable, the type of the optional argument may even change at run-time.

Keyword arguments behave quite differently from ordinary positional arguments. In particular, they do not participate in method dispatch. Methods are dispatched based only on positional arguments, with keyword arguments processed after the matching method is identified.

15.7 Function-like objects

Methods are associated with types, so it is possible to make any arbitrary Julia object “callable” by adding methods to its type. (Such “callable” objects are sometimes called “functors.”)

For example, you can define a type that stores the coefficients of a polynomial, but behaves like a function evaluating the polynomial:

```

struct Poly{T}
    coeffs::Vector{T}
end

function (p::Poly)(x)
    sum(c * x^i for (i, c) in enumerate(p.coeffs))
end

```

Notice that the function is specified by type instead of by name. In the function body, `self` will refer to the object that was called. `A` can be used as follows:

```

A = Poly{Float64}([1, 2, 3])
A(4)

```

This mechanism is also the key to how type constructors and closures (inner functions that refer to their surrounding environment) work in Julia, discussed [later in the manual](#).

15.8 Empty generic functions

Occasionally it is useful to introduce a generic function without yet adding methods. This can be used to separate interface definitions from implementations. It might also be done for the purpose of documentation or code readability. The syntax for this is an empty `block` without a tuple of arguments:

```

function f{T}()
end

```

15.9 Method design and the avoidance of ambiguities

Julia’s method polymorphism is one of its most powerful features, yet exploiting this power can pose design challenges. In particular, in more complex method hierarchies it is not uncommon for [ambiguities](#) to arise.

Above, it was pointed out that one can resolve ambiguities like

by defining a method

This is often the right strategy; however, there are circumstances where following this advice blindly can be counter-productive. In particular, the more methods a generic function has, the more possibilities there are for ambiguities. When your method hierarchies get more complicated than this simple example, it can be worth your while to think carefully about alternative strategies.

Below we discuss particular challenges and some alternative ways to resolve such issues.

Tuple and NTuple arguments

(and) arguments present special challenges. For example,

are ambiguous because of the possibility that : there are no elements to determine whether the or variant should be called. To resolve the ambiguity, one approach is define a method for the empty tuple:

Alternatively, for all methods but one you can insist that there is at least one element in the tuple:

Orthogonalize your design

When you might be tempted to dispatch on two or more arguments, consider whether a "wrapper" function might make for a simpler design. For example, instead of writing multiple variants:

you might consider defining

where converts the argument to type . This is a very specific example of the more general principle of *orthogonal design*, in which separate concepts are assigned to separate methods. Here, will most likely need a fallback definition

A related strategy exploits `toBring` and `toCommonType` to a common type:

```
|
```

One risk with this design is the possibility that if there is no suitable promotion method converting `and` to the same type, the second method will recurse on itself infinitely and trigger a stack overflow. The non-exported function `canBeUsedAsAlternative` can be used as an alternative; when promotion fails it will still throw an error, but one that fails faster with a more specific error message.

Dispatch on one argument at a time

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:

```
|
```

Then the internal methods `and` and `canBeUsedAsAlternative` can dispatch on `withoutConcernAboutAmbiguities` without concern about ambiguities with each other with respect to `.`

Be aware that this strategy has at least one major disadvantage: in many cases, it is not possible for users to further customize the behavior of `by` by defining further specializations of your exported function `.` Instead, they have to define specializations for your internal methods `and` and `,` and this blurs the lines between exported and internal methods.

Abstract containers and element types

Where possible, try to avoid defining methods that dispatch on specific element types of abstract containers. For example,

```
|
```

generates ambiguities for anyone who defines a method

```
|
```

The best approach is to avoid defining *either* of these methods: instead, rely on a generic method `and` and make sure this method is implemented with generic calls (like `and` and `)` that do the right thing for each container type and element type *separately*. This is just a more complex variant of the advice to [orthogonalize](#) your methods.

When this approach is not possible, it may be worth starting a discussion with other developers about resolving the ambiguity; just because one method was defined first does not necessarily mean that it can't be modified or eliminated. As a last resort, one developer can define the "band-aid" method

```
|
```

that resolves the ambiguity by brute force.

Complex method "cascades" with default arguments

If you are defining a method "cascade" that supplies defaults, be careful about dropping any arguments that correspond to potential defaults. For example, suppose you're writing a digital filtering algorithm and you have a method that handles the edges of the signal by applying padding:

```
|
```

This will run afoul of a method that supplies default padding:

```
|
```

Together, these two methods generate an infinite recursion with `padding` constantly growing bigger.

The better design would be to define your call hierarchy like this:

```
|
```

`padding` 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" interface: a user who wants to control the padding explicitly can call the `padding` variant directly.

²Arthur C. Clarke, *Profiles of the Future* (1961): Clarke's Third Law.

Chapter 16

Constructors

Constructors¹ 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:

|

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.

16.1 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 objects that takes only one argument and uses the given value for both the `x` and `y` fields. This is simple:

|

¹Nomenclature: 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.

You could also add a zero-argument constructor method that supplies default values for both of the `x` and `y` fields:

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.

16.2 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.

For example, suppose one wants to declare a type that holds a pair of real numbers, subject to the constraint that the first number is not greater than the second one. One could declare it like this:

Now `Pair` objects can only be constructed such that :

If the type were declared `final`, 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 guarantees that all objects of the 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 , returning the resulting object:

|

This declaration has the same effect as the earlier definition of the type without an explicit inner constructor method. The following two types are equivalent – one with a default constructor, the other with an explicit constructor:

|

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.

16.3 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 fundamental difficulty may not be immediately obvious, let us briefly explain it. Consider the following recursive type declaration:

```
|
```

This type may appear innocuous enough, until one considers how to construct an instance of it. If `is` an instance of , then a second instance can be created by the call:

```
|
```

But how does one construct the first instance when no instance exists to provide as a valid value for its `field`? The only solution is to allow creating an incompletely initialized instance of with an unassigned `field`, and using that incomplete instance as a valid value for the `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 `type`, with a zero-argument inner constructor returning instances having `fields` pointing to themselves:

```
|
```

We can verify that this constructor works and constructs objects that are, in fact, self-referential:

```
|
```

Although it is generally a good idea to return a fully initialized object from an inner constructor, incompletely initialized objects can be returned:

```
|
```

While you are allowed to create objects with uninitialized fields, any access to an uninitialized reference is an immediate error:

This avoids the need to continually check for `nothing` 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:

Arrays of plain data types exhibit the same behavior.

You can pass incomplete objects to other functions from inner constructors to delegate their completion:

As with incomplete objects returned from constructors, if `nothing` or any of its callees try to access the `field` of the `object` before it has been initialized, an error will be thrown immediately.

16.4 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:

As you can see, for constructor calls with explicit type parameters, the arguments are converted to the implied field types: `works`, but `raises an` when converting `to`. When the type is implied by the arguments to the constructor call, as in `,` then the types of the arguments must agree – otherwise the `cannot be determined` – but any pair of real arguments with matching type may be given to the generic `constructor`.

What's really going on here is that `,` and `are` all different constructor functions. In fact, `is` a distinct constructor function for each type `.` Without any explicitly provided inner constructors, the declaration of the composite type automatically provides an inner constructor, `,` for each possible type `,` that behaves just like non-parametric default inner constructors do. It also provides a single general outer `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:

Notice that each definition looks like the form of constructor call that it handles. The call `will` invoke the definition inside the `block`. The outer constructor declaration, on the other hand, defines a method for the general `constructor` which only applies to pairs of values of the same real type. This declaration makes constructor calls without explicit type parameters, like `and` `,` work. Since the method declaration restricts the arguments to being of the same type, calls like `,` with arguments of different types, result in "no method" errors.

Suppose we wanted to make the constructor call `work` by "promoting" the integer value `to` the floating-point value `.` The simplest way to achieve this is to define the following additional outer constructor method:

This method uses the `function` to explicitly convert `to` and then delegates construction to the general constructor for the case where both arguments are `.` With this method definition what was previously a `now` successfully creates a point of type `:`

However, other similar calls still don't work:

```
|
```

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 constructor work as one would expect:

```
|
```

The `convert` function converts all its arguments to a common type – in this case `Rational{Int}`. With this method definition, the `Rational` constructor promotes its arguments the same way that numeric operators like `+` do, and works for all kinds of real numbers:

```
|
```

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.

16.5 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](#):

```
|
```

The first line – – declares that takes one type parameter of an integer type, and is itself a real type. The field declarations and indicate that the data held in a object are a pair of integers of type , one representing the rational value's numerator and the other representing its denominator.

Now things get interesting. has a single inner constructor method which checks that both of and 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 their greatest common divisor, computed using the function. Since returns the greatest common divisor of its arguments with sign matching the first argument (here), after this division the new value of is guaranteed to be non-negative. Because this is the only inner constructor for , we can be certain that objects are always constructed in this normalized form.

also provides several outer constructor methods for convenience. The first is the "standard" general constructor that infers the type parameter 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 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 construct a by applying the constructor to and 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 – a complex number whose real and imaginary parts are rationals:

Thus, although the `+` operator usually returns an instance of `Complex{Int}`, if either of its arguments are complex integers, it will return an instance of `Complex{Complex{Int}}` instead. The interested reader should consider perusing the rest of `Complex{Int}`: it is short, self-contained, and implements an entire basic Julia type.

16.6 Constructors and Conversion

Constructors in Julia are implemented like other callable objects: methods are added to their types. The type of a type is `DataType`, so all constructor methods are stored in the method table for the `DataType` 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{T,U}(x::T)` instead of defining a constructor, because Julia falls back to calling `convert{T,U}(x::T)` if no matching constructor is found. For example, if no constructor `Complex{Complex{Int}}` exists `Complex{Complex{Int}}` is called.

`convert{T,U}` is used extensively throughout Julia whenever one type needs to be converted to another (e.g. in assignment, `+=`, etcetera), and should generally only be defined (or successful) if the conversion is lossless. For example, `Complex{Complex{Int}}` produces `Complex{Complex{Int}}`, but throws an `OverflowError`. If you want to define a constructor for a lossless conversion from one type to another, you should probably define a `convert{T,U}` method instead.

On the other hand, if your constructor 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 method. For example, the `Complex{Complex{Int}}` constructor creates a zero-dimensional array of the type `Complex{Complex{Int}}`, but is not really a "conversion" from `Complex{Int}` to an `Complex{Complex{Int}}`.

16.7 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 `Complex{Int}` but not to `Complex{Complex{Int}}`. Optionally, outer constructors that determine type parameters automatically can be added, for example constructing a `Complex{Complex{Int}}` from the call `Complex{Complex{Int}}()`. 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:

```

using Base: promote_rule

struct SummedVector{T}
    v::Vector{T}
    s::T
end

SummedVector{T}(v::Vector{T}) = SummedVector{T}(v, sum(v))

Base.promote_rule{S, T}(::Type{SummedVector{S}}, ::Type{SummedVector{T}}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{SummedVector{S}}, ::Type{T}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{T}, ::Type{SummedVector{S}}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{T}, ::Type{T}) = T

```

The problem is that we want `SummedVector{T}` to be a larger type than `T`, so that we can sum many elements with less information loss. For example, when `T` is `Int`, we would like `SummedVector{Int}` to be `Complex{Complex{Int}}`. Therefore we want to avoid an interface that allows the user to construct instances of the type `SummedVector{T}`. One way to do this is to provide a constructor only for `SummedVector{Complex{Complex{Int}}}`, but inside the `SummedVector` definition block to suppress generation of default constructors:

```

using Base: promote_rule

struct SummedVector{T}
    v::Vector{T}
    s::T
end

SummedVector{T}(v::Vector{T}) = SummedVector{T}(v, sum(v))

Base.promote_rule{S, T}(::Type{SummedVector{S}}, ::Type{SummedVector{T}}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{SummedVector{S}}, ::Type{T}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{T}, ::Type{SummedVector{S}}) =
    SummedVector{promote_type{S, T}}

Base.promote_rule{S, T}(::Type{T}, ::Type{T}) = T

```

This constructor will be invoked by the syntax `T()`. The syntax `T(a, b, c)` allows specifying parameters for the type to be constructed, i.e. `T(a, b, c)` will return a `T`. `T()` can be used in any constructor definition, but for convenience the parameters to `T()` are automatically derived from the type being constructed when possible.

Chapter 17

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 + 1.0` as the floating-point value `2.0`, 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 before addition since integers and floating-point values cannot be added 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 + 1.0` would be a compilation error in both Ada and ML. Instead one must write `1.0 + 1.0`, 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 should promote to when mixed with other types.

17.1 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> convert{Int64, String}("1234567890")
1234567890
```

Conversion isn't always possible, in which case a `MethodError` is thrown indicating that `convert` doesn't know how to perform the requested conversion:

```
julia> convert{Int64, String}("1234567890a")
MethodError: no method matching convert{Int64, String} at .:convert:10
```

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:

```
julia> convert{Bool, Real}(0.0)
false
```

The type of the first argument of this method is a [singleton type](#), , the only instance of which is . Thus, this method is only invoked when the first argument is the type value . 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:

|

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:

|

Case Study: Rational Conversions

To continue our case study of Julia's type, here are the conversions declared in , right after the declaration of the type and its constructors:

|

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 .

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 `round` operator for truncated integer division (rounded towards zero).

17.2 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 `Value{<T>}` can also be represented as a `Value{<S>}`, `Value{<T>}` is not a subtype of `Value{<S>}`.

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:

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. Rationals mixed 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 :

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 `Base`, 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, `Complex{T}` provides the following outer constructor method:

This allows calls like the following to work:

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` 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:

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:

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{Int, BigInt}` and `promote_rule{BigInt, Int}` – the symmetry is implied by the way `promote` is used in the promotion process.

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`:

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 `src/number.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:

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 promotion of its numerator/denominator type with the other integer type. The 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 18

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.

18.1 Iteration

Required methods		Brief description
		Returns the initial iteration state
		Returns the current item and the next state
		Tests if there are any items remaining
Important optional methods	Default definition	Brief description
		One of <code>iterate</code> , <code>iterkeys</code> , or <code>itervalues</code> as appropriate
		Either <code>iterate</code> or <code>iterkeys</code> as appropriate
		The type the items returned by
	<i>(undefined)</i>	The number of items, if known
	<i>(undefined)</i>	The number of items in each dimension, if known

Value returned by	Required Methods
	and
	<i>(none)</i>
	<i>(none)</i>

Value returned by	Required Methods
	<i>(none)</i>

Sequential iteration is implemented by the methods `iterate`, `iterkeys`, and `itervalues`. 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 `iterate` method returns the initial state for the iterable object. That state gets passed along to `iterkeys`, which tests if there are any elements remaining, and `itervalues`, 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 that 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` loop since the syntax:

```
|
```

is translated into:

```
|
```

A simple example is an iterable sequence of square numbers with a defined length:

```
|
```

With only `iter`, `len`, and `getitem` definitions, the `Iterable` type is already pretty powerful. We can iterate over all the elements:

```
|
```

We can use many of the builtin methods that work with iterables, like `list`, `tuple`, and `dict`:

```
|
```

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 `Sequence` will always be `int`. By extending the `__getitem__` 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 `__len__`, too.

Now, when we ask Julia to fill all the elements into an array it can preallocate an array of the right size instead of blindly filling each element into a new array:

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:

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.

18.2 Indexing

Methods to implement	Brief description
<code>getindex</code>	, indexed element access
<code>setindex!</code>	, indexed assignment
<code>end</code>	The last index, used in <code>end</code>

For the `Iterable` above, we can easily compute the `th` element of the sequence by squaring it. We can expose this as an indexing expression `s[i]`. To opt into this behavior, `s` simply needs to define:

Additionally, to support the syntax `s[1:n]`, we must define `end` to specify the last valid index:

Note, though, that the above *only* defines `s[i]` with one integer index. Indexing with anything other than an integer will throw a `MethodError` saying that there was no matching method. In order to support indexing with ranges or vectors of integers, separate methods must be written:

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 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`.

18.3 Abstract Arrays

Methods to implement		Brief description
		Returns a tuple containing the dimensions of
		(if <code>Linear</code>) Linear scalar indexing
		(if <code>where</code>) N-dimensional scalar indexing
		(if <code>Scalar</code>) Scalar indexed assignment
		(if <code>where</code>) N-dimensional scalar indexed assignment
Optional methods	Default definition	Brief description
		Returns either <code>Linear</code> or <code>Scalar</code> . See the description below.
	defined in terms of <code>scalar</code>	Multidimensional and nonscalar indexing
	defined in terms of <code>scalar</code>	Multidimensional and nonscalar indexed assignment
<code>//</code>	defined in terms of <code>scalar</code>	Iteration
		Number of elements
		Return a mutable array with the same shape and element type
		Return a mutable array with the same shape and the specified element type
		Return a mutable array with the same element type and size <code>dims</code>
		Return a mutable array with the specified element type and size
Non-traditional indices	Default definition	Brief description
		Return the number of valid indices
		Return a mutable array with the specified indices (see below)
		Return an array similar to <code>array</code> with the specified indices (see below)

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 `Linear`. 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 as efficient as possible. 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 `Linear` and `Scalar`. 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. `Linear` arrays are simple: just define `Linear`. When the array is subsequently indexed with a multidimensional set of indices, the fallback `Linear` efficiently converts the indices into one linear index and then calls the above method. `Scalar` arrays, on the other hand, require methods to be defined for each supported dimensionality with `indices`. For example, the built-in `Array{T,N}` type only supports two dimensions, so it just defines `Linear`. The same holds for `Vector{T}`.

Returning to the sequence of squares from above, we could instead define it as a subtype of an :

```
|
```

Note that it's very important to specify the two parameters of the ; the first defines the , and the second defines the . That supertype and those three methods are all it takes for to be an iterable, indexable, and completely functional array:

```
|
```

As a more complicated example, let's define our own toy N-dimensional sparse-like array type built on top of :

```
|
```

Notice that this is an array, so we must manually define `__len__` and `__getitem__` at the dimensionality of the array. Unlike the `list`, we are able to define `__setitem__`, and so we can mutate the array:

The result of indexing an array can itself be an array (for instance when indexing by a tuple). The fallback methods use `__getitem__` 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:

In this example it is accomplished by defining `__getitem__` to create the appropriate wrapped array. (Note that while `array` supports 1- and 2-argument forms, in most case you only need to specialize the 3-argument form.) For this to work it's important that `array` is mutable (supports `__setitem__`). Defining `__len__`, `__getitem__`, and `__setitem__` also makes it possible to `__iter__` the array:

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 `array`:

If you are defining an array type that allows non-traditional indexing (indices that start at something other than 1), you should specialize `__getitem__`. You should also specialize `__setitem__` so that the `size` argument (ordinarily a size-tuple) can accept `objects`, perhaps range-types of your own design. For more information, see [Arrays with custom indices](#).

Chapter 19

Modules

Modules in Julia are separate variable workspaces, i.e. they introduce a new global scope. They are delimited syntactically, inside `module`. 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
    # Module body
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 `my_func` and type `MyType` are exported, and so will be available for importing into other modules. Function `my_private_func` is private to `MyModule`.

The statement `using MyModule` means that a module called `MyModule` 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 `MyModule` 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 `MyModule`.

The statement `import MyModule` is a syntactic shortcut for `using MyModule`.

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:

```
|
```

Standard modules

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 `ls` 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`, 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: `eval` is still imported, as per above). In terms of `using`, a standard `module` looks like this:

```
|
```

Relative and absolute module paths

Given the statement `using Pkg`, the system looks for `Pkg` within `.`. If the module does not exist, the system attempts to `load`, which typically results in loading code from an installed package.

However, some modules contain submodules, which means you sometimes need to access a module that is not directly available in `.`. There are two ways to do this. The first is to use an absolute path, for example `using Pkg: Pkg`. 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:

```
|
```

Here module `cont` contains a submodule `sub`, and code in `cont` wants the contents of `sub` to be visible. This is done by starting the path with a period. Adding more leading periods moves up additional levels in the module hierarchy. For example `cont.sub` would look for `sub` in `cont`'s enclosing module rather than in `cont` itself.

Note that relative-import qualifiers are only valid in `import` and `using` statements.

Module file paths

The global variable `LOAD_PATH` contains the directories Julia searches for modules when calling `import`. It can be extended using :

Putting this statement in the file `init.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. `cont.sub`), 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, for example, you must use `cont.sub` to refer to it. If the operator is more than one character in length you must surround it in brackets, such as: `cont.sub`.

Macro names are written with `macro` in import and export statements, e.g. `import macro`. Macros in other modules can be invoked as `macro` or `macro`.

The syntax `global` does not work to assign a global in another module; global assignment is always module-local.

A variable can be "reserved" for the current module without assigning to it by declaring it as `global` at the top level. This can be used to prevent name conflicts for globals initialized after load time.

Module initialization and precompilation

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 `precompile()`. The resulting cache files will be stored in `cache/`. Subsequently, the module is automatically recompiled upon `precompile()` whenever any of its dependencies change; dependencies are modules it imports, the Julia build, files it includes, or explicit dependencies declared by `using` in the module file(s).

For file dependencies, a change is determined by examining whether the modification time (`mtime`) of each file loaded by `using` or added explicitly by `include` 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 `find_module_path` 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 on the module you changed, and any module that depended on it in which you want to see the change reflected.

Precompiling a module also recursively precompiles 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 in the module file to cause to throw an error (and thereby prevent the module from being imported by any other precompiled module).

should *not* be used in a module unless all of its dependencies are also using . 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 function in your module that executes any initialization steps that must occur at runtime. This function will not be called during compilation (or). 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 (), but not if it is being loaded into a full-compilation process.

In particular, if you define a in a module, then Julia will call immediately *after* the module is loaded (e.g., by , , or) at runtime for the *first* time (i.e., 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 functions called *before* the of the enclosing module.

Two typical uses of are calling runtime initialization functions of external C libraries and initializing global constants that involve pointers returned by external libraries. For example, suppose that we are calling a C library that requires us to call a initialization function at runtime. Suppose that we also want to define a global constant that holds the return value of a function defined by – 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 function in your module:

```

module MyModule
    # ... other code ...
    const ptr_val = init_ptr()
end module

```

Notice that it is perfectly possible to define a global inside a function like ; 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 would also have to be initialized in .

Constants involving most Julia objects that are not produced by do not need to be placed in : 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 and .

Dictionary and set types, or in general anything that depends on the output of a method, are a trickier case. In the common case where the keys are numbers, strings, symbols, ranges, , 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 or and generic user-defined types where you haven't defined a method, the fallback method depends on the memory address of the object (via its) 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 function. Alternatively, you can use the 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:

```
|
```

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 usages of this incrementally compiled module will start from that same counter value.

Note that (which works by hashing the memory pointer) has similar issues (see notes on usage below).

One alternative is to store both and the current value, however, it may be better to redesign the code to not depend on this global state.

2. Associative collections (such as and) need to be re-hashed in . (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):

```
|
```

Several additional restrictions are placed on the operations that can be done while precompiling code to help the user avoid other wrong-behavior situations:

1. Calling to cause a side-effect in another module. This will also cause a warning to be emitted when the incremental precompile flag is set.
2. statements from local scope after has been started (see issue #12010 for plans to add an error for this)
3. Replacing a module (or calling) 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), and no cleanup is done after

2. The memory sharing behavior of a reshaped array is disregarded by precompilation (each view gets its own copy)
3. Expecting the filesystem to be unchanged between compile-time and runtime e.g. `/` to find resources at runtime, or the `BinDeps` 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. `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 `...`, `...`, 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.

It is sometimes helpful during module development to turn off incremental precompilation. The command line flag `--no-precompile` enables you to toggle module precompilation on and off. When Julia is started with `--no-precompile` the serialized modules in the compile cache are ignored when loading modules and module dependencies. `precompile` can still be called manually and it will respect `@precompile` directives for the module. The state of this command line flag is passed to `Base.precompile` to disable automatic precompilation triggering when installing, updating, and explicitly building packages.

Chapter 20

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:

```
|
```

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 `macro` just as well.

Here is a more complex example, still using Markdown:

```
|
```

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 `function foo(x, y=1)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `x=1`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `x{}` and `y{}`). 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 placeholder in the signature (i.e. `kwargs`), and give the complete list under an `kwargs` 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 ...": 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:

```
|
```

5. 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 `>>>` and contains any number of `>>>` prompts together with inputs and expected outputs that mimic the Julia REPL.

For example in the following docstring a variable `var` is defined and the expected result, as printed in a Julia REPL, appears afterwards:

```
|
```

Warning

Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions.

Operating system word size (or) 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 to run all the doctests in the Julia Manual, which will ensure that your example works.

Examples that are untestable should be written within fenced code blocks starting with 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.

6. Use backticks to identify code and equations.

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. rather than .

7. Place the starting and ending characters on lines by themselves.

That is, write:

```
|
```

rather than:

```
|
```

This makes it more clear where docstrings start and end.

8. 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.

20.1 Accessing Documentation

Documentation can be accessed at the REPL or in [IJulia](#) by typing followed by the name of a function or macro, and pressing . For example,

```
|
```

will bring up docs for the relevant function, macro or string macro respectively. In [Juno](#) using will bring up documentation for the object under the cursor.

20.2 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`). 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:

When retrieving documentation for a generic function, the metadata for each method is concatenated with the function, which can of course be overridden for custom types.

20.3 Advanced Usage

The `DocStringExtensions` macro associates its first argument with its second in a per-module dictionary called `DocStringExtensions.docstrings`. By default, documentation is expected to be written in Markdown, and the `DocStringExtensions.docstring` 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 `DocStringExtensions.docstring` macro (or equally, the `DocStringExtensions.docstring` function) will search all 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:

Or for use with Julia's metaprogramming functionality:

```
|
```

Documentation written in non-toplevel blocks, such as ``, ``, ``, ``, and ``, ``, is added to the documentation system as blocks are evaluated. For example:

```
|
```

will add documentation to `MyType` when the condition is `isodd(x)`. Note that even if `isodd(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 `DocStringExtensions.jl` for your custom type that returns the documentation on a per-instance basis. For instance,

```
|
```

will display "Documentation for MyType with value x" while `DocStringExtensions.doc(MyType{y})` will display "Documentation for MyType with value y".

20.4 Syntax Guide

A comprehensive overview of all documentable Julia syntax.

In the following examples `DocStringExtensions.docstring` is used to illustrate an arbitrary docstring which may be one of the follow four variants and contain arbitrary text:

```
|
```

```
|
```

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

```
|
```

Adds docstring to `f`. The first version is the preferred syntax, however both are equivalent.

```
|
```

Adds docstring to `f`.

```
|
```

Adds docstring to two `s`, namely `f` and `g`.

Macros

```
|
```

Adds docstring to the `macro` definition.

```
|
```

Adds docstring to the macro named `my_macro`.

Types

```
|
```

Adds the docstring to types , , and .

```
|
```

Adds docstring to type , to field and to field . Also applicable to types.

Modules

```
|
```

Adds docstring to the . Adding the docstring above the is the preferred syntax, however both are equivalent.

```
|
```

Documenting a by placing a docstring above the expression automatically imports into the module. These imports must be done manually when the module expression is not documented. Empty s cannot be documented.

Global Variables

```
|
```

Adds docstring to the `s`, `,` and `.`

`s` are used to store a reference to a particular `in a` without storing the referenced value itself.

Note

When a definition is only used to define an alias of another definition, such as is the case with the function `and` its alias `in`, do not document the alias and instead document the actual function.

If the alias is documented and not the real definition then the docsystem (`mode`) will not return the docstring attached to the alias when the real definition is searched for.

For example you should write

```
|
```

rather than

```
|
```

```
|
```

Adds docstring to the value associated with `.` Users should prefer documenting `at` its definition.

Multiple Objects

```
|
```

Adds docstring to `and` each of which should be a documentable expression. This syntax is equivalent to

```
|
```

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 `and` `.`

Macro-generated code

Adds `docstring` to expression generated by expanding `.` This allows for expressions decorated with `,` `,` `,` 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 `macro` macro.

The `macro` macro makes use of `to` to allow for documenting `s`. Examining its definition should serve as an example of how to use `correctly`.

– Macro.

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same `docstring` is applied to each expression.

has no effect when a macro that uses it is not documented.

20.5 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.

Italics

Surround words with one asterisk, `, ,` to display the enclosed text in italics.

Literals

Surround text that should be displayed exactly as written with single backticks, `.`

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.

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, `.`

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

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.

It's also possible to add cross-references to other documented functions/methods/variables within the Julia documentation itself. For example:

This will create a link in the generated docs to the `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.

|

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.

|

Headers

A document can be split up into different sections using headers. Headers use the following syntax:

|

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.

|

Additionally, code blocks can be enclosed using triple backticks with an optional "language" to specify how a block of code should be highlighted.

```
def foo():
    """
    A docstring for the foo function.
    """
    pass
```

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.

```
> This is a block quote.
> It can span multiple lines.
> It can also contain code blocks.
```

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 a link to it.

```
!https://example.com/image.png
```

Lists

Unordered lists can be written by prepending each item in a list with either `*`, `o`, or `.`

```
* Item 1
o Item 2
. Item 3
```

Note the two spaces before each `and` 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.

```


- and
- and
- and

```

Note

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 `in`.

Ordered lists are written by replacing the "bullet" character, either `,`, `,`, or `,`, with a positive integer followed by either `.` or `.`.

```


1. and
2. and
3. and

```

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 \LaTeX equations that do not fit inline within a paragraph may be written as display equations using a fenced code block with the "language" `as in the example below.`

```


$$E=mc^2$$


```

Footnotes

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.

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:

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.

Note

As illustrated in the above example each column of `<td>` characters must be aligned vertically.

A `<td>` character on either end of a column's header separator (the row containing `<td>` characters) specifies whether the row is left-aligned, right-aligned, or (when `<td>` appears on both ends) center-aligned. Providing no `<td>` 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:

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. in the case of the admonition.

20.6 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 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 21

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.

21.1 Program representation

Every Julia program starts life as a string:

|

What happens next?

The next step is to [parse](#) each string into an object called an expression, represented by the Julia type :

|

objects contain two parts:

- a identifying the kind of expression. A symbol is an [interned string](#) identifier (more discussion below).

|

- the expression arguments, which may be symbols, other expressions, or literal values:

|

Expressions may also be constructed directly in **prefix notation**:

```
|
```

The two expressions constructed above – by parsing and by direct construction – are equivalent:

```
|
```

The key point here is that Julia code is internally represented as a data structure that is accessible from the language itself.

The `show_sexpr` function provides indented and annotated display of `Expr` objects:

```
|
```

objects may also be nested:

```
|
```

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`:

```
|
```

Symbols

The `Symbol` character has two syntactic purposes in Julia. The first form creates a `Symbol`, an **interned string** used as one building-block of expressions:

```
|
```

The `Symbol{...}` constructor takes any number of arguments and creates a new symbol by concatenating their string representations together:

```
|
```


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 `Expr` are needed to avoid ambiguity in parsing.:

```
|
```

21.2 Expressions and evaluation

Quoting

The second syntactic purpose of the `Expr` character is to create expression objects without using the explicit `Expr{...}` constructor. This is referred to as *quoting*. The `Expr` 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:

```
|
```

(to view the structure of this expression, try `Expr{Int}()` and `Expr{Int}()`, or use `Expr{Int}()` as above)

Note that equivalent expressions may be constructed using `Expr{Int}()` or the direct `Expr{Int}()` form:

```
|
```

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, `Expr{Int}()` and `Expr{Int}()` are symbols, `Expr{Int}()` is a subexpression, and `Expr{Int}()` is a literal 64-bit signed integer.

There is a second syntactic form of quoting for multiple expressions: blocks of code enclosed in `Expr{Int}()`. Note that this form introduces `Expr{Int}()` elements to the expression tree, which must be considered when directly manipulating an expression tree generated from `Expr{Int}()` blocks. For other purposes, `Expr{Int}()` and `Expr{Int}()` blocks are treated identically.

```
|
```

Interpolation

Direct construction of objects with value arguments is powerful, but constructors can be tedious compared to "normal" Julia syntax. As an alternative, Julia allows "splicing" or interpolation of literals or expressions into quoted expressions. Interpolation is indicated by the `␣` prefix.

In this example, the literal value of `1` is interpolated:

Interpolating into an unquoted expression is not supported and will cause a compile-time error:

In this example, the tuple `(1, 2)` is interpolated as an expression into a conditional test:

Interpolating symbols into a nested expression requires enclosing each symbol in an enclosing quote block:

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.

and effects

Given an expression object, one can cause Julia to evaluate (execute) it at global scope using :

Every `module` has its own `function` that evaluates expressions in its global scope. Expressions passed to `are` not limited to returning values – they can also have side-effects that alter the state of the enclosing module's environment:

Here, the evaluation of an expression object causes a value to be assigned to the global variable `.`

Since expressions are just `objects` which can be constructed programmatically and then evaluated, it is possible to dynamically generate arbitrary code which can then be run using `.` Here is a simple example:

The value of `is` used to construct the expression `which` applies the `function` to the value `1` and the variable `.` Note the important distinction between the way `and` `are` used:

- The value of the *variable* at expression construction time is used as an immediate value in the expression. Thus, the value of `when` the expression is evaluated no longer matters: the value in the expression is already `,` independent of whatever the value of `might` be.
- On the other hand, the *symbol* is used in the expression construction, so the value of the variable `at` that time is irrelevant – `is` just a symbol and the variable `need` not even be defined. At expression evaluation time, however, the value of the symbol `is` resolved by looking up the value of the variable `.`

Functions on expressions

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 objects: the `quote` 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:

```
function double_numeric(x::Expr)
    if isa(x, Number)
        Expr(:call, :*, x, x)
    else
        x
    end
end

double_numeric(2) # returns 2 * 2
double_numeric(2 + 3) # returns 2 + 3
```

As another example, here is a function that doubles any numeric argument, but leaves expressions alone:

```
function double_numeric(x::Expr)
    if isa(x, Number)
        Expr(:call, :*, x, x)
    else
        x
    end
end

double_numeric(2) # returns 2 * 2
double_numeric(2 + 3) # returns 2 + 3
```

21.3 Macros

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:

Macros have a dedicated character in Julia's syntax: the `@` (at-sign), followed by the unique name declared in a block. In this example, the compiler will replace all instances of `with` with:

When `with` is entered in the REPL, the expression executes immediately, thus we only see the evaluation result:

Now, consider a slightly more complex macro:

This macro takes one argument: `expr`. When `expr` is encountered, the quoted expression is *expanded* to interpolate the value of the argument into the final expression:

We can view the quoted return expression using the function `show` (**important note:** this is an extremely useful tool for debugging macros):

We can see that the `1` literal has been interpolated into the expression.

There also exists a macro `eval` that is perhaps a bit more convenient than the `eval` function:

Hold up: why macros?

We have already seen a function `eval` in a previous section. In fact, `eval` 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:

The first call to `is` is executed when `is` is called. The resulting expression contains *only* the second :

```
|
```

Macro invocation

Macros are invoked with the following general syntax:

```
|
```

Note the distinguishing `is` before the macro name and the lack of commas between the argument expressions in the first form, and the lack of whitespace after `is` 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 `(1 2)` as one argument to the macro:

```
|
```

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 `macroexpand` function within the macro body:

```
|
```

Building an advanced macro

Here is a simplified definition of Julia's `macro`:

```
|
```

This macro can be used like this:

```
|
```

In place of the written syntax, the macro call is expanded at parse time to its returned result. This is equivalent to writing:

```
|
```

That is, in the first call, the expression is spliced into the test condition slot, while the value of is spliced into the assertion message slot. The entire expression, thus constructed, is placed into the syntax tree where the macro call occurs. Then at execution time, if the test expression evaluates to true, then 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 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:

```
|
```

Now 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 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 function:

```
|
```

There is yet another case that the actual 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., , 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 . Compare:

So now instead of getting a plain string in `str`, 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 `println` call; see for the complete implementation.

The `timeit` 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 `str` in above) follow the [normal scoping block behavior](#).

To demonstrate these issues, let us consider writing a `timeit` 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 times, and then has the value of the expression as its final value. The macro might look like this:

Here, we want `str`, `before`, and `after` to be private temporary variables, and we want `println` to refer to the `println` function in the standard library, not to any variable the user might have (the same applies to `time`). Imagine the problems that could occur if the user expression also contained assignments to a variable called `str`, or defined its own `println` variable. We might get errors, or mysteriously incorrect behavior.

Julia's macro expander solves these problems in the following way. First, variables within a macro result are classified as either local or global. A variable is considered local if it is assigned to (and not declared global), declared local, or used as a function argument name. Otherwise, it is considered global. Local variables are then renamed to be unique (using the `gensym` function, which generates new symbols), and global variables are resolved within the macro definition environment. Therefore both of the above concerns are handled; the macro's locals will not conflict with any user variables, and `using` will refer to the standard library definitions.

One problem remains however. Consider the following use of this macro:

|

Here the user expression is a call to `foo`, but not the same `foo` function that the macro uses. It clearly refers to `foo`. Therefore we must arrange for the code in `foo` to be resolved in the macro call environment. This is done by "escaping" the expression with :

|

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:

|

This kind of manipulation of variables should be used judiciously, but is occasionally quite handy.

21.4 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, this requires an extra build step, and a separate program to generate the repetitive code. In Julia, expression interpolation and `Expr` 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:

```

@inline function f3(x, y, z)
    f2(f2(x, y), z)
end
@inline function f2(x, y)
    f(x, f(y, x))
end
@inline function f(x, y)
    x + y
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 `Expr` prefix quoting form:

```

@inline f3(x, y, z) = f2(f2(x, y), z)
@inline f2(x, y) = f(x, f(y, x))
@inline f(x, y) = x + y

```

This sort of in-language code generation, however, using the `Expr` pattern, is common enough that Julia comes with a macro to abbreviate this pattern:

```

@inline f3(x, y, z) = f2(f2(x, y), z)
@inline f2(x, y) = f(x, f(y, x))
@inline f(x, y) = x + y

```

The `Expr` macro rewrites this call to be precisely equivalent to the above longer versions. For longer blocks of generated code, the expression argument given to `Expr` can be a block:

```

@inline f3(x, y, z) = f2(f2(x, y), z)
@inline f2(x, y) = f(x, f(y, x))
@inline f(x, y) = x + y

```

21.5 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:

- `@regex` produces a regular expression object rather than a string
- `@bytes` is a byte array literal for `String{UInt8}`.

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:

```

@regex s = Regex{s}

```

That's all. This macro says that the literal contents of the string literal `re` should be passed to the `Regex` macro and the result of that expansion should be placed in the syntax tree where the string literal occurs. In other words, the expression is equivalent to placing the following object directly into the syntax tree:

```
|
```

Not only is the string literal form shorter and far more convenient, but it is 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:

```
|
```

Since the regular expression `re` 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:

```
|
```

Moreover, if the compiler could not determine that the `Regex` object was constant over all loops, certain optimizations might not be possible, making this version still less efficient than the more convenient literal form above. Of 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 `cmd`` is parsed as `Cmd{}``. 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 `re`` 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 `Base` and `MyModule` provide non-standard string literal `re``, then one can write `Base.re`` or `MyModule.re`` 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 `cmd`` is implemented with the following innocuous-looking macro:

Of course, a large amount of complexity is hidden in the functions used in 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.

21.6 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 parsing-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 called, the body expression is first evaluated and compiled, then the returned expression is compiled and run. To make this efficient, the result is often cached. And to make this inferable, only a limited subset of the language is usable. Thus, generated functions provide a flexible framework to move work from run-time to compile-time, at the expense of greater restrictions on the allowable 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* of the arguments – not 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 `ccall`). 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 untyped generator.

It’s easiest to illustrate this with an example. We can declare a generated function as

Note that the body returns a quoted expression, namely `:(x + 1)`, rather than just the value of `x + 1`.

From the caller’s perspective, they are very similar to regular functions; in fact, you don’t have to know if you’re calling a regular or generated function – the syntax and result of the call is just the same. Let’s see how `@generated` behaves:

So, we see that in the body of the generated function, `is` 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 `.`

What happens if we evaluate `again` with a type that we have already used?

Note that there is no printout of `.` 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 number of times a generated function is generated *might* be only once, but 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 `in` 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 `,` so it is disallowed.

It is also important to see how `functions` interact with method redefinition. Following the principle that a correct 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 `has` one definition

Define other operations that use `:`

We now add some new definitions for `:`

and compare how these results differ:

Each method of a generated function has its own view of defined functions:

The example generated function above did not do anything a normal function 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:

(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:

Since the body of the generated function is non-deterministic, its behavior, *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 function has side-effects (the call to `rand`), and it is undefined exactly when, how often or how many times these side-effects will occur
- the function solves a problem that is better solved with multiple dispatch - defining `rand` and `randn` will do the same thing, but it is both simpler and faster.
- the function is pathologically insane

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.
4. Taking any locks: C code you call out to may use locks internally, (for example, it is not problematic to call `libc_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 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 that can be used to index into an array using `LinearIndices`, instead of `CartesianIndices`. One possible implementation is the following:

The same thing can be done using recursion:

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 move the iteration to compile-time; in compiler parlance, we use generated 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:

What code will this generate?

An easy way to find out is to extract the body into another (regular) function:

We can now execute and examine the expression it returns:

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 (except in edge cases where the function is generated more than once - see disclaimer above).

Chapter 22

Arrays Multi-dimensionales

Julia, como la mayoría de los lenguajes informáticos técnicos, proporciona una implementación de los arrays de primera clase. La mayoría de los lenguajes informáticos técnicos prestan mucha atención a su implementación de arrays a expensas de otros contenedores. Julia no trata los arrays de manera especial. La biblioteca de arrays se ha implementado casi completamente en el propio lenguaje Julia, y deriva su rendimiento del compilador, al igual que cualquier otro código escrito en Julia. Como tal, es también posible definir tipos de arrays personalizados heredando de `AbstractArray`. Consulte la [sección de manual en la interfaz `AbstractArray`](#) para ms detalles sobre implementar un tipo array personalizado.

Un array es una colección de objetos almacenados en una cuadrícula multidimensional. En el caso más general, un array puede contener objetos de tipo `T`. Para la mayoría de los propósitos computacionales, los arrays deben contener objetos de un tipo más específico, como `Float64` o `String`.

En general, a diferencia de muchos otros lenguajes informáticos técnicos, Julia no espera que los programas se escriban en un estilo vectorizado para el rendimiento. El compilador de Julia utiliza la inferencia de tipos y genera código optimizado para la indexación escalar de arrays, permitiendo que los programas se escriban en un estilo que sea conveniente y legible, sin sacrificar el rendimiento y utilizando menos memoria a veces.

En Julia, todos los argumentos a las funciones se pasan por referencia. Algunos lenguajes informáticos técnicos pasan los arrays por valor, y esto es conveniente en muchos casos. En Julia, las modificaciones hechas a los arrays de entrada dentro de una función serán visibles en la función principal. Toda la biblioteca de arrays de Julia garantiza que las entradas no sean modificadas por las funciones de biblioteca. El código de usuario, si necesita mostrar un comportamiento similar, debe tener cuidado de crear una copia de las entradas que puede modificar.

22.1 Arrays

Funciones Básicas

Function	Description
<code>eltype</code>	Tipo de los elementos contenidos en
<code>length</code>	Número de elementos en
<code>ndims</code>	Número de dimensiones de
<code>size</code>	Una tupla que contiene las dimensiones de
<code>size...{D}</code>	El tamaño de <code>a</code> a lo largo de una dimensión particular
<code>axes</code>	Una tupla que contiene los índices válidos de
<code>axes...{D}</code>	Un rango expresando los índices válidos a lo largo de la dimensión
<code>Iter{D}</code>	Un iterador eficiente para visitar cada posición en
<code>stride</code>	La zancada (<i>stride</i> , distancia de índice lineal entre elementos adyacentes) a lo largo de la dimensión <code>D</code> .
<code>strides</code>	Una tupla de las zancadas en cada dimensión

Construcción e Inicialización

Existen muchas funciones para construir e inicializar matrices. En la siguiente lista de tales funciones, las llamadas con un argumento pueden tomar una sola tupla de tamaños de dimensión o una serie de tamaños de dimensión pasados como un número variable de argumentos. Muchas de estas funciones también aceptan un primer argumento, que es el tipo de los elementos del array. Si este tipo es omitido se asumirá como tipo por defecto por defecto.

Function	Description
	an uninitialized dense
	an array of all zeros
	an array of all zeros with the same type, element type and shape as
	an array of all ones
	an array of all ones with the same type, element type and shape as
	a array with all values
	a array with all values and the same shape as
	a array with all values
	a array with all values and the same shape as
	an array containing the same data as , but with different dimensions
	copy
	copy , recursively copying its elements
	an uninitialized array of the same type as (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 if omitted.
	an array with the same binary data as , but with element type
	an array with random, iid ¹ and uniformly distributed values in the half-open interval $[0, 1)$
	an array with random, iid and standard normally distributed values
	-by- identity matrix
	-by- identity matrix
	range of linearly spaced elements from to
	fill the array with the value
	an array filled with the value

La sintaxis `array` construye un array 1-dimensional (vector) a partir de sus argumentos. Si todos los argumentos tienen un [tipo de promoción](#) común entonces ellos son convertidos a este tipo usando `dtype`.

Concatenación

Los arrays pueden ser contruídos y también concatenados usando las siguientes funciones:

Function	Description
	concatena n-d arrays a lo largo de la dimensión
	abreviatura para
	abreviatura para

Los valores escalares pasados a estas funciones son tratados como arrays de 1 elemento.

Las funciones de concatenación se usan tan frecuentemente que tiene una sintaxis especial:

`concat` tanto en la dimensión 1 (con puntos y coma) como en la dos (con espacios).

¹ *iid*, independently and identically distributed.

Expression	Calls

Inicializadores de Array Tipados

Se puede construir una matriz con un tipo de elemento específico utilizando la sintaxis `dtype([...])`. Esto construirá un array 1-d con el tipo de elemento `dtype`, inicializado para contener los elementos `...`, etc. Por ejemplo, `dtype([1, 2, 3])` construye un array heterogéneo que puede contener cualquier valor.

La sintaxis de concatenación puede ser prefijada de forma similar con un tipo para especificar el tipo de elemento del resultado.

|

Comprensiones

Las comprensiones proporcionan una forma general y potente de construir arrays. Su sintaxis es similar a la notación de construcción de conjuntos en matemáticas:

|

El significado de esta forma es que `expr` es evaluado para las variables `x`, `y`, etc. tomando cada valor de la lista de valores proporcionada. Los valores se pueden especificar mediante cualquier objeto iterable, pero comúnmente serán rangos como `range(10)` o `list('abc')`, o arrays de valores explícitos como `[1, 2, 3]`. El resultado es una matriz N-d densa con dimensiones que son la concatenación de las dimensiones de los rangos de las variables `x`, `y`, etc. y donde cada evaluación `expr` devuelve un escalar.

El siguiente ejemplo calcula la media ponderada del elemento actual y su vecino izquierdo y derecho a lo largo de una rejilla unidimensional:

|

El tipo del array resultante depende de los tipos de los elementos calculados. Para controlar el tipo explícitamente, un tipo puede ser precedido a la comprensión. Por ejemplo, podríamos haber solicitado el resultado en precisión simple escribiendo:

```
|
```

Expresiones Generador

Las comprensiones también se pueden escribir sin los corchetes que las encierran, produciendo un objeto conocido como **generador**. Este objeto puede ser iterado para producir valores bajo demanda, en lugar de reservar espacio para un array y almacenarlos en él de antemano (véase [Iteración](#)). Por ejemplo, la siguiente expresión suma una serie sin asignar memoria:

```
|
```

Quando se escribe una expresión generador con múltiples dimensiones dentro de una lista de argumentos, se necesitan paréntesis para separar el generador de argumentos posteriores:

```
|
```

Todas las expresiones separadas por comas después del se interpretan como rangos. Añadir paréntesis permite añadir un tercer argumento a :

```
|
```

Los rangos en generadores y comprensiones pueden depender de rangos anteriores escribiendo varias palabras clave :

```
|
```

En tales casos, el resultado es siempre unidimensional.

Los valores generados se pueden filtrar usando la palabra clave :

```
|
```

Indexación

La sintaxis general para indexar en un array n-dimensional es:

|

donde cada puede ser un entero escalar, un array de enteros o cualquier otro [índice soportado](#). Esto incluye () para seleccionar todos los índices dentro de la dimensión completa, rangos de la forma o para seleccionar subsecciones contiguas o con salto, y arrays de booleans para seleccionar elementos en sus índices .

Si todos los índices son escalares, entonces el resultado es un solo elemento del array . De lo contrario, es un array con el mismo número de dimensiones que la suma de las dimensionalidades de todos los índices.

Si todos los índices son vectores, por ejemplo, entonces la forma de sería , donde las ubicaciones de contienen el valor . Si se cambia por un array bidimensional, entonces se vuelve un -dimensional array de forma . La matriz añade una dimensión. La ubicación contiene el valor en . Todas las dimensiones indexadas con escalares se eliminan. Por ejemplo, el resultado de es un array de tamaño . Su i-ésimo elemento es poblado por .

Como parte especial de esta sintaxis, se puede usar la palabra clave para representar el último índice de cada dimensión dentro de los corchetes de indexación, según lo determinado por el tamaño del array más interno indexado. La sintaxis de indexación sin la palabra es equivalente a una llamada a :

|

Example:

|

Los rangos vacío de la forma se suelen usar para indicar la localización inter-index entre y . Por ejemplo, la función usa esta convención para indicar el punto de inserción de un valor no encontrados en un array ordenado:

|

Asignación

La sintaxis general para asignar valores en un array n-dimensional es:

|

donde cada `...` puede ser un índice escalar, un array de enteros o cualquier otro [índice soportado](#). Esto incluye `()` para seleccionar todos los índices dentro de la dimensión completa, rangos de la forma `...` o `...` para seleccionar subsecciones contiguas o con salto, y arrays de booleans para seleccionar elementos en sus índices.

Si `...` es un array, debe tener el mismo número de elementos que el producto de las longitudes de los índices. El valor en la localización de `...` es sobrescrito con el valor `...`. Si `...` no es un array, su valor es escrito a todas las localizaciones referenciadas de `...`.

Justo como en [Indexación](#), la palabra clave `...` puede utilizarse para representar el último índice de cada dimensión dentro de los corchetes de los índices, como queda determinado por el tamaño del array en el que se está siendo asignado. La sintaxis de la asignación indexada sin la palabra clave `...` es equivalente a llamar a la función:

Ejemplo:

Tipos de Índices Soportados

En la expresión `...`, cada `...` puede ser un índice escalar, un array de índices escalares o un objeto que representa un array de índices escalares y puede ser convertido a tal mediante:

1. Un índice escalar. Por defecto esto incluye:

- Enteros no booleanos

* `...`s, que se comportan como una `...`-tupla de enteros abarcando múltiples dimensiones (ver abajo para ms detalles)

2. Un array de índices escalares. Esto incluye:

- Vectores y arrays multidimensionales de enteros

* Arrays vacíos como `...`, que no selecciona elementos * `...`s de la forma `...` o `...`, que seleccionan subsecciones contiguas o con salto desde `...` hasta `...` (inclusive) * Cualquier array de índices escalares que sea un subtipo de * Arrays de `...` (ver abajo para ms detalles)

3. Un objeto que representa un array de índice escalares y puede ser convertido a tal mediante `...`. Por defecto esto incluye:

* `...` (), que representa todos los índices dentro de una dimensión entera o a través del array completo * Arrays de booleans, que seleccionan los elementos en los que sus índices son `...` índices (ver abajo para más detalles)

Índices Cartesianos

El objeto especial `np.ndindex` representa un índice escalar que se comporta como una `n`-tupla de enteros que abarcan múltiples dimensiones. Por ejemplo:

```
np.ndindex(2, 3)
Out[1]: (0, 0)
Out[2]: (0, 1)
Out[3]: (0, 2)
Out[4]: (1, 0)
Out[5]: (1, 1)
Out[6]: (1, 2)
```

Considerado solo, esto puede parecer relativamente trivial; simplemente reúne múltiples enteros juntos en un objeto que representa un único índice multidimensional. Sin embargo, cuando se combina con otras formas de indexación e iteradores que producen es, esto puede conducir directamente a un código muy elegante y eficiente. Ver [Iteración a continuación](#), y para algunos ejemplos más avanzados, ver [esta publicación en el blog sobre algoritmos multidimensionales e iteración](#).

Los `arrays` de `np` también están soportados. Representan una colección de índices escalares que abarcan dimensiones cada uno, lo que permite una forma de indexación que a veces se denomina *indexación puntual*. Por ejemplo, permite acceder a los elementos diagonales desde la primera "página" de 'A' desde arriba:

```
A = np.zeros((2, 3, 4))
A[0, 0, 0] = 1
A[1, 1, 1] = 2
A[0, 2, 2] = 3
A[1, 3, 3] = 4
A
Out[1]: array([[[[ 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.,  0.,  0.],
                [ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.]],
               [[ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.]]],
              ...])
```

Esto se puede expresar mucho más simplemente con [dot broadcasting](#) y combinándolo con un índice entero normal (en lugar de extraer la primera página de como un paso separado). Incluso se puede combinar con `np.ndindex` para extraer ambas diagonales de las dos páginas al mismo tiempo:

```
np.ndindex(2, 3)
Out[1]: (0, 0)
Out[2]: (0, 1)
Out[3]: (0, 2)
Out[4]: (1, 0)
Out[5]: (1, 1)
Out[6]: (1, 2)
```

Warning

y los arrays de no son compatibles con la palabra

clave que representa el último índice de una dimensión. No usaremos cuando se indexen expresiones que puedan contener or arrays de ellos.

Indexación Lógica

A menudo denominada indexación lógica o indexación con una máscara lógica, la indexación mediante una matriz booleana selecciona elementos en los índices cuyos valores son . La indexación por un vector booleano es efectivamente igual a la indexación por el vector de enteros que es devuelto por . De forma similar, la indexación por una matriz booleana -dimensional es efectivamente igual a la indexación por el vector de s donde sus valores son . Un índice lógico debe ser un vector de la misma longitud que la dimensión en la que indexa, o debe ser el único índice proporcionado y debe coincidir con el tamaño y la dimensionalidad de la matriz en la que se indexa. En general, es más eficiente usar matrices booleanas como índices directamente en lugar de llamar primero a .

Iteración

Las formas recomendadas de iterar sobre un array completo son:

La primera construcción se usa cuando necesitamos el valor, pero no los índices, de cada elemento. En la segunda construcción, será un `String` si es un tipo array con indexación lineal rápida; en caso contrario será un `Object`:

```

// Iterar sobre un array de Strings
for (String s : array) {
    // ...
}

// Iterar sobre un array de objetos
for (Object o : array) {
    // ...
}

```

En contraste con `for`, iterar con `forEach` proporciona una forma eficiente de iterar sobre cualquier tipo de array.

Rasgos de Array

Si uno escribe un tipo personalizado, uno puede especificar que el tipo tiene indexación lineal rápida usando:

```

// Tipo personalizado con indexación lineal rápida
public class MiArray {
    // ...
}

```

Esta configuración hará que la iteración `forEach` sobre un objeto use enteros. Si no especifica este rasgo, se usa el valor predeterminado `Object`.

Arrays, Funciones y Operadores Vectorizados

Los siguientes operadores están soportados para arrays:

1. Aritmética unaria - `+`, `-`, `~`, `!`
2. Aritmética binaria - `+`, `-`, `*`, `/`, `%`, `&`, `|`, `<<`, `>>`
3. Comparación - `==`, `!=`, `<`, `>`, `<=`, `>=`, `==>`, `!=>`, `<=>`, `>=>`

La mayoría de los operadores aritméticos binarios enumerados anteriormente también funcionan elemento a elemento cuando un argumento es escalar: `+`, `-`, `*`, `/`, `%`, y cuando cualquiera de los argumentos es escalar, y `<`, `>`, `<=`, `>=` cuando el denominador es escalar. Por ejemplo, `1 + array`, `array * 2`.

Además, para permitir una conveniente vectorización de operaciones matemáticas y de otro tipo, Julia [proporciona la sintaxis punto](#), por ejemplo, `array .* array`, o `array ./ array`, para operaciones con elementos sobre arrays o mezclas de matrices y escalares (una [Retransmisión \(broadcasting\)](#)); estos tienen la ventaja adicional de "fusión" en un solo bucle cuando se combina con otras llamadas de puntos, por ejemplo, `array .* array ./ array`.

También, cada operador binario admite una [versión de punto](#) que se puede aplicar a matrices (y combinaciones de matrices y escalares) en tales [operaciones de retransmisión fusionadas](#), por ejemplo, `array .* array ./ array`.

Tenga en cuenta que las comparaciones como `array < array` operan en arrays completos, dando un solo booleano como respuesta. Use operadores de punto como `array .< array` para comparaciones elemento a elemento. (Para operaciones de comparación como `array < array`, solo la versión de elementos es aplicable a las matrices).

También note la diferencia entre `map`, que retransmite elemento a elemento sobre `y`, y `findmax`, que encuentra el mayor valor dentro de `y`. La misma relación se cumple para `map` y `findmin`.

Retransmisión

A veces es útil realizar operaciones binarias elemento por elemento en matrices de diferentes tamaños, como agregar un vector a cada columna de una matriz. Una forma ineficiente de hacer esto sería replicar el vector al tamaño de la matriz:

```

v = zeros(1000000)
for i = 1:1000000
    v[i] = 1
end

```

Esto es un desperdicio cuando las dimensiones son grandes, por lo que Julia ofrece `broadcast`, que expande las dimensiones *singleton* en los argumentos array para hacer coincidir la dimensión correspondiente en el otro array sin usar memoria extra, y aplicar la función dada elemento a elemento:

```

v = zeros(1000000)
for i = 1:1000000
    v[i] = 1
end

```

Los operadores con punto tales como `.*` y `./` son equivalentes a llamadas a `.*` (excepto que se funden, como se describe a continuación). También hay una función `setindex!` para especificar un destino explícito (al que también se puede acceder por fusión mediante asignación), y funciones `setindex` y `setindex!` que retransmiten los índices antes de indexar. Además, `setindex!` es equivalente a `setindex!`, proporcionando una sintaxis conveniente para retransmitir cualquier función (*sintaxis punto*). "Llamadas punto" anidadas (incluidas las llamadas a `setindex!` Etcétera) *se fusionan automáticamente* en una sola llamada `setindex!`.

Además, `setindex!` no está limitado a los array (ver la documentación de la función), también maneja tuplas y trata cualquier argumento que no sea un array, tupla o `Dict{<T, <V>}` (excepto para `Dict{<T, <V>}`) como un "escalar".

```

v = zeros(1000000)
for i = 1:1000000
    v[i] = 1
end

```

Implementation

El tipo de matriz base en Julia es el tipo abstracto `AbstractMatrix`. Este tipo está parametrizado por el número de dimensiones `N` y el tipo de elementos `T`. `Array{T,N}` y `Matrix{T}` son aliases para los casos 1-d y 2-d. Las operaciones en los objetos `AbstractMatrix` se definen usando operadores y funciones de alto nivel, de manera que es independiente del almacenamiento subyacente. Estas operaciones generalmente funcionan correctamente como una alternativa para cualquier implementación de matriz específica.

El tipo `AbstractMatrix` incluye algo vagamente parecido a un array, y las implementaciones de este podrían ser bastante diferentes de los arrays convencionales. Por ejemplo, los elementos se pueden calcular a petición en lugar de ser almacenados. Sin embargo, cualquier tipo concreto de `AbstractMatrix` debería implementar al menos `getindex` (que devolvería una tupla), `setindex!` y `size`; Los arrays mutables también deberían implementar `push!`. Se recomienda que estas operaciones tengan complejidad temporal casi constante, o técnicamente complejidad de orden 1 ($\mathcal{O}(1)$), ya que de lo contrario algunas funciones podrían ser inesperadamente lentas. Los tipos concretos también deberían proporcionar un método `fill!`, que se utiliza para asignar un conjunto similar para `fill` y otras operaciones de actualización. No importa cómo se represente internamente un `AbstractMatrix`, es el tipo de objeto devuelto por la indización entera (`getindex`), cuando `isempty` no está vacío) y debe ser la longitud de la tupla devuelta por `size`.

`AbstractMatrix` es un subtipo abstracto de `AbstractArray` que pretende incluir todos los arrays que están establecidos en bloques regulares de memoria y que, por tanto, se puede pasar a funciones externas C y Fortran que esperan este diseño de memoria. Los subtipos deberían proporcionar un método `stride` que devuelve el "paso" de la dimensión `i`; incrementar el índice de dimensión `i` en `stride[i]` debería incrementar el índice de `i` en 1. Si se proporciona un método de conversión de puntero `ptr`, el diseño de la memoria debe corresponder de la misma manera a estos pasos.

El tipo `Array{S,2}` es una instancia específica de `AbstractMatrix{S}` donde los elementos se almacenan en orden de columnas principales (consulte notas adicionales en [Sugerencias de rendimiento](#)). `Array{S,1}` y `Array{S,2}` son alias para los casos 1-d y 2-d. Las operaciones específicas como la indexación escalar, la asignación y algunas otras operaciones básicas específicas del almacenamiento son todas las que tienen que estar implementadas en `Array{S,N}`, de modo que el resto de la biblioteca de arrays puede implementarse de forma genérica.

`Array{S,2}` es una especialización de `Array{S,N}` que realiza indexación por referencia en lugar de por copia. Un `Array{S,2}` se crea con la función `Array{S,2}(...)`, que es llamada de la misma manera que `Array{S,N}(...)` (con una matriz y una serie de argumentos de índice). El resultado de `Array{S,2}(...)` se ve igual que el resultado de `Array{S,N}(...)`, excepto que los datos se dejan en su lugar. `Array{S,2}` almacena los vectores de índice de entrada en un objeto `Array{S,2}`, que luego puede usarse para indexar la matriz original de forma indirecta. Al colocar la macro `@inplace` delante de una expresión o bloque de código, cualquier segmento `Array{S,2}` en esa expresión se convertirá para crear una vista `Array{S,2}` en su lugar.

`Array{S,2}` y `Array{S,1}` son alias convenientes definidos para que Julia pueda llamar a un rango más amplio de funciones BLAS y LAPACK pasándoles objetos `Array{S,2}` o `Array{S,1}`, y ahorrando así ineficiencias de asignación de memoria y copia.

El siguiente ejemplo calcula la descomposición QR de una pequeña sección de una matriz más grande, sin crear ningún array temporal, y llamando a la función LAPACK apropiada con los parámetros de tamaño de dimensión y salto correctos.

22.2 Vectores y Matrices *Sparse*

Julia tiene soporte integrado para vectores y **matrices dispersas (*sparse*)**. Las matrices *sparse* son matrices que contienen suficientes ceros para almacenarlos en una estructura de datos especial que ahorra espacio y tiempo de ejecución, en comparación con las matrices densas.

Columna Comprimida *Sparse* (CSC) Para Almacenamiento de Matrices *Sparse*

En Julia, las matrices dispersas se almacenan en el formato **Compressed Sparse Column (CSC)**. Las matrices *sparse* de Julia tienen el tipo `Matrix{<T>}`, donde `<T>` es el tipo de los valores almacenados, y `Matrix{<T>}` es el tipo entero para almacenar punteros de columnas e índices de filas. La representación interna de `Matrix{<T>}` es la siguiente:

El almacenamiento de columnas dispersas y comprimidas (CSC) facilita y agiliza el acceso a los elementos en la columna de una matriz *sparse*, mientras que el acceso a la matriz *sparse* por filas es considerablemente más lento. Las operaciones como la inserción de entradas previamente no almacenadas de una en una en la estructura de CSC tienden a ser lentas. Esto se debe a que todos los elementos de la matriz *sparse* que están más allá del punto de inserción deben moverse un lugar más.

Todas las operaciones en matrices *sparse* se implementan cuidadosamente para aprovechar la estructura de datos CSC para el rendimiento y para evitar operaciones costosas.

Si tiene datos en formato CSC desde una aplicación o biblioteca diferente, y desea importarlos a Julia, asegúrese de utilizar la indexación basada en 1. Los índices de fila en cada columna deben estar ordenados. Si su objeto contiene índices de filas sin ordenar, una forma rápida de ordenarlos es hacer una doble transposición.

En algunas aplicaciones, es conveniente almacenar valores cero explícitos en una `Matrix{C}`. Estas *son* aceptadas por funciones en `LinearAlgebra` (pero no hay garantía de que se conservarán en las operaciones de mutación). Tales ceros explícitamente almacenados son tratados como no estructurales por muchas rutinas. La función `nnz` devuelve la cantidad de elementos almacenados explícitamente en la estructura de datos dispersos, incluidos los no-ceros estructurales. Para contar el número exacto de no-ceros numéricos, use `nnznonzero`, que inspecciona todos los elementos almacenados en una matriz `sparse`. `nnz` y `nnznonzero` se pueden usar para eliminar ceros almacenados de la matriz dispersa.

Almacenamiento de Vectores *Sparse*

Los vectores `sparse` se almacenan en un formato de columna análogo al que se usa en las matrices `sparse`. En Julia, los vectores `sparse` tienen el tipo `SpVec{C}` donde `C` es el tipo de los valores almacenados y el tipo entero para los índices. La representación interna es la siguiente:

En cuanto a `SpVec{C}`, el tipo también puede contener ceros almacenados explícitamente. (Consulte [Almacenamiento de matriz `sparse`](#)).

Constructores de Vectores y Matrices *Sparse*

La forma más sencilla de crear matrices `sparse` es usar funciones equivalentes a las funciones `zeros` y `ones` que proporciona Julia para trabajar con matrices densas. Para producir matrices `sparse` en su lugar, puede usar los mismos nombres con el prefijo `sparse_`:

La función `sparse_csc` suele ser una forma útil de construir arrays `sparse`. Por ejemplo, para construir una matriz `sparse`, podemos ingresar un vector `row_idxs` de índices de fila, un vector `col_idxs` de índices de columna, y un vector `vals` de valores almacenados (esto

también se conoce como **formato COO (coordenada)**). construye una matriz *sparse* tal que . El constructor de vector *sparse* equivalente es , que toma el vector de índices (fila) y el vector con los valores almacenados y construye un vector *sparse* tal que .

La inversa de las funciones y es , que recupera las entradas utilizadas para crear el array *sparse*. También hay una función que solo devuelve los vectores índice.

Otra forma de crear un array *sparse* es convertir un array denso en un array *sparse* usando la función :

Puede ir en la otra dirección usando el constructor . La función se puede usar para consultar si una matriz es o no *sparse*.

Operaciones con matrices *sparse*

Las operaciones aritméticas en matrices *sparse* también funcionan como lo hacen en matrices densas. La indexación de, la asignación en y la concatenación de matrices *sparse* funcionan de la misma manera que las matrices densas. Las operaciones de indexación, especialmente la asignación, son costosas, cuando se llevan a cabo un elemento cada vez. En muchos casos, puede ser mejor convertir la matriz dispersa en formato usando , manipular los valores o la estructura en los vectores densos , y luego reconstruir la matriz *sparse*.

Correspondence of dense and sparse methods

La siguiente tabla proporciona una correspondencia entre los métodos incorporados en matrices *sparse* y sus métodos correspondientes en tipos de matrices densas. En general, los métodos que generan matrices *sparse* difieren de sus contrapartes densas en que la matriz resultante sigue el mismo patrón de dispersión que una matriz *sparse* dada , o que la matriz *sparse* resultante tiene densidad , es decir, cada elemento de matriz tiene una probabilidad de ser diferente de cero.

Los detalles se pueden encontrar en la sección [Vectores y Matrices Sparse](#) de la referencia de biblioteca estándar.

Sparse	Dense	Description
		Creates a <i>m</i> -by- <i>n</i> matrix of zeros. (is empty.)
		Creates a matrix filled with ones. Unlike the dense version, has the same sparsity pattern as <i>S</i> .
		Creates a <i>n</i> -by- <i>n</i> identity matrix.
		Interconverts between dense and sparse formats.
		Creates a <i>m</i> -by- <i>n</i> random matrix (of density <i>d</i>) with iid non-zero elements distributed uniformly on the half-open interval $[0, 1)$.
		Creates a <i>m</i> -by- <i>n</i> random matrix (of density <i>d</i>) with iid non-zero elements distributed according to the standard normal (Gaussian) distribution.
		Creates a <i>m</i> -by- <i>n</i> random matrix (of density <i>d</i>) with iid non-zero elements distributed according to the <i>X</i> distribution. (Requires the package.)

Chapter 23

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 `*`, `+`, and `inv` are all supported:

As well as other useful operations, such as finding eigenvalues or eigenvectors:

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 [for more information](#). As an example:

```

A = randn(1000, 1000)
t = @time solve(A, ones{Float64, 1000})

```

Since `A` is not Hermitian, symmetric, triangular, tridiagonal, or bidiagonal, an LU factorization may be the best we can do. Compare with:

```

A = randn(1000, 1000)
t = @time solve(A, ones{Float64, 1000})

```

Here, Julia was able to detect that `A` is in fact symmetric, and used a more appropriate factorization. Often it's possible to write more efficient code for a matrix that is known to have certain properties e.g. it is symmetric, or tridiagonal. Julia provides some special types so that you can "tag" matrices as having these properties. For instance:

```

A = randn(1000, 1000)
t = @time solve(A, ones{Float64, 1000})

```

`A` 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:

```

A = randn(1000, 1000)
t = @time solve(A, ones{Float64, 1000})

```

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 `lu_solve!` while `A' \ b` will end up calling `lu_solve!`, even though we used the same left-division operator. This works for matrices too: `A \ b` would call `lu_solve!`. 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.

23.1 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 for them in LAPACK are available.

Type	Description
	Hermitian matrix
	Upper triangular matrix
	Lower triangular matrix
	Tridiagonal matrix
	Symmetric tridiagonal matrix
	Upper/lower bidiagonal matrix
	Diagonal matrix
	Uniform scaling operator

Elementary operations

Legend:

Matrix factorizations

Legend:

Matrix type					Other functions with optimized methods
				MV	, ,
			MV	MV	,
			MV	MV	,
	M	M	MS	MV	,
	M	M	MS	MV	
	M	M	MS	MV	
	M	M	MV	MV	, , ,
	M	M	MVS	MVS	

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 type	LAPACK					
	HE		ARI			
	TR	A	A	A		
	TR	A	A	A		
	ST	A	ARI	AV		
	GT					
	BD				A	A
	DI		A			

The uniform scaling operator

A operator represents a scalar times the identity operator, `Scalar * I`. The identity operator `I` is defined as a constant and is an instance of `UniformScaling{T}`. The size of these operators are generic and match the other matrix in the binary operations `*`, `*`, and `*`. For `*` and `*` 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.

23.2 Matrix factorizations

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.

Key	Description	Example
A (all)	An optimized method to find all the characteristic values and/or vectors is available	e.g.
R (range)	An optimized method to find the i th through the j th characteristic values are available	
I (interval)	An optimized method to find the characteristic values in the interval $[a, b]$ is available	
V (vectors)	An optimized method to find the characteristic vectors corresponding to the characteristic values λ_i is available	

Type	Description
	Cholesky factorization
	Pivoted Cholesky factorization
	LU factorization
	LU factorization for $n \times n$ matrices
	LU factorization for sparse matrices (computed by UMFPack)
	QR factorization
	Compact WY form of the QR factorization
	Pivoted QR factorization
	Hessenberg decomposition
	Spectral decomposition
	Singular value decomposition
	Generalized SVD

Chapter 24

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.

24.1 Basic Stream I/O

All Julia streams expose at least a `write` and a `read` method, taking the stream as their first argument, e.g.:

```
using Base.Streams
using Base.TCPSocket
using Base.Processes

s = TCPSocket{128}()
write(s, "Enter\n")
r = read(s, 10)
println(r)
```

Note that `write` returns 11, the number of bytes (in `UInt8`) written to `s`, but this return value is suppressed with the `println`.

Here Enter was pressed again so that Julia would read the newline. Now, as 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:

```
using Base.Streams
using Base.TCPSocket
using Base.Processes

s = TCPSocket{128}()
write(s, "Enter\n")
r = read(s, Vector{UInt8}, 10)
println(r)
```

However, since this is slightly cumbersome, there are several convenience methods provided. For example, we could have written the above as:

```
|
```

or if we had wanted to read the entire line instead:

```
|
```

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 :

```
|
```

or if you wanted to read by character instead:

```
|
```

24.2 Text I/O

Note that the `read` 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:

```
|
```

Note that `read` is written to `io` by the `read` function and that the returned value is `Char` (since `read` is one byte).

For text I/O, use the `readline` or `readlines` methods, depending on your needs (see the standard library reference for a detailed discussion of the difference between the two):

```
|
```

24.3 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, `IOContext` adds a hinting parameter to the IO object that the invoked `show` method should print a shorter output (if applicable).

24.4 Working with Files

Like many other environments, Julia has an `open` function, which takes a filename and returns an `IO` object that you can use to read and write things from the file. For example if we have a file, `example.txt`, whose contents are :

```
|
```

If you want to write to a file, you can open it with the `write` flag:

```
|
```

If you examine the contents of `example.txt` at this point, you will notice that it is empty; nothing has actually been written to disk yet. This is because the `IO` object must be closed before the write is actually flushed to disk:

```
|
```

Examining `example.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:

```
|
```

You can call:

```
|
```

to open `example.txt`, call `capitalize`, close 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:

```
|
```

24.5 A simple TCP example

Let's jump right in with a simple example involving TCP sockets. Let's first create a simple server:

```
|
```

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:

```
|
```

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. The difference 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 connects to a server using the specified method. The `connect` function takes the same arguments as `accept`, 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):

```
|
```

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 callbacks or even making sure that the server gets to run. When we called `wait`, 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:

```
|
```

As with other streams, use `close` to disconnect the socket:

```
|
```

24.6 Resolving IP Addresses

One of the methods that does not follow the `Base` methods is `connect`, 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:

```
|
```

At the base of this functionality is , which will do the appropriate address resolution:

|

Chapter 25

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 MPI¹. 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 *remote calls*. A remote reference is an object that can be used from any 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: `RemoteRef{T}` and `RemoteRef{T, S}`.

A remote call returns a `RemoteCall{T}` 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 `RemoteCall`, and you can obtain the full value of the result using `get`.

On the other hand, `RemoteRef`s are rewritable. For example, multiple processes can co-ordinate their processing by referencing the same remote `RemoteRef`.

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 `addprocs` provides `n` worker processes on the local machine. Generally it makes sense for `n` to equal the number of CPU cores on the machine.

The first argument to `@spawn` is the function to call. Most parallel programming in Julia does not reference specific processes or the number of processes available, but `@spawn` is considered a low-level interface providing finer control. The second argument to `@spawn` is the `Process` 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, `f1` and `f2`. The `@spawn` 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 `fetch` exists for this purpose. It is equivalent to `getindex` but is more efficient.

Remember that `fetch` is [equivalent](#) to `getindex`, so this call fetches the first element of the future `f1`.

The syntax of `@spawn` is not especially convenient. The macro `@spawnat` makes things easier. It operates on an expression rather than a function, and picks where to do the operation for you:

Note that we used `@spawnat` instead of `@spawn`. This is because we do not know where the code will run, so in general a `Future` might be required to move to the process doing the addition. In this case, `@spawnat` is smart enough to perform the computation on the process that owns `f1`, so the `fetch` will be a no-op (no work is done).

(It is worth noting that `@spawnat` 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 `Future`s have fetched, the remote stored value is deleted.

25.1 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:

Process 1 knew about the function `f`, 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 `foo.jl`, containing the following code:

Starting Julia with `Julia --pthreads`, you can use this to verify the following:

- `using foo` loads the file on just a single process (whichever one executes the statement).
- `using foo, process=2` 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 `foo` is loaded on process 2, commands like

```
foo.f(2)
```

allow you to store an object of type `foo.F` on process 2 even if `foo` is not in scope on process 2.

You can force a command to run on all processes using the `@eval` macro. For example, `@eval foo.f(2)` can also be used to directly define a function on all processes:

A file can also be preloaded on multiple processes at startup, and a driver script can be used to drive the computation:

The Julia process running the driver script in the example above has an `nprocs` 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 local cluster specified with the `local` option as shown above.
- A cluster spanning machines using the `ssh` 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`, 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 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.

25.2 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.

`ccall` can be considered an explicit data movement operation, since it directly asks that an object be moved to the local machine. `ccall` (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:

```
|
```

Method 2:

```
|
```

The difference seems trivial, but in fact is quite significant due to the behavior of `ccall`. 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 then the first method might be better. Or, if computing 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 and , it might be better to eliminate the parallelism altogether. Or imagine is replaced with a more expensive operation. Then it might make sense to add another statement just for this step.

Chapter 26

Global variables

Expressions executed remotely via `remote`, or closures specified for remote execution using `remote` 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:

```
|
```

Note that `__main__` is a global variable defined in the local workspace. Worker 2 does not have a variable called `__main__`. The act of shipping the closure `remote` to worker 2 results in `__main__` being defined on 2. `__main__` continues to exist on worker 2 even after the call `remote` 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 referenced as part of a remote call.
- 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:

```
|
```

Executing the above snippet results in `__main__` on worker 2 having a different value from `__main__` on worker 3, while the value of `__main__` on node 1 is set to `__main__`.

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. `__main__` can be used to manually reassign specific globals on remote nodes to `__main__` 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 `__main__` blocks to localize global variables.

For example:

As can be seen, global variable `is` is defined on worker 2, but `is` is captured as a local variable and hence a binding for `is` does not exist on worker 2.

26.1 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 `fork()` to flip coins on two processes. First, write the following function in :

The function simply adds together random bits. Here is how we can perform some trials on two machines, and add together the results:

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 `for`, where `acc` is the accumulator, `f` is the reduction function, and the `xs` 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.

Notice that our use of this pattern with `for` can be generalized. We used two explicit `for` 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:

```
|
```

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:

```
|
```

This code will not initialize all of `xs`, 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:

```
|
```

Using "outside" variables in parallel loops is perfectly reasonable if the variables are read-only:

```
|
```

Here each iteration applies `f` to a randomly-chosen sample from a vector `xs` 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{T}` immediately without waiting for completion. The caller can wait for the completions at a later point by calling `wait` on them, or wait for completion at the end of the loop by prefixing it with `@sync`, like `@sync 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 `map` function. For example, we could compute the singular values of several large random matrices in parallel as follows:

```
|
```

Julia's `map` is designed for the case where each function call does a large amount of work. In contrast, `mapreduce` can handle situations where each iteration is tiny, perhaps merely summing two numbers. Only worker processes are used by both `map` and `mapreduce` for the parallel computation. In case of `mapreduce`, the final reduction is done on the calling process.

26.2 Synchronization With Remote References

26.3 Scheduling

Julia's parallel programming platform uses [Tasks \(aka Coroutines\)](#) to switch among multiple computations. Whenever code performs a communication operation like `send` or `recv`, 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:

```
|
```

If one process handles both 800×800 matrices and another handles both 600×600 matrices, we will not get as much scalability as we could. The solution is to make a local task to "feed" work to each process when it completes its current task. For example, consider a simple `map` implementation:

```
|
```


is similar to `waitgroup`, 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 needs to be computed, then waits for its process to finish, then repeats until we run out of indexes. Note that the feeder tasks do not begin to execute until the main task reaches the end of the `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 `block` 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 `block` is called.

26.4 Channels

The section on `s` in [Control Flow](#) discussed the execution of multiple functions in a co-operative manner. `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 `Write` calls.
- Multiple readers in different tasks can read data concurrently via `Read` calls.
- As an example:

- Channels are created via the `makeChannel` constructor. The channel will only hold objects of type `T`. If the type is not specified, the channel can hold objects of any type. `cap` refers to the maximum number of elements that can be held in the

channel at any time. For example, `makeChannel(32)` creates a channel that can hold a maximum of 32 objects of any type. A channel can hold up to 64 objects of any type at any time.

- If a channel is empty, readers (on a `recv` call) will block until data is available.
- If a channel is full, writers (on a `send` call) will block until space becomes available.
- `hasData` tests for the presence of any object in the channel, while `waitData` waits for an object to become available.
- A channel is in an open state initially. This means that it can be read from and written to freely via `recv` and `send` calls. `close` closes a channel. On a closed channel, `recv` will fail. For example:

```
|
```

- `recvAll` and `waitData` (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:

```
|
```

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:

```
|
```

while this will return after reading all data:

Consider a simple example using channels for inter-task communication. We start 4 tasks to process data from a single channel. Jobs, identified by an id (), are written to the channel. Each task in this simulation reads a , waits for a random amount of time and writes back a tuple of and the simulated time to the results channel. Finally all the are printed out.

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.

26.5 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`, `isfull`, and `isempty`. The remote object referred to by a `RemoteRef` is stored in a `RemoteRef`, i.e., a `Vector{Any}` of size 1 capable of holding objects of `Any` type.

`RemoteRef`, which is rewritable, can point to any type and size of channels, or any other implementation of an `AbstractChannel`.

The constructor `RemoteRef{T}` allows us to construct references to channels holding more than one value of a specific type `T`. `RemoteRef{T}` is a function executed on `RemoteRef{T}` and it must return an `AbstractChannel{T}`.

For example, `RemoteRef{Int}` will return a reference to a channel of type `Int` and size 10. The channel exists on worker 1.

Methods `put`, `take`, `isfull`, and `isempty` on a `RemoteRef` are proxied onto the backing store on the remote process.

`RemoteRef` can thus be used to refer to user implemented objects. A simple example of this is provided in `remote_ref.jl` which uses a dictionary as its remote store.

26.6 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, `id`, associated with a `RemoteChannel` identifies the process where the backing store, i.e., the backing `Channel` exists.

- Any process with a reference to a `Channel` can put and take items from the channel. Data is automatically sent to (or retrieved from) the process a `Channel` 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 `Channel` only involves the serialization of an identifier that identifies the location and instance of `Channel` referred to by the handle. A deserialized `Channel` 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 remote channel. Jobs, identified by an id (), are written to the channel. Each remotely executing task in this simulation reads a `Job`, waits for a random amount of time and writes back a tuple of `Job`, time taken and its own `id` to the results channel. Finally all the `Jobs` are printed out on the master process.

|

26.7 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 `Ref` or a `Ref` (unfetched) is serialized to a worker, the node pointed to by the reference is notified. And every time a `Ref` or a `Ref` (unfetched) is garbage collected locally, the node owning the value is again notified.

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 `Refs` are write-once and cached locally, the act of `Ref`ing a `Ref` also updates reference tracking information on the node owning the value.

The node which owns the value frees it once all references to it are cleared.

With `Refs`, serializing an already fetched `Ref` 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 `Ref` on local instances of a `Ref`, or on unfetched `Refs`. Since calling `Ref` on a `Ref` also removes its reference from the remote store, this is not required on fetched `Refs`. Explicitly calling `Ref` 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.

26.8 Shared Arrays

Shared Arrays use system shared memory to map the same array across many processes. While there are some similarities to a `TypedArray`, the behavior of a `SharedArray` is quite different. In a `TypedArray`, 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.

Indexing (assignment and accessing values) works just as with regular arrays, and is efficient because the underlying memory is available to the local process. Therefore, most algorithms work naturally on `SharedArrays`, 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 `ArrayBuffer.prototype.slice`. For other types, `ArrayBuffer.prototype.slice` just returns the object itself, so it's safe to use `ArrayBuffer.prototype.slice` on any `ArrayBuffer`-type object.

The constructor for a shared array is of the form:

```
|
```

which creates an `n`-dimensional shared array of a `bits` type and size `length` across the processes specified by `workers`. Unlike distributed arrays, a shared array is accessible only from those participating workers specified by the `workers` named argument (and the creating process too, if it is on the same host).

If an `init` function, of signature `init(index)`, 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:

```
|
```

`ArrayBuffer.prototype.slice` 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:

```
|
```

Since all processes have access to the underlying data, you do have to be careful not to set up conflicts. For example:

```
|
```

would result in undefined behavior. Because each process fills the *entire* array with its own , whichever process is the last to execute (for any particular element of) will have its retained.

As a more extended and complex example, consider running the following "kernel" in parallel:

```
|
```

In this case, if we try to split up the work using a one-dimensional index, we are likely to run into trouble: if is near the end of the block assigned to one worker and is near the beginning of the block assigned to another, it's very likely that will not be ready at the time it's needed for computing . In such cases, one is better off chunking the array manually. Let's split along the second dimension. Define a function that returns the indexes assigned to this worker:

```
|
```

Next, define the kernel:

```
|
```


We also define a convenience wrapper for a `map` implementation

```
|
```

Now let's compare three different versions, one that runs in a single process:

```
|
```

one that uses :

```
|
```

and one that delegates in chunks:

```
|
```

If we create `s` and time these functions, we get the following results (with `%timeit`):

Run the functions once to JIT-compile and them on the second run:

The biggest advantage of `SharedArrays` is that it minimizes traffic among the workers, allowing each to compute for an extended time on the assigned piece.

26.9 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.

26.10 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:

- The initial Julia process, also called the `master`, is special and has an `id` 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 `WorkerOptions` object.
- `addprocs` calls the appropriate `spawn` 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 `tcp`, it is possible for a Julia cluster to provide its own transport.

Julia provides two in-built cluster managers:

- `addprocs`, used when `addprocs` or `addprocs!` are called
- `addprocs_with`, used when `addprocs_with` is called with a list of hostnames

`addprocs` 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 `spawn`, a method responsible for launching new workers
- implement `kill`, which is called at various events during a worker's lifetime (for example, sending an interrupt signal)

`addprocs` requires `ClusterManager` to implement:

|

As an example let us see how the `LocalClusterManager`, the manager responsible for starting workers on the same host, is implemented:

|

The `launch` method takes the following arguments:

- `cm` : the cluster manager that `launch` is called with
- `kwargs` : all the keyword arguments passed to `launch`
- `args` : the array to append one or more `Worker` objects to
- `done` : 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` argument results in the launched processes initializing themselves as workers and connections being set up via TCP/IP sockets. Optionally, `enable` may also be specified to enable other workers to connect to it at the specified `port` and `host`. 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 `enable` must NOT be specified. Instead, newly launched workers should call `mpi_init` before using any of the parallel constructs.

For every worker launched, the `launch` method must add a `Worker` object (with appropriate fields initialized) to

Most of the fields in `Worker` are used by the inbuilt managers. Custom cluster managers would typically specify only `cm` or `kwargs` :

- If `bind` 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 `bind` is not specified, `host` and `port` are used to connect.
- `workers`, `cores`, and `cmd` 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.
 - `workers` with an integer value will launch a total of `workers` workers.
 - `cores` with a value of `cores` will launch as many workers as the number of cores on that machine.
 - `cmd` is the name of the executable including the full path.
 - `args` should be set to the required command line arguments for new workers.
- `ssh`, `ssh_host`, and `ssh_port` are used when a ssh tunnel is required to connect to the workers from the master process.
- `custom_manager` is provided for custom cluster managers to store their own worker-specific information.

`add_worker` is called at different times during the worker's lifetime with appropriate `values`:

- with `add` / `remove` when a worker is added / removed from the Julia worker pool.
- with `kill` when `kill` is called. The `kill` should signal the appropriate worker with an interrupt signal.
- with `cleanup` for cleanup purposes.

26.11 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 `Object` (for example, a `Condition` in the default implementation), reads an entire message, processes it and waits for the next one.
- Sending messages to a process is done directly from any Julia task—not just communication tasks—again, via the appropriate `Object`.

Replacing the default transport requires the new implementation to set up connections to remote workers and to provide appropriate `Object`s that the message-processing loops can wait on. The manager-specific callbacks to be implemented are:

|

The default implementation (which uses TCP/IP sockets) is implemented as `TransportTCP`.

`TransportTCP` should return a pair of `IOStream` objects, one for reading data sent from worker, and the other to write data that needs to be sent to worker. Custom cluster managers can use an in-memory `Transport` as the plumbing to proxy data between the custom, possibly non-transport and Julia's in-built parallel infrastructure.

`TransportMemory` is an in-memory `Transport` which behaves like an `IOStream`—it is a stream which can be handled asynchronously.

Folder `examples/zeromq` contains an example of using ZeroMQ to connect Julia workers in a star topology with a OMQ 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 OMQ being used as the transport layer.

When using custom transports:

- Julia workers must NOT be started with `addprocs`. Starting with `addprocs` will result in the newly launched workers defaulting to the TCP/IP socket transport implementation.
- For every incoming logical connection with a worker, `accept_worker` must be called. This launches a new task that handles reading and writing of messages from/to the worker represented by the `IOStream` objects.
- `accept_worker` MUST be called as part of worker process initialization.
- Field `info` in `WorkerInfo` can be set by the cluster manager when `addprocs` is called. The value of this field is passed in in all `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 tuple at which to connect to a worker.

`remove_worker` is called to remove a worker from the cluster. On the master process, the corresponding `IOStream` objects must be closed by the implementation to ensure proper cleanup. The default implementation simply executes an `close` call on the specified remote worker.

`examples/unixsockets` is an example that shows a simple implementation using UNIX domain sockets for cluster setup.

26.12 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`:

- The master process does not listen on any port. It only connects out to the workers.
- Each worker binds to only one of the local interfaces and listens on the first free port starting from `10000`.
- `LocalManager`, used by `addprocs`, 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` followed by an `addprocs` 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 `bind` keyword argument: `addprocs(...; bind="0.0.0.0")`.
- `SSHManager`, used by `addprocs`, 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 `ssh_options`.
- `SSHManager` 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 `ssh_options`, for example `ssh_options = Dict{String, String}()`.

Note that worker-worker connections are still plain TCP and the local security policy on the remote cluster must allow for free connections between worker nodes, at least for ports 9009 and above.

Securing and encrypting all worker-worker traffic (via SSH) or encrypting individual messages can be done via a custom ClusterManager.

26.13 Cluster Cookie

All processes in a cluster share the same cookie which, by default, is a randomly generated string on the master process:

- `get_cookie()` returns the cookie, while `set_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 must be passed to the workers at startup via argument `--cluster-cookie`. Custom ClusterManagers can retrieve the cookie on the master by calling `get_cookie()`. Cluster managers not using the default TCP/IP transport (and hence not specifying `transport`) must call `set_cookie()` 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.

26.14 Specifying Network Topology (Experimental)

The keyword argument `cluster_topology` passed to `ClusterManager` is used to specify how the workers must be connected to each other:

- `all`, the default: all workers are connected to each other.
- `driver`: only the driver process, i.e. 1, has connections to the workers.
- `custom`: the `method` of the cluster manager specifies the connection topology via the fields `cluster_topology` and `cluster_topology_kwargs` in `ClusterManager`. A worker with a cluster-manager-provided identity `id` will connect to all workers specified in `cluster_topology[id]`.

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.

26.15 Multi-Threading (Experimental)

In addition to tasks, remote calls, and remote references, Julia from `1.4` forwards will natively support multi-threading. Note that this section is experimental and the interfaces may change in the future.

Setup

By default, Julia starts up with a single thread of execution. This can be verified by using the command :

```
|
```

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:

```
|
```

(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 `setenv`. If you're on Windows, start up the command line in the location of `cmd` and use `set` instead of `setenv`.)

Let's verify there are 4 threads at our disposal.

```
|
```

But we are currently on the master thread. To check, we use the command

```
|
```

The Macro

Let's work a simple example using our native threads. Let us create an array of zeros:

```
|
```

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 `@parallel` macro. This macro is affixed in front of a `for` loop to indicate to Julia that the loop is a multi-threaded region:

```
|
```

The iteration space is split amongst the threads, after which each thread writes its thread ID to its assigned locations:

```
|
```


Note that `@threadcall` does not have an optional reduction parameter like `@mpi`.

26.16 @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 `@threadcall` explicitly allows for other tasks to be scheduled.

Thus, a task executing a `@threadcall` 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 `jlcall` (C equivalent of `ccall`).

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 `@threadcall` 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 `JL_NUM_THREADS`. While waiting for a free thread, and during function execution 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.

¹In 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 27

Date and DateTime

The `java.time` module provides two types for working with dates: `LocalDate` and `LocalDateTime`, representing day and millisecond precision, respectively; both are subtypes of the abstract `TemporalAccessor`. 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 `LocalDate` 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 `LocalDate` and `LocalDateTime` are basically immutable wrappers. The single `Instant` field of either type is actually a `TemporalAccessor` type, which represents a continuously increasing machine timeline based on the UT second¹. The `LocalDate` 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 `LocalDate` and `LocalDateTime` 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 no year zero exists. The ISO standard, however, states that 1 BC/BCE is year zero, so `-1` is the day before `0`, and year `-1` (yes, negative one for the year) is 2 BC/BCE, year `-2` is 3 BC/BCE, etc.

27.1 Constructors

`LocalDate` and `LocalDateTime` types can be constructed by integer or `TemporalAccessor` types, by parsing, or through adjusters (more on those later):



¹The 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 `LocalDate` and `LocalDateTime` 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.

or 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 or constructor, of the form or .

Delimited slots are marked by specifying the delimiter the parser should expect between two subsequent periods; so lets the parser know that between the first and second slots in a date string like , it should find the character. The , , and characters let the parser know which periods to parse in each slot.

Fixed-width slots are specified by repeating the period character the number of times corresponding to the width with no delimiter between characters. So would correspond to a date string like . The parser distinguishes a fixed-width slot by the absence of a delimiter, noting the transition from one period character to the next.

Support for text-form month parsing is also supported through the and characters, for abbreviated and full-length month names, respectively. By default, only English month names are supported, so corresponds to "Jan", "Feb", "Mar", etc. And corresponds to "January", "February", "March", etc. Similar to other name=>value mapping functions and , custom locales can be loaded by passing in the mapping to the and dicts for abbreviated and full-name month names, respectively.

One note on parsing performance: using the 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 , and pass it instead of a raw format string.

You can also use the string macro. This macro creates the object once when the macro is expanded and uses the same object even if a code snippet is run multiple times.

A full suite of parsing and formatting tests and examples is available in .

27.2 Durations/Comparisons

Finding the length of time between two `Instant` or `OffsetDateTime` is straightforward given their underlying representation as `long` and `int`, respectively. The difference between `Instant` is returned in the number of `seconds`, and `OffsetDateTime` in the number of `seconds`. Similarly, comparing `Instant` is a simple matter of comparing the underlying machine instants (which in turn compares the internal `long` values).

27.3 Accessor Functions

Because the `date` and `datetime` types are stored as single values, date parts or fields can be retrieved through accessor functions. The lowercase accessors return the field as an integer:

While propercase return the same value in the corresponding type:

Compound methods are provided, as they provide a measure of efficiency if multiple fields are needed at the same time:

One may also access the underlying `year` or integer value:

27.4 Query Functions

Query functions provide calendrical information about a . They include information about the day of the week:

Month of the year:

As well as information about the 's year and quarter:

The and methods can also take an optional 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 and . First the mapping is loaded into the variable:

The above mentioned functions can then be used to perform the queries:

Since the abbreviated versions of the days are not loaded, trying to use the function will error.

27.5 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 module approach tries to follow the simple principle of trying to change as little as possible when doing 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 module, it gives the result of February 28th. How does it figure that 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, . 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:

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 means `add` will always be added first, then `subtract`, then `add`, etc. Hence the following *does* result in associativity and Just Works:

Tricky? Perhaps. What is an innocent 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:

27.6 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 `dateutil` 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 `date` as input and return or *adjust* to the first or last of the desired period relative to the input.

The next two higher-order methods, `date_range` and `datetime_range`, generalize working with temporal expressions by taking a `start` as first argument, along with a starting `date`. `A` is just a function, usually anonymous, that takes a single `date` as input and returns a `bool`, indicating a satisfied adjustment criterion. For example:

This is useful with the `do-block` syntax for more complex temporal expressions:

The `date_range` method can be used to obtain all valid dates/moments in a specified range:

Additional examples and tests are available in .

27.7 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. types are simple wrappers and are constructed by wrapping any convertible type, i.e. or . Arithmetic between of the same type behave like integers, and limited arithmetic is available.

27.8 Rounding

and values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with , , or :

Chapter 28

Interacting With Julia

Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the 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> 1 + 2
2
```

To exit the interactive session, type `Ctrl-C` – the control key together with the `C` key on a blank line – or type `exit` followed by the return or enter key. The REPL greets you with a banner and a `julia>` prompt.

28.1 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> 1 + 2
2
```

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> 1 + 2;
julia> 1 + 2
2
```

Help mode

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.

Search modes

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 `~` – the control key together with the `key`. The prompt will change to `~`, 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 `again`.

Just as `~` is a reverse search, `!` is a forward search, with the prompt `!`. The two may be used in conjunction with each other to move through the previous or next matching results, respectively.

28.2 Key bindings

The Julia REPL makes great use of key bindings. Several control-key bindings were already introduced above (`q` to exit, and `~` 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 `set_keymap!`. The keys of this dictionary may be characters or strings. The key `default` refers to the default action. Control plus character bindings are indicated with `C-`. Meta plus `key` can be written `M-key`. The values of the custom keymap must be `nothing` (indicating that the input should be ignored) or functions that accept the signature `(key::AbstractChar, repl::REPLState)`. The function must be called before the REPL is initialized, by registering the operation with `add_custom_keymap!`. For example, to bind the up and down arrow keys to move through history without prefix search, one could put the following code in `~/.julia/replkeymap.jl`:

Users should refer to `REPLState` to discover the available actions on key input.

28.3 Tab completion

In both the Julia 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:

Keybinding	Description
Program control	
	Exit (when buffer is empty)
	Interrupt or cancel
	Clear console screen
Return/Enter,	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)
,	Incremental history search, described above
Cursor movement	
Right arrow,	Move right one character
Left arrow,	Move left one character
Home,	Move to beginning of line
End,	Move to end of line
	Change to the previous or next history entry
	Change to the next history entry
Up arrow	Move up one line (or to the previous history entry)
Down arrow	Move down one line (or to the next history entry)
Page-up	Change to the previous history entry that matches the text before the cursor
Page-down	Change to the next history entry that matches the text before the cursor
	Move right one word
	Move left one word
Editing	
Backspace,	Delete the previous character
Delete,	Forward delete one character (when buffer has text)
meta-Backspace	Delete the previous word
	Forward delete the next word
	Delete previous text up to the nearest whitespace
	"Kill" to end of line, placing the text in a buffer
	"Yank" insert the text from the kill buffer
	Transpose the characters about the cursor
	Write a number in REPL and press to open editor at corresponding stackframe

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:


```
|
```

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:

```
|
```

Tab completion can help with investigation of the available methods matching the input arguments:

```
|
```

Keywords are also displayed in the suggested methods, see second line after `where` and `and` are keyword arguments:

```
|
```

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:

```
|
```

Fields for output from functions can also be completed:

```
|
```

The completion of fields for output from functions uses type inference, and it can only suggest fields if the function is type stable.

28.4 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 `file`, which is to be placed inside your home directory:

```
|
```

The available color keys can be seen by typing `?` 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 `ENV` in the `ENV` function above (respectively, `help`, `shell`, `input`, and `answer`). For the latter two, be sure that the `bold` field is also set to false.

It is also possible to apply boldface formatting by using `bold` as a color. For instance, to print answers in boldface font, one can use the following as a :

```
|
```

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 `file`:

```
|
```

Chapter 29

Running External Programs

Julia borrows backtick notation for commands from the shell, Perl, and Ruby. However, in Julia, writing

```
|
```

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 `'s` 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 `'s` immediate child process, using `run` and `run!` calls.

Here's a simple example of running an external program:

```
|
```

The `stdout` is the output of the `cmd` command, sent to `stdout`. The `run` method itself returns `Cmd`, 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:

```
|
```

More generally, you can use `read` to read from or write to an external command.

```
|
```

29.1 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 `$(file)` for interpolation much as you would in a string literal (see [Strings](#)):

```
|
```

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 `file`, we wanted to sort the contents of the file `file`. Let's try it:

```
|
```

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:

```
|
```

As you can see, the space in the 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):

```
|
```

If you interpolate an array as part of a shell word, Julia emulates the shell's argument generation:

```
|
```

Moreover, if you interpolate multiple arrays into the same word, the shell's Cartesian product generation behavior is emulated:

```
|
```

Since you can interpolate literal arrays, you can use this generative functionality without needing to create temporary array objects first:

```
|
```

29.2 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:

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 `$foo` (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:

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:

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 first. Since Julia shows commands to you before running them, you can easily and safely just examine its interpretation without doing any damage.

29.3 Pipelines

Shell metacharacters, such as `;`, `&`, and `&&`, need to be quoted (or escaped) inside of Julia's backticks:

This expression invokes the `cat` command with three words as arguments: `cat`, `/dev/null`, and `&`. The result is that a single line is printed: `cat /dev/null &`. How, then, does one construct a pipeline? Instead of using `&` inside of backticks, one uses `|`:

```
|
```

This pipes the output of the `cat` command to the `cat` 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:

```
|
```

This prints the highest five user IDs on a UNIX system. The `cat`, `sort`, and `head` commands are all spawned as immediate children of the current 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:

```
|
```

The order of the output here is non-deterministic because the two processes are started nearly simultaneously, and race to make the first write to the descriptor they share with each other and the parent process. Julia lets you pipe the output from both of these processes to another program:

```
|
```

In terms of UNIX plumbing, what's happening here is that a single UNIX pipe object is created and written to by both processes, and the other end of the pipe is read from by the `cat` command.

IO redirection can be accomplished by passing keyword arguments `stdin`, `stdout`, and `stderr` to the `run` function:

```
|
```

Avoiding Deadlock in Pipelines

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 `readall`, not `read`, 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:

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:

This is a classic example of a single producer feeding two concurrent consumers: one 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 `$!=$|` in Perl causes each print statement to flush the 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:

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 30

Llamando a código C y Fortran

Aunque la mayoría del código se puede escribir en Julia, hay muchas bibliotecas maduras de alta calidad para computación numérica ya escritas en C y Fortran. Para permitir el uso fácil de este código existente, Julia hace que sea sencillo y eficiente llamar a las funciones C y Fortran. Julia tiene una filosofía de "no repetitivo": las funciones se pueden llamar directamente desde Julia sin ningún código de "pegamento", generación de código o compilación, incluso desde el aviso interactivo. Esto se logra haciendo una llamada apropiada con la sintaxis , que se parece a una llamada de función ordinaria.

El código que se debe llamar debe estar disponible como una biblioteca compartida. La mayoría de las bibliotecas C y Fortran ya se han compilado como bibliotecas compartidas, pero si está compilando el código usted mismo usando GCC (o Clang), necesitará usar las opciones `-fPIC` y `-shared`. Las instrucciones de la máquina generadas por el JIT de Julia son las mismas que una llamada C nativa, por lo que la sobrecarga resultante es lo mismo que llamar a una función de biblioteca desde el código C. (Las llamadas a funciones que no son de la biblioteca en C y Julia pueden estar incluidas y, por lo tanto, pueden tener incluso menos gastos generales que las llamadas a funciones de biblioteca compartidas. Cuando LLVM genera bibliotecas y ejecutables, es posible realizar optimizaciones de todo el programa que incluso optimizar a través de este límite, pero Julia aún no lo admite. Sin embargo, en el futuro, puede hacerlo, produciendo ganancias de rendimiento incluso mayores).

Las bibliotecas y funciones compartidas están referenciadas por una tupla de la forma `(lib, func)` donde `lib` es el nombre de función exportado por C. `func` se refiere al nombre de la biblioteca compartida: las bibliotecas compartidas disponibles en la ruta de carga (específica de la plataforma) se resolverán por nombre y, si es necesario, se puede especificar una ruta directa.

El nombre de una función se puede usar solo en lugar de la tupla (solo `func`). En este caso, el nombre se resuelve dentro del proceso actual. Este formulario se puede usar para llamar funciones de biblioteca C, funciones en el tiempo de ejecución de Julia o funciones en una aplicación vinculada a Julia.

Por defecto, los compiladores Fortran **generan nombres destrozados** (por ejemplo, convirtiendo nombres de funciones a minúsculas o mayúsculas, a menudo añadiendo un guión bajo), y para llamar a una función Fortran a través de `ccall` debe pasar el identificador mutilado correspondiente a la regla seguida por su compilador Fortran . Además, cuando se llama a una función Fortran, todas las entradas se deben pasar por referencia.

Por último, se puede usar `ccall` para generar de hecho una llamada a la función de librería. Los argumentos a `ccall` son los siguientes:

1. Una pareja , que debe ser escrita como una constante literal,
`(C_ptr_t, C_ptr_t)`

un puntero a función (por ejemplo, de `ccall`).

2. Tipo de retorno (ver abajo para la correspondencia entre el tipo declarado en C y Julia)

* Este argumento será evaluado en tiempo de compilación, cuando se defina el método que lo contiene.

- Una tupla de tipos de entrada. Los tipos de entrada deben ser escritos como un literal tupla, no como una variable o expresión de valor tupla.

- Este argumento será evaluado en tiempo de compilación, cuando se defina el método que lo contiene.

4. Los siguientes argumentos, si los hay, son los valores de los argumentos actuales pasados a la función.

Como un ejemplo completo pero simple, el siguiente código llama a la función `getenv` de la librería estándar C:

no toma argumentos y devuelve un `void`. Un problema común es que una 1-tupla debe escribirse con una coma al final. Por ejemplo, para llamar a la función `getenv` para obtener un puntero al valor de una variable de entorno, se realiza una llamada como esta:

Note que la tupla del tipo de argumento debe ser escrita como `(int)`, en lugar de como `int`. Esto es debido a que `int` es justo la expresión rodeada entre paréntesis, en lugar de una tupla que contiene a :

En la práctica, especialmente cuando se proporciona funcionalidad reutilizable, generalmente se envuelve el uso de funciones de Julia que configuran argumentos y luego se comprueban los errores de cualquier forma que la función C o Fortran los indique, propagándose al código que llama desde Julia como excepciones. Esto es especialmente importante ya que las API de C y Fortran son notoriamente inconsistentes sobre cómo indican las condiciones de error. Por ejemplo, la función de biblioteca C está envuelta en la siguiente función Julia, que es una versión simplificada de la definición real de :

La función `C` indica un error al devolver, pero otras funciones C estándar indican errores de varias maneras diferentes, incluyendo al devolver `-1`, `0`, `1` y otros valores especiales. Este contenedor arroja una excepción que indica claramente el problema si la persona que llama intenta obtener una variable de entorno inexistente:

Aquí hay un ejemplo ligeramente ms complejo que descubre el nombre de host de la máquina local:

Este ejemplo primero asigna un array de bytes, luego llama a la función de biblioteca C `gethostname` para llenar el array con el nombre de host, toma un puntero al buffer de nombre de host, y convierte el puntero a una cadena Julia, asumiendo que es una cadena C terminada en NUL. Es común que las bibliotecas C usen este patrón de requerir al llamador que asigne memoria para que la pase al llamado y la complete. La asignación de la memoria de Julia se logra generalmente creando un array no inicializado y pasando un puntero a sus datos a la función C. Es por eso que no usamos el tipo `String` aquí: como la matriz no está inicializada, podría contener bytes NUL. Convertir a `String` como parte de `convert` comprueba si hay bytes NUL contenidos y, por lo tanto, puede arrojar un error de conversión.

30.1 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:

The function `ccall` generates the C-compatible function pointer for a call to a Julia library function. Arguments to `ccall` are as follows:

1. A Julia Function
2. Return type
3. A tuple of input types

A classic example is the standard C library `qsort` function, declared as:

```
void qsort(void *base, rsize_t nobj, rsize_t size,
           int (*comp)(const void *, const void *))
```

The `base` argument is a pointer to an array of length `nobj`, with elements of `size` bytes each. `comp` 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 `arr` 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`:

```
function compare(a::T, b::T) where T
    return < a < b ? -1 : 1
end
```

Notice that we have to be careful about the return type: `qsort` expects a function returning a `C`, so we must be sure to return `1` via a call to `1` and a `-1`.

In order to pass this function to `qsort`, we obtain its address using the function :

```
qsort(arr, length(arr), sizeof(T), @cfunction(compare, C, (T, T)))
```

`@cfunction` accepts three arguments: the Julia function `compare`, the return type `C`, and a tuple of the argument types, in this case to sort an array of `T` elements.

The final call to `qsort` looks like this:

```
qsort(arr, length(arr), sizeof(T), @cfunction(compare, C, (T, T)))
```

As can be seen, `arr` is changed to the sorted array `sorted_arr`. Note that Julia knows how to convert an array into a `C`, how to compute the size of a type in bytes (identical to C's `sizeof` operator), and so on. For fun, try inserting a `@cfunction` line into `qsort`, 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).

30.2 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 `convert` function to convert each argument to the specified type. For example, the following call:

```
|
```

will behave as if the following were written:

```
|
```

normally just calls `convert`, but can be defined to return an arbitrary new object more appropriate for passing to C. For example, this is used to convert an array of objects (e.g. strings) to an array of pointers.

`Ptr{Ctype}` handles conversion to C 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:

Syntax / Keyword	Example	Description
		"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>Any</code> are allowed) in order for the instance to be constructed.
	<code>Union{...}</code>	"Super Type" :: A super-type (not a leaf-type) that cannot be instantiated, but can be used to describe a group of types.
		"Type Parameter" :: A specialization of a type (typically used for dispatch or storage optimization).
		"TypeVar" :: The <code>T</code> in the type parameter declaration is referred to as a TypeVar (short for type variable).
	<code>Ptr{Ctype}</code>	"Primitive Type" :: A type with no fields, but a size. It is stored and defined by-value.
		"Struct" :: A type with all fields defined to be constant. It is defined by-value, and may be stored with a type-tag.
	<code>Array{Ctype, N}</code>	"Is-Bits" :: A <code>Ptr{Ctype}</code> , or a <code>Struct{Ctype...}</code> type where all fields are other <code>Ptr{Ctype}</code> types. It is defined by-value, and is stored without a type-tag.
		"Singleton" :: a Leaf Type or Struct with no fields.
or		"Tuple" :: an immutable data-structure similar to an anonymous struct type, or a constant array. Represented as either an array or a struct.

Bits Types:

There are several special types to be aware of, as no other type can be defined to behave the same:

- `Ptr{Ctype}` Exactly corresponds to the `void*` type in C (or `void**` in Fortran).

- Exactly corresponds to the `type` in C (or `in Fortran`).
- Exactly corresponds to the `type` in C (or `in Fortran`).
- Exactly corresponds to the `type` in C (or `in Fortran`).
- Exactly corresponds to the `type` annotation in C (or any `type` in Fortran). Any Julia type that is not a subtype of `is assumed to be unsigned`.
- Behaves like a `that can manage its memory via the Julia GC`.
- When an array is passed to C as a `argument`, it is not reinterpret-cast: Julia requires that the element type of the array matches `, and the address of the first element is passed`.
Therefore, if an `contains data in the wrong format, it will have to be explicitly converted using a call such as`.
To pass an array `as a pointer of a different type without converting the data beforehand` (for example, to pass a `array to a function that operates on uninterpreted bytes`), you can declare the argument as `.
If an array of eltype is passed as a argument, will attempt to first make a null-terminated copy of the array with each element replaced by its version. This allows, for example, passing an pointer array of type to an argument of type .`

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 `in C is not the same as an in Julia).`

System Independent:

The `type` is essentially a synonym for `, except the conversion to 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 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 NUL and want to skip the check, you can use as the argument type. can also be used as the 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 `or wrapper around their type specification.`

Warning

For string arguments `() the Julia type should be (if NUL-terminated data is expected) or either or otherwise (these two pointer types have the same effect), as described above, not . Similarly, for array arguments (or), the Julia type should again be , not .`

C name	Fortran name	Standard Julia Alias	Julia Base Type
(only in C++)			
	,		
, (C, typical)	,		
	,		
and or			
(where T represents an appropriately defined type)			
(or , e.g. a string)			if NUL-terminated, or if not
(or)			
(any Julia Type)			
(a reference to a Julia Type)			
			Not supported
(variadic function specification)			(where is one of the above types, variadic functions of different argument types are not supported)

C name	Standard Julia Alias	Julia Base Type
		(x86, x86_64), (powerpc, arm)
		(UNIX), (Windows)
		(UNIX), (Windows)
		(UNIX), (Windows)

Warning

Julia's type is 32 bits, which is not the same as the wide character type (or) on all platforms.

Warning

A return type of means the function will not return i.e. C++11 or C11 (e.g. or). Do not use this for functions that return no value () but do return, use instead.

Note

For arguments, the Julia type should be (if the C routine expects a NUL-terminated string) or 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 `std::string` 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 `std::string` can be called by using a `std::string` type within Julia. For example, C functions of the form:

```
|
```

can be called via the following Julia code:

```
|
```

Note

A C function declared to return `std::string` will return the value `""` in Julia.

Struct Type correspondences

Composite types, aka `struct` in C or `type` in Fortran90 (or `type` / `type` in some variants of F77), can be mirrored in Julia by creating a definition with the same field layout.

When used recursively, `struct` 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 `struct` type and use that instead. Unnamed `structs` are 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 `struct` 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 :

```
|
```

Arrays of unknown size (C99-compliant variable length `structs` specified by `...` or `...`) 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:

```
|
```

In Julia, we can access the parts independently to make a copy of that string:

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 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 signature, as long as they don't affect the layout of the type. For example, `ccall{Ptr{Cint}, Cvoid, (Cint, Cvoid), c}` is valid, since `Ptr{Cint}` is always a word-size primitive type. But, `ccall{Ptr{Cint}, Cvoid, (Cint, Cvoid), c}` is not valid, since the type layout of `Ptr{Cint}` 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 `Ptr{T}` 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 bytes.
- The element type of the tuple must be an instance of `Ptr{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:

The following Julia code calls `__m128i` using :

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 any C program. Do not try to free an object received from a C library with `free` in Julia, as this may result in the `free` function being called via the wrong 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, `Ptr{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 `ccall`. 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 `T`, as Fortran passes all variables by reference. The return type should either be `Ptr{T}` for Fortran subroutines, or a `Ref{T}` for Fortran functions returning the type `T`.

30.3 Mapping C Functions to Julia

/ argument translation guide

For translating a C argument list to Julia:

- `T`, where `T` is one of the primitive types: `Int`, `Float`, `Char`, `Ptr{T}`, or any of their equivalents
 - `T`, where `T` is an equivalent Julia Bits Type (per the table above)
 - if `T` is an `Ptr`, the argument type should be equivalent to `Ptr{T}` or `Ref{T}`
 - argument value will be copied (passed by value)
- `Ptr{T}` (including typedef to a struct)
 - `T`, where `T` is a Julia leaf type
 - argument value will be copied (passed by value)
- `Ref{T}`
 - 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{T}`, if it really is just an unknown pointer
- `Any`
 - argument value must be a valid Julia object
 - currently unsupported by `ccall`
- `void`

-
- argument value must be a valid Julia object (or)
- currently unsupported by
-
- , where is the Julia type corresponding to
- argument value will be copied if it is an type otherwise, the value must be a valid Julia object
- (e.g. a pointer to a function)
- (you may need to use explicitly to create this pointer)
- (e.g. a vararg)
- , where is the Julia type
-
- not supported

/ return type translation guide

For translating a C return type to Julia:

-
- (this will return the singleton instance)
- , where is one of the primitive types: , , , , , , or any of their equivalents
- , where is an equivalent Julia Bits Type (per the table above)
- if is an , the argument type should be equivalent to or
- argument value will be copied (returned by-value)
- (including typedef to a struct)
- , where is a Julia Leaf Type
- argument value will be copied (returned by-value)
-
- 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 , if it really is just an unknown pointer
-
-
- argument value must be a valid Julia object
-

-
- argument value must be a valid Julia object (or)
- - If the memory is already owned by Julia, or is an type, and is known to be non-null:
 - * , where is the Julia type corresponding to
 - * a return type of is invalid, it should either be (corresponding to) or (corresponding to)
 - * **C MUST NOT** modify the memory returned via if is an type
 - If the memory is owned by C:
 - * , where is the Julia type corresponding to
- (e.g. a pointer to a function)
 - (you may need to use explicitly to create this pointer)

Passing Pointers for Modifying Inputs

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 , you need to first encapsulate the value inside an of the appropriate type. When you pass this object as an argument, Julia will automatically pass a C pointer to the encapsulated data:

Upon return, the contents of and can be retrieved (if they were changed by) by and ; that is, they act like zero-dimensional arrays.

Special Reference Syntax for ccall (deprecated):

The syntax is deprecated, use the argument type instead.

A prefix is used on an argument to 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.

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). 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 or .

When a scalar value is passed with as an argument of type , the value will first be converted to type .

30.4 Some Examples of C Wrappers

Here is a simple example of a C wrapper that returns a `type`:

```

using C

# C wrapper function
function wrapper()
    # ... C code ...
endfunction

# Julia wrapper function
function wrapper()::type
    # ... Julia code ...
endfunction

```

The [GNU Scientific Library](#) (here assumed to be accessible through `libgsl`) defines an opaque pointer, `gsl_vector`, as the return type of the C function `gsl_vector_alloc`. As user code never has to look inside the `gsl_vector` struct, the corresponding Julia wrapper simply needs a new type declaration, `Vector`, that has no internal fields and whose sole purpose is to be placed in the type parameter of a `function`. The return type of the `wrapper` is declared as `Vector`, since the memory allocated and pointed to by `gsl_vector` is controlled by C (and not Julia).

The input `x` is passed by value, and so the function's input signature is simply declared as `Vector` without any `ref` or `out` necessary. (If the wrapper was calling a Fortran function instead, the corresponding function input signature should instead be `ref Vector`, since Fortran variables are passed by reference.) Furthermore, `Vector` can be any type that is convertible to a `Integer`; the `wrapper` implicitly calls `Integer`.

Here is a second example wrapping the corresponding destructor:

```

using C

# C wrapper function
function wrapper(x::Vector)
    # ... C code ...
endfunction

# Julia wrapper function
function wrapper(x::Vector)
    # ... Julia code ...
endfunction

```

Here, the input `x` is declared to be of type `Vector`, meaning that the memory that `x` points to may be managed by Julia or by C. A pointer to memory allocated by C should be of type `Vector`, but it is convertible using `Vector` and therefore can be used in the same (covariant) context of the input argument to a `function`. A pointer to memory allocated by Julia must be of type `Vector`, 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 `Vector` 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 `Vector` for the method signature of the wrapper and similarly in the `function` is also acceptable.

Here is a third example passing Julia arrays:

The C function wrapped returns an integer error code; the results of the actual evaluation of the Bessel J function populate the Julia array `res`. This variable can only be used with corresponding input type declaration, since its memory is allocated and managed by Julia, not C. The implicit call to `ccall` unpacks the Julia pointer to a Julia array data structure into a form understandable by C.

Note that for this code to work correctly, `res` must be declared to be of type `Ptr{Cint}` and not `Ptr{Cint}`. The memory is managed by Julia and the `ccall` signature alerts Julia's garbage collector to keep managing the memory for `res` while the `ccall` executes. If `res` were used instead, the `ccall` may still work, but Julia's garbage collector would not be aware that the memory declared for `res` is being used by the external C function. As a result, the code may produce a memory leak if `res` never gets freed by the garbage collector, or if the garbage collector prematurely frees `res`, the C function may end up throwing an invalid memory access exception.

30.5 Garbage Collection Safety

When passing data to a C function, it is best to avoid using the `ccall` function. Instead define a `convert` method and pass the variables directly to the C function. `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 `Ptr{Cint}` 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 `ccall` and `ccallcc` 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 `ccallcc` 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 `ccallcc` contained a reference to `ccallcc` and both `ccallcc` and `ccallcc` are due for garbage collection, there is no guarantee that `ccallcc` would be finalized after `ccallcc`. If proper finalization of `ccallcc` depends on `ccallcc` being valid, it must be handled in other ways.

30.6 Non-constant Function Specifications

A function specification must be a constant expression. However, it is possible to use computed values as function names by staging through `ccallcc` as follows:

This expression constructs a name using `__attribute__((__externally_visible__))`, then substitutes this name into a new `__attribute__((__externally_visible__))` expression, which is then evaluated. Keep in mind that `__attribute__((__externally_visible__))` only operates at the top level, so within this expression local variables will not be available (unless their values are substituted with `__attribute__((__externally_visible__))`). For this reason, `__attribute__((__externally_visible__))` 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.

30.7 Indirect Calls

The first argument to `__attribute__((__externally_visible__))` can also be an expression evaluated at run time. In this case, the expression must evaluate to a `void*`, which will be used as the address of the native function to call. This behavior occurs when the first 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 `__attribute__((__externally_visible__))`, then cache it in a global variable for that session. For example:

```

// Example: Indirect call using a global variable
// This code is a placeholder for the example content.

```

30.8 Calling Convention

The second argument to `__attribute__((__externally_visible__))` 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`, and `__fastcall`. For example (from `__attribute__((__externally_visible__))`) we see the same `__attribute__((__externally_visible__))` as above, but with the correct signature for Windows:

```

// Example: Calling convention specifier
// This code is a placeholder for the example content.

```

For more information, please see the [LLVM Language Reference](#).

There is one additional special calling convention `__asm__`, 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:

```

// Example: __asm__ calling convention
// This code is a placeholder for the example content.

```

As with any `__attribute__((__externally_visible__))`, 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 `__attribute__((__externally_visible__))`.

30.9 Accessing Global Variables

Global variables exported by native libraries can be accessed by name using the `ccall` function. The arguments to `ccall` are a symbol specification identical to that used by `ccall`, and a type describing the value stored in the variable:

The result is a pointer giving the address of the value. The value can be manipulated through this pointer using `ccall` and `ccall`.

30.10 Accessing Data through a Pointer

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 `ccall`. 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 `ccall` and `ccall` (e.g. `ccall` 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 `ccall` is `Ptr{Cvoid}`, then the memory is assumed to contain a reference to a Julia object (`Ptr{Cvoid}`), 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 `ccall` itself is actually a `Ptr{Cvoid}`, it can be converted back to a Julia object reference by `ccall`. (Julia values `ccall` can be converted to `Ptr{Cvoid}` pointers, as `ccall`, by calling `ccall`.)

The reverse operation (writing data to a `Ptr{T}`), can be performed using `ccall`. Currently, this is only supported for primitive types or other pointer-free () 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 pointer of interest is a plain-data array (primitive type or immutable struct), the function `ccall` may be more useful. The final parameter should be true if Julia should "take ownership" of the underlying buffer and call `ccall` when the returned object is finalized. If the `ccall` parameter is omitted or false, the caller must ensure the buffer remains in existence until all access is complete.

Arithmetic on the `Ptr{T}` type in Julia (e.g. using `ccall`) does not behave the same as C's pointer arithmetic. Adding an integer to a `Ptr{T}` 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.

30.11 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 `Condition{Cvoid}` object and wait on it:

The callback you pass to C should only execute a `ccall` to `ccall`, passing `ccall` 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 `ccall` may result in a single wakeup notification to the condition.

30.12 More About Callbacks

For more details on how to pass callbacks to C libraries, see this [blog post](#).

30.13 C++

For direct C++ interfacing, see the [Cxx](#) package. For tools to create C++ bindings, see the [CxxWrap](#) package.

Chapter 31

Handling Operating System Variation

When dealing with platform libraries, it is often necessary to provide special cases for various platforms. The variable `__PLATFORM__` can be used to write these special cases. There are several functions intended to make this easier: `__if`, `__endif`, `__else`, and `__elseif`. These may be used as follows:

```
|
```

Note that `__if` and `__endif` are mutually exclusive subsets of `__PLATFORM__`. Additionally, there is a macro `__platform_if` which makes it possible to use these functions to conditionally hide invalid code, as demonstrated in the following examples.

Simple blocks:

```
|
```

Complex blocks:

```
|
```

When chaining conditionals (including `if/elseif/end`), the `__platform_if` must be repeated for each level (parentheses optional, but recommended for readability):

```
|
```


Chapter 32

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_PATH` to `/usr/local/bin`, then either type `ENV["JULIA_PATH"] = "/usr/local/bin"` for instance in the REPL to make this change on a case by case basis, or add the same to the user configuration file `~/.julia/config/userconfig.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_PATH"]`.

The environment variables that Julia uses generally start with `JULIA_`. If `JULIA_PATH` is called with `equal to`, then the output will list defined environment variables relevant for Julia, including those for which `JULIA_` appears in the name.

32.1 File locations

The absolute path of the directory containing the Julia executable, which sets the global variable `JULIA_PATH`. If `JULIA_PATH` is not set, then Julia determines the value at run-time.

The executable itself is one of

|

by default.

The global variable `JULIA_DATA_PATH` determines a relative path from `JULIA_PATH` to the data directory associated with Julia. Then the path

|

determines the directory in which Julia initially searches for source files (via `JULIA_LOAD_PATH`).

Likewise, the global variable `JULIA_CONFIG_PATH` determines a relative path to the configuration file directory. Then Julia searches for a file at

|

by default (via `JULIA_LOAD_PATH`).

For example, a Linux installation with a Julia executable located at `/usr/local/bin/julia`, a `JULIA_PATH` of `/usr/local/bin`, and a `JULIA_DATA_PATH` of `data` will have `JULIA_PATH` set to `/usr/local/bin`, a source-file search path of

|

and a global configuration search path of

|

A separated list of absolute paths that are to be appended to the variable `JULIA_LOAD_PATH`. (In Unix-like systems, the path separator is `:`; in Windows systems, the path separator is `;`.) The variable is where `code` and `look` for code; it defaults to the absolute paths

|

so that, e.g., version 0.6 of Julia on a Linux system with a Julia executable at `/usr/bin/julia` will have a default of

|

The path of the parent directory `for` the version-specific Julia package repositories. If the path is relative, then it is taken with respect to the working directory. If `is` not set, then `defaults` to

|

Then the repository location `for` a given Julia version is

|

For example, for a Linux user whose home directory is `~/`, the directory containing the package repositories would by default be

|

and the package repository for version 0.6 of Julia would be

|

The absolute path `of` the REPL's history file. If `is` not set, then `defaults` to

|

A positive `that` determines how much time the max-sum subroutine `of` the package dependency resolver `will` devote to attempting satisfying constraints before giving up: this value is by default `,` and larger values correspond to larger amounts of time.

Suppose the value of `is` `.` Then

- the number of pre-decimation iterations is `,`
- the number of iterations between decimation steps is `,` and
- at decimation steps, at most one in every `packages` is decimated.

32.2 External applications

The absolute path of the shell with which Julia should execute external commands (via `ENV["SHELL"]`). 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.

The editor returned by `Base.julia_editor` and used in, e.g., `Base.showerror`, referring to the command of the preferred editor, for instance `vim`.

`Base.julia_editor` takes precedence over `EDITOR`, which in turn takes precedence over `vim`. If none of these environment variables is set, then the editor is taken to be `vim` on Windows and OS X, or `vim` if it exists, or `vim` otherwise.

Note

`Base.julia_editor` is *not* used in the determination of the editor for `Base.showerror`: this function checks `EDITOR` and `vim` alone.

32.3 Parallelization

Overrides the global variable `Base.CPU_THREADS`, the number of logical CPU cores available.

A function that sets the value of `Base.CPU_THREADS` (default: `Base.CPU_THREADS`). This function gives the number of seconds a worker process will wait for a master process to establish a connection before dying.

An unsigned 64-bit integer (`UInt64`) that sets the maximum number of threads available to Julia. If `Base.CPU_THREADS` exceeds the number of available physical CPU cores, then the number of threads is set to the number of cores. If `Base.CPU_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 `Base.CPU_THREADS`.

If set to a string that starts with the case-insensitive substring `never`, then spinning threads never sleep. Otherwise, `Base.CPU_THREADS` is interpreted as an unsigned 64-bit integer (`UInt64`) and gives, in nanoseconds, the amount of time after which spinning threads should sleep.

If set to anything besides `nothing`, then Julia's thread policy is consistent with running on a dedicated machine: the master thread is on `proc 0`, and threads are affinityized. Otherwise, Julia lets the operating system handle thread policy.

32.4 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](#).

The formatting (default: light red,) that errors should have at the terminal.

The formatting (default: yellow,) that warnings should have at the terminal.

The formatting (default: cyan,) that info should have at the terminal.

The formatting (default: normal,) that input should have at the terminal.

The formatting (default: normal,) that output should have at the terminal.

The formatting (default: bold,) that line info should have during a stack trace at the terminal.

The formatting (default: bold,) that function calls should have during a stack trace at the terminal.

32.5 Debugging and profiling

’, ’

If set, these environment variables take strings that optionally start with the character , followed by a string interpolation of a colon-separated list of three signed 64-bit integers (). This triple of integers represents the arithmetic sequence , ,

- If it's the n th time that has been called, and belongs to the arithmetic sequence represented by , then garbage collection is forced.
- If it's the n th time that has been called, and belongs to the arithmetic sequence represented by , then garbage collection is forced.
- If it's the n th time that has been called, and belongs to the arithmetic sequence represented by , then counts for the number of calls to and are printed.

If the value of the environment variable begins with the character , 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 is set to in the build configuration).

If set to anything besides , 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 is set to in the build configuration).

If set to anything besides , 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 is set to in the build configuration).

If set to anything besides , 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](#) (set to in the build configuration), or
- [OProfile](#) (set to in the build configuration).

Arguments to be passed to the LLVM backend.

Note

This environment variable has an effect only if Julia was compiled with set — in particular, the executable is always compiled with this build variable.

If set, then Julia prints detailed information about the cache in the loading process of .

Chapter 33

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#).

33.1 High-Level Embedding

We start with a simple C program that initializes Julia and calls some Julia code:

In order to build this program you have to put the path to the Julia header into the include path and link against `libjulia.so`. For instance, when Julia is installed to `/usr/local`, one can compile the above test program `test.c` with `gcc` using:

Then if the environment variable `LD_LIBRARY_PATH` is set to `/usr/local/lib`, the output program can be executed.

Alternatively, look at the `program` in the Julia source tree in the `src` folder. The file `program` is another simple example of how to set options while linking against `lib`.

The first thing that has to be done before calling any other Julia C function is to initialize Julia. This is done by calling `julia_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 `julia_init_with_image` instead.

The second statement in the test program evaluates a Julia statement using a call to `julia_eval`.

Before the program terminates, it is strongly recommended to call `julia_close`. The above example program calls this before returning from `main`.

Note

Currently, dynamically linking with the `libjulia` shared library requires passing the `-Wl,-rpath,$ORIGIN` option. In Python, this looks like:

```
|
```

Note

If the `julia` program needs to access symbols from the main executable, it may be necessary to add `-Wl,-rpath,$ORIGIN` linker flag at compile time on Linux in addition to the ones generated by `julia-config` described below. This is not necessary when compiling a shared library.

Using `julia-config` to automatically determine build parameters

The script `julia-config` 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

```
|
```

On the command line

A simple use of this script is from the command line. Assuming that `julia-config` is located in `./libjulia`, it can be invoked on the command line directly and takes any combination of 3 flags:

```
|
```

If the above example source is saved in the file `test.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 `gcc` for `cc`:

```
|
```

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 `$(shell)` may be found in the directory, this is not necessarily the case, but Julia can be used to locate `$(shell)` too, and the makefile can be used to take advantage of that. The above example is extended to use a Makefile:

```


```

Now the build command is simply `make`.

33.2 Converting Types

Real applications will not just need to execute expressions, but also return their values to the host program. `jlcall` returns a `void*`, which is a pointer to a heap-allocated Julia object. Storing simple data types like `double` 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:

```


```

In order to check whether `val` is of a specific Julia type, we can use the `jl_is_xxx_ptr`, `jl_is_xxx_val`, or `jl_typeof` functions. By typing `jl_typeof(val)` into the Julia shell we can see that the return type is `jl_value_t` (in C). To convert the boxed Julia value into a C double the `jl_double_val` function is used in the above code snippet.

Corresponding `jl_double_ptr` functions are used to convert the other way:

```


```

As we will see next, boxing is required to call Julia functions with specific arguments.

33.3 Calling Julia Functions

While `jlcall` 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_fcall_ptr`:

```


```

In the first step, a handle to the Julia function `is` retrieved by calling `jl_get_function_handle`. The first argument passed to `jl_get_function_handle` is a pointer to the module in which `is` is defined. Then, the double value is boxed using `jl_box_double`. Finally, in the last step, the function is called using `jl_call2`, `jl_call3`, and `jl_call4` functions also exist, to conveniently handle different numbers of arguments. To pass more arguments, use :

Its second argument `args` is an array of arguments and `nargs` is the number of arguments.

33.4 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_new_struct` 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_gc_collect` calls. But in order to make sure that values can survive `jl_gc_collect` calls, we have to tell Julia that we hold a reference to a Julia value. This can be done using the `jl_gc_add_ref` macros:

The `jl_gc_add_ref` call releases the references established by the previous `jl_gc_add_ref`. Note that `jl_gc_add_ref` is working on the stack, so it must be exactly paired with a `jl_gc_remove_ref` before the stack frame is destroyed.

Several Julia values can be pushed at once using the `jl_gc_add_ref_array`, `jl_gc_add_ref_ptr`, and `jl_gc_add_ref_ptr_array` macros. To push an array of Julia values one can use the `jl_gc_add_ref_array` macro, which can be used as follows:

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:

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 `jl_gc_collect` object was just allocated and garbage collection was not run since then. Remember that most `jl_gc_collect` functions can sometimes invoke garbage collection.

The write barrier is also necessary for arrays of pointers when updating their data directly. For example:

Manipulating the Garbage Collector

There are some functions to control the GC. In normal use cases, these should not be necessary.

Function	Description
	Force a GC run
	Disable the GC, return previous state as int
	Enable the GC, return previous state as int
	Return current state as int

33.5 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, it 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:

```
|
```

Alternatively, if you have already allocated the array you can generate a thin wrapper around its data:

```
|
```

The last argument is a boolean indicating whether Julia should take ownership of the data. If this argument is non-zero, the GC will call `jl_free` on the data pointer when the array is no longer referenced.

In order to access the data of `x`, we can use :

```
|
```

Now we can fill the array:

```
|
```

Now let us call a Julia function that performs an in-place operation on :

```
|
```

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 `jlcall` and `jlvalue` can be cast to a `jl_array_t*`:

Now the content of `arr` can be accessed as before using `jl_array_ptr`. 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:

Notice that while Julia arrays use 1-based indexing, the C API uses 0-based indexing (for example in calling `jl_array_ptr`) in order to read as idiomatic C code.

33.6 Exceptions

Julia code can throw exceptions. For example, consider:

This call will appear to do nothing. However, it is possible to check whether an exception was thrown:

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 `jlcall` with a function that checks whether an exception was thrown, and then rethrows the exception in the host language.

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:

General exceptions can be raised using the functions:

|

takes a C string, and is called like :

|

where in this example is assumed to be an integer.

Chapter 34

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 `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 `Manifest.toml` file for when you're ready to add your code to the curated METADATA repository.

34.1 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:

```
pkg.status()
# Packages: 0
# Manifest.toml: 0
# Manifest.toml: 0
```

Your package directory is automatically initialized the first time you run a `pkg.add` command that expects it to exist – which includes `pkg.status`. Here's an example non-trivial set of required and additional packages:

```
pkg.add("Distributions")
pkg.add("DataFrames")
pkg.add("Plots")
pkg.add("Gadfly")
pkg.add("Makie")
pkg.add("IJulia")
pkg.add("REPL.jl")
pkg.add("Base64")
pkg.add("JSON")
pkg.add("UUIDs")
pkg.add("Dates")
pkg.add("Logging")
pkg.add("Printf")
pkg.add("Random")
pkg.add("Statistics")
pkg.add("Test")
pkg.add("TOML")
pkg.add("Unicode")
pkg.add("Base")
pkg.add("Base64")
pkg.add("JSON")
pkg.add("UUIDs")
pkg.add("Dates")
pkg.add("Logging")
pkg.add("Printf")
pkg.add("Random")
pkg.add("Statistics")
pkg.add("Test")
pkg.add("TOML")
pkg.add("Unicode")
pkg.add("Base")
```

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.status` returns a dictionary, mapping installed package names to the version of that package which is installed:

```
pkg.status()
# Packages: 0
# Manifest.toml: 0
# Manifest.toml: 0
```

34.2 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 to install (or remove) to satisfy those requirements optimally – and 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 `Manifest.toml`. You can edit this file by hand and then call `update` to install, upgrade or remove packages to optimally satisfy the requirements, or you can do `update!()`, which will open `Manifest.toml` in your editor (configured via the `JULIA_EDITOR` or `JULIA_EDITOR_PATH` environment variables), and then automatically call `update` afterwards if necessary. If you only want to add or remove the requirement for a single package, you can also use the non-interactive `add` and `remove` commands, which add or remove a single requirement to `Manifest.toml` and then call `update`.

You can add a package to the list of requirements with the `add` function, and the package and all the packages that it depends on will be installed:

What this is doing is first adding `MyPackage` to your `Manifest.toml` file:

It then runs `update` using these new requirements, which leads to the conclusion that the `MyPackage` package should be installed since it is required but not installed. As stated before, you can accomplish the same thing by editing your `Manifest.toml` file by hand and then running yourself:

This is functionally equivalent to calling `pip install`, except that `pip install` doesn't change `requirements.txt` until *after* installation has completed, so if there are problems, `requirements.txt` will be left as it was before calling `pip install`. The format of the `requirements.txt` 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 `pip uninstall` to remove the requirement for it from the `requirements.txt` file:

Once again, this is equivalent to editing the `requirements.txt` file to remove the line with each package name on it then running `pip install` to update the set of installed packages to match. While `pip install` and `pip uninstall` are convenient for adding and removing requirements for a single package, when you want to add or remove multiple packages, you can call `pip install` to manually change the contents of `requirements.txt` and then update your packages accordingly. `pip install` does not roll back the contents of `requirements.txt` if `pip install` fails – rather, you have to run again to fix the file's 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 `pip install`. By default, all GitHub-hosted packages will be accessed via 'https'; this default can be modified by calling `pip config set git_protocol https`. 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:

However, this change will be system-wide and thus the use of `pip config set git_protocol https` is preferable.

Note

The package manager functions also accept the `suffix` on package names, though the suffix is stripped internally. For example:

34.3 Offline Installation of Packages

For machines with no Internet connection, packages may be installed by copying the package root directory (given by `pkg_root`) from a machine with the same operating system and environment.

`install_offline` does the following within the package root directory:

1. Adds the name of the package to `pkg_root/Manifest.toml`.
2. Downloads the package to `pkg_root/Manifest.toml`, then copies the package to the package root directory.
3. Recursively performs step 2 against all the packages listed in the package's `Manifest.toml` file.
4. Runs `run`.

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.

34.4 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 ¹. The `add` and `install` 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 `add`, where `url` is a git URL from which the package can be cloned:

By convention, Julia repository names end with `.jl` (the additional `jl` 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 `pkg_root` directory, however, the extension is redundant so we leave it off.

If unregistered packages contain a `Manifest.toml` 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.

¹The official set of packages is at <https://github.com/JuliaLang/METADATA.jl>, but individuals and organizations can easily use a different meta-data 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.

34.5 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 :

The first step of updating packages is to pull new changes to and see if any new registered package versions have been published. After this, 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 – you installed it with .
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 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 function. In that case, only the packages provided as arguments and their dependencies will be updated:

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.

34.6 Checkout, Pin and Free

You may want to use the 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 or some other development branch. In such cases, you can do to checkout the branch of or to checkout some other branch:

Immediately after installing with it is on the current most recent registered version – at the time of writing this. Then after running , you can see from the output of that is on an unregistered version greater than , indicated by the "pseudo-version" number .

When you checkout an unregistered version of a package, the copy of the file in the package repo takes precedence over any requirements registered in , 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 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 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 :

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 won't change the version the package is on, you can use the function:

After this, the package will remain pinned at version – or more specifically, at commit , but since versions are permanently associated a given git hash, this is the same thing. works by creating a throw-away branch for the commit you want to pin the package at and then 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:

Now the package is pinned at commit , which corresponds to version . When you decide to "unpin" a package and let the package manager update it again, you can use like you would to move off of any branch:

After this, the package is managed by the package manager again, and future calls to will upgrade it to newer versions when they are published. The throw-away branch remains in your local 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 ².

34.7 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 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 branch. In order to use a custom repository and branch, issue the following command:

²Packages 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.

The `branch` argument is optional and defaults to `.`. Once initialized, a file named `branch` in your `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 file directly (be careful!) or remove the `branch` directory and re-initialize your METADATA repository using the `branch` command.

Chapter 35

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.

35.1 Initial Setup

Since packages are git repositories, before doing any package development you should setup the following standard global git configuration settings:

where `is your actual full name` (spaces are allowed between the double quotes) and `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 account](#) on GitHub and then do:

where `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 `to access the functions starting with` in the document below.

35.2 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.,) 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.

Code changes

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:

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 private machine, you first *push* your 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 package installed. In the description below, anything starting with `jl>` or `jl` is meant to be typed at the Julia prompt; anything starting with `jl>` 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:

Now suppose you're ready to make some changes to `Images.jl`. While there are several possible approaches, here is one that is widely used:

- From the Julia prompt, type `jl> Pkg.update()`. 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, see [Dirty packages](#) below.
- Create a branch for your changes: navigate to the package folder (the one that Julia reports from `jl> Pkg.status()`) and (in shell mode) create a new branch using `git checkout -b <name>`, where `<name>` might be some descriptive name (e.g., `fix-bug`). By creating a branch, you ensure that you can easily go back and forth between your new work and the current `master` branch (see <https://git-scm.com/book/en/v2/Git-Branching-Branches-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 `test`: this will run your tests in a separate (new) process.
 - From Julia, `test` from the package's `src` 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 `test` 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>.
- Submit your changes: From the Julia prompt, type `push`. 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 `checkout` (from shell mode) or `checkout` (from the Julia prompt).
 - As above, make your changes, run the tests, and commit your changes.
 - From the shell, type `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 status` to see what these changes are; you can either discard them (`git reset`) or commit them before switching branches. If you can't easily resolve the problems manually, as a last resort you can delete the entire `src` folder and reinstall a fresh copy with `install`. 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` 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 `main` but wish to go back to the official `main` (called `main`), use the following procedure:

- Create a new branch. This branch will hold your changes.

- Make sure everything is committed to this branch.
- . If this fails, *do not* proceed further until you have resolved the problems, or you may lose your changes.
- *Reset* (your current branch) back to an earlier state with (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:

```

commit 1234567890abcdef1234567890abcdef12345678
Author: John Doe <john.doe@example.com>
Date:   Mon Jul 11 12:34:56 2011 -0700
    Initial commit
  
```

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 `branch` and create a new branch with `branch`. Since you started from `branch`, this will be a copy. Now go back to the one you intend to modify with `branch`.
- From the shell, type `branch`.
- To combine commits, change `branch` to `branch` (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:

```

branch
branch
branch
  
```

Now let's assume you've rebased successfully. Since your repository has now diverged from the one in your GitHub fork, you're going to have to do a *force push*:

- To make it easy to refer to your GitHub fork, create a "handle" for it with `branch`, where the URL comes from the "clone URL" on your GitHub fork's page.
- Force-push to your fork with `branch`. The `branch` indicates that this should replace the `branch` found at `branch`.

35.3 Creating a new Package

REQUIRE speaks for itself

You should have a `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 `file` in this file. While this line is partly informational, it also has the consequence of whether `file` will update

code found in version directories. It will not update code found in version directories beneath the floor of what's specified in your .

As the development version 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 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 to your file. In this case, you will still have in your . The 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 expect your users to be on, you can set an upper bound. In this case, you would put in your 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 and do not want to support the current stable version of Julia. If you choose to do this, simply add to your . Just remember to change the to in your file once 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 .

Lastly, in many cases you may need extra packages for testing. Additional packages which are only required for tests should be specified in the file. This file has the same specification as the standard 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 to follow in naming your package:

1. Avoid jargon. In particular, avoid acronyms unless there is minimal possibility of confusion.
 - It's ok to say if you're talking about the USA.
 - It's not ok to say , even if you're talking about positive mental attitude.
2. Avoid using 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.
 - provides the type.
 - provides the type.
 - In contrast, provides no new type, but instead new functionality in the function.
4. Err on the side of clarity, even if clarity seems long-winded to you.

- is a less ambiguous name than or , 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, , , and other packages each implement a unique approach based on a particular design philosophy.
 - In contrast, provides a consistent interface to use many well-established sorting algorithms.
6. Packages that wrap external libraries or programs should be named after those libraries or programs.
- wraps the library, which can be identified easily in a web search.
 - provides an interface to call the MATLAB engine from within Julia.

Generating the package

Suppose you want to create a new Julia package called . To get started, do where is the new package name and is the name of a license that the package generator knows about:

```
|
```

This creates the directory , initializes it as a git repository, generates a bunch of files that all packages should have, and commits them to the repository:

```
|
```


At the moment, the package manager knows about the MIT "Expat" License, indicated by `MIT_EXPAT`, the Simplified BSD License, indicated by `SIMPLIFIED_BSD`, and version 2.0 of the Apache Software License, indicated by `APACHE_2_0`. 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 `LICENSE` file after it has been generated.

If you created a GitHub account and configured git to know about it, `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 Pkg` 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 `foo.jl` needs to load `bar.txt`, use the following code:

Making Your Package Available

Once you've made some commits and you're happy with how `Pkg` 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 ³. 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:

For your package, it will be your GitHub user name and the name of your package, but you get the idea. People you send this URL to can use `add` to install the package and try it out:

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 `Pkg` is ready to be registered as an official package, you can add it to your local copy of `using Pkg` :

³Installing and using GitHub's "[hub](#)" tool is highly recommended. It allows you to do things like run `push` in the package repo and have it automatically created via GitHub's API.

This creates a commit in the `repo`:

This commit is only locally visible, however. To make it visible to the Julia community, you need to merge your local upstream into the official repo. The `push` command will fork the repository on GitHub, push your changes to your fork, and open a pull request:

Tip

If `push` fails with error:

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.

Other failures may require you to circumvent by [creating a pull request on GitHub](#). See: [Publishing METADATA manually](#) below.

Once the package URL for `is` registered in the official `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 `push` command:

This tags in the repo:

```
|
```

It also creates a new version entry in your local repo for :

```
|
```

The command takes an optional second argument that is either an explicit version number object like or one of the symbols , or . 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 first. Since Julia itself hasn't achieved status, it's best to be conservative in your package's tagged versions.

As with , these changes to aren't available to anyone else until they've been included upstream. Again, use the 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 :

```
|
```

Publishing METADATA manually

If 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):

3. Push your changes to your fork:

4. If all of that works, then go back to the GitHub page for your fork, and click the "pull request" link.

35.4 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:

Since the commit hash stays the same, the contents of the file that will be checked out in the repo will **not** match the requirements in after such a change; this is unavoidable. When you fix the requirements in for a previous version of a package, however, you should also fix the file in the current version of the package.

35.5 Requirements Specification

The file, the file inside packages, and the package files use a simple line-based format to express the ranges of package versions which need to be installed. Package and files should also include the range of versions of the package is expected to work with. Additionally, packages can include a 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:

This requirement is satisfied by any version of the package. The package name can be followed by zero or more version numbers in ascending order, indicating acceptable intervals of versions of that package. One version opens an interval, while 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:

is satisfied by any version of greater than or equal to . Suffixing a version with allows any pre-release versions as well. For example:

is satisfied by pre-release versions such as `1.0.0-rc1` or `1.0.0-rc2`, or by any version greater than or equal to `1.0.0`.

This requirement entry:

is satisfied by versions from `1.0.0` up to, but not including `2.0.0`. If you want to indicate that any version will do, you will want to write:

If you want to start accepting versions after `1.0.0`, you can write:

If a requirement line has leading words that begin with `system`, 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:

will require the `package` only on systems where the operating system is OS X. The system conditions that are currently supported are (hierarchically):

- - `osx`
 - `linux`
 - `bsd`
 - `unix`
 - `*`
-

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 `system`. Examples:

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 `String`. Such code is occasionally necessary to keep track of new or deprecated functionality between various releases of Julia. Examples of runtime checks:

See the section on [version number literals](#) for a more complete description.

Chapter 36

Profiling

The 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.

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 operating system load). Moreover, as discussed further below, because samples 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.

36.1 Basic usage

Let's work with a simple test case:

|

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):

Now we're ready to profile this function:

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:

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 `.jl`. This is the function that runs the REPL when you launch Julia. If you examine line 97 of `.jl`, you'll see this is where the function `repl_readline` 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 we entered `.`. The next line reflects actions taken in the `macro`.

The first line shows that 80 backtraces were taken at line 73 of `.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 `repl_readline`, 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:

`repl_readline` refers to the fact that we defined `repl_readline` in the REPL, rather than putting it in a file; if we had used a file, this would show the file name. The `repl_readline` shows that the function `repl_readline` was the first expression evaluated in this REPL session. Line 2 of `.jl` contains the call to `repl_readline`, and there were 52 (out of 80) backtraces that occurred at this line. Below that, you can see a call to `repl_readline` inside `repl_readline`.

A little further down, you see:

Line 3 of `.jl` contains the call to `repl_readline`, and there were 28 (out of 80) backtraces taken here. Below that, you can see the specific places in `.jl` that carry out the time-consuming operations in the `repl_readline` 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:

In general, if you have samples collected at a line, you can expect an uncertainty on the order of (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 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:

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:

If you were to profile `recurse`, and a backtrace was taken while it was executing `recurse`, the backtrace would look like this:

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.

36.2 Accumulation and clearing

Results from `accumulate` in a buffer; if you run multiple pieces of code under `accumulate`, then `show` will show you the combined results. This can be very useful, but sometimes you want to start fresh; you can do so with `clear`.

36.3 Options for controlling the display of profile results

`Profile` has more options than we've described so far. Let's see the full declaration:

Let's first discuss the two positional arguments, and later the keyword arguments:

- `buffer` – Allows you to save the results to a buffer, e.g. a file, but the default is to print to `IO` (the console).
- `backtraces` – Contains the data you want to analyze; by default that is obtained from `backtraces`, which pulls out the backtraces from a pre-allocated buffer. For example, if you want to profile the profiler, you could say:

The keyword arguments can be any combination of:

- `showtree` – Introduced above, determines whether backtraces are printed with (default, `true`) or without (`false`) indentation indicating tree structure.
- `showc` – If `true`, backtraces from C and Fortran code are shown (normally they are excluded). Try running the introductory example with `showc=true`. This can be extremely helpful in deciding whether it's Julia code or C code that is causing a bottleneck; setting `showc=true` also improves the interpretability of the nesting, at the cost of longer profile dumps.
- `showops` – Some lines of code contain multiple operations; for example, `sum(a)` contains both an array reference (`a`) and a sum operation. These correspond to different lines in the generated machine code, and hence there may be two or more different addresses captured during backtraces on this line. `showops=true` lumps them together, and is probably what you typically want, but you can generate an output separately for each unique instruction pointer with `showops=false`.
- `depth` – Limits frames at a depth higher than `depth` in the `format` format.
- `order` – Controls the order in `format` format. (default) sorts by the source line, whereas `order=:samples` sorts in order of number of collected samples.
- `noise` – Limits frames that are below the heuristic noise floor of the sample (only applies to `format=:samples`). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which `noise` is the number of samples on this line, and `noise` is the number of samples for the callee.

- – Limits frames with less than occurrences.

File/function names are sometimes truncated (with `...`), and indentation is truncated with a `~` 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 `indent` in an `IOContext`:

```
|
```

36.4 Configuration

`Profile` just accumulates backtraces, and the analysis happens when you call `show`. 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:

```
|
```

`max_backtraces` is the total number of instruction pointers you can store, with a default value of `50000`. 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 `0.1`. 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 37

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 `@timev`, 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 `--profile` command-line option, for which you can choose `none` (the default, do not measure allocation), `all` (measure memory allocation everywhere except Julia's core code), or `line` (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 `profile` appended after the file name, residing in the same directory as the source file. Each line lists the total number of bytes allocated. The `ProfileView` package contains some elementary analysis tools, for example `sort` to sort the lines in order of number of bytes allocated.

In interpreting the results, there are a few important details. Under the `line` 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 significantly, JIT-compilation also adds to allocation counts, because much of Julia's 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 `ProfileView.reset()` to reset all allocation counters. Finally, execute the desired commands and quit Julia to trigger the generation of the `profile` files.

Chapter 38

Stack Traces

The `trace` module provides simple stack traces that are both human readable and easy to use programmatically.

38.1 Viewing a stack trace

The primary function used to obtain a stack trace is :

|

Calling `traceback.print_stack()` returns a vector of `s`. For ease of use, the alias `traceback.print_stack` can be used in place of `s`. (Examples with `s` indicate that output may vary depending on how the code is run.)

|

Note that when calling `stacktrace` you'll typically see a frame with `stacktrace`. When calling `stacktrace` from the REPL you'll also have a few extra frames in the stack from `stacktrace`, usually looking something like this:

38.2 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 `stacktrace`:

This makes stack trace information available programmatically for logging, error handling, and more.

38.3 Error handling

While having easy access to information about the current state of the callstack can be helpful in many places, the most obvious application is in error handling and debugging.

You may notice that in the example above the first stack frame points to line 4, where `is` is called, rather than line 2, where `bad_function` is called, and `'s` frame is missing entirely. This is understandable, given that `is` 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 `std::current_exception()` instead of `std::current_exception_ptr()`. Instead of returning callstack information for the current context, `std::current_exception()` returns stack information for the context of the most recent exception:

Notice that the stack trace now indicates the appropriate line number and the missing frame.

38.4 Comparison with

A call to `stacktrace` returns a vector of `stacktrace_entry`, which may then be passed into `stacktrace_to_string` for translation:

Notice that the vector returned by `ls` had 21 pointers, while the vector returned by `ls2` only has 5. This is because, by default, `ls` 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:

Individual pointers returned by `ls` can be translated into `ls2` by passing them into :

Chapter 39

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.

39.1 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:

|

Uses of non-constant globals can be optimized by annotating their types at the point of use:

|

Writing functions is better style. It leads to more reusable code and 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:

|

is equivalent to:

|

so all the performance issues discussed previously apply.

39.2 Measure performance with `bench` and pay attention to memory allocation

A useful tool for measuring performance is the `bench` macro. The following example illustrates good working style:

```
def fib(n) = {
  // ...
}
```

On the first call `(fib 10)`, `bench` gets compiled. (If you've not yet used `bench` 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. This is the single biggest advantage of `bench` vs. functions like `time` and `timeit`, which only report time.

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 `bench` macro in global scope) and has an order of magnitude faster execution after the first call:

```
def fib(n) = {
  // ...
}
```

Below you'll learn how to spot the problem with `bench` 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](#)).

39.3 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 `@memory`, `@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 `--profile` option and examine the resulting `profile.log` files to see information about where those allocations occur. See [Memory allocation analysis](#).
- `@code_warntest` generates a representation of your code that can be helpful in finding expressions that result in type uncertainty. See [below](#).
- The [Lint](#) package can also warn you of certain types of programming errors.

39.4 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:

```
|
```

Because `AbstractArray{T}` is an array of abstract type `T`, it must be able to hold any `T` value. Since `T` objects can be of arbitrary size and structure, `AbstractArray{T}` must be represented as an array of pointers to individually allocated `T` objects. Because `AbstractArray{T}` will always be a `Vector{T}`, we should instead, use:

```
|
```

which will create a contiguous block of 64-bit floating-point values that can be manipulated efficiently.

See also the discussion under [Parametric Types](#).

39.5 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 Julia, the compiler generally knows the types of all function arguments, local 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:

```
|
```

This allows `o` to be of any type. This can often be useful, but it does have a downside: for objects of type `o`, 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 `o` :

`o` and `o2` have the same type, yet their underlying representation of data in memory is very different. Even if you stored just numeric values in field `o`, the fact that the memory representation of a `Double` differs from a `Float` 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 `o`. Here, we are focused on the case where `o` might be any one of several types, in which case the natural solution is to use parameters. For example:

This is a better choice than

because the first version specifies the type of `o` from the type of the wrapper object. For example:

The type of field `o` can be readily determined from the type of `o`, but not from the type of `o2`. Indeed, in `o2` it's possible to change the type of field `o` :

In contrast, once `is` is constructed, the type of `is` cannot change:

```
|
```

The fact that the type of `is` is known from `is`'s type—coupled with the fact that its type cannot change mid-function—allows the compiler to generate highly-optimized code for objects like `is` but not for objects like `is`.

Of course, all of this is true only if we construct `is` with a concrete type. We can break this by explicitly constructing it with an abstract type:

```
|
```

For all practical purposes, such objects behave identically to those of `is`.

It's quite instructive to compare the sheer amount code generated for a simple function

```
|
```

using

```
|
```

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:

```
|
```

For example:

For `Vec`, 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 `Vec`. Usually the best way to achieve this is to wrap your specific operation (here, `dot`) in a separate function:

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 `Vec`. You could do it like this:

This works fine for `Vec`, but we'd also have to write explicit versions for `Vec2` or other abstract types. To prevent such tedium, you can use two parameters in the declaration of `dot`:

Note the somewhat surprising fact that `Array` doesn't appear in the declaration of `field`, a point that we'll return to in a moment. With this approach, one can write functions such as:

Note

Because we can only define `for` for `Array`, and any unspecified parameters are arbitrary, the first function above could have been written more succinctly as

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 `field` has element type `T`, so it's perfectly possible to construct an object like this:

To prevent this, we can add an inner constructor:

The inner constructor requires that the element type of `be` be `.`

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 `.`). 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:

Here, we happened to know that the first element of `be` would be an `.`. 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.

Declare types of keyword arguments

Keyword arguments can have declared types:

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 `.`, can be slow and should be avoided in performance-sensitive code.

39.6 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:

```
|
```

This can be written more concisely and efficiently as:

```
|
```

39.7 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:

```
|
```

Although this seems innocent enough, the problem is that `i` is an integer (of type `int`) and `val` might be of any type. Thus, depending on the value of `val`, 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:

```
|
```

There is also a `float` function, and a more general `convert` function, which returns `val` converted to the type of `i`.

39.8 Avoid changing the type of a variable

An analogous "type-stability" problem exists for variables used repeatedly within a function:

```
|
```

Local variable `i` starts as an integer, and after one loop iteration becomes a floating-point number (the result of `op` operator). This makes it more difficult for the compiler to optimize the body of the loop. There are several possible fixes:

- Initialize `arr` with
- Declare the type of `arr`:
- Use an explicit conversion:
- Initialize with the first loop iteration, to `0`, then loop

39.9 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:

```
int* rand_type(int n) {
    int i;
    int* arr = (int*) malloc(n * sizeof(int));
    for (i = 0; i < n; i++) {
        arr[i] = rand() % 1000000;
    }
    return arr;
}
```

This should be written as:

```
int* rand_type(int n) {
    int* arr = (int*) malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 1000000;
    }
    return arr;
}
```

Julia's compiler specializes code for argument types at function boundaries, so in the original implementation it does not know the type of `rand` 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 `for` for different types of `T`.

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 `rand` in `Base`, or the `rand` function, which we could have used instead of writing our own `rand`.

Functions like `rand` occur when dealing with data of uncertain type, for example data loaded from an input file that might contain either integers, floats, strings, or something else.

39.10 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:

This approach works very well: the compiler can figure out that `fill` is an `Array{T,N}` because it knows the type of the fill value (`0`) and the dimensionality (`N`). This implies that the compiler can generate very efficient code for any future usage of `fill` in the same function.

But now let's say you want to write a function that creates a `3×3×...` array in arbitrary dimensions; you might be tempted to write a function

This works, but (as you can verify for yourself using `typeof`) the problem is that the output type cannot be inferred: the argument `dim` 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 `z`; 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 `dim` (see ["Value types"](#)):

```

function f(x::Vector{<T>}, y::Vector{<T>}, z::Vector{<T>}, dim::Int) where {<T>}
    for i = 1:dim
        z[i] = x[i] + y[i]
    end
end

```

Julia has a specialized version of `f` that accepts `dim` as the second parameter; by passing `dim` as a type-parameter, you make its "value" known to the compiler. Consequently, this version of `f` allows the compiler to predict the return type.

However, making use of such techniques can be surprisingly subtle. For example, it would be of no help if you called from a function like this:

```

function g(x::Vector{<T>}, y::Vector{<T>}, z::Vector{<T>}) where {<T>}
    f(x, y, z, dim)
end

```

Here, you've created the same problem all over again: the compiler can't guess the type of `dim`, so it doesn't know the type of `f`. Attempting to use `f`, but doing so incorrectly, can easily make performance *worse* in many situations. (Only in situations where you're effectively combining `f` 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 `f` would be:

```

function h(x::Vector{<T>}, y::Vector{<T>}, z::Vector{<T>}) where {<T>}
    f(x, y, z, Val{length(x)})
end

```

In this example, `dim` is passed as a parameter, so its "value" is known to the compiler. Essentially, `f` works only when `dim` is either hard-coded (`Val{length(x)}`) or already specified in the type-domain.

39.11 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 S
    x::Int
    y::Int
end

```


and then dispatch on objects like .

This might be worthwhile when the following are true:

- You require CPU-intensive processing on each , and it becomes vastly more efficient if you know the and at compile time.
- You have homogenous lists of the same type of to process, so that you can store them all in an .

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 has a different type than , 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 compile-time impact: Julia will compile specialized functions for each different ; if you have hundreds or thousands of such types, then every function that accepts such an object as a parameter (from a custom function you might write yourself, to the generic 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.

39.12 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 function or the syntax as shown below (notice that the array is ordered , not):

```

julia> reshape{Int64}(1:100, 10, 10)
100-element Array{Int64,2}:
 1  11  21  31  41  51  61  71  81  91
10 20 30 40 50 60 70 80 90
21 31 41 51 61 71 81 91
31 41 51 61 71 81 91
41 51 61 71 81 91
51 61 71 81 91
61 71 81 91
71 81 91
81 91
91

```

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 row-major ordering, which is the convention adopted by C and Python () 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` and returns a square 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 `:`):

```
|
```

Now we will time each of these functions using the same random `by` input vector:

```
|
```

Notice that `col` is much faster than `row`. This is expected because `col` respects the column-based memory layout of the `matrix` and fills it one column at a time. Additionally, `col` is much faster than `row` 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.

39.13 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

```
|
```

with

```
|
```

Timing results:

```
|
```

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 `Vector{Float64}` rather than an `Array{Float64,1}`, had we so desired.

Taken to its extreme, pre-allocation can make your code uglier, so performance measurements and some judgment may be required. However, for "vectorized" (element-wise) functions, the convenient syntax `dot` can be used for in-place operations with fused loops and no temporary arrays (see the [dot syntax for vectorizing functions](#)).

39.14 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 `dot` and `mul!` are defined, it can be advantageous to instead use `dot!` and `mul!` because the resulting loops can be fused with surrounding computations. For example, consider the two functions:

```
function f1(A::Array{Float64,2})
    for i in 1:size(A,1)
        for j in 1:size(A,2)
            A[i,j] = A[i,j] + A[i,j]
        end
    end
end
```

Both `f1` and `f2` compute the same thing. However, `f2` (defined with the help of the `@dot` macro) is significantly faster when applied to an array:

```
function f2(A::Array{Float64,2})
    @dot for i in 1:size(A,1)
        for j in 1:size(A,2)
            A[i,j] = A[i,j] + A[i,j]
        end
    end
end
```

That is, `f2` is three times faster and allocates 1/7 the memory of `f1`, because each `+=` operation in `f1` allocates a new temporary array and executes in a separate loop. (Of course, if you just do `A += A` then it is as fast as `f2` 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.)

39.15 Consider using views for slices

In Julia, an array "slice" expression like `A[1:10]` creates a copy of that data (except on the left-hand side of an assignment, where `A[1:10] = ...` assigns in-place to that portion of `A`). 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{T, N, A, I}`) 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(A, 1:10)`, or more simply for a whole expression or block of code by putting `viewof` in front of that expression. For example:

```
|
```

Notice both the 3× speedup and the decreased memory allocation of the version of the function.

39.16 Avoid string interpolation for I/O

When writing data to a file (or other I/O device), forming extra intermediate strings is a source of overhead. Instead of:

```
|
```

use:

```
|
```

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:

```
|
```

versus:

```
|
```

39.17 Optimize network I/O during parallel execution

When executing a remote function in parallel:

```
|
```

is faster than:

```
|
```

The former results in a single network round-trip to every worker, while the latter results in two network calls - first by the `for` and the second due to the `map` (or even a `do`). The `map` is also being executed serially resulting in an overall poorer performance.

39.18 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.

39.19 Tweaks

These are some minor points that might help in tight inner loops.

- Avoid unnecessary arrays. For example, instead of `use` `Array{...}` use `Vector{...}`.
- Use `for` instead of `foreach` for complex `for`. In general, try to rewrite code to use `for` instead of `foreach` for complex arguments.
- Use `div` for truncating division of integers instead of `÷`, instead of `÷`, and `÷` instead of `÷`.

39.20 Performance Annotations

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 `fast-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.

Note: While `@inbounds` needs to be placed directly in front of a loop, both `@fastmath` and `@simd` can be applied to several statements at once, e.g. using `...; ...`, or even to a whole function.

Here is an example with both `@fastmath` and `@simd` markup:

```

@fastmath @simd for i in 1:N
    for j in 1:M
        x[i,j] = x[i,j] + 1
    end
end

```

```

// Loop with reduction
double sum = 0.0;
for (int i = 0; i < N; i++)
    sum += a[i];

```

On a computer with a 2.4GHz Intel Core i5 processor, this produces:

```

// Loop with reduction
double sum = 0.0;
for (int i = 0; i < N; i++)
    sum += a[i];

```

(measures the performance, and larger numbers are better.) The range for a loop should be a one-dimensional range. A variable used for accumulating, such as in the example, is called a *reduction variable*. By using , 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 .
- No iteration ever waits on another iteration to make forward progress.

A loop containing , , or will cause a compile-time error.

Using 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 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 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 .

Here is an example with all three kinds of markup. This program first calculates the finite difference of a one-dimensional array, and then evaluates the L2-norm of the result:

On a computer with a 2.7 GHz Intel Core i7 processor, this produces:

Here, the option disables the macro, so that we can compare results.

In this case, the speedup due to 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 is small enough to fit into the L1 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 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 in the function is hoisted out of the loop (i.e. calculated outside the loop), as if one had written . In the loop, the expression then becomes , 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 function.

39.21 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 grants permission for floating-point operations to treat subnormal inputs or outputs as zeros, which may improve performance on some hardware. A call enforces strict IEEE behavior for subnormal numbers.

Below is an example where subnormals noticeably impact performance on some hardware:



This example generates many subnormal numbers because the values in 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 implies :

In some applications, an alternative to zeroing subnormal numbers is to inject a tiny bit of noise. For example, instead of initializing with zeros, initialize it with:

39.22

The macro (or its function variant) can sometimes be helpful in diagnosing type-related problems. Here's an example:

Interpreting the output of `type`, like that of its cousins `typeof`, `typeof`, and `typeof`, 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 `<type>` (where `<type>` might be `number`, for example). The most important characteristic of `type` 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 `x`, one of the variables you created, is a `number`, due to the type-instability of `Math.random()`. There is another variable, `y`, which you can see also has the same type.

The next lines represent the body of `foo`. The lines starting with a number followed by a colon (`1:`) are labels, and represent targets for jumps (via `goto`) in your code. Looking at the body, you can see that `foo` has been *inlined* into `foo`—everything before comes from code defined in `foo`.

Starting at `1:`, the variable `x` is defined, and again annotated as a `number`. Next, we see that the compiler created the temporary variable `temp` to hold the result of `Math.random()`. Because a `number` times *either* an `number` or yields a `number`, all type-instability ends here. The net result is that `foo` 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 `foo` to be type-stable: if you did so, all of the variables in `foo` 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 `foo` is never used in isolation, the fact that `foo`'s output is type-stable (for `number` 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
 - Interpretation: function with unstable return type
 - Suggestion: make the return value type-stable, even if you have to annotate it
- - Interpretation: call to a type-unstable function
 - Suggestion: fix the function, or if necessary annotate the return value
- - Interpretation: accessing elements of poorly-typed arrays
 - Suggestion: use arrays with better-defined types, or if necessary annotate the type of individual element accesses

- - Interpretation: getting a field that is of non-leaf type. In this case, `had` a field `.`. But `needs` the dimension `,` too, to be a concrete type.
 - Suggestion: use concrete types like `or` , where `is` now a parameter of

Chapter 40

Workflow Tips

Here are some tips for working with Julia efficiently.

40.1 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 `command` line. A common pattern includes the following elements:

- **Put code under development in a temporary module.** Create a file, say `temp.jl`, and include within it

```
|
```

- **Put your test code in another file.** Create another file, say `test.jl`, which begins with

```
|
```

and includes tests for the contents of `temp.jl`. The value of using `temp.jl` versus `test.jl` is that you can call `temp.jl` instead of having to restart the REPL when your definitions change. Of course, the cost is the need to prepend `temp.jl` 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

```
|
```

The advantage is that you can now do `temp.jl` in your test code and can therefore avoid prepending `temp.jl` 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 `command` prompt. Save good ideas in `.`. Occasionally restart the REPL, issuing

```
|
```

Simplify initialization

To simplify restarting the REPL, put project-specific initialization code in a file, say `.`, which you can run on startup by issuing the command:

```
|
```

If you further add the following to your `file`

```
|
```

then calling `from` that directory will run the initialization code without the additional command line argument.

40.2 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 41

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.

41.1 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 `π`).

41.2 Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
|
```

it's better to use available generic functions:

```
|
```

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 `Integer` or `Int` 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 `TypeError` 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 `add` that returns one plus its argument:

```
|
```

The last definition of `is_integer` handles any type supporting `is_integer` (which returns 1 in the same type as `is_integer`, which avoids unwanted type promotion) and the `is_integer` function with those arguments. The key thing to realize is that there is *no performance penalty* to defining *only* the general `is_integer`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `is_integer`, Julia will automatically compile a specialized `is_integer` function for `Int` arguments, with the call to `is_integer` replaced by its inlined value. Therefore, the first three definitions of `is_integer` above are completely redundant with the fourth definition.

41.3 Handle excess argument diversity in the caller

Instead of:

```
|
```

use:

```
|
```

This is better style because `is_integer` does not really accept numbers of all types; it really needs `is_integer` `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.

41.4 Append to names of functions that modify their arguments

Instead of:

```
|
```

use:

```
|
```

The Julia standard library uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `copy` and `copy!()`), and others which are just modifying (e.g., `sort`, `sort!`, `sortperm`, `sortperm!`). It is typical for such functions to also return the modified array for convenience.

41.5 Avoid strange type s

Types such as are often a sign that some design could be cleaner.

41.6 Avoid type Unions in fields

When creating a type such as:

```
|
```

ask whether the option for to be (of type) is really necessary. Here are some alternatives to consider:

- Find a safe default value to initialize with
- Introduce another type that lacks
- If there are many fields like , store them in a dictionary
- Determine whether there is a simple rule for when is . For example, often the field will start as but get initialized at some well-defined point. In that case, consider leaving it undefined at first.
- If really needs to hold no value at some times, define it as instead, as this guarantees type-stability in the code accessing this field (see [Nullable types](#)).

41.7 Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
|
```

In this case is better. It is also more helpful to the compiler to annotate specific uses (e.g.) than to try to pack many alternatives into one type.

41.8 Use naming conventions consistent with Julia's

- modules and type names use capitalization and camel case: , .
- functions are lowercase (,) and, when readable, with multiple words squashed together (,). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (as a more efficient implementation of) or as modifiers ().
- conciseness is valued, but avoid abbreviation (rather than) 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.

41.9 Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

41.10 Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
|
```

instead of:

```
|
```

41.11 Don't overuse

Splicing function arguments can be addictive. Instead of `..., ...`, use simply `...`, which already concatenates arrays. `...` is better than `...`, but since `...` is already iterable it is often even better to leave it alone, and not convert it to an array.

41.12 Don't use unnecessary static parameters

A function signature:

```
|
```

should be written as:

```
|
```

instead, especially if `...` is not used in the function body. Even if `...` is used, it can be replaced with `...` 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 `Vector`, may need type parameters in function calls. See the FAQ [Avoid fields with abstract containers](#) for more information.

41.13 Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
|
```

Decide whether the concept in question will be written as `...` or `...`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `...` 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.

41.14 Don't overuse macros

Be aware of when a macro could really be a function instead.

Calling `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.

41.15 Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
|
```

don't write definitions like the following:

```
|
```

The problem is that users of this type can write `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.

41.16 Don't overload methods of base container types

It is possible to write definitions like the following:

```
|
```

This would provide custom 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{T}` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

41.17 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:

```
|
```

The problem is that now any other module that uses `is` will also see this definition. Since `is` 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 `is` 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 might provide some types useful for working with colors; another package could define methods for those types that enable conversions between 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.

41.18 Be careful with type equality

You generally want to use `is` and `isa` for testing types, not `==`. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `Int`), or if you *really, really* know what you're doing.

41.19 Do not write

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 `f(x) = ...`, write `f(x)`.

41.20 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,

```
|
```

while

```
|
```

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. `1.0`, and promotion happens with the multiplication. Similarly, `1.0` literals are less type disruptive than `1` literals, but more disruptive than `s`:

|

Thus, use `1.0` literals when possible, with `1.0` for literal non-integer numbers, in order to make it easier to use your code.

Chapter 42

Frequently Asked Questions

42.1 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 `Base`), it is always present.

If memory usage is your concern, you can always replace objects with ones that consume less memory. For example, if `x` is a gigabyte-sized array that you no longer need, you can free the memory with `x = nothing`. The memory will be released the next time the garbage collector runs; you can force this to happen with `@gc`.

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:

```
|
```

Types in module `Base` cannot be redefined.

While this can be inconvenient when you are developing new code, there's an 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 `Base` 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:

```
|
```

42.2 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:

```
function foo(x)
  x = 1
end

foo(2)
```

In Julia, the binding of a variable cannot be changed by passing as an argument to a function. When calling in the above example, is a newly created variable, bound initially to the value of , i.e. ; then is rebound to the constant , while the variable of the outer scope is left untouched.

But here is a thing you should pay attention to: suppose is bound to an object of type (or any other *mutable* type). From within the function, you cannot "unbind" from this Array, but you can change its content. For example:

```
function foo(x)
  x[1] = 1
end

x = [2, 3, 4]
foo(x)
x
```

Here we created a function , that assigns to the first element of the passed array (bound to at the call site, and bound to within the function). Notice that, after the function call, is still bound to the same array, but the content of that array changed: the variables and were distinct bindings referring to the same mutable object.

Can I use or inside a function?

No, you are not allowed to have a or 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 :

```
|
```

This loads the module and defines a variable that refers to the module, but does not import any of the other symbols from the module into the current namespace. You refer to the symbols by their qualified names etc.

2. Wrap your function in a module:

```
|
```

This imports all the symbols from , but only inside the module .

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:

```
|
```

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:

If Julia were a language that made more liberal use of ASCII characters, the splatting operator might have been written as `...` instead of `...`.

42.3 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:

It returns either an `Int` or a `Complex{<:Real}` depending on the value of its argument. Since Julia can't predict the return type of this function at compile-time, any computation that uses it will have to guard against both types possibly occurring, making generation of fast machine code difficult.

Why does Julia give a `Complex{<:Real}` for certain seemingly-sensible operations?

Certain operations make mathematical sense but result in errors:

```
julia> 1 + 2im
Complex{Im{Float64}}(1.0 + 2.0im)
```

This behavior is an inconvenient consequence of the requirement for type-stability. In the case of `+`, most users want to give a real number, and would be unhappy if it produced the complex number `Complex{Im{Float64}}`. One could write the `+` function to switch to a complex-valued output only when passed a negative number (which is what `+` does in some other languages), but then the result would not be `type-stable` and the `+` 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> 1 + 2im::Complex{Im{Float64}}
Complex{Im{Float64}}(1.0 + 2.0im)
```

Why does Julia use native machine integer arithmetic?

Julia uses machine arithmetic for integer computations. This means that the range of `Integer` values is bounded and wraps around at either end so that adding, subtracting and multiplying integers can overflow or underflow, leading to some results that can be unsettling at first:

```
julia> 2^31 + 1
-2147483648
```

One alternative to consider would be to check each integer operation for overflow and promote results to bigger integer types such as `int64_t` or `long` 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.

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:

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 or 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:

```
|
```

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 `a` and `b` in Julia using the expression :

```
|
```

See? No problem. That's the correct midpoint between 2^{62} and 2^{63} , despite the fact that `a + b` is -4611686018427387904. Now try it in Matlab:

```
|
```

Oops. Adding a `>>` operator to Matlab wouldn't help, because saturation that occurs when adding `a` and `b` has already destroyed the information necessary to compute the correct midpoint.

Not only is lack of associativity unfortunate for programmers who cannot rely 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 `midpoint`. The machine code for this function is just this:

```
|
```

The actual body of the function is a single instruction, which computes the integer multiply and add at once. This is even more beneficial when gets inlined into another function:

Since the call to gets inlined, the loop body ends up being just a single instruction. Next, consider what happens if we make the number of loop iterations fixed:

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 having integer arithmetic silently overflow 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 `ArgumentError` during remote execution?

As the error states, an immediate cause of an `ArgumentError` on a remote node is that a binding by that name does not exist. Let us explore some of the possible causes.

The closure `closure` carries a reference to `env`, and since `env` is unavailable on node 2, an `ArgumentError` is thrown.

Globals under modules other than `Base` are not serialized by value to the remote node. Only a reference is sent. Functions which create global bindings (except under `Base`) may cause an `ArgumentError` to be thrown later.

In the above example, `defined` on all nodes. However the call to `created` created a new global binding on the local node, but this was not found on node 2 resulting in an error.

Note that this does not apply to globals created under `module`. Globals under `module` are serialized and new bindings created under on the remote node.

This does not apply to `or` declarations. However, anonymous functions bound to global variables are serialized as can be seen below.

42.4 Packages and Modules

What is the difference between "using" and "importall"?

There is only one difference, and on the surface (syntax-wise) it may seem very minor. The difference between `and` is that with `you` need to say `to` extend module `Foo`'s function bar with a new method, but with `or`, you only need to say `and` it automatically extends module `Foo`'s function bar.

If you use `,` then `and` become equivalent. If you use `,` then they are different.

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 `using Foo`, then you'll replace the other module's implementation of `foo` 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 `foo`).

42.5 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 `isnothing` 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 `Nothing`. This is an ordinary type with no fields; there is nothing special about it except for this convention, and that the REPL does not print anything for it. Some language constructs that would not otherwise have a value also yield `nothing`, for example `pop!`.

For situations where a value exists only sometimes (for example, missing statistical data), it is best to use the `Missing` 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 `Union{}`, 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 `Any` (an empty union type). You will generally not need to use this type.

42.6 Memory

Why does `push!` allocate memory when `push` and `push!` are arrays?

In Julia, `push` gets replaced during parsing by `push!`. For arrays, this has the consequence that, rather than storing the result in the same location in memory as `push`, 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 `push!(arr, 1)` do not modify the meaning of `arr`, they modify the value bound to `arr`. For an immutable, the only way to change the value is to reassign it.

To amplify a bit further, consider the following function:

```
function f(x)
```

After a call like `f([1, 2, 3])`, you would get the expected result: `[1, 2, 3]`. However, now suppose that `f`, when used with matrices, instead mutated the left hand side. There would be two problems:

- For general square matrices, `matrixProduct` cannot be implemented without temporary storage: `matrixProduct` 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 `matrixProduct` work in-place); if you took advantage of the mutability of `Matrix`, then this function would behave differently for mutable vs. immutable inputs. In particular, for immutable `Matrix`, after the call you'd have (in general) `matrixProduct`, but for mutable `Matrix` you'd have `matrixProduct`.

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.

42.7 Asynchronous IO and concurrent synchronous writes

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:

```
|
```

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.

and "lock" the stream during a call. Consequently changing `lock` to `lock` in the above example results in:

```
|
```

You can lock your writes with a `lock` like this:

```
|
```

42.8 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, but find the slightly faster rate of changes 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 43

Noteworthy Differences from other Languages

43.1 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, `[]`.
- Julia arrays are assigned by reference. After `a = b`, changing elements of `a` will modify `b` 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 = zeros(1,5)` can create the array `a` and `a = zeros(1,10)` can grow it into size 10, the corresponding Julia statement `a = zeros{Float64,1}(10)` 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 `zeros{Float64,1}(10)` and `zeros{Float64,1}(5,5)`, which grow `a` much more efficiently than MATLAB's `a`.
- The imaginary unit `i` is represented in Julia as `im`, not `i` or `1i` as in MATLAB.
- In Julia, literal numbers without a decimal point (such as `1`) create integers instead of floating point numbers. Arbitrarily large integer literals are supported. As a result, some operations such as `1/2` 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 = func()` or `(a, b) = func()`. MATLAB's `switch`, 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 `10`, not `10x1`. For example, `zeros{Float64,1}(10)` makes a 1-dimensional array.
- In Julia, `zeros{Float64,1}(10)` will always construct a 3-element array containing `0`, `0`, and `0`.
 - To concatenate in the first ("vertical") dimension use either `zeros{Float64,1}(10,2)` or separate with semicolons `[zeros{Float64,1}(10), zeros{Float64,1}(10)]`.
 - To concatenate in the second ("horizontal") dimension use either `zeros{Float64,1}(10,2)` or separate with spaces `[zeros{Float64,1}(10) zeros{Float64,1}(10)]`.
 - To construct block matrices (concatenating in the first two dimensions), use either `zeros{Float64,1}(10,2)` or combine spaces and semicolons `[zeros{Float64,1}(10) zeros{Float64,1}(10)]`.
- In Julia, `zeros{Float64,1}(10)` and `zeros{Float64,1}(10)` construct objects. To construct a full vector like in MATLAB, use `zeros{Float64,1}(10)`. Generally, there is no need to call `zeros{Float64,1}(10)` though. `zeros{Float64,1}(10)` 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 `zeros{Float64,1}(10)`, or with iterators such as `zeros{Float64,1}(10)` and `zeros{Float64,1}(10)`. 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 `all` 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 `sum` that operate column-wise by default (`sum` is equivalent to `sum'`) do not have special behavior for arrays; the argument is returned unmodified since it still performs `sum`. To sort a matrix like a vector, use `sort'`.
- In Julia, if `a` is a 2-dimensional array, `fft(a)` computes a 2D FFT. In particular, it is not equivalent to `fft'`, which computes a 1D FFT acting column-wise.
- In Julia, parentheses must be used to call a function with zero arguments, like in `sum()` and `sort()`.
- 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 `show` 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 `isequal(a, b)`, and similarly for the other boolean operators like `<`, `>`, and `<=`.
- In Julia, the operators `&`, `&&`, and `!&` perform the bitwise operations equivalent to `&`, `&&`, and `!&` 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[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 `sum(a...)`.
- Julia's `svdvals` 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 `_` is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, `_` is not set when Julia code is run in non-interactive mode.
- Julia's `struct`s do not support dynamically adding fields at runtime, unlike MATLAB's `struct`s. 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 `a(a == 0) = []` or in the statement `a(a == 0) = NaN` to modify `a` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(a == 0, a)` and `filter!(a == 0, a)` as alternatives to the corresponding transliterations `a(a == 0) = []` and `a(a == 0) = NaN`. Using `filter` reduces the use of temporary arrays.
- The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in `cell{1:n}` in MATLAB, is written using the splat operator in Julia, e.g. as `cell[1:n]...`.

43.2 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 `%`, not `mod`. `rem` 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, `x[x > 0]` is equivalent to `x[x != 0]`.
 - In R, `x[x < 0]` is equivalent to `x[x <= 0]`.
 - In Julia, `x[x > 0]` throws a `BoundsError`.
 - In Julia, `x[x < 0]` produces `Array{Union{Nothing, T}, 1}`.
- 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, `x + y` is valid R but the equivalent `x + y` will throw an error in Julia.
- Julia's `do` takes the function first, then its arguments, unlike `do.call` in R. Similarly Julia's equivalent of `do.call` in R is `do` where the function is the first argument.
- Multivariate apply in R, e.g. `apply`, can be written as `map` in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions `map`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `for`, and functions. In lieu of the one-line `if`, Julia allows statements of the form `if`, `while`, and `for`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `(x = 1)`.
- In Julia, `:=`, `+=`, and `-=` are not assignment operators.
- Julia's `begin` creates an anonymous function, like Python.
- Julia constructs vectors using brackets. Julia's `Array{T, N}` is the equivalent of R's `matrix`.
- 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 `conj'` operator. Julia's `conj'` is therefore equivalent to R's `conjT`.
- Julia does not require parentheses when writing `if` statements or `for` loops: use `if` instead of `if()` and `for` instead of `for()`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if 0` in Julia, because `if` statements accept only booleans. Instead, you can write `if false`, `if true`, or `if 1`.
- Julia does not provide `for` and `while`. Instead, use `for` and `while`.
- Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `1.0` are the same. In Julia, they cannot be used interchangeably.
- Julia's `1` and `1.0` are not like R's.

- Julia cannot assign to the results of function calls on the left hand side of an assignment operation: you cannot write `x = f()`.
- 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 `[x, y]`, use `(x, y)`.
- 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 `isnumeric` and `isinteger` act like R's `is.numeric` and `is.integer`.
- 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` and `vcat`, not `c()` and `r()` like in R.
- In Julia, a range like `1:n` is not shorthand for a vector like `1:n` in R, but is a specialized `UnitRange` that is used for iteration without high memory overhead. To convert a range into a vector, use `collect`.
- Julia's `size` and `length` are the equivalent of `nrow` and `ncol` respectively in R, but both arguments need to have the same dimensions. While `length` and `replace` and `in` in R, there are important differences.
- Julia's `sum`, `prod`, `all`, and `any` 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 `M` in Julia and `M` be the same matrix in R. Then `sum(M)` gives the same result as `sum(M)`, but `sum(M, dims=1)` is a row vector containing the sum over each column and `sum(M, dims=2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where `sum(M)` and `sum(M)`. If the second argument is a vector, then it specifies all the dimensions over which the sum is performed, e.g., `sum(M, dims=c(1,2))`. 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 `typeof` type.
- Julia lacks the equivalent of R's `list` or `list`.
- In Julia, `if` does not require parentheses.
- In R, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x[x != 0]` or in the statement `x[x != 0] = NA` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(x, !iszero)` and `filter!(x, !iszero)` as alternatives to the corresponding transliterations `x[x != 0]` and `x[x != 0] = NA`. Using `filter` reduces the use of temporary arrays.

43.3 Noteworthy differences from Python

- Julia requires `end` to end a block. Unlike Python, Julia has no `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. `1:n` in Julia is `1:n+1` 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 `begin`, `end`, 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](#)).
- Julia's updating operators (e.g. `+=`, `-=`, ...) are *not in-place* whereas NumPy's are. This means `x += 1` doesn't change values in `x`, it rather rebinds the name `x` to the result of the right-hand side `x + 1`, which is a new array. Use `x[x] += 1`, explicit loops, or `view`.
- 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 `rand()` returns a new random number every time it is invoked without argument. On the other hand, the function `rand(0.0, 1.0)` returns every time it is called as `rand()`.
- In Julia `%` is the remainder operator, whereas in Python it is the modulus.

43.4 Noteworthy differences from C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension. 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 `a` will modify `b` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `a = a + 1` 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](#)).
- Julia values are passed and assigned by reference. If a function modifies 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 `1`) 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 `BigInt` (if `isbigint`), `Float64`, or the arbitrarily large `Complex{<T>}` type. There are no numeric literal suffixes, such as `u`, `U`, `s`, `S`, 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, `0x1` and `0x01` have type `UInt8`, and `0x100000000` have type `UInt64`, then literals with 5 to 8 hex digits have type `UInt8`, 9 to 16 hex digits type `UInt16` and 17 to 32 hex digits type `UInt32`. This needs to be taken into account when defining hexadecimal masks, for example `0xffffffff` is very different from `0xffffffffu`. 64 bit and 32 bit literals are expressed as `0xffffffffu` and `0xffffffffu < 1 < 0xffffffffu` respectively. Floating point literals are rounded (and not promoted to the `Float64` 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 C/C++. String literals can have values of other variables or expressions interpolated into them, indicated by `{<expr>}` or `{<var>}`, which evaluates the variable name or the expression in the context of the function.
- `#<comment>` indicates a number, and not a single-line comment (which is `#<comment>` in Julia)
- `"""<comment>"""` 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. `(x, y)` or `(x, y) = ...`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `&x, &y`).
- 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 `print` can be used to print specific output. In the REPL, `println` 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 `⊻` performs the bitwise XOR operation, i.e. `⊻` in C/C++. Also, the bitwise operators do not have the same precedence as C/C++, so parenthesis may be required.
- Julia's `^` is exponentiation (pow), not bitwise XOR as in C/C++ (use `⊻` or `⊕` 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.
- Julia does not require parentheses when writing `if` statements or `for` loops: use `if <cond>` instead of `if (<cond>)` and `for <var> in <iter>` instead of `for (<var> in <iter>)`.
- Julia does not treat the numbers `true` and `false` as Booleans. You cannot write `if true` in Julia, because statements accept only booleans. Instead, you can write `if <cond>`, `if !<cond>`, or `if <var>`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `for`, and functions. In lieu of the one-line `if`, Julia allows statements of the form `if <cond> <stmt> end`, and `if <cond> <stmt1> else <stmt2> end`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `(x, y) = ...`, 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, `@<macro>(<args>)`, and a statement-like syntax, `@<macro> <stmt>`. 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` construct: `parallel { ... }`. Where the end of the macro construct may be unclear, use the function-like form.

- Julia now has an enumeration type, expressed using the macro `enum`. For example:

```
enum Color{Red, Green, Blue}
```
- By convention, functions that modify their arguments have a `!` at the end of the name, for example `sort!`.
- 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 `Any`, using the most-specific-declaration rule).

Chapter 44

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 `?<symbol>` 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 45

Essentials

45.1 Introducción

La librería estándar de Julia contiene un rango de funciones y marcos apropiados para realizar computación científica y numérica, pero es también tan amplia como la de muchos lenguajes de programación de propósito general. También hay funcionalidad adicional disponible en una colección creciente de paquetes disponibles. Las funciones están agrupadas abajo por temas.

Algunas notas generales:

- Excepto para las funciones en los módulos predefinidos (`Base`, `Math`, `Statistics`, `Random`, `Compat`, `Logging`, `Printf`, `Base64`, `JSON`, `UUIDs`, `Dates`, `Time`, `TimeSeries`, `Repl`, `InteractiveUtils`, `Base64`, `JSON`, `UUIDs`, `Dates`, `Time`, `TimeSeries`, `Repl`, `InteractiveUtils`), todas las funciones documentadas aquí están disponibles directamente para ser usadas en programas.

- Para usar funciones de módulos, usar `using` para importar el módulo, y

para usar las funciones.

- Alternativamente `import` importará todas las funciones exportadas por el módulo en

el espacio de nombres actual.

- Por convenio, los nombres de funciones que acaban con un signo de admiración (`!`)

modifican sus argumentos. Algunas funciones tienen las dos versiones (con y sin modificación de los argumentos).

45.2 Moviéndose

– `Function`.

|

Quit (or control-D at the prompt). The default exit code is zero, indicating that the processes completed successfully.

– Function.

|

Quit the program indicating that the processes completed successfully. This function calls (see).

– Function.

|

Register a zero-argument function to be called at process exit. hooks are called in last in first out (LIFO) order and run before object finalizers.

– Function.

|

Register a one-argument function to be called before the REPL interface is initialized in interactive sessions; this is useful to customize the interface. The argument of is the REPL object. This function should be called from within the initialization file.

– Function.

|

Determine whether Julia is running an interactive session.

– Function.

|

Print information about exported global variables in a module, optionally restricted to those matching .

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

– Function.

|

Compute the amount of memory used by all unique objects reachable from the argument.

Keyword Arguments

- : specifies the types of objects to exclude from the traversal.
- : specifies the types of objects to always charge the size of all of their fields, even if those fields would normally be excluded.

– Method.

|

Edit a file or directory optionally providing a line number to edit the file at. Returns to the prompt when you quit the editor. The editor can be changed by setting , or as an environment variable.

– Method.

|

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. The editor can be changed by setting , or as an environment variable.

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls the function on the resulting expression.

– Method.

|

Show a file using the default pager, optionally providing a starting line number. Returns to the prompt when you quit the pager.

– Method.

|

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls the function on the resulting expression.

– Method.

|

Send a printed form of to the operating system clipboard ("copy").

- Method.

```
|
```

Return a string with the contents of the operating system clipboard ("paste").

- Function.

```
|
```

Force reloading of a package, even if it has been loaded before. This is intended for use during package development as code is modified.

- Function.

```
|
```

This function is part of the implementation of `using` / `import`, if a module is not already defined in `ENV`. It can also be called directly to force reloading a module, regardless of whether it has been loaded before (for example, when interactively developing libraries).

Loads a source file, in the context of the `module`, on every active node, searching standard locations for files. `load` is considered a top-level operation, so it sets the current `path` but does not use it to search for files (see help for `path`). This function is typically used to load library code, and is implicitly called by `using` to load packages.

When searching for files, `load` first looks for package code under `lib`, then tries paths in the global array `LOAD_PATH`. `load` is case-sensitive on all platforms, including those with case-insensitive filesystems like macOS and Windows.

- Function.

```
|
```

Creates a precompiled cache file for a module and all of its dependencies. This can be used to reduce package load times. Cache files are stored in `cache`, which defaults to `cache`. See [Module initialization and precompilation](#) for important notes.

- Function.

```
|
```

Specify whether the file calling this function is precompilable. If `is_precompilable` is `true`, then `load` throws an exception when the file is loaded by `using` // *unless* the file is being precompiled, and in a module file it causes the module to be automatically precompiled when it is imported. Typically, `is_precompilable` should occur before the `module` declaration in the file, or better yet in order to be backward-compatible with Julia 0.3.

If a module or file is *not* safely precompilable, it should call `is_precompilable` in order to throw an error if Julia attempts to precompile it.

`is_precompilable` should *not* be used in a module unless all of its dependencies are also using `is_precompilable`. Failure to do so can result in a runtime error when loading the module.

- Function.

|

Evaluate the contents of the input source file in the current context. Returns the result of the last evaluated expression of the input file. During including, a task-local include path is set to the directory containing the file. Nested calls to `include` will search relative to that path. All paths refer to files on node 1 when running in parallel, and files will be fetched from node 1. This function is typically used to load source interactively, or to combine files in packages that are broken into multiple source files.

- Function.

|

Like `include`, except reads code from the given string rather than from a file. Since there is no file path involved, no path processing or fetching from node 1 is done.

- Function.

|

In a module, declare that the file specified by `file` (relative or absolute) is a dependency for precompilation; that is, the module will need to be recompiled if this file changes.

This is only needed if your module depends on a file that is not used via `include`. It has no effect outside of compilation.

- Function.

|

Search through all documentation for a string, ignoring case.

- Method.

|

Returns the method of `obj` (a `QObject` object) that would be called for arguments of the given `args`.

If `obj` is an abstract type, then the method that would be called by `obj` is returned.

- Method.

|

Return the module in which the binding for the variable referenced by `name` was created.

- Macro.

|

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the function.

– Function.

|

Returns the method table for .

If is specified, returns an array of methods whose types match.

– Function.

|

Return an array of methods with an argument of type .

The optional second argument restricts the search to a particular module or function (the default is all modules, starting from Main).

If optional is , also return arguments with a parent type of , excluding type .

– Macro.

|

Show an expression and result, returning the result.

– Function.

|

Print information about the version of Julia in use. If the argument is , detailed system information is shown as well.

– Function.

|

Replace the top-level module () with a new one, providing a clean workspace. The previous module is made available as . A previously-loaded package can be accessed using a statement such as .

This function should only be used interactively.

– Keyword.

|

A variable referring to the last computed value, automatically set at the interactive prompt.

45.3 Todos los Objetos

- Function.

|

Determine whether `obj1` and `obj2` are identical, in the sense that no program could distinguish them. Compares mutable objects by address in memory, and compares immutable objects (such as numbers) by contents at the bit level. This function is sometimes called `id`.

|

- Function.

|

Determine whether `obj` is of the given type `type`. Can also be used as an infix operator, e.g. `obj is type`.

- Method.

|

Similar to `is`, except treats all floating-point values as equal to each other, and treats `NaN` as unequal to `NaN`. The default implementation of `is` calls `is`, so if you have a type that doesn't have these floating-point subtleties then you probably only need to define `is`.

`is` is the comparison function used by hash tables (`Dict`). `is` must imply that `is`.

This typically means that if you define your own `is` function then you must define a corresponding `is` (and vice versa). Collections typically implement `is` by calling `is` recursively on all contents.

Scalar types generally do not need to implement `is` separate from `is`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `is`, `is`, and `is`).

|

– Method.

|

Similar to `compare`, except treats all floating-point values as equal to each other, and treats `NaN` as unequal to `NaN`. The default implementation of `compare` calls `compare`, so if you have a type that doesn't have these floating-point subtleties then you probably only need to define `compare`.

`compare` is the comparison function used by hash tables (`HashMap`). `compare` must imply that `compare`.

This typically means that if you define your own `compare` function then you must define a corresponding `compare` (and vice versa). Collections typically implement `compare` by calling `compare` recursively on all contents.

Scalar types generally do not need to implement `compare` separate from `compare`, unless they represent floating-point numbers amenable to a more efficient implementation than that provided as a generic fallback (based on `compare`, `compare`, and `compare`).

|

|

If neither `lhs` nor `rhs` is null, compare them according to their values (i.e. `compare`). Else, return `0` if both arguments are null, and if one is null but not the other: nulls are considered equal.

– Function.

|

Test whether `lhs` is less than `rhs`, 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 `compare`. 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`.

– Method.

|

If neither `lhs` nor `rhs` is null, compare them according to their values (i.e. `compare`). Else, return `0` if only `lhs` is null, and `1` otherwise: nulls are always considered greater than non-nulls, but not greater than another null.

– Function.

|

Return `if` `is` , otherwise return `.` This differs from `or` in that it is an ordinary function, so all the arguments are evaluated first. In some cases, using `instead` of an `statement` can eliminate the branch in generated code and provide higher performance in tight loops.

|

– Function.

|

Compare `and` lexicographically and return -1, 0, or 1 depending on whether `is` less than, equal to, or greater than `, respectively`. This function should be defined for lexicographically comparable types, and `will` call `by` default.

|

– Function.

|

Determine whether `is` lexicographically less than `.`

|

– Function.

|

Get the concrete type of `.`

– Function.

|

Construct a tuple of the given objects.

Example

|

- Function.

|

Create a tuple of length `n`, computing each element as `fn(i)`, where `i` is the index of the element.

|

- Function.

|

Get a hash value for `obj` based on object identity. if `key`.

- Function.

|

Compute an integer hash code such that `hash(x) == hash(y)` implies `x == 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 `__eq__` (hence `__hash__`) to guarantee the property mentioned above.

- Function.

|

Register a function `callback` to be called when there are no program-accessible references to `obj`. The type of `callback` must be a `Callable`, otherwise the behavior of this function is unpredictable.

- Function.

|

Immediately run finalizers registered for object `obj`.

- Function.

|

Create a shallow copy of `obj`: 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.

– Function.

|

Create a deep copy of : 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 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 behavior by defining a specialized version of the function (which shouldn't otherwise be used), where is the type to be specialized for, and keeps track of objects copied so far within the recursion. Within the definition, should be used in place of , and the variable should be updated as appropriate before returning.

– Function.

|

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. With a single symbol argument, tests whether a global variable with that name is defined in .

– Function.

|

Convert to a value of type .

If is an type, an will be raised if is not representable by , for example if is not integer-valued, or is outside the range supported by .

Examples

|

If is a or type, then it will return the closest value to representable by .

|

If `l` is a collection type and `c` a collection, the result of `l[c]` may alias `c`.

Similarly, if `l` is a composite type and `i` a related instance, the result of `l[i]` may alias part or all of `i`.

– Function.

Convert all arguments to their common promotion type (if any), and return them all (as a tuple).

Example

– Function.

Convert `x` to the type of `y` ().

– Function.

|

If `t` 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 `t` is converted to `Object`.

Examples

|

– Function.

|

The identity function. Returns its argument.

|

45.4 Tipos

– Function.

|

Return the supertype of `DataType`.

|

– Function.

|

Return `true` if and only if all values of `t1` are also of `t2`. Can also be written using the infix operator as `t1 <: t2`.

Examples

|

– Function.

|

Subtype operator, equivalent to .

|

– Function.

|

Supertype operator, equivalent to .

– Function.

|

Return a list of immediate subtypes of `DataType` . Note that all currently loaded subtypes are included, including those not visible in the current module.

|

– Function.

|

The lowest value representable by the given (real) numeric `DataType` .

Examples

|

- Function.

|

The highest value representable by the given (real) numeric .

- Function.

|

The smallest in absolute value non-subnormal value representable by the given floating-point DataType .

- Function.

|

The highest finite value representable by the given floating-point DataType .

Examples

|

- Function.

|

The largest integer losslessly representable by the given floating-point DataType .

|

The largest integer losslessly representable by the given floating-point DataType that also does not exceed the maximum integer representable by the integer DataType .

- Method.

|

Size, in bytes, of the canonical binary representation of the given DataType , if any.

Examples

|

If `size` does not have a specific size, an error is thrown.

|

– Method.

|

Returns the *machine epsilon* of the floating point type (by default). This is defined as the gap between 1 and the next largest value representable by `type`, and is equivalent to `type(1) - type(1) - 1`.

|

– Method.

|

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

|

The exceptions to this rule are the smallest and largest finite values (e.g. `type(0)` and `type(1)` for `type`), which round to the smaller of the values.

The rationale for this behavior is that `eps(x)` bounds the floating point rounding error. Under the default rounding mode, if `y` is a real number and `x` is the nearest floating point number to `y`, then

$$|y - x| \leq \text{eps}(x)/2.$$

|

|

– Function.

|

Determine a type big enough to hold values of each argument type without loss, whenever possible. In some cases, where no type exists to which both types can be promoted losslessly, some loss is tolerated; for example, `returns` even though strictly, not all `values` can be represented exactly as `values`.

|

– Function.

|

Specifies what type should be used by `when` given values of types `and` . This function should not be called directly, but should have definitions added to it for new types as appropriate.

– Function.

|

Extract a named field from a `of` composite type. The syntax `calls` .

Example

|

- Function.

|

Assign to a named field in of composite type. The syntax calls .

- Function.

|

The byte offset of field of a type relative to the data start. For example, we could use it in the following manner to summarize information about a struct:

|

- Function.

|

Determine the declared type of a field (specified by name or index) in a composite DataType .

|

- Function.

|

Return iff value 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 is mutable.

|

– Function.

|

Return if is a "plain data" type, meaning it is immutable and contains no references to other values. Typical examples are numeric types such as , , and .

|

– Function.

|

Determine whether 's only subtypes are itself and . This means is a concrete type that can have instances.

|

– Function.

|

Compute a type that contains both and .

– Function.

|

Compute a type that contains the intersection of `T` and `S`. Usually this will be the smallest such type or one close to it.

- `Type`.

|

Create a "value type" out of `T`, which must be an `enum` value. The intent of this construct is to be able to dispatch on constants, e.g., `enum` allows you to dispatch directly (at compile-time) to an implementation `T`, without having to test the boolean value at runtime.

- `Macro`.

|

Create an `enum` subtype with name `T` and enum member values of `T` and `S` with optional assigned values of `T` and `S`, respectively. `enum` can be used just like other types and enum member values as regular values, such as

|

`enum`, which defaults to `T`, must be a primitive subtype of `T`. Member values can be converted between the enum type and `T`. `enum` and `T` perform these conversions automatically.

- `Function`.

|

Return a collection of all instances of the given type, if applicable. Mostly used for enumerated types (see `enum`).

|

45.5 Funciones Genéricas

- `Type`.

|

Abstract type of all functions.

|

– Function.

|

Determine whether the given generic function has a method matching the given of argument types with the upper bound of world age given by .

|

– Function.

|

Determine whether the given generic function has a method applicable to the given arguments.

Examples

|

– Function.

|

Invoke a method for the given generic function matching the specified types, on the specified arguments. The arguments must be compatible with the specified types. This 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).

– Function.

|

Calls `f`, 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 `f` is somewhat slower than calling `f` directly, and the type of the result cannot be inferred by the compiler.)

- Function.

|

Applies a function to the preceding argument. This allows for easy function chaining.

|

- Function.

|

Compose functions: i.e. `f ∘ g` means `f(g(x))`. The `∘` symbol can be entered in the Julia REPL (and most editors, appropriately configured) by typing `\circ`. Example:

|

45.6 Syntax

- Function.

|

Evaluate an expression in the given module and return the result. Every module (except those defined with `using`) has its own 1-argument definition of `eval`, which evaluates expressions in that module.

- Macro.

|

Evaluate an expression with values interpolated into it using `$(...)`. If two arguments are provided, the first is the module to evaluate in.

- Function.

|

Load the file using , evaluate all expressions, and return the value of the last one.

- Function.

|

Only valid in the context of an 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.

- Macro.

|

Eliminates array bounds checking within expressions.

In the example below the bound check of array A is skipped to improve performance.

|

Warning

Using may return incorrect results/crashes/corruption for out-of-bounds indices. The user is responsible for checking it manually.

- Macro.

|

Give a hint to the compiler that this function is worth inlining.

Small functions typically do not need the annotation, as the compiler does it automatically. By using on bigger functions, an extra nudge can be given to the compiler to inline it. This is shown in the following example:

|

- Macro.

|

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:

|

– Function.

|

Generates a symbol which will not conflict with other variable names.

– Macro.

|

Generates a gensym symbol for a variable. For example, `__gensym(x)` is transformed into `__gensym(x)`.

– Macro.

|

Tells the compiler to apply the polyhedral optimizer Polly to a function.

– Method.

|

Parse the expression string and return an expression (which could later be passed to eval for execution). `__parse_expr` is the index of the first character to start parsing. If `__parse_expr` is (default), 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 `__parse_expr`. If `__parse_expr` is (default), syntax errors other than incomplete expressions will raise an error. If `__parse_expr` is `__parse_expr`, will return an expression that will raise an error upon evaluation.

|

– Method.

|

Parse the expression string greedily, returning a single expression. An error is thrown if there are additional characters after the first expression. If `is (default)`, syntax errors will raise an error; otherwise, `will return an expression that will raise an error upon evaluation.`

|

45.7 Nullables

– Type.

|

Wrap value in an object of type , which indicates whether a value is present. yields a non-empty wrapper and yields an empty instance of a wrapper that might contain a value of type .

yields with stored in the result's field.

Examples

|

– Method.

|

Attempt to access the value of `o`. Returns the value if it is present; otherwise, returns `default` if provided, or throws a `NullPointerException` if not.

– **Function.**

|

Return whether or not `o` is null for `default`; return `o` for all other `o`.

Examples

|

– **Function.**

|

Return the value of `o` for `default`; return `o` for all other `o`.

This method does not check whether or not `o` is null before attempting to access the value of `o` for `default` (hence "unsafe").

|

|

45.8 Sistema

- Function.

|

Run a command object, constructed with backticks. Throws an error if anything goes wrong, including the process exiting with a non-zero status.

- Function.

|

Run a command object asynchronously, returning the resulting object.

- Constant.

|

Used in a stream redirect to discard all data written to it. Essentially equivalent to /dev/null on Unix or NUL on Windows. Usage:

|

- Function.

|

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.

- Function.

|

Determine whether a process is currently running.

- Function.

|

Determine whether a process has exited.

- Method.

|

Send a signal to a process. The default is to terminate the process.

- Function.

|

Set the process title. No-op on some operating systems.

- Function.

|

Get the process title. On some systems, will always return an empty string.

- Function.

|

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.

- Function.

|

Mark a command object so that running it will not throw an error if the result code is non-zero.

- Function.

|

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.

- Type.

|

Construct a new object, representing an external program and arguments, from , while changing the settings of the optional keyword arguments:

- : If (defaults to), then the will not throw an error if the return code is nonzero.
- : If (defaults to), then the will be run in a new process group, allowing it to outlive the process and not have Ctrl-C passed to it.

- : If (defaults to), then on Windows the 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 to a program 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. is useful for launching programs that parse their command line in nonstandard ways.) Has no effect on non-Windows systems.
- : If (defaults to), then on Windows no new console window is displayed when the is executed. This has no effect if a console is already open or on non-Windows systems.
- : Set environment variables to use when running the . is either a dictionary mapping strings to strings, an array of strings of the form , an array or tuple of pairs, or . In order to modify (rather than replace) the existing environment, create by and then set as desired.
- : Specify a working directory for the command (instead of the current directory).

For any keywords that are not specified, the current settings from are used. Normally, to create a object in the first place, one uses backticks, e.g.

```
|
```

– Function.

```
|
```

Set environment variables to use when running the given . is either a dictionary mapping strings to strings, an array of strings of the form , or zero or more pair arguments. In order to modify (rather than replace) the existing environment, create by and then setting as desired, or use .

The keyword argument can be used to specify a working directory for the command.

– Function.

```
|
```

Execute in an environment that is temporarily modified (not replaced as in) by zero or more arguments . is generally used via the syntax. A value of can be used to temporarily unset an environment variable (if it is set). When returns, the original environment has been restored.

– Method.

```
|
```

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 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 is equivalent to . This provides a more concise way to specify multi-stage pipelines.

Examples:

```
|
```

- Method.

```
|
```

Redirect I/O to or from the given . Keyword arguments specify which of the command's streams should be redirected. controls whether file output appends to the file. This is a more general version of the 2-argument function. is equivalent to when is a command, and to when is another kind of data source.

Examples:

```
|
```

- Function.

```
|
```

Get the local machine's host name.

- Function.

```
|
```

Get the IP address of the local machine.

- Function.

```
|
```

Get Julia's process ID.

- Method.

```
|
```

Get the system time in seconds since the epoch, with fairly high (typically, microsecond) resolution.

- Function.

```
|
```

Get the time in nanoseconds. The time corresponding to 0 is undefined, and wraps every 5.8 years.

- Function.

```
|
```

Set a timer to be read by the next call to `time` or `time-elapsed`. The macro call `time` can also be used to time evaluation.

– Function.

Print and return the time elapsed since the last `time`. The macro call `time` can also be used to time evaluation.

– Function.

Return, but do not print, the time elapsed since the last `time`. The macro calls `time` and `time-elapsed` also return evaluation time.

– Macro.

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 `time`, `time-elapsed`, and `time`.

|

– Macro.

|

This is a verbose version of the macro. It first prints the same information as , then any non-zero memory allocation counters, and then returns the value of the expression.

See also , , and .

|

– Macro.

|

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.

See also , , and .

|

|

– Macro.

|

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 `, , ,` and `.`

|

– Macro.

|

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 `macros`, which do not try to adjust for the effects of compilation.

See also `, , ,` and `.`

|

– Type.

|

A singleton of this type provides a hash table interface to environment variables.

– Constant.

|

Reference to the singleton `, providing a dictionary interface to system environment variables.`

- Function.

|

Predicate for testing if the OS provides a Unix-like interface. See documentation in [Handling Operating System Variation](#).

- Function.

|

Predicate for testing if the OS is a derivative of Apple Macintosh OS X or Darwin. See documentation in [Handling Operating System Variation](#).

- Function.

|

Predicate for testing if the OS is a derivative of Linux. See documentation in [Handling Operating System Variation](#).

- Function.

|

Predicate for testing if the OS is a derivative of BSD. See documentation in [Handling Operating System Variation](#).

- Function.

|

Predicate for testing if the OS is a derivative of Microsoft Windows NT. See documentation in [Handling Operating System Variation](#).

- Function.

|

Returns the version number for the Windows NT Kernel as a (major, minor) pair, or if this is not running on Windows.

- Macro.

|

Partially evaluates an expression at parse time.

For example, will evaluate and insert either or into the expression. This is useful in cases where a construct would be invalid on other platforms, such as a to a non-existent function. and are also valid syntax.

45.9 Errores

- Function.

|

Raise an with the given message.

- Function.

|

Throw an object as an exception.

- Function.

|

Throw an object without changing the current exception backtrace. The default argument is the current exception (if called within a block).

- Function.

|

Get a backtrace object for the current program point.

- Function.

|

Get the backtrace of the current exception, for use within blocks.

- Function.

|

Throw an if is . Also available as the macro .

- Macro.

|

Throw an if is . Preferred syntax for writing assertions. Message is optionally displayed upon assertion failure.

- Type.

|

The parameters to a function call do not match a valid signature. Argument is a descriptive error string.

- Type.

|

The asserted condition did not evaluate to . Optional argument is a descriptive error string.

- Type.

|

An indexing operation into an array, , tried to access an out-of-bounds element, .

- Type.

|

The objects called do not have matching dimensionality. Optional argument is a descriptive error string.

- Type.

|

Integer division was attempted with a denominator value of 0.

- Type.

|

The arguments to a function or constructor are outside the valid domain.

- Type.

|

No more data was available to read from a file or stream.

- Type.

|

Generic error type. The error message, in the field, may provide more specific details.

- Type.

|

Type conversion cannot be done exactly.

- Type.

|

The process was stopped by a terminal interrupt (CTRL+C).

- Type.

|

An indexing operation into an () or like object tried to access or delete a non-existent element.

- Type.

|

An error occurred while ing, ing, or a file. The error specifics should be available in the field.

- Type.

|

A method with the required type signature does not exist in the given generic function. Alternatively, there is no unique most-specific method.

- Type.

|

An attempted access to a with no defined value.

- Type.

|

An operation allocated too much memory for either the system or the garbage collector to handle properly.

- Type.

|

An operation tried to write to memory that is read-only.

- Type.

|

The result of an expression is too large for the specified type and will cause a wraparound.

- Type.

|

The expression passed to the `eval` function could not be interpreted as a valid Julia expression.

- Type.

|

After a client Julia process has exited, further attempts to reference the dead child will throw this exception.

- Type.

|

The function call grew beyond the size of the call stack. This usually happens when a call recurses infinitely.

- Type.

|

A system call failed with an error code (in the `errno` global variable).

- Type.

|

A type assertion failure, or calling an intrinsic function with an incorrect argument type.

- Type.

|

The item or field is not defined for the given object.

- Type.

|

A symbol in the current scope is not defined.

– Type.

|

An error occurred when running a module's function. The actual error thrown is available in the field.

– Function.

|

Returns an anonymous function that calls function . If an exception arises, is repeatedly called again, each time returns , after waiting the number of seconds specified in . should input 's current state and the .

Examples

|

– Type.

|

A iterator of length whose elements exponentially increase at a rate in the interval $*(1 \pm)$. The first element is and all elements are clamped to .

45.10 Eventos

– Method.

|

Create a timer to call the given function. The is passed one argument, the timer object itself. The callback will be invoked after the specified initial , and then repeating with the given interval. If is , the timer is only triggered once. Times are in seconds. A timer is stopped and has its resources freed by calling on it.

– Type.

|

Create a timer that wakes up tasks waiting for it (by calling `on` on the timer object) at a specified interval. Times are in seconds. Waiting tasks are woken with an error when the timer is closed (by `close`). Use `is_active` to check whether a timer is still active.

– Type.

|

Create a async condition that wakes up tasks waiting for it (by calling `on` on the object) when notified from C by a call to `notify`. Waiting tasks are woken with an error when the object is closed (by `close`). Use `is_active` to check whether it is still active.

– Method.

|

Create a async condition that calls the given `function`. The `function` is passed one argument, the async condition object itself.

45.11 Reflexión

– Function.

|

Get the name of a `module` as a `string`.

|

– Function.

|

Get a module's enclosing `module`. `module` is its own parent, as is `__main__` after `__init__`.

|

– Function.

|

Get the *dynamically* current , which is the code is currently being read from. In general, this is not the same as the module containing the call to this function.

– Function.

|

Get the fully-qualified name of a module as a tuple of symbols. For example,

|

– Function.

|

Get an array of the names exported by a , excluding deprecated names. If is true, then the list also includes non-exported names defined in the module, deprecated names, and compiler-generated names. If is true, then names explicitly imported from other modules are also included.

As a special case, all names defined in are considered "exported", since it is not idiomatic to explicitly export names from .

– Function.

|

Get the number of fields of a .

– Function.

|

Get an array of the fields of a .

|

– Function.

|

Get the name of field of a .

|

– Function.

|

Determine the module containing the definition of a .

– Function.

|

Get the name of a (potentially UnionAll-wrapped) (without its parent module) as a symbol.

– Function.

|

Determine whether a global is declared in a given . The default argument is .

– Function.

|

Get the name of a generic as a symbol, or .

– Method.

|

Determine the module containing the (first) definition of a generic function.

– Method.

|

Determine the module containing a given definition of a generic function.

– Method.

|

Returns a tuple giving the location of a generic definition.

– Method.

|

Returns a tuple giving the location of a definition.

– Macro.

|

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple giving the location for the method that would be called for those arguments. It calls out to the function.

45.12 Interioridades

– Function.

|

Perform garbage collection. This should not generally be used.

– Function.

|

Control whether garbage collection is enabled using a boolean argument (for enabled, 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.

– Function.

|

Takes the expression and returns an equivalent expression with all macros removed (expanded).

– Macro.

|

Return equivalent expression with all macros removed (expanded).

There is a subtle difference between and in that expansion takes place in different contexts. This is best seen in the following example:

|

With `with` the expression expands where `where` appears in the code (module in the example). With `with` the expression expands in the current module where the code was finally called (REPL in the example). Note that when calling `with` or directly from the REPL, both of these contexts coincide, hence there is no difference.

- Function.

|

Takes the expression `expr` and returns an equivalent expression in lowered form. See also `lower`.

- Function.

|

Returns an array of lowered ASTs for the methods matching the given generic function and type signature.

- Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls `call` on the resulting expression.

- Function.

|

Returns an array of lowered and type-inferred ASTs for the methods matching the given generic function and type signature. The keyword argument `controls` whether additional optimizations, such as inlining, are also applied.

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls `on` the resulting expression.

– Function.

|

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to `which` defaults to `.` 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 [for more information](#).

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls `on` the resulting expression.

– Function.

|

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to `which` defaults to `.`

All metadata and `dbg.*` calls are removed from the printed bitcode. Use `code_llvm_raw` for the full IR.

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls `on` the resulting expression.

– Function.

|

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to which defaults to . Switch assembly syntax using symbol parameter set to for AT&T syntax or for Intel syntax. Output is AT&T syntax by default.

– Macro.

|

Evaluates the arguments to the function or macro call, determines their types, and calls on the resulting expression.

– Function.

|

Compile the given function for the argument tuple (of types) , but do not execute it.

Chapter 46

Colecciones y Estructuras de Datos

46.1 Iteración

La iteración secuencial es implementada por los métodos `iter()` y `next()`. El bucle `for` general:

```
|
```

es traducido a:

```
|
```

El objeto `iterable` puede ser cualquier cosa, y debería ser elegido apropiadamente para cada tipo iterable. Ver la [sección de manual sobre la interfaz de iteración](#) para ms detalles sobre definir un tipo iterable personalizado.

– Function.

```
|
```

Get initial iteration state for an iterable object.

Examples

```
|
```

– Function.

|

Test whether we are done iterating.

Examples

|

– Function.

|

For a given iterable object and iteration state, return the current item and the next iteration state.

Examples

|

– Function.

|

Given the type of an iterator, returns one of the following values:

- if the length (number of elements) cannot be determined in advance.
- if there is a fixed, finite length.
- if there is a known length plus a notion of multidimensional shape (as for an array). In this case the function is valid for the iterator.
- if the iterator yields values forever.

The default value (for iterators that do not define this function) is `None`. This means that most iterators are assumed to implement `__len__`.

This trait is generally used to select between algorithms that pre-allocate space for their result, and algorithms that resize their result incrementally.

|

– Function.

|

Given the type of an iterator, returns one of the following values:

- if the type of elements yielded by the iterator is not known in advance.
- if the element type is known, and would return a meaningful value.

is the default, since iterators are assumed to implement .

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.

|

Completamente implementada por:

-
-
-
-
-
-
-
-
-
-
-

46.2 Colecciones generales

– Function.

|

Determine whether a collection is empty (has no elements).

Examples

|

– Function.

|

Remove all elements from a .

|

– Method.

|

For ordered, indexable collections, returns the maximum index for which is valid. For unordered collections, returns the number of elements.

Examples

|

– Function.

|

Returns the last index of the collection.

Example

|

Completamente implementado por:

-
-
-
-
-
-
-
-
-
-

46.3 Colecciones Iterables

– Function.

|

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 s check whether the item to one of the elements. s look for pairs, and the key is compared using . To test for the presence of a key in a dictionary, use or .

|

– Function.

|

Determine the type of the elements generated by iterating a collection of the given . For associative collection types, this will be a . The definition 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.

|

– Function.

|

Returns a vector containing the highest index in `vec` for each value in `vec2` that is a member of `vec`. The output vector contains 0 wherever `vec2` is not a member of `vec`.

Examples

|

– Function.

|

Returns the indices of elements in `collection1` that appear in `collection2`.

Examples

|

|

– Function.

|

Returns an array containing one value from `iterable` for each unique value, as determined by `key`.

|

|

Returns an array containing one value from `iterable` for each unique value produced by `key` applied to elements of `iterable`.

|

|

Returns an array containing only the unique elements of the iterable `iterable`, in the order that the first of each set of equivalent elements originally appears. If `key` is specified, returns unique regions of the array `iterable` along `key`.

|

- Function.

Return `True` if all values from `arr1` are distinct when compared with `arr2`.

- Method.

Reduce the given collection with the given binary operator . must be a neutral element for that will be returned for empty collections. It is unspecified whether is used for non-empty collections.

Reductions for certain commonly-used operators have special implementations which should be used instead: , ,

， ， ， 。

The associativity of the reduction is implementation dependent. This means that you can't use non-associative operations like `<` because it is undefined whether `<` should be evaluated as `<` or `<`. Use `<` or `<` instead for guaranteed left or right associativity.

Some operations accumulate error, and parallelism will also 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

– Method.

|

Like `reduce`. This cannot be used with empty collections, except for some special cases (e.g. when `initial` is one of `, , , , ,`) when Julia can determine the neutral element of `op`.

|

– Method.

|

Like `reduce`, but with guaranteed left associativity. `initial` will be used exactly once.

|

– Method.

|

Like `reduce`, but using the first element of `collection` as `initial`. In general, this cannot be used with empty collections (see `reduce`).

|

– Method.

|

Like `reduce`, but with guaranteed right associativity. `initial` will be used exactly once.

|

– Method.

|

Like `reduce`, but using the last element of `collection` as `initial`. In general, this cannot be used with empty collections (see `reduce`).

|

– Method.

|

Returns the largest element in a collection.

|

– Method.

|

Compute the maximum value of an array over the given dimensions. See also the `max` function to take the maximum of two or more arguments, which can be applied elementwise to arrays via `map`.

|

– Function.

|

Compute the maximum value of `arr` over the singleton dimensions of `dims`, and write results to `out`.

Examples

|

– Method.

|

Returns the smallest element in a collection.

|

– Method.

|

Compute the minimum value of an array over the given dimensions. See also the `min` function to take the minimum of two or more arguments, which can be applied elementwise to arrays via `min`.

Examples

|

– Function.

|

Compute the minimum value of `arr` over the singleton dimensions of `arr`, and write results to `out`.

Examples

|

|

– Method.

|

Compute both the minimum and maximum element in a single pass, and return them as a 2-tuple.

|

– Method.

|

Compute the minimum and maximum elements of an array over the given dimensions.

Example

|

– Function.

|

Returns the index of the maximum element in a collection. If there are multiple maximal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

|

– Function.

|

Returns the index of the minimum element in a collection. If there are multiple minimal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

|

– Method.

|

Returns the maximum element of the collection and its index. If there are multiple maximal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

|

– Method.

|

For an array input, returns the value and index of the maximum over the given region.

Examples

|

– Method.

|

Returns the minimum element of the collection and its index. If there are multiple minimal elements, then the first one will be returned. values are ignored, unless all elements are .

The collection must not be empty.

Examples

|

– Method.

|

For an array input, returns the value and index of the minimum over the given region.

Examples

|

– Function.

|

Find the maximum of `array` and the corresponding linear index along singleton dimensions of `array` and `dim`, and store the results in `max` and `idx`.

– Function.

|

Find the minimum of `array` and the corresponding linear index along singleton dimensions of `array` and `dim`, and store the results in `min` and `idx`.

– Function.

|

Sum the results of calling function `func` on each element of `array`.

|

|

Returns the sum of all elements in a collection.

|

|

Sum elements of an array over the given dimensions.

Examples

|

– Function.

|

Sum elements of over the singleton dimensions of , and write results to .

Examples

|

– Function.

|

Returns the product of applied to each element of .

|

|

Returns the product of all elements of a collection.

|

|

Multiply elements of an array over the given dimensions.

Examples

|

|

– Function.

|

Multiply elements of over the singleton dimensions of , and write results to .

Examples

|

– Method.

|

Test whether any elements of a boolean collection are , returning as soon as the first value in is encountered (short-circuiting).

|

– Method.

|

Test whether any values along the given dimensions of an array are .

Examples

|

– Function.

|

Test whether any values in along the singleton dimensions of are , and write results to .

Examples

|

– Method.

|

Test whether all elements of a boolean collection are , returning as soon as the first value in is encountered (short-circuiting).

|

– Method.

|

Test whether all values along the given dimensions of an array are .

Examples

|

– Function.

|

Test whether all values in along the singleton dimensions of are , and write results to .

Examples

|

|

– Function.

|

Count the number of elements in `collection` for which predicate `predicate` returns `true`. If `predicate` is omitted, counts the number of `true` elements in `collection` (which should be a collection of boolean values).

|

– Method.

|

Determine whether predicate `predicate` returns `true` for any elements of `collection`, returning `true` as soon as the first item in `collection` for which `predicate` returns `true` is encountered (short-circuiting).

|

– Method.

|

Determine whether predicate `predicate` returns `true` for all elements of `collection`, returning `true` as soon as the first item in `collection` for which `predicate` returns `false` is encountered (short-circuiting).

|

– Function.

|

Call function on each element of iterable . For multiple iterable arguments, is called elementwise. should be used instead of when the results of are not needed, for example in .

Example

|

– Function.

|

Transform collection by applying to each element. For multiple collection arguments, apply elementwise.

Examples

|

|

Return applied to the value of if it has one, as a . If is null, then return a null value of type . is guaranteed to be either or a concrete type. Whichever of these is chosen is an implementation detail, but typically the choice that maximizes performance would be used. If has a value, then the return type is guaranteed to be of type .

– Function.

|

Like , but stores the result in rather than a new collection. must be at least as large as the first collection.

Example

|

– Method.

|

Apply function `f` to each element in `xs`, and then reduce the result using the binary function `g`. `z` must be a neutral element for `g` that will be returned for empty collections. It is unspecified whether `z` is used for non-empty collections.

`foldLeft` is functionally equivalent to calling `fold`, but will in general execute faster since no intermediate collection needs to be created. See documentation for `fold` and `foldLeft`.

|

The associativity of the reduction is implementation-dependent. Additionally, some implementations may reuse the return value of `f` for elements that appear multiple times in `xs`. Use `foldLeft` or `foldRight` instead for guaranteed left or right associativity and invocation of `f` for every value.

– Method.

|

Like `foldLeft`. In general, this cannot be used with empty collections (see [see](#)).

– Method.

|

Like `foldLeft`, but with guaranteed left associativity, as in `foldLeft`. `f` will be used exactly once.

– Method.

|

Like `foldLeft`, but using the first element of `xs` as `z`. In general, this cannot be used with empty collections (see [see](#)).

– Method.

|

Like `pop()`, but with guaranteed right associativity, as in `pop()`. `pop()` will be used exactly once.

– Method.

|

Like `pop()`, but using the first element of `self` as `pop()`. In general, this cannot be used with empty collections (see `pop()`).

– Function.

|

Get the first element of an iterable collection. Returns the start point of a `range()` even if it is empty.

|

– Function.

|

Get the last element of an ordered collection, if it can be computed in $O(1)$ time. This is accomplished by calling `pop()` to get the last index. Returns the end point of a `range()` even if it is empty.

|

– Function.

|

Get the step size of a `range()` object.

|

– Method.

|

Return an of all items in a collection or iterator. For associative collections, returns . If the argument is array-like or is an iterator with the trait, the result will have the same shape and number of dimensions as the argument.

Example

|

– Method.

|

Return an with the given element type of all items in a collection or iterable. The result has the same shape and number of dimensions as .

|

– Method.

|

Determine whether every element of is also in , using .

Examples

|

– Function.

|

Return a copy of , removing elements for which is . For associative collections, the function is passed two arguments (key and value).

Examples

|

|

Return null if either is null or is false, and otherwise.

– Function.

|

Update , removing elements for which is . For associative collections, the function is passed two arguments (key and value).

Example

|

46.4 Colecciones Indexables

– Method.

|

Retrieve the value(s) stored at the given key or index within a collection. The syntax is converted by the compiler to .

Example

|

– Method.

|

Store the given value at the given key or index within a collection. The syntax is converted by the compiler to .

Completamente implementado por:

-
-
-
-
-
-
-
-

Parcialmente implementado por:

-
-
-

46.5 Colecciones asociativas

es la colección asociativa estándar. Su implementación usa `hash` como función de hashing para la clave, e `eq` para determinar la igualdad. Si redefine estas dos funciones en un tipo personalizado sobrescriban como se almacenan dichos tipos en una tabla hash.

es una tabla hash especial donde las claves son siempre identidades de objeto.

es una implementación de tabla hash donde las claves son referencias débiles a los objetos y, por lo tanto, permiten recolección de basura recogida incluso cuando se referencian en una tabla hash.

s se pueden crear pasando pares de objetos contruidos con `cons` a un constructor `Dict`. Esta llamada intentará inferir información de tipo de las claves y valores (es decir, este ejemplo crea un `Dict`). Para especificar explícitamente los tipos, use la sintaxis `Dict::new`. Por ejemplo, `Dict::new(1, 2)`.

Las colecciones asociativas pueden también ser creadas con generadores. Por ejemplo, `Dict::new(1..10)`.

Dado un diccionario `dict`, la sintaxis `dict[key]` devuelve el valor de la clave (si existe) o arroja un error, y `dict[key] = value` almacena el par de clave-valor en `dict` (reemplazando cualquier valor existente por la clave). Múltiples argumentos para `Dict::new` se convierten a tuplas; por ejemplo, la sintaxis `Dict::new(1, 2, 3)` es equivalente a `Dict::new(1, 2, 3)`, es decir, se refiere al valor introducido por la tupla `(1, 2, 3)`.

– Type.

|

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 generated by the argument.

|

Alternatively, a sequence of pair arguments may be passed.

|

– Type.

|

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 `key` is always `Object`.

See `Dict::new` for further help.

– Type.

|

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 `WeakKeyDictionary` for further help.

– **Function.**

|

Determine whether a collection has a mapping for a given key.

|

– **Method.**

|

Return the value stored for the given key, or the given default value if no mapping for the key is present.

Examples

|

– **Function.**

|

Return the value stored for the given key, or if no mapping for the key is present, return `default`. Use `default` to also store the default value in the dictionary.

This is intended to be called using `block` syntax

|

– Method.

|

Return the value stored for the given key, or if no mapping for the key is present, store , and return .

Examples

|

– Method.

|

Return the value stored for the given key, or if no mapping for the key is present, store , and return .

This is intended to be called using block syntax:

|

– Function.

|

Return the key matching argument if one exists in , otherwise return .

|

- Function.

|

Delete the mapping for the given key in a collection, and return the collection.

Example

|

- Method.

|

Delete and return the mapping for if it exists in , otherwise return , or throw an error if is not specified.

Examples

|

- Function.

|

Return an iterator over all keys in a collection. returns an array of keys. Since the keys are stored internally in a hash table, the order in which they are returned may vary. But and both iterate and return the elements in the same order.

|

|

– Function.

|

Return an iterator over all values in a collection. 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 and both iterate and return the elements in the same order.

|

– Function.

|

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. 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.

|

Construct a merged collection from the given collections. If necessary, the types of the resulting collection will be promoted to accommodate the types of the merged collections. Values with the same key will be combined using the combiner function.

– Function.

Merge changes into current head

Internal implementation of merge. Returns `Boolean` if merge was successful, otherwise

Perform a git merge on the repository, merging commits with diverging history into the current branch. Returns `Boolean` if the merge succeeded, if not.

The keyword arguments are:

- `commit` : Merge the named commit(s) in.
- `branch` : Merge the branch and all its commits since it diverged from the current branch.
- `ff-only` : If `ff-only` 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`.
- `options` : specifies options for the merge, such as merge strategy in case of conflicts.
- `checkoutOptions` : specifies options for the checkout step.

Equivalent to `git merge`.

Note

If you specify a `branch`, this must be done in reference format, since the string will be turned into a `Branch`. For example, if you wanted to merge `branch`, you would call `merge(branch)`.

|

Update collection with pairs from the other collections. See also .

|

|

Update collection with pairs from the other collections. Values with the same key will be combined using the combiner function.

|

– Function.

|

Suggest that collection reserve capacity for at least elements. This can improve performance.

– Function.

|

Get the key type of an associative collection type. Behaves similarly to .

|

– Function.

|

Get the value type of an associative collection type. Behaves similarly to .

|

Completamente implementado por:

-
-
-

Parcialmente implementado por:

-
-
-
-
-

46.6 Colecciones tipo Conjunto

– Type.

|

Construct a of the values generated by the given iterable object, or an empty set. Should be used instead of for sparse integer sets, or for sets of arbitrary objects.

– Type.

|

Construct a sorted set of positive s generated by the given iterable object, or an empty set. Implemented as a bit string, and therefore designed for dense integer sets. Only s greater than 0 can be stored. If the set will be sparse (for example holding a few very large integers), use instead.

– Function.

|

Construct the union of two or more sets. Maintains order with arrays.

Examples

|

– Function.

|

Union each element of into set in-place.

– Function.

|

Construct the intersection of two or more sets. Maintains order and multiplicity of the first argument for arrays and ranges.

– Function.

|

Construct the set of elements in but not . Maintains order with arrays. Note that both arguments must be collections, and both will be iterated over. In particular, where is a potential member of , will not work in general.

Example

|

– Function.

|

Remove each element of from set in-place.

– Function.

|

Construct the symmetric difference of elements in the passed in sets or arrays. Maintains order with arrays.

Example

|

– Method.

|

The set is destructively modified to toggle the inclusion of integer .

– Method.

|

For each element in , destructively toggle its inclusion in set .

– Method.

|

For each element in , destructively toggle its inclusion in set .

– Function.

|

Intersects sets `set1` and `set2` and overwrites the set `set1` with the result. If needed, `set1` will be expanded to the size of `set2`.

– Function.

```
|
```

Return `True` if `set1` is a subset of or equal to `set2`.

Completamente implementado por:

-
-

Parcialmente implementado por:

-

46.7 Acciones relacionadas con colas

– Function.

```
|
```

Insert one or more `elements` at the end of `queue`.

Example

```
|
```

Use `queue.extend()` to add all the elements of another collection to `queue`. The result of the preceding example is equivalent to `queue.extend([1, 2, 3])`.

– Method.

```
|
```

Remove the last item in `queue` and return it.

Examples

|

– Function.

|

Insert one or more at the beginning of .

Example

|

– Function.

|

Remove the first from .

Example

|

|

– Function.

|

Insert an into at the given . is the index of in the resulting .

Example

|

– Function.

|

Remove the item at the given and return the modified . Subsequent items are shifted to fill the resulting gap.

Example

|

|

Remove the items at the indices given by , and return the modified . Subsequent items are shifted to fill the resulting gap.

can be either an iterator or a collection of sorted and unique integer indices, or a boolean vector of the same length as with indicating entries to delete.

Examples

– Function.

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

|

To insert `before` an index `without` removing any items, use `.`

|

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 `before` an index `without` removing any items, use `.`

Example

|

– Function.

|

Resize `to` contain `elements`. If `is` smaller than the current collection length, the first `elements` will be retained. If `is` larger, the new elements are not guaranteed to be initialized.

Examples

|

|

– Function.

|

Add the elements of to the end of .

Examples

|

Use to add individual items to which are not already themselves in another collection. The result is of the preceding example is equivalent to .

– Function.

|

Insert the elements of to the beginning of .

Example

|

Completamente implementado por:

- (a.k.a. 1-dimensional)
- (a.k.a. 1-dimensional)

Chapter 47

Mathematics

47.1 Operadores Matemáticos

- Method.

|

Unary minus operator.

- Function.

|

Addition operator. calls this function with all arguments, i.e. .

- Method.

|

Subtraction operator.

- Method.

|

Multiplication operator. calls this function with all arguments, i.e. .

- Function.

|

Right division operator: multiplication of by the inverse of on the right. Gives floating-point results for integer arguments.

- Method.

|

Left division operator: multiplication of `x` by the inverse of `A` on the left. Gives floating-point results for integer arguments.

|

– Method.

|

Exponentiation operator. If `A` is a matrix, computes matrix exponentiation.

If `x` is an integer literal (e.g. `2` in `x^2` or `2.0` in `x^2.0`), the Julia code `x^x` is transformed by the compiler to `expm2(x)`, to enable compile-time specialization on the value of the exponent. (As a default fallback we have `expm2(x)`, where usually `expm2` unless `expm2` has been defined in the calling namespace.)

|

– Function.

|

Computes `expm2(x)` without rounding the intermediate result `x`. On some systems this is significantly more expensive than `expm2(x)`. `expm2` is used to improve accuracy in certain algorithms. See `expm2`.

– Function.

|

Combined multiply-add, computes $a * b + c$ in an efficient manner. This may on some systems be equivalent to $a * b + c$, or to $a * (b + c/b)$. is used to improve performance. See [.1](#).

Example

|

– Function.

|

The quotient from Euclidean division. Computes a / b , truncated to an integer.

|

– Function.

|

Largest integer less than or equal to x .

|

– Function.

|

Smallest integer larger than or equal to x .

|

– Function.

|

The reduction of x modulo y , or equivalently, the remainder of x after floored division by y , i.e.

|

if computed without intermediate rounding.

The result will have the same sign as y , and magnitude less than $|y|$ (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 $-y$.

|

|

Find x such that $x \equiv a \pmod{n}$, where n is the number of integers representable in T , and a is an integer in T . If T can represent any integer (e.g. \mathbb{Z}), then this operation corresponds to a conversion to \mathbb{Z} .

|

– Function.

|

Remainder from Euclidean division, returning a value of the same sign as y , and smaller in magnitude than $|y|$. This value is always exact.

|

|

– Function.

|

Compute the remainder of x after integer division by y , with the quotient rounded according to the rounding mode $round$. In other words, the quantity

|

without any intermediate rounding. This internally uses a high precision approximation of 2π , and so will give a more accurate result than

- if $x < 0$, then the result is in the interval $[-y, 0]$. This will generally be the most accurate result.
- if $x \geq 0$, then the result is in the interval $[0, y]$ if $round$ is positive, or $[-y, 0]$ otherwise.
- if $y < 0$, then the result is in the interval $[0, -y]$.
- if $y \geq 0$, then the result is in the interval $[-y, 0]$.

Example

|

– Function.

|

Modulus after division by y , returning in the range $[0, y)$.

This function computes a floating point representation of the modulus after division by numerically exact y , and is therefore not exactly the same as $\text{fmod}(x, y)$, which would compute the modulus of x relative to division by the floating-point number y .

Example

|

– Function.

|

The quotient and remainder from Euclidean division. Equivalent to $\text{divmod}(x, y)$ or $\text{divmod}(x, y)$.

|

– Function.

|

The floored quotient and modulus after division. Equivalent to .

– Function.

|

Flooring division, returning a value consistent with

See also: .

|

– Function.

|

Modulus after flooring division, returning a value such that in the range $(0, y]$ for positive and in the range $[y, 0)$ for negative .

|

– Function.

|

Return .

See also: , .

– Function.

|

Divide two integers or rational numbers, giving a result.

|

– Function.

|

Approximate floating point number as a number with components of the given integer type. The result will differ from by no more than . If is not provided, it defaults to .

|

– Function.

|

Numerator of the rational representation of .

|

– Function.

|

Denominator of the rational representation of .

– Function.

Left bit shift operator, `<<`. For `x << n`, the result is `x` shifted left by `n` bits, filling with `s`. This is equivalent to `x * 2^n`. For `x << 1`, this is equivalent to `x * 2`.

See also `>>`, `>>=`.

Left bit shift operator, `<<`. For `x << n`, the result is `x` with elements shifted `n` positions backwards, filling with `0` values. If `n` is negative, elements are shifted forwards. Equivalent to `x * 2^n`.

Examples

– Function.

|

Right bit shift operator, \gg . For x , the result is x shifted right by n bits, where n is the number of bits, filling with 0s if $n > 0$, preserving the sign of x if $n < 0$. This is equivalent to $x \gg n$. For x , this is equivalent to $x \gg n$.

|

See also [§ 47.1.1](#), [§ 47.1.2](#).

|

Right bit shift operator, \gg . For x , the result is x with elements shifted n positions forward, filling with 0 values. If $n < 0$, elements are shifted backwards. Equivalent to $x \gg n$.

Example

|

|

– Function.

|

Unsigned right bit shift operator, `>>`. For `x >> y`, the result is `x` shifted right by `y` bits, where `y` is filled with `s`. For `x >> y`, this is equivalent to `x / 2y`.

For integer types, this is equivalent to `x / 2y`. For integer types, this is equivalent to `x / 2y`.

|

`s` are treated as if having infinite size, so no filling is required and this is equivalent to `x / 2y`.

See also `>>`, `>>`.

|

Unsigned right bitshift operator, `>>`. Equivalent to `>>`. See `>>` for details and examples.

– Function.

|

Called by `>>` syntax for constructing ranges.

|

|

Range operator. `>>` constructs a range from `>>` to `>>` with a step size of 1, and `>>` is similar but uses a step size of `>>`. These syntaxes call the function `>>`. The colon is also used in indexing to select whole dimensions.

– Function.

|

Construct a range by length, given a starting value and optional step (defaults to 1).

– Type.

|

Define an `IntRange` that behaves like `Range`, with the added distinction that the lower limit is guaranteed (by the type system) to be 1.

– Type.

|

A range `where` produces values of type `T`, parametrized by a reference value, `a`, and the `length`. By default `a` is the starting value, but alternatively you can supply it as the value of `from` for some other index. In conjunction with `where` this can be used to implement ranges that are free of roundoff error.

– Function.

|

Generic equality operator, giving a single `Boolean` 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 `forall` by calling `forall` 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.

– Function.

|

Not-equals comparison operator. Always gives the opposite answer as `==`. New types should generally not implement this, and rely on the fallback definition `!=` instead.

|

– Function.

|

Equivalent to .

|

– Function.

|

Less-than comparison operator. New numeric types should implement 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 .

|

– Function.

|

Less-than-or-equals comparison operator.

|

– Function.

|

Greater-than comparison operator. Generally, new types should implement `>` instead of this function, and rely on the fallback definition `>=`.

|

– Function.

|

Greater-than-or-equals comparison operator.

|

– Function.

|

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 `<`. For floating-point numbers, uses `<=` but throws an error for unordered arguments.

|

– Function.

|

Bitwise not.

Examples

|

– Function.

|

Bitwise and.

Examples

|

– Function.

|

Bitwise or.

Examples

|

– Function.

|

Bitwise exclusive or of `x` and `y`. The infix operation `x ⊕ y` is a synonym for `xor(x, y)`, and `x ⊕ y` can be typed by tab-completing `x ⊕` or `y ⊕` in the Julia REPL.

|

– Function.

|

Boolean not.

|

|

Predicate function negation: when the argument of `not` is a function, it returns a function which computes the boolean negation of `f`. Example:

|

– Keyword.

|

Short-circuiting boolean AND.

– Keyword.

|

Short-circuiting boolean OR.

47.2 Funciones Matemáticas

- Function.

|

Inexact equality comparison: `if`. The default `is` zero and the default `depends` on the types of `and`. The keyword argument `determines` whether or not NaN values are considered equal (defaults to `false`).

For real or complex floating-point values, `defaults` to `.` This corresponds to requiring equality of about half of the significant digits. For other types, `defaults` to zero.

`and` may also be arrays of numbers, in which case `defaults` to `but` may be changed by passing a `keyword` argument. (For numbers, `is` the same thing as `.`) When `and` are arrays, if `is` not finite (i.e. `or`), the comparison falls back to checking whether all elements of `and` are approximately equal component-wise.

The binary operator `is` is equivalent to `with` the default arguments, and `is` is equivalent to `.`

|

- Function.

|

Compute sine of `,` where `is` in radians.

- Function.

|

Compute cosine of `,` where `is` in radians.

- Function.

|

Compute tangent of `,` where `is` in radians.

- Function.

|

Compute sine of `,` where `is` in degrees.

– Function.

|

Compute cosine of , where is in degrees.

– Function.

|

Compute tangent of , where is in degrees.

– Function.

|

Compute $\sin(\pi x)$ more accurately than , especially for large .

– Function.

|

Compute $\cos(\pi x)$ more accurately than , especially for large .

– Function.

|

Compute hyperbolic sine of .

– Function.

|

Compute hyperbolic cosine of .

– Function.

|

Compute hyperbolic tangent of .

– Function.

|

Compute the inverse sine of , where the output is in radians.

– Function.

|

Compute the inverse cosine of , where the output is in radians

– Function.

|

Compute the inverse tangent of , where the output is in radians.

– Function.

|

Compute the inverse tangent of , using the signs of both and to determine the quadrant of the return value.

– Function.

|

Compute the inverse sine of , where the output is in degrees.

– Function.

|

Compute the inverse cosine of , where the output is in degrees.

– Function.

|

Compute the inverse tangent of , where the output is in degrees.

– Function.

|

Compute the secant of , where is in radians.

– Function.

|

Compute the cosecant of , where is in radians.

– Function.

|

Compute the cotangent of , where is in radians.

– Function.

|

Compute the secant of , where is in degrees.

– Function.

|

Compute the cosecant of , where is in degrees.

– Function.

|

Compute the cotangent of , where is in degrees.

– Function.

|

Compute the inverse secant of , where the output is in radians.

– Function.

|

Compute the inverse cosecant of , where the output is in radians.

– Function.

|

Compute the inverse cotangent of , where the output is in radians.

– Function.

|

Compute the inverse secant of , where the output is in degrees.

– Function.

|

Compute the inverse cosecant of , where the output is in degrees.

– Function.

|

Compute the inverse cotangent of , where the output is in degrees.

– Function.

|

Compute the hyperbolic secant of

– Function.

|

Compute the hyperbolic cosecant of .

– Function.

|

Compute the hyperbolic cotangent of .

– Function.

|

Compute the inverse hyperbolic sine of .

– Function.

|

Compute the inverse hyperbolic cosine of .

– Function.

|

Compute the inverse hyperbolic tangent of .

– Function.

|

Compute the inverse hyperbolic secant of .

– Function.

|

Compute the inverse hyperbolic cosecant of .

– Function.

|

Compute the inverse hyperbolic cotangent of .

– Function.

|

Compute $\sin(\pi x)/(\pi x)$ if $x \neq 0$, and 1 if $x = 0$.

– Function.

|

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 .

– Function.

|

Convert from degrees to radians.

|

– Function.

|

Convert from radians to degrees.

|

– Function.

|

Compute the hypotenuse $\sqrt{x^2 + y^2}$ avoiding overflow and underflow.

Examples

|

|

Compute the hypotenuse $\sqrt{\sum x_i^2}$ avoiding overflow and underflow.

– Method.

|

Compute the natural logarithm of . Throws for negative arguments. Use complex negative arguments to obtain complex results.

There is an experimental variant in the module, which is typically faster and more accurate.

– Method.

|

Compute the base logarithm of . Throws for negative arguments.

|

Note

If is a power of 2 or 10, or should be used, as these will typically be faster and more accurate. For example,

```

|

```

– Function.

```

|

```

Compute the logarithm of to base 2. Throws for negative arguments.

Example

```

|

```

– Function.

```

|

```

Compute the logarithm of to base 10. Throws for negative arguments.

Example

```

|

```

– Function.

```

|

```

Accurate natural logarithm of . Throws for arguments less than -1.

There is an experimental variant in the module, which is typically faster and more accurate.

Examples

```

|

```

– Function.

|

Return such that has a magnitude in the interval $[1/2, 1)$ or 0, and is equal to $x \times 2^{exp}$.

– Function.

|

Compute the natural base exponential of , in other words e^x .

– Function.

|

Compute 2^x .

|

– Function.

|

Compute 10^x .

Examples

|

– Function.

|

Compute $x \times 2^n$.

Example

|

– Function.

|

Return a tuple (fpart,i part) of the fractional and integral parts of a number. Both parts have the same sign as the argument.

Example

|

– Function.

|

Accurately compute $e^x - 1$.

– Method.

|

Rounds to an integer value according to the provided , returning a value of the same type as . When not specifying a rounding mode the global mode will be used (see), which by default is round to the nearest integer (mode), with ties (fractional values of 0.5) being rounded to the nearest even integer.

|

The optional argument will change how the number gets rounded.

converts the result to type , throwing an if the value is not representable.

rounds to the specified number of digits after the decimal place (or before if negative). rounds using a base other than 10.

|

Note

Rounding to specified digits in bases other than 2 can be inexact when operating on binary floating point numbers. For example, the value represented by is actually *less* than 1.15, yet will be rounded to 1.2.

– Type.

|

A type used for controlling the rounding mode of floating point operations (via / functions), or as optional arguments for rounding to the nearest integer (via the function).

Currently supported rounding modes are:

- (default)
-
-
-
- (only)
-
-

– Constant.

|

The default rounding mode. Rounds to the nearest integer, with ties (fractional values of 0.5) being rounded to the nearest even integer.

– Constant.

|

Rounds to nearest integer, with ties rounded away from zero (C/C++ behaviour).

– Constant.

|

Rounds to nearest integer, with ties rounded toward positive infinity (Java/JavaScript behaviour).

- Constant.

|

using this rounding mode is an alias for .

- Constant.

|

using this rounding mode is an alias for .

- Constant.

|

using this rounding mode is an alias for .

- Method.

|

Returns the nearest integral value of the same type as the complex-valued `to` , breaking ties using the specified `s`. The first `s` is used for rounding the real components while the second is used for rounding the imaginary components.

- Function.

|

returns the nearest integral value of the same type as `that` is greater than or equal to .

converts the result to type `type` , throwing an `if` the value is not representable.

and `work` as for .

- Function.

|

returns the nearest integral value of the same type as `that` is less than or equal to .

converts the result to type `type` , throwing an `if` the value is not representable.

and `work` as for .

- Function.

|

returns the nearest integral value of the same type as whose absolute value is less than or equal to .

converts the result to type , throwing an if the value is not representable.

and work as for .

– Function.

|

returns the nearest integral value of type whose absolute value is less than or equal to . If the value is not representable by , an arbitrary value will be returned.

– Function.

|

Rounds (in the sense of) so that there are significant digits, under a base representation, default 10. E.g., is , and is .

– Function.

|

Return the minimum of the arguments. See also the function to take the minimum element from a collection.

|

– Function.

|

Return the maximum of the arguments. See also the function to take the maximum element from a collection.

|

– Function.

|

Return . See also: that returns .

|

– Function.

|

Return if . If , return . If , return . Arguments are promoted to a common type.

|

– Function.

|

Restrict values in to the specified range, in-place. See also .

– Function.

|

The absolute value of .

When is applied to signed integers, overflow may occur, resulting in the return of a negative value. This overflow occurs only when is applied to the minimum representable value of a signed integer. That is, when , , not as might be expected.

|

– Function.

|

Calculates , checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g.) cannot represent , thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable. For example, standard two's complement signed integers (e.g.) cannot represent , thus leading to an overflow.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates \sqrt{x} , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates \sqrt{x} , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates \sqrt{x} , checking for overflow errors where applicable.

The overflow protection may impose a perceptible performance penalty.

– Function.

|

Calculates \sqrt{x} , with the flag indicating whether overflow has occurred.

– Function.

|

Calculates \sqrt{x} , with the flag indicating whether overflow has occurred.

– Function.

|

Calculates \sqrt{x} , with the flag indicating whether overflow has occurred.

– Function.

|

Squared absolute value of x .

|

– Function.

|

Return which has the magnitude of and the same sign as .

Examples

|

– Function.

|

Return zero if and $x/|x|$ otherwise (i.e., ± 1 for real).

– Function.

|

Returns if the value of the sign of is negative, otherwise .

Examples

|

– Function.

|

Return with its sign flipped if is negative. For example .

|

– Function.

|

Return \sqrt{x} . Throws `ValueError` for negative arguments. Use complex negative arguments instead. The prefix operator `abs` is equivalent to `sqrt(x)`.

– Function.

|

Integer square root: the largest integer `y` such that `y*y ≤ x`.

|

– Function.

|

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 `abs` is equivalent to `sqrt(x)`.

|

– Method.

|

Return the real part of the complex number `z`.

|

– Function.

|

Return the imaginary part of the complex number `z`.

|

– Function.

|

Return both the real and imaginary parts of the complex number .

|

– Function.

|

Compute the complex conjugate of a complex number .

|

|

Returns a lazy view of the input, where each element is conjugated.

Example

|

– Function.

|

Compute the phase angle in radians of a complex number .

– Function.

|

Return $\exp(iz)$.

– Function.

|

Number of ways to choose out of items.

Example

|

– Function.

|

Factorial of . If x is an integer, the factorial is computed as an integer (promoted to at least 64 bits). Note that this may overflow if x is not small, but you can use `gamma` to compute the result exactly in arbitrary precision. If x is not an integer, `gamma` is equivalent to $\Gamma(x+1)$.

|

– Function.

|

Greatest common (positive) divisor (or zero if a and b are both zero).

Examples

|

– Function.

|

Least common (non-negative) multiple.

Examples

|

– Function.

|

Computes the greatest common (positive) divisor of x and y and their Bézout coefficients, i.e. the integer coefficients u and v that satisfy $ux + vy = d = \text{gcd}(x, y)$. $\text{gcdx}(x, y)$ returns (d, u, v) .

Examples

|

Note

Bézout coefficients are *not* uniquely defined. gcdx returns the minimal Bézout 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 max , and the identity then holds only via the unsigned integers' modulo arithmetic.

– Function.

|

Test whether x is a power of two.

Examples

|

– Function.

|

The smallest power of two not less than x . Returns 0 for $x = 0$, and returns max for negative arguments.

Examples

|

– Function.

|

The largest power of two not greater than x . Returns 0 for $x < 0$, and returns $-x$ for negative arguments.

Examples

|

– Function.

|

The smallest 2^m not less than x , where m is a non-negative integer. x must be greater than 1, and m must be greater than 0.

Examples

|

See also .

– Function.

|

The largest 2^m not greater than x , where m is a non-negative integer. x must be greater than 1, and m must not be less than 1.

Examples

|

See also .

– Function.

|

Next integer greater than or equal to that can be written as $\prod k_i^{p_i}$ for integers p_1, p_2 , etc.

Example

|

– Function.

|

Take the inverse of modulo : 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

|

– Function.

|

Compute $x^p \pmod{m}$.

Examples

|

– Function.

|

Compute the gamma function of .

– Function.

|

Compute the logarithm of the absolute value of for , while for compute the principal branch cut of the logarithm of (defined for negative by analytic continuation from positive).

– Function.

|

Compute the logarithmic factorial of a nonnegative integer . Equivalent to of , but extends this function to non-integer .

– Function.

|

Euler integral of the first kind $B(x, y) = \Gamma(x)\Gamma(y)/\Gamma(x + y)$.

– Function.

|

Natural logarithm of the absolute value of the function $\log(|B(x, y)|)$.

– Function.

|

Compute the number of digits in integer `x` written in base `b`. The base `b` must not be in `[0, 1]`.

Examples

|

– Function.

|

Multiply `x` and `y`, giving the result as a larger type.

|

– Macro.

|

Evaluate the polynomial $\sum_k c[k]z^{k-1}$ for the coefficients `c[0], ..., c[n-1]`; 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.

|

47.3 Estadística

– Function.

|

Apply the function `f` to each element of `x` and take the mean.

|

|

Compute the mean of whole array `x`, or optionally along the dimensions in `dims`.

Note

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `Statistics` package is recommended.

– Function.

|

Compute the mean of `x` over the singleton dimensions of `x`, and write results to `out`.

– Function.

|

Compute the sample standard deviation of a vector or array `x`, optionally along dimensions in `dims`. The algorithm returns an estimator of the generative distribution's standard deviation under the assumption that each entry of `x` is an IID drawn from that generative distribution. This computation is equivalent to calculating $\sqrt{\frac{1}{n} \sum (x_i - \bar{x})^2}$. A pre-computed `mean` may be provided. If `mean` is `nothing`, then the sum is scaled with `n`, whereas the sum is scaled with `n-1` if `mean` is `mean(x)` where `n` is the number of elements in `x`.

Note

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `Statistics` package is recommended.

– Function.

|

Compute the sample standard deviation of a vector `x` with known mean `μ`. If `μ` is `nothing`, then the sum is scaled with `n`, whereas the sum is scaled with `n-1` if `μ` is `mean(x)` where `n` is the number of elements in `x`.

Note

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `Statistics` package is recommended.

– Function.

|

Compute the sample variance of a vector or array `x`, optionally along dimensions `dims`. The algorithm will return an estimator of the generative distribution's variance under the assumption that each entry of `x` is an IID drawn from that generative distribution. This computation is equivalent to calculating $\frac{1}{n} \sum (x_i - \bar{x})^2$. If `dims` is `1`, then the sum is scaled with `n`, whereas the sum is scaled with `n - 1` if `dims` is `2` where `n` is the number of elements. The mean `μ` 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 `Statistics` package is recommended.

– Function.

|

Compute the sample variance of a collection `xs` with known mean(s) `μ`, optionally over `dims`. `xs` may contain means for each dimension of `xs`. If `dims` is `1`, then the sum is scaled with `n`, whereas the sum is scaled with `n - 1` if `dims` is `2` where `n` is the number of elements.

Note

Julia does not ignore `NaN` values in the computation. For applications requiring the handling of missing data, the `Statistics` package is recommended.

– Function.

|

Compute the middle of a scalar value, which is equivalent to `x` itself, but of the type of `μ` for consistency.

|

Compute the middle of two reals `x` and `y`, which is equivalent in both value and type to computing their mean `(x + y) / 2`.

|

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.

|

|

Compute the middle of an array `x`, which consists of finding its extrema and then computing their mean.

|

– Function.

|

Compute the median of an entire array , or, optionally, along the dimensions in . 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 values in the computation. For applications requiring the handling of missing data, the package is recommended.

– Function.

|

Like , but may overwrite the input vector.

– Function.

|

Compute the quantile(s) of a vector at a specified probability or vector . The keyword argument indicates whether can be assumed to be sorted.

The should be on the interval [0,1], and should not have any values.

Quantiles are computed via linear interpolation between the points , for where . This corresponds to Definition 7 of Hyndman and Fan (1996), and is the same as the R default.

Note

Julia does not ignore values in the computation. For applications requiring the handling of missing data, the package is recommended. will throw an in the presence of 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

– Function.

|

Compute the quantile(s) of a vector `x` at the probabilities `prob`, with optional output into array `out` (if not provided, a new output array is created). The keyword argument `sorted` indicates whether `x` can be assumed to be sorted; if `sorted` (the default), then the elements of `prob` may be partially sorted.

The elements of `prob` should be on the interval `[0,1]`, and `prob` should not have any `NaN` values.

Quantiles are computed via linear interpolation between the points `x`, for `prob` where `prob` is `sorted`. 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 `Missings.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

– Function.

|

Compute the variance of the vector `x`. If `axis` is `1` (the default) then the sum is scaled with `1/(n-1)`, whereas the sum is scaled with `1/n` if `axis` is `2` where `n` is the length of `x`.

|

Compute the covariance matrix of the matrix `X` along the dimension `axis`. If `axis` is `1` (the default) then the sum is scaled with `1/(n-1)`, whereas the sum is scaled with `1/n` if `axis` is `2` where `n` is the number of rows in `X`.

|

Compute the covariance between the vectors `x` and `y`. If `axis` is `1` (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 \bar{x} is the mean of `x`. If `axis` is `2`, computes $\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})^*$.

|

Compute the covariance between the vectors or matrices `x` and `y` along the dimension `axis`. If `axis` is `1` (the default) then the sum is scaled with `1/(n-1)`, whereas the sum is scaled with `1/n` if `axis` is `2` where `n` is the number of rows in `x` and `y`.

– Function.

|

Return the number one.

|

Compute the Pearson correlation matrix of the matrix `X` along the dimension `axis`.

|

Compute the Pearson correlation between the vectors `x` and `y`.

|

Compute the Pearson correlation between the vectors or matrices `x` and `y` along the dimension `dim`.

47.4 Procesamiento de Señales

Las funciones de transformada rápida de Fourier (*Fast Fourier transform* – FFT) en Julia están implementadas mediante llamadas a funciones de la librería [FFTW](#).

– Function.

|

Performs a multidimensional FFT of the array `A`. The optional `dims` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `A` along the transformed dimensions is a product of small primes; see [FFTW docs](#). See also [FFTW docs](#) for even greater efficiency.

A one-dimensional FFT computes the one-dimensional discrete Fourier transform (DFT) as defined by

$$\text{DFT}(A)[k] = \sum_{n=1}^{\text{length}(A)} \exp\left(-i \frac{2\pi(n-1)(k-1)}{\text{length}(A)}\right) A[n].$$

A multidimensional FFT simply performs this operation along each transformed dimension of `A`.

Note

- Julia starts FFTW up with 1 thread by default. Higher performance is usually possible by increasing number of threads. Use `Threads.nthreads()` to use as many threads as cores on your system.
- This performs a multidimensional FFT by default. FFT libraries in other languages such as Python and Octave perform a one-dimensional FFT along the first non-singleton dimension of the array. This is worth noting while performing comparisons. For more details, refer to the [Noteworthy Differences from other Languages](#) section of the manual.

– Function.

|

Same as `fft`, but operates in-place on `A`, which must be an array of complex floating-point numbers.

– Function.

|

Multidimensional inverse FFT.

A one-dimensional inverse FFT computes

$$\text{IDFT}(A)[k] = \frac{1}{\text{length}(A)} \sum_{n=1}^{\text{length}(A)} \exp\left(+i \frac{2\pi(n-1)(k-1)}{\text{length}(A)}\right) A[n].$$

A multidimensional inverse FFT simply performs this operation along each transformed dimension of .

– Function.

|

Same as , but operates in-place on .

– Function.

|

Similar to , but computes an unnormalized inverse (backward) transform, which must be divided by the product of the sizes of the transformed dimensions in order to obtain the inverse. (This is slightly more efficient than because it omits a scaling step, which in some applications can be combined with other computational steps elsewhere.)

$$\text{BDFT}(A)[k] = \text{length}(A) \text{IDFT}(A)[k]$$

– Function.

|

Same as , but operates in-place on .

– Function.

|

Pre-plan an optimized FFT along given dimensions () of arrays matching the shape and type of . (The first two arguments have the same meaning as for .) Returns an object which represents the linear operator computed by the FFT, and which contains all of the information needed to compute quickly.

To apply to an array , use ; in general, the syntax for applying plans is much like that of matrices. (A plan can only be applied to arrays of the same size as the for which the plan was created.) You can also apply a plan with a preallocated output array by calling . (For , however, the input array must be a complex floating-point array like the output .) You can compute the inverse-transform plan by and apply the inverse plan with (the inverse plan is cached and reused for subsequent calls to or), and apply the inverse plan to a pre-allocated output array with .

The argument is a bitwise-or of FFTW planner flags, defaulting to . e.g. passing or will instead spend several seconds (or more) benchmarking different possible FFT algorithms and picking the fastest one; see the FFTW

manual for more information on planner flags. The optional `maxtime` argument specifies a rough upper bound on the allowed planning time, in seconds. Passing `0` or `None` may cause the input array `x` to be overwritten with zeros during plan creation.

`ifft` is the same as `fft` but creates a plan that operates in-place on its argument (which must be an array of complex floating-point numbers). `ifft2` and so on are similar but produce plans that perform the equivalent of the inverse transforms `ifft` and so on.

– Function.

|

Same as `fft`, but produces a plan that performs inverse transforms.

– Function.

|

Same as `fft`, but produces a plan that performs an unnormalized backwards transform.

– Function.

|

Same as `fft`, but operates in-place on `x`.

– Function.

|

Same as `fft`, but operates in-place on `x`.

– Function.

|

Same as `fft`, but operates in-place on `x`.

– Function.

|

Multidimensional FFT of a real array `x`, exploiting the fact that the transform has conjugate symmetry in order to save roughly half the computational time and storage costs compared with `fft`. If `x` has size `(N1, N2, ...)`, the result has size `(N1/2, N2, ...)`.

The optional `axes` argument specifies an iterable subset of one or more dimensions of `x` to transform, similar to `fft`. Instead of (roughly) halving the first dimension of `x` in the result, the `axes` dimension is (roughly) halved in the same way.

- Function.

|

Inverse of : for a complex array , gives the corresponding real array whose FFT yields in the first half. As for , is an optional subset of dimensions to transform, defaulting to .

is the length of the transformed real array along the dimension, which must satisfy . (This parameter cannot be inferred from since both as well as are valid sizes for the transformed real array.)

- Function.

|

Similar to but computes an unnormalized inverse transform (similar to), which must be divided by the product of the sizes of the transformed dimensions (of the real output array) in order to obtain the inverse transform.

- Function.

|

Pre-plan an optimized real-input FFT, similar to except for instead of . The first two arguments, and the size of the transformed result, are the same as for .

- Function.

|

Pre-plan an optimized real-input unnormalized transform, similar to except for instead of . The first two arguments and the size of the transformed result, are the same as for .

- Function.

|

Pre-plan an optimized inverse real-input FFT, similar to except for and , respectively. The first three arguments have the same meaning as for .

- Function.

|

Performs a multidimensional type-II discrete cosine transform (DCT) of the array , using the unitary normalization of the DCT. The optional argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of along the transformed dimensions is a product of small primes; see . See also for even greater efficiency.

- Function.

|

Same as `scipy.fftpack.dct`, except that it operates in-place on `a`, which must be an array of real or complex floating-point values.

– Function.

|

Computes the multidimensional inverse discrete cosine transform (DCT) of the array `a` (technically, a type-III DCT with the unitary normalization). The optional `axes` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. Most efficient if the size of `a` along the transformed dimensions is a product of small primes; see `scipy.fftpack.dct`. See also `scipy.fftpack.dctn` for even greater efficiency.

– Function.

|

Same as `scipy.fftpack.dctn`, but operates in-place on `a`.

– Function.

|

Pre-plan an optimized discrete cosine transform (DCT), similar to `scipy.fftpack.dctn` except producing a function that computes `a`. The first two arguments have the same meaning as for `scipy.fftpack.dctn`.

– Function.

|

Same as `scipy.fftpack.dctn`, but operates in-place on `a`.

– Function.

|

Pre-plan an optimized inverse discrete cosine transform (DCT), similar to `scipy.fftpack.dctn` except producing a function that computes `a`. The first two arguments have the same meaning as for `scipy.fftpack.dctn`.

– Function.

|

Same as `scipy.fftpack.dctn`, but operates in-place on `a`.

– Method.

|

Swap the first and second halves of each dimension of .

– Method.

|

Swap the first and second halves of the given dimension or iterable of dimensions of array .

– Function.

|

Undoes the effect of .

– Function.

|

Apply filter described by vectors and to vector , with an optional initial filter state vector (defaults to zeros).

– Function.

|

Same as but writes the result into the argument, which may alias the input to modify it in-place.

– Function.

|

Construct vector such that . Equivalent to polynomial division.

– Function.

|

Convolution of two vectors. Uses FFT algorithm.

– Function.

|

2-D convolution of the matrix with the 2-D separable kernel generated by the vectors and . Uses 2-D FFT algorithm.

|

2-D convolution of the matrix `x` with the matrix `y`. Uses 2-D FFT algorithm.

– Function.

|

Compute the cross-correlation of two vectors.

Las siguientes funciones están definidas dentro del módulo `scipy.fftw`.

– Function.

|

Performs a multidimensional real-input/real-output (r2r) transform of type `type` of the array `x`, as defined in the FFTW manual. `type` specifies either a discrete cosine transform of various types (`'DCT-1'`, `'DCT-2'`, `'DCT-3'`, or `'DCT-4'`), a discrete sine transform of various types (`'DST-1'`, `'DST-2'`, or `'DST-3'`), a real-input DFT with halfcomplex-format output (`'DFT'` and its inverse `'IDFT'`), or a discrete Hartley transform (`'HARTLEY'`). The `axes` argument may be an array or tuple in order to specify different transform types along the different dimensions of `x`; `type` is used for any unspecified dimensions. See the FFTW manual for precise definitions of these transform types, at <http://www.fftw.org/doc>.

The optional `axes` argument specifies an iterable subset of dimensions (e.g. an integer, range, tuple, or array) to transform along. `type` is then the transform type for `axes`, with `type` being used for `axes`.

See also `scipy.fftw.interfaces.numpy_fft` to pre-plan optimized r2r transforms.

– Function.

|

Same as `scipy.fftw.r2r`, but operates in-place on `x`, which must be an array of real or complex floating-point numbers.

– Function.

|

Pre-plan an optimized r2r transform, similar to `scipy.fftw.r2r` except that the transforms (and the first three arguments) correspond to `axes` and `type`, respectively.

– Function.

|

Similar to `scipy.fftw.r2r`, but corresponds to `axes`.

Chapter 48

Números

48.1 Tipos Numéricos Estándar

Tipos Numéricos Abstractos

- Type.

|

Abstract supertype for all number types.

- Type.

|

Abstract supertype for all real numbers.

- Type.

|

Abstract supertype for all floating point numbers.

- Type.

|

Abstract supertype for all integers.

- Type.

|

Abstract supertype for all signed integers.

- Type.

|

Abstract supertype for all unsigned integers.

Tipos Numéricos Concretos

- Type.

|

16-bit floating point number type.

- Type.

|

32-bit floating point number type.

- Type.

|

64-bit floating point number type.

- Type.

|

Arbitrary precision floating point number type.

- Type.

|

Boolean type.

- Type.

|

8-bit signed integer type.

- Type.

|

8-bit unsigned integer type.

– Type.

|

16-bit signed integer type.

– Type.

|

16-bit unsigned integer type.

– Type.

|

32-bit signed integer type.

– Type.

|

32-bit unsigned integer type.

– Type.

|

64-bit signed integer type.

– Type.

|

64-bit unsigned integer type.

– Type.

|

128-bit signed integer type.

– Type.

|

128-bit unsigned integer type.

– Type.

|

Arbitrary precision integer type.

– Type.

|

Complex number type with real and imaginary part of type .

, and are aliases for , and respectively.

– Type.

|

Rational number type, with numerator and denominator of type .

– Type.

|

Irrational number type.

48.2 Formatos de Datos

– Function.

|

Convert an integer to a binary string, optionally specifying a number of digits to pad to.

|

– Function.

|

Convert an integer to a hexadecimal string, optionally specifying a number of digits to pad to.

|

– Function.

|

Convert an integer to a decimal string, optionally specifying a number of digits to pad to.

Examples

|

– Function.

|

Convert an integer to an octal string, optionally specifying a number of digits to pad to.

|

– Function.

|

Convert an integer to a string in the given , optionally specifying a number of digits to pad to.

|

– Function.

|

Returns an array with element type (default) of the digits of in the given base, optionally padded with zeros to a specified size. More significant digits are at higher indexes, such that .

Examples

|

– Function.

|

Fills an array of the digits of 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

|

– Function.

|

A string giving the literal bit representation of a number.

Example

|

– Method.

|

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.

|

– Function.

|

Like , but returns a of the requested type. The result will be null if the string does not contain a valid number.

– Function.

|

Convert a number to a maximum precision representation (typically or). See for information about some pitfalls with floating-point numbers.

– Function.

|

Convert a number to a signed integer. If the argument is unsigned, it is reinterpreted as signed without checking for overflow.

– Function.

|

Convert a number to an unsigned integer. If the argument is signed, it is reinterpreted as unsigned without checking for negative values.

Examples

|

– Method.

|

Convert a number or array to a floating point data type. When passed a string, this function is equivalent to .

– Function.

|

Extract the (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 `0` is returned.

Examples

|

– Function.

|

Get the exponent of a normalized floating-point number.

– Method.

|

Convert real numbers or arrays to complex. defaults to zero.

– Function.

|

Byte-swap an integer. Flip the bits of its binary representation.

Examples

|

– Function.

|

Get a hexadecimal string of the binary representation of a floating point number.

Example

|

– Function.

|

Convert a hexadecimal string to the floating point number it represents.

– Function.

|

Convert an arbitrarily long hexadecimal string to its binary representation. Returns an , i.e. an array of bytes.

– Function.

Convert an array of bytes to its hexadecimal representation. All characters are in lower-case.

48.3 Constantes y Funciones de Números Generales

– Function.

Return a multiplicative identity for : a value such that . Alternatively can take a type , in which case returns a multiplicative identity for any of type .

If possible, returns a value of the same type as , and returns a value of type . 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, should return an identity value of the same precision (and shape, for matrices) as .

If you want a quantity that is of the same type as , or of type , even if is dimensionful, use instead.

– Function.

|

Returns π , where π is either the type of the argument or (if a type is passed) the argument. This differs from π for dimensionful quantities: π is dimensionless (a multiplicative identity) while π is dimensionful (of the same type as π , or of type π).

|

– Function.

|

Get the additive identity element for the type of π (π can also specify the type itself).

|

– Constant.

|

The constant π .

|

– Constant.

|

The imaginary unit.

– Constant.

|

The constant e .

|

– Constant.

|

Catalan's constant.

|

– Constant.

|

Euler's constant.

|

– Constant.

|

The golden ratio.

|

– Constant.

|

Positive infinity of type ∞ .

– Constant.

|

Positive infinity of type .

– Constant.

|

Positive infinity of type .

– Constant.

|

A not-a-number value of type .

– Constant.

|

A not-a-number value of type .

– Constant.

|

A not-a-number value of type .

– Function.

|

Test whether a floating point number is subnormal.

– Function.

|

Test whether a number is finite.

|

- Function.

|

Test whether a number is infinite.

- Function.

|

Test whether a floating point number is not a number (NaN).

- Function.

|

Return if ; if is an array, this checks whether all of the elements of are zero.

- Function.

|

The result of iterative applications of to if , or applications of if .

|

Returns the smallest floating point number of the same type as such . If no such exists (e.g. if is or), then returns .

- Function.

|

Returns the largest floating point number of the same type as such . If no such exists (e.g. if is or), then returns .

- Function.

|

Test whether is numerically equal to some integer.

|

- Function.

|

Test whether `arr` or all its elements are numerically equal to some real number.

|

– Method.

|

Create a `Float32` from `x`. If `x` is not exactly representable then `rounding` determines how `x` is rounded.

Examples

|

See [here](#) for available rounding modes.

– Method.

|

Create a `Float64` from `x`. If `x` is not exactly representable then `rounding` determines how `x` is rounded.

Examples

|

See [here](#) for available rounding modes.

– Method.

|

Create an arbitrary precision integer. `bits` may be an integer (or anything that can be converted to an integer). The usual mathematical operators are defined for this type, and results are promoted to a `BigInt`.

Instances can be constructed from strings via `parse_int`, or using the `"..."` string literal.

– Method.

Create an arbitrary precision floating point number. `Decimal` may be an `int`, a `float` or a `str`. The usual mathematical operators are defined for this type, and results are promoted to a `Decimal`.

Note that because decimal literals are converted to floating point numbers when parsed, `Decimal('1.1')` may not yield what you expect. You may instead prefer to initialize constants from strings via `Decimal('1.1')`, or using the `Decimal` string literal.

– Function.

Get the current floating point rounding mode for type `float`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `int()`) and type conversion.

See `decimal.FROUNDINGMODES` for available modes.

– Method.

Set the rounding mode of floating point type `float`, controlling the rounding of basic arithmetic functions (`+`, `-`, `*`, `/` and `int()`) and type conversion. Other numerical functions may give incorrect or invalid values when using rounding modes other than the default `ROUND_HALF_UP`.

Note that this may affect other types, for instance changing the rounding mode of `float` will change the rounding mode of `Decimal`. See `decimal.FROUNDINGMODES` for available modes.

Warning

This feature is still experimental, and may give unexpected or incorrect values.

– Method.

Change the rounding mode of floating point type `float` for the duration of `context`. It is logically equivalent to:

See [fenvt](#) 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.

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.

– Function.

Returns `1` if operations on subnormal floating-point values ("denormals") obey rules for IEEE arithmetic, and `0` if they might be converted to zeros.

– Function.

If `is_fast` is `1`, 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 `1` unless `is_fast` is `0` but the hardware does not support zeroing of subnormal numbers.

`is_fast` can speed up some computations on some hardware. However, it can break identities such as `is_fast & is_fast == 1`.

Enteros

– Function.

Number of ones in the binary representation of `x`.

|

– Function.

|

Number of zeros in the binary representation of .

|

– Function.

|

Number of zeros leading the binary representation of .

|

– Function.

|

Number of ones leading the binary representation of .

|

– Function.

|

Number of zeros trailing the binary representation of .

|

– Function.

|

Number of ones trailing the binary representation of .

|

– Function.

|

Returns `if` is odd (that is, not divisible by 2), and `otherwise`.

|

– Function.

|

Returns `is` is even (that is, divisible by 2), and `otherwise`.

|

48.4 BigFloats

El tipo `implements` el punto flotante de precisión arbitraria usando la librería [GNU MPFR library](#).

– Function.

|

Get the precision of a floating point number, as defined by the effective number of bits in the mantissa.

– Method.

|

Get the precision (in bits) currently used for `arithmetic`.

– Function.

|

Set the precision (in bits) to be used for `arithmetic`.

|

|

Reseed the random number generator. If `seed` is provided, the RNG will give a reproducible sequence of numbers, otherwise Julia will get entropy from the system. For `seed`, the value may be a non-negative integer, a vector of integers or a filename, in which case the seed is read from a file (`bytes` bytes are read from the file, where `bytes` is an optional argument). `RandomDevice` does not support seeding.

– Type.

|

Create a `RandomDevice` object. Different RNG objects can have their own seeds, which may be useful for generating different streams of random numbers.

Example

|

– Type.

|

Create a `RandomDevice` object. Two such objects will always generate different streams of random numbers.

– Function.

|

Pick a random element or array of random elements from the set of values specified by `values`; `dims` can be

- an indexable collection (for example `Array{Int,1}` or `Vector{Int}`), or
- a type: the set of values to pick from is then equivalent to `Int` for integers (this is not applicable to `Float64`), and to `[0, 1)` for floating point numbers;

defaults to `1`.

– Function.

|

Populate the array `arr` with random values. If the indexable collection `values` is specified, the values are picked randomly from `values`. This is equivalent to `fill(arr, rand(values, dims))` but without allocating a new array.

Example

|

– Function.

|

Generate a of random boolean values.

Example

|

– Function.

|

Generate a normally-distributed random number of type with mean 0 and standard deviation 1. Optionally generate an array of normally-distributed random numbers. The module currently provides an implementation for the types , , and (the default).

Examples

|

– Function.

|

Fill the array with normally-distributed (mean 0, standard deviation 1) random numbers. Also see the function.

Example

|

– Function.

|

Generate a random number of type according to the exponential distribution with scale 1. Optionally generate an array of such random numbers. The module currently provides an implementation for the types , , and (the default).

Examples

|

– Function.

|

Fill the array with random numbers following the exponential distribution (with scale 1).

Example

|

– Function.

Create an array of the size of initialized RNG objects. The first RNG object given as a parameter and following 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 .

Default jump polynomial moves forward RNG state by steps.

Chapter 49

Cadenas

– Method.

|

The number of characters in string .

Example

|

– Method.

|

The number of bytes in string .

Example

|

– Method.

|

Multiplication operator. calls this function with all arguments, i.e. .

– Method.

|

Repeat times the string . The function is an alias to this operator.

|

- Function.

|

Create a string from any values using the `strcat` function.

|

- Function.

|

Create a string from any value using the `strdup` function.

- Method.

|

Convert a string to a contiguous byte array representation encoded as UTF-8 bytes. This representation is often appropriate for passing strings to C.

- Function.

|

Convert string data between Unicode encodings. `strncpy` is either a `char*` or a `wchar_t*` of UTF-XX code units, where `XX` is 8, 16, or 32. `strncpy` indicates the encoding of the return value: `strncpy` to return a (UTF-8 encoded) `char*` or `strncpy` to return a `wchar_t*` of UTF- data. (The alias `strncpy` can also be used as the integer type, for converting `strings` used by external C libraries.)

The `strncpy` 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.

- Function.

|

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 `ptr` is not a valid memory address to data of the requested length.

- Method.

|

Get the `th` code unit of an encoded string. For example, returns the `th` byte of the representation of a UTF-8 string.

– Function.

|

Convert a string to `type` and check that it contains only ASCII data, otherwise throwing an `indicating the position of the first non-ASCII byte`.

|

– Macro.

|

Construct a regex, such as `.` The regex also accepts one or more flags, listed after the ending quote, to change its behaviour:

- enables case-insensitive matching
- treats the `and` tokens as matching the start and end of individual lines, as opposed to the whole string.
- allows the `modifier` to match newlines.
- 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:

|

– Macro.

|

Create an `object` from a literal string.

– Macro.

|

Create a `object` from a literal string.

– Function.

|

Normalize the string according to one of the four "normal forms" of the Unicode standard: can be , , , or . 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 , where any number of the following boolean keywords options (which all default to except for) are specified:

- : do not perform canonical composition
- : do canonical decomposition instead of canonical composition (is ignored if present)
- : compatibility equivalents are canonicalized
- : perform Unicode case folding, e.g. for case-insensitive string comparison
- , , or : convert various newline sequences (LF, CRLF, CR, NEL) into a linefeed (LF), line-separation (LS), or paragraph-separation (PS) character, respectively
- : strip diacritical marks (e.g. accents)
- : strip Unicode's "default ignorable" characters (e.g. the soft hyphen or the left-to-right marker)
- : 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
- : throw an error if unassigned code points are found
- : enforce Unicode Versioning Stability

For example, NFKC corresponds to the options .

– Function.

|

Returns an iterator over substrings of 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.)

– Method.

|

Returns if the given value is valid for its type, which currently can be either or .

– Method.

|

Returns `True` if the given value is valid for that type. Types currently can be either `int` or `float`. Values for `int` can be of type `int` or `float`. Values for `float` can be of that type, or `int`.

– Method.

|

Tells whether index `i` is valid for the given string.

Examples

|

– Function.

|

Returns `True` if the given char or integer is an assigned Unicode code point.

– Function.

|

Test whether a string contains a match of the given regular expression.

– Function.

|

Search for the first match of the regular expression `re` in `s` and return a `Match` object containing the match, or `None` if the match failed. The matching substring can be retrieved by accessing `match.group()` and the captured sequences can be retrieved by accessing `match.groups()`. The optional `pos` argument specifies an index at which to start the search.

– Function.

|

Search for all matches of a the regular expression `in` 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.

– Function.

```
|
```

Return a vector of the matching substrings from `s`.

– Function.

```
|
```

Make a string at least `width` columns wide when printed by padding `str` on the left with copies of `fill`.

```
|
```

– Function.

```
|
```

Make a string at least `width` columns wide when printed by padding `str` on the right with copies of `fill`.

```
|
```

– Function.

```
|
```

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 :

`begin` = such that `str[begin, end)` matches `pat`, or `0` if unmatched.

`end` = such that `str[begin, end)` matches `pat`, or `str.size()` if unmatched.

```
|
```

– Function.

|

Similar to , but returning the last occurrence of the given characters within the given string, searching in reverse from .

|

– Function.

|

Similar to , but return only the start index at which the substring is found, or if it is not.

|

– Function.

|

Similar to , but return only the start index at which the substring is found, or if it is not.

|

– Method.

|

Determine whether the second argument is a substring of the first.

|

– Method.

|

Reverses a string.

|

– Function.

|

Search for the given pattern , and replace each occurrence with . If is provided, replace at most occurrences. As with search, the second argument may be a single character, a vector or a set of characters, a string, or a regular expression. If is a function, each occurrence is replaced with where is the matched substring. If is a regular expression and is a , then capture group references in are replaced with the corresponding matched text.

– Function.

|

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 's second argument (i.e. a single character, collection of characters, string, or regular expression). If is omitted, it defaults to the set of all space characters, and is taken to be . 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.

|

– Function.

|

Similar to , but starting from the end of the string.

|

– Function.

Return with any leading and trailing whitespace removed. If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

– Function.

Return with any leading whitespace and delimiters removed. The default delimiters to remove are , , , , and . If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

– Function.

Return with any trailing whitespace and delimiters removed. The default delimiters to remove are , , , , and . If (a character, or vector or set of characters) is provided, instead remove characters contained in it.

– Function.

Returns if starts with . If is a vector or set of characters, tests whether the first character of belongs to that set. See also .

|

– Function.

|

Returns if ends with . If is a vector or set of characters, tests whether the last character of belongs to that set.

See also .

|

– Function.

|

Returns with all characters converted to uppercase.

Example

|

– Function.

|

Returns with all characters converted to lowercase.

Example

|

– Function.

|

Capitalizes the first character of each word in .

Example

|

– Function.

|

Returns with the first character converted to uppercase.

Example

|

– Function.

|

Returns with the first character converted to lowercase.

Example

|

– Function.

|

Join an array of into a single string, inserting the given delimiter between adjacent strings. If is given, it will be used instead of between the last two strings. For example,

|

can be any iterable over elements which are convertible to strings via . will be printed to .

– Function.

|

Remove the last character from .

|

– Function.

|

Remove a single trailing newline from a string.

|

– Function.

|

Convert a byte index to a character index with respect to string .

See also .

Example

|

– Function.

|

Convert a character index to a byte index.

See also .

Example

|

– Function.

|

Get the next valid string index after . Returns a value greater than at or after the end of the string.

Examples

|

|

– Function.

|

Get the previous valid string index before . Returns a value less than at the beginning of the string.

Examples

|

– Function.

|

Create a random ASCII string of length , consisting of upper- and lower-case letters and the digits 0-9. The optional argument specifies a random number generator, see [Random Numbers](#).

Example

|

– Function.

|

Gives the number of columns needed to print a character.

– Function.

|

Gives the number of columns needed to print a string.

Example

|

- Function.

|

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'.

- Function.

|

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'.

- Function.

|

Tests whether a character belongs to the ASCII character set, or whether this is true for all elements of a string.

- Function.

|

Tests whether a character is a control character. Control characters are the non-printing characters of the Latin-1 subset of Unicode.

- Function.

|

Tests whether a character is a numeric digit (0-9).

- Function.

|

Tests whether a character is printable, and not a space. Any character that would cause a printer to use ink should be classified with .

- Function.

|

Tests whether a character is a lowercase letter. A character is classified as lowercase if it belongs to Unicode category Ll, Letter: Lowercase.

– Function.

|

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'.

– Function.

|

Tests whether a character is printable, including spaces, but not a control character.

– Function.

|

Tests whether a character belongs to the Unicode general category Punctuation, i.e. a character whose category code begins with 'P'.

– Function.

|

Tests whether a character is any whitespace character. Includes ASCII characters '\t', '\n', '\v', '\f', '\r', and ' ', Latin-1 character U+0085, and characters in Unicode category Zs.

– Function.

|

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.

– Function.

|

Tests whether a character is a valid hexadecimal digit. Note that this does not include (as in the standard prefix).

Example

|

– Type.

|

Create a by concatenating the string representations of the arguments together.

– Function.

|

General escaping of traditional C and Unicode escape sequences. Any characters in are also escaped (with a backslash). See also .

– Function.

|

General unescaping of traditional C and Unicode escape sequences. Reverse of .

Chapter 50

Arrays

50.1 Constructores y Tipos

- Type.

|

Abstract array supertype which arrays inherit from.

- Type.

|

Construct an uninitialized -dimensional dense array with element type , where is determined from the length or number of . may be a tuple or a series of integer arguments corresponding to the lengths in each dimension. If the rank is supplied explicitly as in , then it must match the length or number of .

Example

|

- Method.

|

Construct a 1-d array of the specified type. This is usually called with the syntax . Element values can be specified using .

Example

|

– Function.

|

Create an array of all zeros with the same layout as , element type and size . The argument can be skipped, which behaves like was passed. For convenience may also be passed in variadic form.

Examples

|

See also , .

– Function.

|

Create an array of all ones with the same layout as , element type and size . The argument can be skipped, which behaves like was passed. For convenience may also be passed in variadic form.

Examples

|

See also , .

– Type.

|

Construct an uninitialized with the given dimensions. Behaves identically to the constructor.

|

|

|

Construct a `ndarray` generated by the given iterable object. The shape is inferred from the `obj` object.

|

– `ones` Function.

|

Create a `ndarray` with all values set to `1`.

|

|

Create a `ndarray` with all values set to `1` of the same shape as `obj`.

|

– Function.

|

Create a with all values set to .

|

|

Create a with all values set to of the same shape as .

|

– Function.

|

Create an array filled with the value . For example, returns a 5×5 array of floats, with each element initialized to .

|

If is an object reference, all elements will refer to the same object. will return an array filled with the result of evaluating once.

– Function.

|

Fill array with the value . If is an object reference, all elements will refer to the same object. will return filled with the result of evaluating once.

Examples

– Method.

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 `elementType` and `length`. 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`.

For example, `Array.uninitializedMutableOfRange` returns an uninitialized `Array` since ranges are neither mutable nor support 2 dimensions:

Conversely, `Array.uninitializedMutableOf2D` returns an uninitialized `Array` with two elements since `s` are both mutable and can support 1-dimensional arrays:

Since `s` can only store elements of type `String`, however, if you request a different element type it will create a regular `Array` instead:

– Method.

|

Create an uninitialized mutable array analogous to that specified by , but with specified by the last argument. might be a type or a function.

Examples:

|

creates an array that "acts like" an (and might indeed be backed by one), but which is indexed identically to . If has conventional indexing, this will be identical to , but if has unconventional indexing then the indices of the result will match .

|

would create a 1-dimensional logical array whose indices match those of the columns of .

|

would create an array of , initialized to zero, matching the indices of .

– Function.

|

-by- identity matrix. The default element type is .

Examples

|

|

-by- identity matrix.

|

-by- identity matrix. The default element type is .

Examples

```
|
```

```
|
```

Constructs an identity matrix of the same dimensions and type as .

```
|
```

Note the difference from .

– Function.

```
|
```

Construct a range of linearly spaced elements from to .

```
|
```

– Function.

```
|
```

Construct a vector of logarithmically spaced numbers from to .

```
|
```

– Function.

|

Return a vector consisting of a random subsequence of the given array , where each element of is included (in order) with independent probability . (Complexity is linear in , so this function is efficient even if is small and is large.) Technically, this process is known as "Bernoulli sampling" of .

– Function.

|

Like , but the results are stored in (which is resized as needed).

50.2 Funciones básicas

– Function.

|

Returns the number of dimensions of .

|

– Function.

|

Returns a tuple containing the dimensions of . 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.

|

– Method.

|

Returns the tuple of valid indices for array .

|

– Method.

|

Returns the valid range of indices for array along dimension .

|

– Method.

|

Returns the number of elements in .

|

– Function.

|

Creates an iterable object for visiting each index of an `AbstractArray` in an efficient manner. For array types that have opted into fast linear indexing (like `ndarray`), this is simply the `range()`. 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:

|

```
|
```

If you supply more than one argument, will create an iterable object that is fast for all arguments (a if all inputs have fast linear indexing, a otherwise). If the arrays have different sizes and/or dimensionalities, returns an iterable that spans the largest range along each dimension.

– Function.

```
|
```

Returns a specifying the valid range of indices for where is an . For arrays with conventional indexing (indices start at 1), or any multidimensional array, this is ; however, for one-dimensional arrays with unconventional indices, this is .

Calling this function is the "safe" way to write algorithms that exploit linear indexing.

```
|
```

– Type.

```
|
```

specifies the "native indexing style" for array . When you define a new 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:

```
|
```

The default is .

Julia's internal indexing machinery will automatically (and invisibly) convert all indexing operations into the preferred style using or . 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 , 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 depending on the most efficient access pattern. In particular, creates an iterator whose type depends on the setting of this trait.

– Function.

|

Counts the number of nonzero values in array (dense or sparse). Note that this is not a constant-time operation. For sparse matrices, one should usually use , which returns the number of stored values.

|

– Function.

|

Transform an array to its complex conjugate in-place.

See also .

Example

|

– Function.

|

Returns the distance in memory (in number of elements) between adjacent elements in dimension .

|


```
|
```

– Function.

```
|
```

Returns a tuple of the memory strides in each dimension.

```
|
```

– Function.

```
|
```

Returns a tuple of subscripts into array corresponding to the linear index .

```
|
```

```
|
```

Returns a tuple of subscripts into an array with dimensions , corresponding to the linear index .

Example:

```
|
```

provides the indices of the maximum element.

```
|
```

– Function.

```
|
```

The inverse of `linear_index`, returns the linear index corresponding to the provided subscripts.

```
|
```

– Function.

```
|
```

Check that a matrix is square, then return its common dimension. For multiple arguments, return a vector.

Example

```
|
```

50.3 Retransmisión y Vectorización

Ver también la [sintaxis de puntos para vectorizar funciones](#); por ejemplo, `sum` llama implícitamente a `sumv`. En lugar de confiar en los métodos "vectorizados" de funciones como `sum` para operar en arrays, debe usar `sumv` para vectorizar a través de `sumv`.

– Function.

```
|
```

Broadcasts the arrays, tuples, `s`, nullables, and/or scalars to a container of the appropriate type and dimensions. In this context, anything that is not a subtype of `Array`, (except for `s`), `s`, or `s` 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 `s`, it returns an array (expanding singleton dimensions), and treats `s` as 0-dimensional arrays, and tuples as 1-dimensional arrays.

The following additional rule applies to arguments: If there is at least one `s`, and all the arguments are scalars or `s`, it returns a `s` treating `s` as "containers".

A special syntax exists for broadcasting: `s` is equivalent to `s`, and nested `s` calls are fused into a single broadcast loop.



|

– Function.

|

Like `dot`, but store the result of `in` in the array. Note that `in` is only used to store the result, and does not supply arguments to `in` unless it is also listed in the `in`, as in `in` to perform `in`.

– Macro.

|

Convert every function call or operator in `in` into a "dot call" (e.g. convert `in` to `in`), and convert every assignment in `in` to a "dot assignment" (e.g. convert `in` to `in`).

If you want to *avoid* adding dots for selected function calls in `in`, splice those function calls in with `in`. For example, `in` is equivalent to `in` (no dot for `in`).

(`in` is equivalent to a call to `in`.)

|

– Function.

|

Broadcasts the `in` arrays to a common size like `in` and returns an array of the results `in`, where `in` goes over the positions in the broadcast result `in`.

|

|

– Function.

|

Broadcasts the `arr1` and `arr2` arrays to a common size and stores the value from each position in `arr1` at the indices in `indices` given by the same positions in `arr2`.

50.4 Indexación y Asignación

– Method.

|

Returns a subset of array `arr` as specified by `indices`, where each `indices[i]` may be an integer, a slice, or a list of integers. See the manual section on [array indexing](#) for details.

Examples

|

- Method.

|

Store values from array within some subset of as specified by .

- Method.

|

Copy the block of in the range of to the block of in the range of . The sizes of the two regions must match.

- Function.

|

Tests whether the given array has a value associated with index . Returns if the index is out of bounds, or has an undefined reference.

|

- Type.

|

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 an internal vector type () to represent the collection of indices they span before being used.

- Type.

|

Create a multidimensional index , which can be used for indexing a multidimensional array . In particular, is equivalent to . One can freely mix integer and indices; for example, (where and are indices and is an) can be a useful expression when writing algorithms that work along a single dimension of an array of arbitrary dimensionality.

A is sometimes produced by , and always when iterating with an explicit .

Example

– Type.

Define a region spanning a multidimensional rectangular range of integer indices. These are most commonly encountered in the context of iteration, where will return indices equivalent to the nested loops

Consequently these can be useful for writing algorithms that work in arbitrary dimensions.

|

– Function.

|

Convert the tuple to a tuple of indices for use in indexing into array .

The returned tuple must only contain either s or s of scalar indices that are supported by array . It will error upon encountering a novel index type that it does not know how to process.

For simple index types, it defers to the unexported to process each index . While this internal function is not intended to be called directly, 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, calls , which then recursively walks through both the given tuple of indices and the dimensional indices of in tandem. As such, not all index types are guaranteed to propagate to .

– Function.

|

Return if the specified indices are in bounds for the given array . Subtypes of should specialize this method if they need to provide custom bounds checking behaviors; however, in many cases one can rely on 's indices and .

See also .

|

|

Throw an error if the specified indices are not in bounds for the given array .

– Function.

|

Return if the given is within the bounds of . 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.

|

50.5 Vistas (SubArrays y otros tipos de vistas)

– Function.

|

Like , but returns a view into the parent array with the given indices instead of making a copy. Calling or on the returned computes the indices to the parent array on the fly without checking bounds.

|

– Macro.

|

Creates a from an indexing expression. This can only be applied directly to a reference expression (e.g.), and should *not* be used as the target of an assignment (e.g.). See also to switch an entire block of code to use views for slicing.

|

– Macro.

|

Convert every array-slicing operation in the given expression (which may be a / block, loop, function, etc.) to return a view. Scalar indices, non-array types, and explicit calls (as opposed to) are unaffected.

Note that the macro only affects expressions that appear explicitly in the given , not array slicing that occurs in functions called by that code.

– Function.

|

Returns the "parent array" of an array view type (e.g.,), or the array itself if it is not a view.

– Function.

|

From an array view , returns the corresponding indexes in the parent.

– Function.

|

Return all the data of where the index for dimension equals . Equivalent to where is in position .

Example

|

– Function.

|

Change the type-interpretation of a block of memory. For arrays, this constructs an array with the same binary data as the given array, but with the specified element type. For example, `view(a, dtype='float64')` interprets the 4 bytes corresponding to `a` as a `float64`.

Warning

It is not allowed to `view` an array to an element type with a larger alignment than the alignment of the array. For a normal array, this is the alignment of its element type. For a reinterpreted array, this is the alignment of the array it was reinterpreted from. For example, `view(a, dtype='float128')` is not allowed but `view(a, dtype='float64')` is allowed.

Examples

|

– Function.

|

Return an array 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 `view` 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 `-1`, 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. The total number of elements must not change.

|

|

– Function.

|

Remove the dimensions specified by `axis` from array `a`. Elements of `a` must be unique and within the range `[0, a.ndim - 1]`. `axis` must equal 1 for all `axis` in `axis`.

Example

|

– Function.

|

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.

Example

|

See also .

50.6 Concatenación and permutación

– Function.

|

Concatenate the input arrays along the specified dimensions in the iterable . For dimensions not in , all input arrays should have the same size, which will also be the size of the output array along that dimension. For dimensions in , the size of the output array is the sum of the sizes of the input arrays along that dimension. If is a single number, the different arrays are tightly stacked along that dimension. If 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, builds a block diagonal matrix, i.e. a block matrix with , , ... as diagonal blocks and matching zero blocks away from the diagonal.

– Function.

|

Concatenate along dimension 1.

|

|

– Function.

|

Concatenate along dimension 2.

|

– Function.

|

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.

|

If the first argument is a single integer , then all block rows are assumed to have block columns.

– Function.

|

Reverse in dimension .

Example

|

– Function.

|

Circularly shift the data in an array. The second argument is a vector giving the amount to shift in each dimension.

Example

|

See also .

– Function.

|

Circularly shift the data in , storing the result in . specifies the amount to shift in each dimension.

The array must be distinct from the array (they cannot alias each other).

See also .

– Function.

|

Copy to , indexing each dimension modulo its length. and 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 agrees with .

Example

|

|

– Method.

|

Returns if there is at least one element in such that is .

|

– Method.

|

Return a vector of the linear indexes of the non-zeros in (determined by). A common use of this is to convert a boolean array to an array of indexes of the elements. If there are no non-zero elements of , returns an empty array.

Examples

|

– Method.

|

Return a vector of the linear indexes of where returns . If there are no such elements of , find returns an empty array.

Examples

|

– Function.

|

Return a vector of indexes for each dimension giving the locations of the non-zeros in (determined by). If there are no non-zero elements of , returns a 2-tuple of empty arrays.

Examples

|

– Function.

|

Return a tuple where and are the row and column indexes of the non-zero values in matrix , and is a vector of the non-zero values.

Example

|

– Method.

|

Return the linear index of the first non-zero value in (determined by). Returns if no such value is found.

Examples

|

– Method.

|

Return the linear index of the first element equal to in . Returns if is not found.

Examples

|

– Method.

|

Return the linear index of the first element of `array` for which `predicate` returns `true`. Returns `-1` if there is no such element.

Examples

|

– Method.

|

Return the linear index of the last non-zero value in `array` (determined by `predicate`). Returns `-1` if there is no non-zero value in `array`.

Examples

|

– Method.

|

Return the linear index of the last element equal to `value` in `array`. Returns `-1` if there is no element of `array` equal to `value`.

Examples

|

|

– Method.

|

Return the linear index of the last element of `for which` returns `.` Returns `if there is no such element.`

Examples

|

– Method.

|

Find the next linear index `>=` of a non-zero element of `, or` if not found.

Examples

|

– Method.

|

Find the next linear index \geq of an element of for which returns , or if not found.

Examples

|

– Method.

|

Find the next linear index \geq of an element of equal to (using), or if not found.

Examples

|

– Method.

|

Find the previous linear index \leq of a non-zero element of , or if not found.

Examples

|

– Method.

|

Find the previous linear index \leq of an element of for which returns , or if not found.

Examples

|

– Method.

|

Find the previous linear index \leq of an element of equal to (using), or if not found.

Examples

|

– Function.

|

Permute the dimensions of array . is a vector specifying a permutation of length . This is a generalization of transpose for multi-dimensional arrays. Transpose is equivalent to .

See also: .

Example

|

|

– Function.

|

Permute the dimensions of array and store the result in the array . is a vector specifying a permutation of length . The preallocated array should have and is completely overwritten. No in-place permutation is supported and unexpected results will happen if and have overlapping memory regions.

See also .

– Type.

|

Given an `AbstractArray` , create a view such that the dimensions appear to be permuted. Similar to , except that no copying occurs (shares storage with).

See also: .

Example

|

– Function.

|

Check two array shapes for compatibility, allowing trailing singleton dimensions, and return whichever shape has more dimensions.

|

50.7 Funciones de Arrays

– Method.

|

Cumulative operation along a dimension (defaults to 1). See also 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 , see: ,

|

|

Like , but using a starting element . The first entry of the result will be . For example:

|

– Function.

|

Cumulative operation `on` along a dimension, storing the result in `.` The dimension defaults to 1. See also `.`

– Function.

|

Cumulative product along a dimension (defaults to 1). See also `.` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

|

– Function.

|

Cumulative product of `along` a dimension, storing the result in `.` The dimension defaults to 1. See also `.`

– Function.

|

Cumulative sum along a dimension (defaults to 1). See also `.` to use a preallocated output array, both for performance and to control the precision of the output (e.g. to avoid overflow).

|

|

– Function.

|

Cumulative sum of along a dimension, storing the result in . The dimension defaults to 1. See also .

– Function.

|

Cumulative sum along a dimension, using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy. The dimension defaults to 1.

– Function.

|

Finite difference operator of matrix or vector . If is a matrix, compute the finite difference over a dimension (default).

Example

|

– Function.

|

Compute differences along vector , using as the spacing between points. The default spacing is one.

Example

|

– Function.

|

Rotate matrix 180 degrees.

Example

|

|

Rotate matrix 180 degrees an integer number of times. If is even, this is equivalent to a .

Examples

|

– Function.

|

Rotate matrix left 90 degrees.

Example

|

|

|

Rotate matrix left 90 degrees an integer number of times. If is zero or a multiple of four, this is equivalent to a .

Examples

|

– Function.

|

Rotate matrix right 90 degrees.

Example

|

|

Rotate matrix right 90 degrees an integer number of times. If is zero or a multiple of four, this is equivalent to a .

Examples

|

– Function.

|

Reduce 2-argument function along dimensions of . is a vector specifying the dimensions to reduce, and is the initial value to use in the reductions. For , , and the 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 .

Examples

|

```
|
```

– Function.

```
|
```

Evaluates to the same as , but is generally faster because the intermediate array is avoided.

Examples

```
|
```

– Function.

```
|
```

Transform the given dimensions of array using function . is called on each slice of of the form . is an integer vector specifying where the colons go in this expression. The results are concatenated along the remaining dimensions. For example, if is and is 4-dimensional, is called on for all and .

Examples

```
|
```

```
|
```

– Function.

```
|
```

Returns the sum of all elements of , using the Kahan-Babuska-Neumaier compensated summation algorithm for additional accuracy.

50.8 Combinatoria

– Function.

```
|
```

Construct a random permutation of length . The optional argument specifies a random number generator (see [Random Numbers](#)). To randomly permute a arbitrary vector, see or .

Example

```
|
```

– Function.

```
|
```


Return the inverse permutation of `p`. If `p[i] = j`, then `inv[j] = i`.

Example

```
|
```

– Function.

```
|
```

Returns `True` if `p` is a valid permutation.

Examples

```
|
```

– Method.

```
|
```

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 `list(p)`. Note that this is generally faster than `list(p)` for large vectors.

See also `list(p)`.

Example

|

– Function.

|

Like , but the inverse of the given permutation is applied.

Example

|

– Function.

|

Construct a random cyclic permutation of length . The optional argument specifies a random number generator, see [Random Numbers](#).

Example

|

– Function.

|

Return a randomly permuted copy of . The optional argument specifies a random number generator (see [Random Numbers](#)). To permute in-place, see . To obtain randomly permuted indices, see .

Example

|

– Function.

|

In-place version of : randomly permute the array in-place, optionally supplying the random-number generator .

Example

|

– Function.

|

Return a copy of reversed from start to stop.

Examples

|

– Function.

|

Given an index in , return the corresponding index in so that . (This can be nontrivial in the case where is a Unicode string.)

– Function.

|

In-place version of .

50.9 BitArrays

s son matrices booleanas "compactas" eficientes en el uso del espacio, que almacenan un bit por valor booleano. Se pueden usar de forma similar a los arrays (que almacenan un byte por valor booleano), y se pueden convertir a/desde este último a través de `toArray` y `fromArray`, respectivamente.

– Function.

```
|
```

Performs a bitwise not operation on `array`. See `invert`.

Example

```
|
```

– Function.

```
|
```

Performs a left rotation operation on `array` and puts the result into `array`. `amount` controls how far to rotate the bits.

```
|
```

Performs a left rotation operation in-place on `array`. `amount` controls how far to rotate the bits.

– Function.

```
|
```

Performs a left rotation operation, returning a new `BitArray`. `amount` controls how far to rotate the bits. See also `rotateLeftInPlace`.

Examples

```
|
```

|

– Function.

|

Performs a right rotation operation on and puts the result into . controls how far to rotate the bits.

|

Performs a right rotation operation in-place on . controls how far to rotate the bits.

– Function.

|

Performs a right rotation operation on , returning a new . controls how far to rotate the bits. See also .

Examples

|

50.10 Matrices y Vectores *Sparse*

Los vectores y las matrices *sparse* soportan ampliamente el mismo conjunto de operaciones que sus contrapartidas densas. las siguientes funcioens son específicas para arrays *sparse*.

– Type.

|

Vector type for storing sparse vectors.

– Type.

|

Matrix type for storing sparse matrices in the [Compressed Sparse Column](#) format.

– Function.

|

Convert an `AbstractMatrix` into a sparse matrix.

Example

|

|

|

Create a sparse matrix of dimensions such that . The function is used to combine duplicates. If and are not specified, they are set to and respectively. If the function is not supplied, defaults to unless the elements of are Booleans in which case defaults to . All elements of must satisfy , and all elements of must satisfy . Numerical zeros in (,) are retained as structural nonzeros; to drop numerical zeros, use .

For additional documentation and an expert driver, see .

Example

|

– Function.

|

Create a sparse vector of length such that . Duplicates are combined using the function, which defaults to if no argument is provided, unless the elements of are Booleans in which case defaults to .

|

|

Create a sparse vector of length `n` where the nonzero indices are keys from the dictionary, and the nonzero values are the values from the dictionary.

|

|

Convert a vector `v` into a sparse vector of length `n`.

Example

|

– Function.

|

Returns `True` if `x` is sparse, and `False` otherwise.

– Function.

|

Convert a sparse matrix or vector `x` into a dense matrix or vector.

Example

|

– Function.

|

Returns the number of stored (filled) elements in a sparse array.

Example

|

– Function.

|

Create a sparse vector of length or sparse matrix of size . This sparse array will not contain any nonzero values. No storage will be allocated for nonzero values during construction. The type defaults to if not specified.

Examples

|

– Function.

|

Create a sparse array with the same structure as that of , but with every nonzero element having the value .

Example

|

Note the difference from .

– Method.

|

Create a sparse identity matrix of size . When is supplied, create a sparse identity matrix of size . The type defaults to if not specified.

is equivalent to , and can be used to efficiently create a sparse multiple of the identity matrix.

– Method.

|

Create a sparse identity matrix with the same size as .

Example

|

Note the difference from .

|

Create a sparse identity matrix of size . When is supplied, create a sparse identity matrix of size . The type defaults to if not specified.

is equivalent to , and can be used to efficiently create a sparse multiple of the identity matrix.

– Function.

|

Construct a sparse diagonal matrix. is a tuple of vectors containing the diagonals and is a tuple containing the positions of the diagonals. In the case the input contains only one diagonal, can be a vector (instead of a tuple) and can be the diagonal position (instead of a tuple), defaulting to 0 (diagonal). Optionally, and specify the size of the resulting sparse matrix.

Example

|

– Function.

|

Create a random length `sparse` vector or `by` 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 `dist` and have the type `dtype`. The uniform distribution is used in case `dist` is not specified. The optional `rng` argument specifies a random number generator, see [Random Numbers](#).

Example

|

– Function.

|

Create a random sparse vector of length `n` 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](#).

Example

|

– Function.

|

Return a vector of the structural nonzero values in sparse array . This includes zeros that are explicitly stored in the sparse array. The returned vector points directly to the internal nonzero storage of , and any modifications to the returned vector will mutate as well. See and .

Example

|

– Function.

|

Return a vector of the row indices of . Any modifications to the returned vector will mutate 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 and .

Example

|

– Function.

|

Return the range of indices to the structural nonzero values of a sparse matrix column. In conjunction with and , this allows for convenient iterating over a sparse matrix :

|

– Method.

|

Removes stored numerical zeros from , optionally trimming resulting excess space from and when is .

For an out-of-place version, see . For algorithmic information, see .

– Method.

|

Generates a copy of and removes stored numerical zeros from that copy, optionally trimming excess space from the result's and arrays when is .

For an in-place version and algorithmic information, see .

Example

|

– Method.

|

Removes stored numerical zeros from , optionally trimming resulting excess space from and when is .

For an out-of-place version, see . For algorithmic information, see .

– Method.

|

Generates a copy of `A` and removes numerical zeros from that copy, optionally trimming excess space from the result's `row` and `col` arrays when `trim` is `true`.

For an in-place version and algorithmic information, see `sparsify!`.

Example

|

– `perm` Function.

|

Bilaterally permute `A`, returning `(Aperm, P, Q)`. Column-permutation `P`'s length must match `A`'s column count `ncol(A)`. Row-permutation `Q`'s length must match `A`'s row count `nrow(A)`.

For expert drivers and additional information, see `perm!`.

Example

|

|

– Method.

|

Bilaterally permute `a`, storing result `(b)` in `c`. Stores intermediate result `(d)` in optional argument `out` if present. Requires that none of `a`, `b`, and, if present, `out` alias each other; to store result `b` back into `a`, use the following method lacking `out`:

|

`a`'s dimensions must match those of `(b)` (and `c`), and `c` must have enough storage to accommodate all allocated entries in `(b)` (and `c`). Column-permutation `a`'s length must match `a`'s column count `(n)`. Row-permutation `a`'s length must match `a`'s row count `(m)`.

`a`'s dimensions must match those of `(b)` (and `c`), and `c` must have enough storage to accommodate all allocated entries in `(b)` (and `c`).

For additional (algorithmic) information, and for versions of these methods that forgo argument checking, see (unexported) parent methods `perm` and `perm2`.

See also: `perm`.

Chapter 51

Tareas y Computación Paralela

51.1 Tareas

– Type.

|

Create a (i.e. coroutine) to execute the given function (which must be callable with no arguments). The task exits when this function returns.

Example

|

In this example, is a runnable that hasn't started yet.

– Function.

|

Get the currently running .

– Function.

|

Determine whether a task has exited.

|

|

– Function.

|

Determine whether a task has started executing.

|

– Function.

|

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.

|

A fast, unfair-scheduling version of which immediately yields to before calling the scheduler.

– Function.

|

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, is returned from the task's last call to . This is a low-level call that only switches tasks, not considering states or scheduling in any way. Its use is discouraged.

– Method.

|

Look up the value of a key in the current task's task-local storage.

– Method.

|

Assign a value to a key in the current task's task-local storage.

– Method.

|

Call the function with a modified task-local storage, in which is assigned to ; the previous value of , or lack thereof, is restored afterwards. Useful for emulating dynamic scoping.

– Type.

|

Create an edge-triggered event source that tasks can wait for. Tasks that call on a are suspended and queued. Tasks are woken up when is later called on the . Edge triggering means that only tasks waiting at the time 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 type does this, and so can be used for level-triggered events.

– Function.

|

Wake up tasks waiting for a condition, passing them . If is (the default), all waiting tasks are woken, otherwise only one is. If is , 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 .

– Function.

|

Add a 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 .

If a second argument is provided, it will be passed to the task (via the return value of) when it runs again. If is , the value is raised as an exception in the woken task.

|

– Macro.

|

Wrap an expression in a `task` and add it to the local machine's scheduler queue. Similar to `task` except that an enclosing `task` does NOT wait for tasks started with an `task`.

– Macro.

|

Wrap an expression in a `task` without executing it, and return the `task`. This only creates a task, and does not run it.

|

– Function.

|

Block the current task for a specified number of seconds. The minimum sleep time is 1 millisecond or input of `0`.

– Type.

|

Constructs a `channel` with an internal buffer that can hold a maximum of `capacity` objects of type `T`. `channel` calls on a full channel block until an object is removed with `recv`.

`unbufferedChannel` constructs an unbuffered channel. `blocks` until a matching `isCalled` is called. And vice-versa.

Other constructors:

- `channelOf` : equivalent to `channel`
- `unbufferedChannelOf` : equivalent to `unbufferedChannel`

– Method.

|

Appends an item to the channel. Blocks if the channel is full.

For unbuffered channels, blocks until a is performed by a different task.

– Method.

|

Removes and returns a value from a. Blocks until data is available.

For unbuffered channels, blocks until a is performed by a different task.

– Method.

|

Determine whether a has a value stored to it. Returns immediately, does not block.

For unbuffered channels returns if there are tasks waiting on a.

– Method.

|

Waits for and gets the first available item from the channel. Does not remove the item. is unsupported on an unbuffered (0-size) channel.

– Method.

|

Closes a channel. An exception is thrown by:

- on a closed channel.
- and on an empty, closed channel.

– Method.

|

Associates the lifetime of with a task. Channel is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on.

The 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.

– Function.

Uses multiple concurrent tasks to map over a collection (or multiple equal length collections). For multiple collection arguments, is applied elementwise.

specifies the number of tasks to run concurrently. Depending on the length of the collections, if is unspecified, up to 100 tasks will be used for concurrent mapping.

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 is less than the current number of tasks.

If is specified, the collection is processed in batch mode. must then be a function that must accept a of argument tuples and must return a vector of results. The input vector will have a length of or less.

The following examples highlight execution in different tasks by returning the of the tasks in which the mapping function is executed.

First, with undefined, each element is processed in a different task.

|

With all elements are processed in 2 tasks.

|

With defined, the mapping function needs to be changed to accept an array of argument tuples and return an array of results. is used in the modified mapping function to achieve this.

|

Note

Currently, all tasks in Julia are executed in a single OS thread co-operatively. Consequently, is beneficial only when the mapping function involves any I/O - disk, network, remote worker invocation, etc.

- Function.

|

Like , but stores output in rather than returning a collection.

51.2 Soporte General a la Computación Paralela

– Function.

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package .

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable in the worker process's environment. Relevant only when using TCP/IP as transport.

Add processes on remote machines via SSH. Requires to be installed in the same location on each node, or to be available via a shared file system.

is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string or a tuple - .

is a string of the form . defaults to current user, to the standard ssh port. If is specified, other workers will connect to this worker at the specified and .

is the number of workers to be launched on the specified host. If specified as it will launch as many workers as the number of cores on the specific host.

Keyword arguments:

- : if then SSH tunneling will be used to connect to the worker from the master process. Default is .
- : specifies additional ssh options, e.g. ' .
- : specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- : specifies the working directory on the workers. Defaults to the host's current directory (as found by)
- : if then BLAS will run on multiple threads in added processes. Default is .
- : name of the executable. Defaults to or as the case may be.
- : additional flags passed to the worker processes.
- : Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
 - : All processes are connected to each other. The default.
 - : Only the driver process, i.e. 1 connects to the workers. The workers do not connect to each other.
 - : The method of the cluster manager specifies the connection topology via fields and in . A worker with a cluster manager identity will connect to all workers specified in .

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 terminates. This timeout can be controlled via environment variable . The value of on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

Equivalent to

Note that workers do not run a 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.

|

Launches workers using the in-built which only launches workers on the local host. This can be used to take advantage of multiple cores. will add 4 processes on the local machine. If is , binding is restricted to . Keyword args , , , and have the same effect as documented for .

– Function.

|

Get the number of available processes.

– Function.

|

Get the number of available worker processes. This is one less than . Equal to if .

– Method.

|

Returns a list of all process identifiers.

– Method.

|

Returns a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as are returned.

– Function.

|

Returns a list of all worker process identifiers.

– Function.

|

Removes the specified workers. Note that only process 1 can add or remove workers.

Argument specifies how long to wait for the workers to shut down: - If unspecified, will wait until all requested are removed. - An is raised if all workers cannot be terminated before the requested seconds. - With a value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled object is returned. The user should call on the task before invoking any other parallel calls.

- Function.

|

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.

|

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.

- Function.

|

Get the id of the current process.

- Function.

|

Transform collection by applying to each element using available workers and tasks.

For multiple collection arguments, apply elementwise.

Note that 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, distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify . This is equivalent to using . For example, is equivalent to

can also use a mix of processes and tasks via the argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length or less. A batch is sent as a single request to a free worker, where a local processes elements from the batch using multiple concurrent tasks.

Any error stops from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument 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:

|

Errors can also be handled by retrying failed computations. Keyword arguments `and` are passed through to `as` keyword arguments `and` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `and` are specified, the hook is called before retrying. If `and` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `on` an element a maximum of 3 times without any delay between retries.

|

Example: Retry `only` if the exception is not of type `,` with exponentially increasing delays up to 3 times. Return a `in` place for all occurrences.

|

– Type.

|

Exceptions on remote computations are captured and rethrown locally. `A` wraps the `of` the worker and a captured exception. `A` captures the remote exception and a serializable form of the call stack when the exception was raised.

– Type.

|

Create a `on` process `.` The default `is` the current process.

– Method.

|

Make a reference to a `on` process `.` The default `is` the current process.

– Method.

|

Create references to remote channels of a specific size and type. is a function that when executed on must return an implementation of an .

For example, , will return a reference to a channel of type and size 10 on .

The default is the current process.

– Function.

|

Block the current task until some event occurs, depending on the type of the argument:

- : Wait for a value to become available on the specified remote channel.
- : Wait for a value to become available for the specified future.
- : Wait for a value to be appended to the channel.
- : Wait for on a condition.
- : Wait for a process or process chain to exit. The field of a process can be used to determine success or failure.
- : Wait for a 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 for changes on a file descriptor (see for keyword arguments and return code)

If no argument is passed, the task blocks for an undefined period. A task can only be restarted by an explicit call to or .

Often is called within a loop to ensure a waited-for condition is met before proceeding.

– Method.

|

Waits and fetches a value from depending on the type of :

- : Wait for and get the value of a . The fetched value is cached locally. Further calls to on the same reference return the cached value. If the remote value is an exception, throws a which captures the remote exception and backtrace.
- : Wait for and get the value of a remote reference. Exceptions raised are same as for a .

Does not remove the item fetched.

– Method.

|

Call a function `asynchronously` on the given arguments on the specified process. Returns a `.` Keyword arguments, if any, are passed through to `.`

– Method.

|

Perform a faster `in` one message on the `specified` by worker id `.` Keyword arguments, if any, are passed through to `.`

See also `and` `.`

– Method.

|

Perform `in` one message. Keyword arguments, if any, are passed through to `.` Any remote exceptions are captured in a `and` thrown.

See also `and` `.`

– Method.

|

Executes `on` worker `asynchronously`. Unlike `,` 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 `s` to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `will` serialize the call to `,` followed by `and` `in` that order. However, it is not guaranteed that `is` executed before `on` worker 2.

Any exceptions thrown by `are` printed to `on` the remote worker.

Keyword arguments, if any, are passed through to `.`

– Method.

|

Store a set of values to the `.` If the channel is full, blocks until space is available. Returns its first argument.

– Method.

|

Store a value to a `.` `s` are write-once remote references. A `on` an already set `throws` an `.` All asynchronous remote calls return `s` and set the value to the return value of the call upon completion.

– Method.

|

Fetch value(s) from a , removing the value(s) in the processs.

– Method.

|

Determine whether a 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 since they are assigned only once.

– Method.

|

Determine whether a has a value stored to it.

If the argument is owned by a different node, this call will block to wait for the answer. It is recommended to wait for in a separate task instead or to use a local as a proxy:

|

– Type.

|

Create a WorkerPool from a vector of worker ids.

– Type.

|

An implementation of an , , (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 object. To clear the cache earlier, use .

For global variables, only the bindings are captured in a closure, not the data. blocks can be used to capture global data.

For example:

|

The above would transfer only once to each worker.

– Function.

|

containing idle - used by and (by default).

– Method.

|

Removes all cached functions from all participating workers.

– Function.

|

Returns an anonymous function that executes function on an available worker using .

– Method.

|

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

|

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

|

variant of . Waits for and takes a free worker from and performs a on it.

– Method.

|

variant of . Waits for and takes a free worker from and performs a on it.

– Function.

|

Waits until `returns` or for `seconds`, whichever is earlier. `is` polled every `seconds`.

– Macro.

|

Creates a closure around an expression and runs it on an automatically-chosen process, returning a `to` the result.

– Macro.

|

Accepts two arguments, `and` an expression. A closure is created around the expression and run asynchronously on process `.` Returns a `to` the result.

– Macro.

|

Equivalent to `.` See `and` `.`

– Macro.

|

Equivalent to `.` See `and` `.`

– Macro.

|

Like `,` wraps an expression in a `and` adds it to the local machine's scheduler queue. Additionally it adds the task to the set of items that the nearest enclosing `waits` for.

– Macro.

|

Wait until all dynamically-enclosed uses of `,` `,` and `are` complete. All exceptions thrown by enclosed async operations are collected and thrown as a `.`

– Macro.

|

A parallel for loop of the form :

```
|
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, 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 , like :

```
|
```

– Macro.

```
|
```

Execute an expression under everywhere. Equivalent to calling on all processes. Errors on any of the processes are collected into a and thrown. For example :

```
|
```

will define on all processes.

Unlike and , does not capture any local variables. Prefixing with allows us to broadcast local variables using interpolation :

```
|
```

The expression is evaluated under irrespective of where is called from. For example :

```
|
```

will result in being defined on all processes and not .

– Method.

```
|
```

Clears global bindings in modules by initializing them to . should be of type or a collection of s . and identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under are cleared.

An exception is raised if a global constant is requested to be cleared.

– Function.

|

s and s are identified by fields:

- - refers to the node where the underlying object/storage referred to by the reference actually exists.
- - 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 from the master process would result in a value of 2 and a value of 1.
- is unique across all references created from the worker specified by .

Taken together, and uniquely identify a reference across all workers.

is a low-level API which returns a object that wraps and values of a remote reference.

– Function.

|

A low-level API which returns the backing for an returned by . The call is valid only on the node where the backing channel exists.

– Function.

|

A low-level API which given a connection or a , returns the of the worker it is connected to. This is useful when writing custom methods for a type, which optimizes the data written out depending on the receiving process id.

– Method.

|

Returns the cluster cookie.

– Method.

|

Sets the passed cookie as the cluster cookie, then returns it.

51.3 Arrays Compartidos

– Type.

|

Construct a of a bits type and size across the processes specified by - all of which have to be on the same host. If is specified by calling , then must match the length of .

If is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, and 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 function of the type is specified, it is called on all the participating workers.

The shared array is valid as long as a reference to the object exists on the node which created the mapping.

|

Construct a backed by the file , with element type (must be a bits type) and size , across the processes specified by - 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.,) will also change the values on disk

If is left unspecified, the shared array will be mapped across all processes on the current host, including the master. But, and 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.

must be one of , , , or , and defaults to if the file specified by already exists, or if not. If an function of the type is specified, it is called on all the participating workers. You cannot specify an function if the file is not writable.

allows you to skip the specified number of bytes at the beginning of the file.

– Method.

|

Get the vector of processes mapping the shared array.

– Function.

|

Returns the actual object backing .

– Function.

|

Returns the current worker's index in the list of workers mapping the (i.e. in the same list returned by), or 0 if the is not mapped locally.

– Function.

|

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:n`. In multi-process contexts, returns an empty range in the parent process (or any process for which `myid() > 0`).

It's worth emphasizing that `myid()` exists purely as a convenience, and you can partition work on the array among workers any way you wish. For a `distributed` array, all indexes should be equally fast for each worker process.

51.4 Multi-Threading

Este interfaz experimental soporta las capacidades multi-hilo de Julia. Los tipos y funciones descritos aquí pueden cambiar en el futuro (y probablemente lo harán).

- Function.

```
|
```

Get the ID number of the current thread of execution. The master thread has ID `0`.

- Function.

```
|
```

Get the number of threads available to the Julia process. This is the inclusive upper bound on `myid()`.

- Macro.

```
|
```

A macro to parallelize a for-loop to run with multiple threads. This spawns `n` 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.

- Type.

```
|
```

Holds a reference to an object of type `T`, ensuring that it is only accessed atomically, i.e. in a thread-safe manner.

Only certain "simple" types can be used atomically, namely the primitive integer and float-point types. These are `Int`, `Float64`, and `Float32`.

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 `atomic[]` notation:

```
|
```

Atomic operations use an prefix, such as , , etc.

– Function.

|

Atomically compare-and-set

Atomically compares the value in with . If equal, write to . Otherwise, leaves unmodified. Returns the old value in . By comparing the returned value to (via) one knows whether was modified and now holds the new value .

For further details, see LLVM's instruction.

This function can be used to implement transactional semantics. Before the transaction, one records the value in . After the transaction, the new value is stored only if has not been modified in the mean time.

|

– Function.

|

Atomically exchange the value in

Atomically exchanges the value in with . Returns the **old** value.

For further details, see LLVM's instruction.

|

– Function.

|

Atomically add to

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

|

– Function.

|

Atomically subtract from

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

|

– Function.

|

Atomically bitwise-and with

Performs atomically. Returns the **old** value.

For further details, see LLVM's instruction.

|

– Function.

|

Atomically bitwise-nand (not-and) with
Performs atomically. Returns the **old** value.
For further details, see LLVM's instruction.

|

– Function.

|

Atomically bitwise-or with
Performs atomically. Returns the **old** value.
For further details, see LLVM's instruction.

|

– Function.

|

Atomically bitwise-xor (exclusive-or) with
Performs atomically. Returns the **old** value.
For further details, see LLVM's instruction.

|

– Function.

|

Atomically store the maximum of `val` and `old` in `ptr`.

Performs `max` atomically. Returns the **old** value.

For further details, see LLVM's `max` instruction.

|

– Function.

|

Atomically store the minimum of `val` and `old` in `ptr`.

Performs `min` atomically. Returns the **old** value.

For further details, see LLVM's `min` instruction.

|

– Function.

|

Insert a sequential-consistency memory 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.

51.5 ccall using a threadpool (Experimental)

- Macro.

|

The `ccall` 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 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 `LIBUV_THREADPOOL_SIZE` environment variable and restarting the process.

Note that the called function should never call back into Julia.

51.6 Primitivas de Sincronización

- Type.

|

Abstract supertype describing types that implement the thread-safe synchronization primitives: `Lock`, `ReentrantLock`, and `ReadWriteLock`.

- Function.

|

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`.

- Function.

|

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.

- Function.

|

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`.

- Function.

|

Check whether the lock is held by any task/thread. This should not be used for synchronization (see instead).

- Type.

|

Creates a reentrant lock for synchronizing Tasks. The same task can acquire the lock as many times as required. Each must be matched with an .

This lock is NOT threadsafe. See for a threadsafe lock.

- Type.

|

These are standard system mutexes for locking critical sections of logic.

On Windows, this is a critical section object, on pthreads, this is a .

See also SpinLock for a lighter-weight lock.

- Type.

|

Creates a non-reentrant lock. Recursive use will result in a deadlock. Each must be matched with an .

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 RecursiveSpinLock for a version that permits recursion.

See also Mutex for a more efficient version on one core or if the lock may be held for a considerable length of time.

- Type.

|

Creates a reentrant lock. The same thread can acquire the lock as many times as required. Each must be matched with an .

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.

- Type.

|

Creates a counting semaphore that allows at most `acquires` to be in use at any time. Each acquire must be matched with a release.

This construct is NOT threadsafe.

– Function.

|

Wait for one of the `permits` to be available, blocking until one can be acquired.

– Function.

|

Return one permit to the pool, possibly allowing another task to acquire it and resume execution.

51.7 Interfaz de Administración de Cluster

Esta interfaz proporciona un mecanismo para lanzar y gestionar *workers* Julia sobre diferentes entornos cluster. Hay dos tipos de administradores presentes en Base: `LocalClusterManager`, para lanzar *workers* adicionales sobre el mismo host, y `RemoteClusterManager`, para lanzarlos sobre hosts remotos vía `SSH`. Para conectar y transportar mensajes entre procesos se usan los sockets TCP/IP. Es posible que los administradores de clusters proporcionen un transporte diferente.

– Function.

|

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `worker` entry to `workers` and notify `callbacks`. The function MUST exit once all workers, requested by `workers`, have been launched. `kwargs` is a dictionary of all keyword arguments `launch_worker` was called with.

– Function.

|

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate values:

- with `worker` / when a worker is added / removed from the Julia worker pool.
- with `signal` when `worker` is called. The `manager` should signal the appropriate worker with an interrupt signal.
- with `cleanup` for cleanup purposes.

– Method.

|

Implemented by cluster managers. It is called on the master process, by `addprocs`. It should cause the remote worker specified by `id` to exit. `addprocs` executes a remote `on` on `id`.

– Function.

|

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--addprocs` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `addprocs` is a `Function`.

– Method.

|

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with `id`, specified by `id` and return a pair of `IOStream` objects. Messages from `id` to current process will be read off `in`, while messages to be sent to `id` will be written to `out`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `addprocs` sets up TCP/IP socket connections in-between workers.

– Function.

|

Called by cluster managers using 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 connection to the remote worker and provide two `IOStream` objects, one for incoming messages and the other for messages addressed to the remote worker. If `id` is `nothing`, 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 `addprocs`.

Chapter 52

Álgebra Lineal

52.1 Funciones Estándar

Las funciones de álgebra lineal en Julia está ampliamente implementadas llamando a funciones de [LAPACK](#). Las factorizaciones *sparse* llaman a funciones de [SuiteSparse](#)

– Method.

|

Multiplication operator. calls this function with all arguments, i.e. .

– Method.

|

Left division operator: multiplication of by the inverse of on the left. Gives floating-point results for integer arguments.

|

– Function.

|

Dot product of two vectors consisting of `elements` of array `a` with stride `stride` and `elements` of array `b` with stride `stride`.

Example:

|

– Function.

|

For any iterable containers `a` and `b` (including arrays of any dimension) of numbers (or any element type for which `isdefined` is defined), compute the Euclidean dot product (the sum of $a_i b_i$) as if they were vectors.

Examples

|

– Function.

|

Compute the cross product of two 3-vectors.

Example

|

– Function.

|

Compute a convenient factorization of , based upon the type of the input matrix. checks to see if it is symmetric/triangular/etc. if is passed as a generic matrix. checks every element of 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: .

Properties of	type of factorization
Positive-definite	Cholesky (see)
Dense Symmetric/Hermitian	Bunch-Kaufman (see)
Sparse Symmetric/Hermitian	LDLt (see)
Triangular	Triangular
Diagonal	Diagonal
Bidiagonal	Bidiagonal
Tridiagonal	LU (see)
Symmetric real tridiagonal	LDLt (see)
General square	LU (see)
General non-square	QR (see)

If is called on a Hermitian positive-definite matrix, for instance, then will return a Cholesky factorization.

Example

|

This returns a , which can now be passed to other linear algebra functions (e.g. eigensolvers) which will use specialized methods for types.

– Type.

|

Constructs a matrix from the diagonal of .

Example

|

|

Constructs a matrix with `diag` as its diagonal.

Example

|

– Type.

|

Constructs an upper () or lower () bidiagonal matrix using the given diagonal () and off-diagonal () vectors. The result is of type `ComplexBidiagonal` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `toMatrix()` (or `toRealMatrix()` for short). `offDiagonal`'s length must be one less than the length of `diagonal`.

Example

|

|

|

Constructs an upper () or lower () bidiagonal matrix using the given diagonal () and off-diagonal () vectors. The result is of type `ComplexDoubleMatrix` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `toRegularMatrix()` (or `toRegularMatrix()` for short). `offDiagonal`'s length must be one less than the length of `diagonal`.

Example

|

|

Construct a `ComplexDoubleMatrix` matrix from the main diagonal of `diagonal` and its first super- (if `upper`) or sub-diagonal (if `lower`).

Example

– Type.

Construct a symmetric tridiagonal matrix from the diagonal and first sub/super-diagonal, respectively. The result is of type `Tridiagonal` and provides efficient specialized eigensolvers, but may be converted into a regular matrix with `toMatrix()` (or `toDense()` for short).

Example

– Type.

|

Construct a tridiagonal matrix from the first subdiagonal, diagonal, and first superdiagonal, respectively. The result is of type `Tridiagonal{S}` and provides efficient specialized linear solvers, but may be converted into a regular matrix with `toMatrix()` (or for short). The lengths of `sub` and `super` must be one less than the length of `diag`.

Example

|

returns a `Tridiagonal{S}` array based on (abstract) matrix `A`, using its first lower diagonal, main diagonal, and first upper diagonal.

Example

|

– Type.

|

Construct a view of the upper (if) or lower (if) triangle of the matrix .

Example

|

Note that will not be equal to unless is itself symmetric (e.g. if).

– Type.

|

Construct a view of the upper (if) or lower (if) triangle of the matrix .

Example

|

|

Note that $\text{conj}(A)$ will not be equal to A unless A is itself Hermitian (e.g. if A is real).

– Type.

|

Construct a `view` of the the matrix `A`.

Example

|

– Type.

|

Construct an `view` of the the matrix `A`.

Example

|

– Function.

|

Compute the LU factorization of A , such that $A = LU$. By default, pivoting is used. This can be overridden by passing `usePivoting` for the second argument.

See also `lu`.

Example

```
|
```

– Function.

```
|
```

Compute the LU factorization of a sparse matrix A .

For sparse A with real or complex element type, the return type of `lu` is (L, U, P) , with L = or U = respectively and P is an integer type (or int).

The individual components of the factorization can be accessed by indexing:

Component	Description
<code>L</code>	(lower triangular) part of
<code>U</code>	(upper triangular) part of
<code>P</code>	right permutation
<code>L</code>	left permutation
<code>U</code>	of scaling factors
<code>P</code>	components

The relation between L and U is

further supports the following functions:

-
-
-

Note

uses the UMFPACK library that is part of SuiteSparse. As this library only supports sparse matrices with 2^{31} elements, converts A into a copy that is of type `int` or `long` as appropriate.

```
|
```

Compute the LU factorization of `A`.

In most cases, if `A` is a subtype of `Matrix` with an element type supporting `+`, `*`, and `inv`, the return type is `Matrix`. If pivoting is chosen (default) the element type should also support `max` and `min`.

The individual components of the factorization can be accessed by indexing:

Component	Description
<code>L</code>	(lower triangular) part of <code>A</code>
<code>U</code>	(upper triangular) part of <code>A</code>
<code>P</code>	(right) permutation
<code>Q</code>	(right) permutation

The relationship between `L` and `U` is

`LU` further supports the following functions:

Supported function		
<code>LU.L</code>		
<code>LU.U</code>		
<code>LU.P</code>		
<code>LU.Q</code>		
<code>LU.LU</code>		
<code>LU.LUinv</code>		
<code>LU.LUinvT</code>		
<code>LU.LUinvTinv</code>		

Example

|

– Function.

|

`LU.LU` is the same as `LU`, but saves space by overwriting the input `A`, instead of creating a copy. An `IllegalArgumentException` is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

– Function.

Compute the Cholesky factorization of a positive definite matrix and return the matrix such that .

Example

Compute the square root of a non-negative number .

Example

– Function.

Compute the Cholesky factorization of a sparse positive definite matrix . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. is most frequently used to solve systems of equations with , but also the methods , , and are defined for . You can also extract individual factors from , using . However, since pivoting is on by default, the factorization is internally represented as with a permutation matrix ; using just without accounting for will give incorrect answers. To include the effects of permutation, it's typically preferable to extract "combined" factors like (the equivalent of) and (the equivalent of).

Setting the optional keyword argument computes the factorization of instead of . If the argument is nonempty, it should be a permutation of 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 or as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the module.

|

Compute the Cholesky factorization of a dense symmetric positive definite matrix and return a factorization. The matrix can either be a or or a *perfectly* symmetric or Hermitian . In the latter case, the optional argument may be for using the lower part or for the upper part of . The default is to use . The triangular Cholesky factor can be obtained from the factorization with: and . The following functions are available for objects: , , , and . A exception is thrown in case the matrix is not positive definite.

Example

|

|

Compute the pivoted Cholesky factorization of a dense symmetric positive semi-definite matrix and return a factorization. The matrix can either be a or or a *perfectly* symmetric or Hermitian . In the latter case, the optional argument may be for using the lower part or for the upper part of . The default is to use . The triangular Cholesky factor can be obtained from the factorization with: and . The following functions are available for objects: , , , , and . The argument determines the tolerance for determining the rank. For negative values, the tolerance is the machine precision.

– Function.

|

Compute the Cholesky (LL') factorization of , reusing the symbolic factorization . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian.

See also .

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 `double` or `ComplexDouble` as appropriate.

|

The same as `cholmod`, but saves space by overwriting the input `A`, instead of creating a copy. An `Exception` is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

Example

|

|

The same as `cholmod`, but saves space by overwriting the input `A`, instead of creating a copy. An `Exception` is thrown if the factorization produces a number not representable by the element type of `A`, e.g. for integer types.

– Function.

|

Update a Cholesky factorization `U` with the vector `x`. If `U` is `double` then `U` but the computation of `U` only uses `n` operations.

– Function.

|

Downdate a Cholesky factorization `U` with the vector `x`. If `U` is `double` then `U` but the computation of `U` only uses `n` operations.

– Function.

|

Update a Cholesky factorization `U` with the vector `x`. If `U` is `double` then `U` but the computation of `U` only uses `n` operations. The input factorization `U` is updated in place such that on exit `U`. The vector `x` is destroyed during the computation.

– Function.

|

Downdate a Cholesky factorization with the vector . If then but the computation of only uses operations. The input factorization is updated in place such that on exit . The vector is destroyed during the computation.

– Function.

|

Compute the LDL' factorization of a sparse matrix . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian. A fill-reducing permutation is used. is most frequently used to solve systems of equations with . The returned factorization object also supports the methods , , and . You can extract individual factors from using . However, since pivoting is on by default, the factorization is internally represented as with a permutation matrix ; using just without accounting for will give incorrect answers. To include the effects of permutation, it is typically preferable to extract "combined" factors like (the equivalent of) and (the equivalent of). The complete list of supported factors is .

Setting the optional keyword argument computes the factorization of instead of . If the argument is nonempty, it should be a permutation of 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 or as appropriate.

Many other functions from CHOLMOD are wrapped but not exported from the module.

|

Compute an factorization of a real symmetric tridiagonal matrix such that where is a unit lower triangular matrix and is a vector. The main use of an factorization is to solve the linear system of equations with .

– Function.

|

Compute the LDL' factorization of , reusing the symbolic factorization . must be a or a / view of a . Note that even if doesn't have the type tag, it must still be symmetric or Hermitian.

See also .

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 or as appropriate.

|

Same as , but saves space by overwriting the input , instead of creating a copy.

– Function.

|

Compute the (pivoted) QR factorization of `A` such that either `Q` or `R`. Also see `qr`. The default is to compute a thin factorization. Note that `A` is not extended with zeros when the full `Q` is requested.

|

Computes the polar decomposition of a vector. Returns `u`, a unit vector in the direction of `x`, and `r`, the norm of `x`.

See also `norm`, `dot`, and `abs`.

Example

|

– Function.

|

Computes the polar decomposition of a vector. Instead of returning a new vector as `u`, this function mutates the input vector `x` in place. Returns `u`, a unit vector in the direction of `x` (this is a mutation of `x`), and `r`, the norm of `x`.

See also `norm`, `dot`, and `abs`.

– Function.

|

Compute the `qr` factorization of a sparse matrix `A`. A fill-reducing permutation is used. The main application of this type is to solve least squares problems with `lsqr`. The function calls the C library SPQR and a few additional functions from the library are wrapped but not exported.

|

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 `qr` stores the factorization in a packed format:

- if then is a object,
- otherwise if the element type of is a BLAS type (, , or), then is a object,
- otherwise is a object.

The individual components of the factorization can be accessed by indexing with a symbol:

- : the orthogonal/unitary matrix
- : the upper triangular matrix
- : the permutation vector of the pivot (only)
- : the permutation matrix of the pivot (only)

The following functions are available for the objects: , , and . When 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 thin or full is allowed, i.e. both and are supported. A matrix can be converted into a regular matrix with which has a named argument .

Example

|

Note

returns multiple types because LAPACK uses several representations that minimize the memory storage requirements of products of Householder elementary reflectors, so that the and matrices can be stored compactly rather as two separate dense matrices.

– Function.

|

is the same as when is a subtype of , but saves space by overwriting the input , instead of creating a copy. An exception is thrown if the factorization produces a number not representable by the element type of , e.g. for integer types.

– Type.

|

A QR matrix factorization stored in a packed format, typically obtained from `qr`. 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 τ_i where:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has two fields:

- `R` is an $m \times n$ matrix.
 - The upper triangular part contains the elements of R , that is `R[1:n, 1:n]` for a `qr` object.
 - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the i th column of the matrix.
- `tau` is a vector of length `n` containing the coefficients τ_i .

– Type.

|

A QR matrix factorization stored in a compact blocked format, typically obtained from `qr`. 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* format¹, as a lower trapezoidal matrix V and an upper triangular matrix T where

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T) = I - VTV^T$$

such that v_i is the i th column of V , and τ_i is the i th diagonal element of T .

The object has two fields:

- `R`, as in the `qr` type, is an $m \times n$ matrix.
 - The upper triangular part contains the elements of R , that is `R[1:n, 1:n]` for a `qr` object.
 - The subdiagonal part contains the reflectors v_i stored in a packed format such that `R[1:n, 1:n]`.
- `T` is a square matrix with `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 ².

– Type.

|

A QR matrix factorization with column pivoting in a packed format, typically obtained from `qr`. 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:

$$Q = \prod_{i=1}^{\min(m,n)} (I - \tau_i v_i v_i^T).$$

The object has three fields:

- `u` is an $m \times n$ matrix.
 - The upper triangular part contains the elements of R , that is `u[i,j]` for a `qr` object.
 - The subdiagonal part contains the reflectors v_i stored in a packed format where v_i is the i th column of the matrix.
- `tau` is a vector of length `n` containing the coefficients τ_i .
- `p` is an integer vector of length `n` corresponding to the permutation P .

– Function.

|

Compute the LQ factorization of `A`, using the input matrix as a workspace. See also `qr`.

– Function.

|

Compute the LQ factorization of `A`. See also `qr`.

²C 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

¹R 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

– Function.

```
|
```

Perform an LQ factorization of `x` such that `L`. The default is to compute a thin factorization. The LQ factorization is the QR factorization of `x`. `isnot` is not extended with zeros if the full `is` is requested.

– Function.

```
|
```

Compute the Bunch-Kaufman³ factorization of a symmetric or Hermitian matrix `x` and return a `Dict{<T, T>}` object. `isupper` indicates which triangle of matrix `x` to reference. If `isupper` is `true`, `x` is assumed to be symmetric. If `isupper` is `false`, `x` is assumed to be Hermitian. If `isupper` is `true`, rook pivoting is used. If `isupper` is `false`, rook pivoting is not used. The following functions are available for `Dict{<T, T>}` objects: `get`, `set`, `keys`, `values`, `pairs`.

– Function.

```
|
```

`inplace!` is the same as `inplace`, but saves space by overwriting the input `x`, instead of creating a copy.

– Function.

```
|
```

Computes eigenvalues (`eigenvalues`) and eigenvectors (`eigenvectors`) of `x`. See `eigenvalues` for details on the `isupper`, `islower`, and `isreal` arguments (for `isupper`, `islower`, and `isreal` matrices) and the `isupper` and `islower` keyword arguments. The eigenvectors are returned columnwise.

Example

```
|
```

`eigenvalues` is a wrapper around `eigenvalues`, extracting all parts of the factorization to a tuple; where possible, using `eigenvalues` is recommended.

```
|
```

Computes generalized eigenvalues (`eigenvalues`) and vectors (`eigenvectors`) of `x` with respect to `y`.

`generalized_eigenvalues` is a wrapper around `generalized_eigenvalues`, extracting all parts of the factorization to a tuple; where possible, using `generalized_eigenvalues` is recommended.

Example

³J 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](#).

|

– Function.

|

Returns the eigenvalues of .

For general non-symmetric matrices it is possible to specify how the matrix is balanced before the eigenvalue calculation. The option `perm` permutes the matrix to become closer to upper triangular, and `scale` scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is `True` for both options.

|

Computes the generalized eigenvalues of `A` and `B`.

Example

|

|

Returns the eigenvalues of `A`. It is possible to calculate only a subset of the eigenvalues by specifying a `subset` covering indices of the sorted eigenvalues, e.g. the 2nd to 8th eigenvalues.

|

|

|

Returns the eigenvalues of . It is possible to calculate only a subset of the eigenvalues by specifying a pair and for the lower and upper boundaries of the eigenvalues.

|

– Function.

|

Same as , but saves space by overwriting the input , instead of creating a copy. The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm.

|

Same as , but saves space by overwriting the input (and), instead of creating copies.

|

Same as , but saves space by overwriting the input , instead of creating a copy. is a range of eigenvalue *indices* to search for - for instance, the 2nd to 8th eigenvalues.

|

Same as , but saves space by overwriting the input , instead of creating a copy. is the lower bound of the interval to search for eigenvalues, and is the upper bound.

– Function.

|

Returns the largest eigenvalue of . The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of are complex, this method will fail, since complex numbers cannot be sorted.

Example

|

– Function.

|

Returns the smallest eigenvalue of . The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. Note that if the eigenvalues of are complex, this method will fail, since complex numbers cannot be sorted.

Example

|

|

– Function.

|

Returns a matrix whose columns are the eigenvectors of . (The i th eigenvector can be obtained from the slice .)

If the optional vector of eigenvalues is specified, returns the specific corresponding eigenvectors.

Example

|

|

Returns a matrix whose columns are the eigenvectors of . (The i th eigenvector can be obtained from the slice .)
The `and` keywords are the same as for .

Example

|

|

|

Returns a matrix whose columns are the generalized eigenvectors of and . (The i th eigenvector can be obtained from the slice .)

Example

|

– Function.

|

Computes the eigenvalue decomposition of , returning an factorization object which contains the eigenvalues in and the eigenvectors in the columns of the matrix . (The i th eigenvector can be obtained from the slice .)

The following functions are available for objects: , , and .

For general nonsymmetric matrices it is possible to specify how the matrix is balanced before the eigenvector calculation. The option permutes the matrix to become closer to upper triangular, and scales the matrix by its diagonal elements to make rows and columns more equal in norm. The default is for both options.

Example

|

|

Computes the generalized eigenvalue decomposition of A and B , returning a factorization object which contains the generalized eigenvalues in λ and the generalized eigenvectors in the columns of the matrix V . (The i th generalized eigenvector can be obtained from the slice $V[:, i]$.)

|

Computes the eigenvalue decomposition of A , returning an object which contains the eigenvalues in λ and the eigenvectors in the columns of the matrix V . (The i th eigenvector can be obtained from the slice $V[:, i]$.)

The following functions are available for objects: `sort`, `sort_eigenvalues`, and `sort_eigenvectors`.

The `indices` specifies indices of the sorted eigenvalues to search for.

Note

If k is not n , where n is the dimension of A , then the returned factorization will be a *truncated* factorization.

|

Computes the eigenvalue decomposition of A , returning an object which contains the eigenvalues in λ and the eigenvectors in the columns of the matrix V . (The i th eigenvector can be obtained from the slice $V[:, i]$.)

The following functions are available for objects: `sort`, `sort_eigenvalues`, and `sort_eigenvectors`.

`lower` is the lower bound of the window of eigenvalues to search for, and `upper` is the upper bound.

Note

If `indices` does not contain all eigenvalues of A , then the returned factorization will be a *truncated* factorization.

– Function.

|

Same as `sort_eigenvalues`, but saves space by overwriting the input `A` (and `B`), instead of creating a copy.

– Function.

|

Compute the Hessenberg decomposition of A and return a object. If U is the factorization object, the unitary matrix can be accessed with `U.U` and the Hessenberg matrix with `U.H`. When `U.H` is extracted, the resulting type is the `ComplexMatrix` object, and may be converted to a regular matrix with `U.H.toRegularMatrix()` (or `U.H.toRegularMatrixShort()` for short).

Example

|

– Function.

|

is the same as , but saves space by overwriting the input , instead of creating a copy.

– Function.

|

Computes the Schur factorization of the matrix . The (quasi) triangular Schur factor can be obtained from the object with either or and the orthogonal/unitary Schur vectors can be obtained with or such that . The eigenvalues of can be obtained with .

Example

|

|

Computes the Generalized Schur (or QZ) factorization of the matrices A and B . The (quasi) triangular Schur factors can be obtained from the object `schur` with `schur.T` and `schur.BT`, the left unitary/orthogonal Schur vectors can be obtained with `schur.U` or `schur.UT` and the right unitary/orthogonal Schur vectors can be obtained with `schur.V` or `schur.VT` such that $U^H A V = T$ and $U^H B V = BT$. The generalized eigenvalues of A and B can be obtained with `schur.eigenvals`.

– Function.

|

Same as `schur` but uses the input argument as workspace.

|

Same as `schur` but uses the input matrices A and B as workspace.

– Function.

|

Computes the Schur factorization of the matrix A . The methods return the (quasi) triangular Schur factor `schur.T` and the orthogonal/unitary Schur vectors `schur.U` such that $U^H A U = T$. The eigenvalues of A are returned in the vector `schur.eigenvals`.

See [Schur](#).

Example

|

|

See [Schur](#).

– Function.

|

Reorders the Schur factorization of a matrix according to the logical array returning the reordered factorization object. The selected eigenvalues appear in the leading diagonal of and the corresponding leading columns of 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 .

|

Reorders the Schur factorization of a real matrix according to the logical array returning the reordered matrices and as well as the vector of eigenvalues . The selected eigenvalues appear in the leading diagonal of and the corresponding leading columns of 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 .

|

Reorders the Generalized Schur factorization of a matrix pair according to the logical array and returns a GeneralizedSchur object . The selected eigenvalues appear in the leading diagonal of both and , and the left and right orthogonal/unitary Schur vectors are also reordered such that still holds and the generalized eigenvalues of and can still be obtained with .

|

Reorders the Generalized Schur factorization of a matrix pair according to the logical array and returns the matrices , , and vectors and . The selected eigenvalues appear in the leading diagonal of both and , and the left and right unitary/orthogonal Schur vectors are also reordered such that still holds and the generalized eigenvalues of and can still be obtained with .

– Function.

|

Same as but overwrites the factorization .

|

Same as but overwrites the input arguments.

|

Same as but overwrites the factorization .

|

Same as but overwrites the factorization the input arguments.

– Function.

Compute the singular value decomposition (SVD) of A and return an `svd` object.

U , Σ , and V can be obtained from the factorization with `U`, `S`, and `V`, such that $A = U \Sigma V^T$. The algorithm produces Σ and hence U is more efficient to extract than V . The singular values in Σ are sorted in descending order.

If `full = True` (default), a thin SVD is returned. For a $M \times N$ matrix A , Σ is $M \times M$ for a full SVD and $M \times \min(M, N)$ for a thin SVD.

Example

Compute the generalized SVD of A and B , returning a factorization object `gsvd`, such that $A = U \Sigma V^T$ and $B = P \Lambda Q^T$.

For an M -by- N matrix A and P -by- N matrix B ,

- U is a M -by- M orthogonal matrix,
- P is a P -by- P orthogonal matrix,
- V is a N -by- N orthogonal matrix,
- Σ is a $(K+L)$ -by- N matrix whose rightmost $(K+L)$ -by- $(K+L)$ block is nonsingular upper block triangular,
- Λ is a M -by- $(K+L)$ diagonal matrix with 1s in the first K entries,
- Q is a P -by- $(K+L)$ matrix whose top right L -by- L block is diagonal,

`rank` is the effective numerical rank of the matrix A .

The entries of `U` and `P` 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).

– Function.

`svd` is the same as `svd`, but saves space by overwriting the input `A`, instead of creating a copy.

|

is the same as , but modifies the arguments and in-place, instead of making copies.

– Function.

|

Computes the SVD of , returning , vector , and such that . The singular values in are sorted in descending order.

If (default), a thin SVD is returned. For a $M \times N$ matrix , is $M \times M$ for a full SVD () and $M \times \min(M, N)$ for a thin SVD.

is a wrapper around , extracting all parts of the factorization to a tuple. Direct use of is therefore more efficient.

Example

|

|

Wrapper around extracting all parts of the factorization to a tuple. Direct use of is therefore generally more efficient. The function returns the generalized SVD of and , returning , , , , and such that and .

– Function.

|

Returns the singular values of in descending order.

Example

|

|

|

Return the generalized singular values from the generalized singular value decomposition of A and B . See also `svdvals`.

– Type.

|

A Givens rotation linear operator. The fields `cos` and `sin` 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 `det` and can therefore be multiplied with matrices of arbitrary size as long as `m` for `G * A` or `n` for `A * G'`.

See also:

– Function.

|

Computes the Givens rotation G and scalar c such that for any vector v where

|

the result of the multiplication

|

has the property that

|

See also:

|

Computes the Givens rotation G and scalar c such that the result of the multiplication

|

has the property that

|

See also:

|

Computes the Givens rotation and scalar such that the result of the multiplication

|

has the property that

|

See also:

– Function.

|

Upper triangle of a matrix.

Example

|

|

Returns the upper triangle of starting from the th superdiagonal.

Example

|

|

– Function.

|

Upper triangle of a matrix, overwriting in the process. See also .

|

Returns the upper triangle of starting from the th superdiagonal, overwriting in the process.

Example

|

– Function.

|

Lower triangle of a matrix.

Example

|

|

|

Returns the lower triangle of `a` starting from the `k`th superdiagonal.

Example

|

– Function.

|

Lower triangle of a matrix, overwriting `a` in the process. See also `triu`.

|

Returns the lower triangle of `a` starting from the `k`th superdiagonal, overwriting `a` in the process.

Example

|

|

– Function.

|

A giving the indices of the i th diagonal of the matrix .

Example

|

– Function.

|

The i th diagonal of a matrix, as a vector. Use to construct a diagonal matrix.

Example

|

– Function.

|

Construct a matrix by placing \mathbf{v} on the k th diagonal.

Example

```
|
```

– Function.

```
|
```

Scale an array \mathbf{a} by a scalar α overwriting \mathbf{a} in-place.

If \mathbf{A} is a matrix and \mathbf{v} is a vector, then `scalb` scales each column of \mathbf{A} by \mathbf{v} (similar to `scalb`), while `scalb` scales each row of \mathbf{A} by \mathbf{v} (similar to `scalb`), again operating in-place on \mathbf{A} . An exception is thrown if the scaling produces a number not representable by the element type of \mathbf{A} , e.g. for integer types.

Example

```
|
```

– Function.

```
|
```

Compute the rank of a matrix by counting how many singular values of \mathbf{A} have magnitude greater than ϵ . By default, the value of ϵ is the largest dimension of \mathbf{A} multiplied by the ϵ of the `double` type.

Example

– Function.

Compute the p -norm of a vector or the operator norm of a matrix, defaulting to the 2-norm.

For vectors, this is equivalent to `linalg.norm` 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, `p=inf` returns the largest value in A , whereas `p=-inf` returns the smallest.

Example

For matrices, the matrix norm induced by the vector p -norm is used, where valid values of p are 1, 2, or ∞ . (Note that for sparse matrices, `linalg.norm` is currently not implemented.) Use `linalg.frobenius` to compute the Frobenius norm.

When $p=1$, the matrix norm is the maximum absolute column sum of :

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^m |a_{ij}|$$

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 $p=\infty$, the matrix norm is the maximum absolute row sum of A :

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^n |a_{ij}|$$

Example

|

|

For numbers, return $(|x|^p)^{1/p}$. This is equivalent to $|x|$.

|

For row vectors, return the q -norm of A , which is equivalent to the p -norm with value $q=1/p$. 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 A matrix.

– Function.

|

For any iterable container (including arrays of any dimension) of numbers (or any element type for which `abs` is defined), compute the p -norm (defaulting to 2) as if `arr` 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.

can assume any numeric value (even though not all values produce a mathematically valid vector norm). In particular, `l1` returns the largest value in `abs(x)`, whereas `l2` returns the smallest. If `A` is a matrix and `l2`, then this is equivalent to the Frobenius norm.

Example

```
|
```

```
|
```

For numbers, return $(|x|^p)^{1/p}$.

– Function.

```
|
```

Normalize the vector `x` in-place so that its `l2`-norm equals unity, i.e. `norm(x, 2) == 1`. See also `norm` and `norml2`.

– Function.

```
|
```

Normalize the vector `x` so that its `l2`-norm equals unity, i.e. `norm(x, 2) == 1`. See also `norm` and `norml2`.

Examples

```
|
```

– Function.

|

Condition number of the matrix , computed using the operator -norm. Valid values for are , (default), or .

– Function.

|

$$\kappa_S(M, p) = \left\| |M| |M^{-1}| \right\|_p$$

$$\kappa_S(M, x, p) = \left\| |M| |M^{-1}| |x| \right\|_p$$

Skeel condition number κ_S of the matrix , optionally with respect to the vector , as computed using the operator -norm. $|M|$ denotes the matrix of (entry wise) absolute values of M ; $|M|_{ij} = |M_{ij}|$. Valid values for are , and (default).

This quantity is also known in the literature as the Bauer condition number, relative condition number, or componentwise relative condition number.

– Function.

|

Matrix trace. Sums the diagonal elements of .

Example

|

– Function.

|

Matrix determinant.

Example

|

– Function.

|

Log of matrix determinant. Equivalent to , but may provide increased accuracy and/or speed.

Examples

|

– Function.

|

Log of absolute value of matrix determinant. Equivalent to , but may provide increased accuracy and/or speed.

– Function.

|

Matrix inverse. Computes matrix such that , where is the identity matrix. Computed by solving the left-division .

Example

|

– Function.

|

Computes the Moore-Penrose pseudoinverse.

For matrices with floating point elements, it is convenient to compute the pseudoinverse by inverting only singular values above a given threshold, .

The optimal choice of varies both with the value of and the intended application of the pseudoinverse. The default value of is , 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, is recommended.

For more information, see ⁴, ⁵, ⁶, ⁷.

Example

|

– Function.

|

Basis for nullspace of .

Example

|

⁴Issue 8859, "Fix least squares", <https://github.com/JuliaLang/julia/pull/8859>

⁵Åke 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)

⁶G. 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)

⁷Konstantinos 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)

– Function.

|

Construct a matrix by repeating the given matrix (or vector) times in dimension 1 and times in dimension 2.

Examples

|

– Function.

|

Construct an array by repeating the entries of . The i -th element of specifies the number of times that the individual entries of the i -th dimension of should be repeated. The i -th element of specifies the number of times that a slice along the i -th dimension of should be repeated. If or are omitted, no repetition is performed.

Examples

|

|

– Function.

|

Kronecker tensor product of two vectors or two matrices.

Example

|

– Function.

|

Concatenate matrices block-diagonally. Currently only implemented for sparse matrices.

Example

|

– Function.

|

Perform simple linear regression using Ordinary Least Squares. Returns β_0 and β_1 such that $\beta_0 + \beta_1 x$ is the closest straight line to the given points (x_i, y_i) , i.e., such that the squared error between $\beta_0 + \beta_1 x_i$ and y_i is minimized.

Examples:

|

See also:

, , ,

– Function.

|

Compute the matrix exponential of , defined by

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

For symmetric or Hermitian , an eigendecomposition () is used, otherwise the scaling and squaring algorithm (see ⁸) is chosen.

Example

|

– Function.

|

If has no negative real eigenvalue, compute the principal matrix logarithm of , i.e. the unique matrix X such that $e^X = A$ and $-\pi < \text{Im}(\lambda) < \pi$ for all the eigenvalues λ of X . If has nonpositive eigenvalues, a nonprincipal matrix function is returned whenever possible.

If is symmetric or Hermitian, its eigendecomposition () is used, if is triangular an improved version of the inverse scaling and squaring method is employed (see ⁹ and ¹⁰). For general matrices, the complex Schur form () is computed and the triangular algorithm is used on the triangular factor.

⁸Nicholas 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

⁹Awad 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

¹⁰Awad H. Al-Mohy, Nicholas J. Higham and Samuel D. Relton, "Computing the Fréchet derivative of the matrix logarithm and estimating the condition number", SIAM Journal on Scientific Computing, 35(4), 2013, C394-C410. doi:10.1137/120885991

Example

|

– Function.

|

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 (`eig`) is used to compute the square root. Otherwise, the square root is determined by means of the Björck-Hammarling method ¹¹, which computes the complex Schur form (`schur`) and then the complex square root of the triangular factor.

Example

|

– Function.

|

Computes the solution X to the continuous Lyapunov equation $A^T X + X A = -Q$, where no eigenvalue of A has a zero real part and no two eigenvalues are negative complex conjugates of each other.

– Function.

|

Computes the solution X to the Sylvester equation $A X + X B = C$, where A , B , and C have compatible dimensions and A and B have no eigenvalues with equal real part.

¹¹ Åke 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

– Function.

|

Test whether a matrix is symmetric.

Examples

|

– Function.

|

Test whether a matrix is positive definite.

Example

|

– Function.

|

Test whether a matrix is positive definite, overwriting in the process.

Example

|

|

– Function.

|

Test whether a matrix is lower triangular.

Examples

|

– Function.

|

Test whether a matrix is upper triangular.

Examples

|

– Function.

|

Test whether a matrix is diagonal.

Examples

|

– Function.

|

Test whether a matrix is Hermitian.

Examples

|

– Type.

|

A lazy-view wrapper of an `array`, which turns a length-`n` vector into a `(n, 1)` 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 `array_view` function or `array_view` operator (or related `array_view` operator).

By convention, a vector can be multiplied by a matrix on its left (`matrix * vector`) whereas a row vector can be multiplied by a matrix on its right (such that `vector * matrix`). It differs from a `matrix`-sized matrix by the facts that its transpose returns a vector and the inner product `inner_product` returns a scalar, but will otherwise behave similarly.

– Type.

|

A lazy-view wrapper of an `array`, taking the elementwise complex conjugate. This type is usually constructed (and unwrapped) via the `array_view` function (or related `array_view`), but currently this is the default behavior for `array_view` only. For other arrays, the constructor can be used directly.

Examples

|

– Function.

|

The transposition operator (`array_view::operator T()`).

Example

|

|

The transposition operator (`array_view::operator T()`).

Example

|

– Function.

|

Transpose array and store the result in the preallocated array , which should have a size corresponding to . No in-place transposition is supported and unexpected results will happen if and have overlapping memory regions.

– Function.

|

The conjugate transposition operator ().

Example

|

– Function.

|

Conjugate transpose array and store the result in the preallocated array , which should have a size corresponding to . No in-place transposition is supported and unexpected results will happen if and have overlapping memory regions.

– Method.

|

Computes eigenvalues of using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- : Number of eigenvalues
- : Number of Krylov vectors used in the computation; should satisfy for real symmetric problems and for other problems, where is the size of the input matrix . The default is . Note that these restrictions limit the input matrix to be of dimension at least 2.
- : type of eigenvalues to compute. See the note below.

	type of eigenvalues
	eigenvalues of largest magnitude (default)
	eigenvalues of smallest magnitude
	eigenvalues of largest real part
	eigenvalues of smallest real part
	eigenvalues of largest imaginary part (nonsymmetric or complex only)
	eigenvalues of smallest imaginary part (nonsymmetric or complex only)
	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric only)

- : parameter defining the relative tolerance for convergence of Ritz values (eigenvalue estimates). A Ritz value is considered converged when its associated residual is less than or equal to the product of and $max^{(2/3, ||)}$, where is LAPACK's machine epsilon. The residual associated with and its corresponding Ritz vector v is defined as the norm $||Av - v||$. The specified value of should be positive; otherwise, it is ignored and is used instead. Default: .
- : Maximum number of iterations (default = 300)
- : Specifies the level shift used in inverse iteration. If (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to using shift and invert iterations.
- : Returns the Ritz vectors (eigenvectors) if
- : starting vector from which to start the iterations

returns the requested eigenvalues in , the corresponding Ritz vectors (only if), the number of converged eigenvalues , the number of iterations and the number of matrix vector multiplications , as well as the final residual vector .

Example

Note

The and keywords interact: the description of eigenvalues searched for by do not necessarily refer to the eigenvalues of , but rather the linear operator constructed by the specification of the iteration mode implied by .

Note

Although has a default value, the best choice depends strongly on the matrix . We recommend that users _always_ specify a value for which suits their specific needs.

For details of how the errors in the computed eigenvalues are estimated, see:

	iteration mode	refers to eigenvalues of
	ordinary (forward)	A
real or complex	inverse with level shift	$(A - \sigma I)^{-1}$

- B. N. Parlett, "The Symmetric Eigenvalue Problem", SIAM: Philadelphia, 2/e (1998), Ch. 13.2, "Assessing 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/S0895479895281484

– Method.

Computes generalized eigenvalues of and using implicitly restarted Lanczos or Arnoldi iterations for real symmetric or general nonsymmetric matrices respectively.

The following keyword arguments are supported:

- : Number of eigenvalues
- : Number of Krylov vectors used in the computation; should satisfy for real symmetric problems and for other problems, where is the size of the input matrices and . The default is . Note that these restrictions limit the input matrix to be of dimension at least 2.
- : type of eigenvalues to compute. See the note below.

	type of eigenvalues
	eigenvalues of largest magnitude (default)
	eigenvalues of smallest magnitude
	eigenvalues of largest real part
	eigenvalues of smallest real part
	eigenvalues of largest imaginary part (nonsymmetric or complex only)
	eigenvalues of smallest imaginary part (nonsymmetric or complex only)
	compute half of the eigenvalues from each end of the spectrum, biased in favor of the high end. (real symmetric only)

- : relative tolerance used in the convergence criterion for eigenvalues, similar to in the 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 and the accompanying note about .
- : Maximum number of iterations (default = 300)
- : Specifies the level shift used in inverse iteration. If (default), defaults to ordinary (forward) iterations. Otherwise, find eigenvalues close to using shift and invert iterations.
- : Returns the Ritz vectors (eigenvectors) if
- : starting vector from which to start the iterations

returns the requested eigenvalues in , the corresponding Ritz vectors (only if), the number of converged eigenvalues , the number of iterations and the number of matrix vector multiplications , as well as the final residual vector .

Example

|

Note

The `method` and `kwargs` keywords interact: the description of eigenvalues searched for by `kwargs` 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 `method`.

	iteration mode	refers to the problem
	ordinary (forward)	$Av = Bv\lambda$
real or complex	inverse with level shift	$(A - \sigma B)^{-1}B = v\nu$

– Function.

|

Computes the largest singular values of `A` using implicitly restarted Lanczos iterations derived from `scipy.linalg.sgv`.

Inputs

- `A`: Linear operator whose singular values are desired. `A` may be represented as a subtype of `scipy.sparse.linalg.LinearOperator`, e.g., a sparse matrix, or any other type supporting the four methods `__matmul__`, `__rmatmul__`, `__rmul__`, and `__mul__`.
- `k`: Number of singular values. Default: 6.
- `u`: If `True`, return the left and right singular vectors `U` and `V`. If `False`, omit the singular vectors. Default: `True`.
- `tol`: tolerance, see `scipy.linalg.sgv`.
- `maxiter`: Maximum number of iterations, see `scipy.linalg.sgv`. Default: 1000.
- `maxkrylov`: Maximum size of the Krylov subspace, see `scipy.linalg.sgv` (there called `k`). Default: `min(k, maxkrylov)`.
- `u0`: Initial guess for the first left Krylov vector. It may have length `u0.shape[0]` (the first dimension of `A`), or 0.
- `v0`: Initial guess for the first right Krylov vector. It may have length `v0.shape[0]` (the second dimension of `A`), or 0.

Outputs

- `U`: An `ndarray` object containing the left singular vectors, the requested values, and the right singular vectors. If `u` is `False`, the left and right singular vectors will be empty.
- `sigma`: Number of converged singular values.
- `iterations`: Number of iterations.
- `products`: Number of matrix–vector products used.
- `residual`: Final residual vector.

Example

Implementation

is formally equivalent to calling `svdvals` to perform implicitly restarted Lanczos tridiagonalization on the Hermitian matrix $\begin{pmatrix} 0 & A' \\ A & 0 \end{pmatrix}$, whose eigenvalues are plus and minus the singular values of A .

– Function.

computes the peak flop rate of the computer by using double precision `Float64`. By default, if no arguments are specified, it multiplies a matrix of size `n`, where `n` is the number of processors. If the underlying BLAS is using multiple threads, higher flop rates are realized. The number of BLAS threads can be set with `BLAS_THREADS`.

If the keyword argument `parallel` is set to `true`, `svdvals` 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.

52.2 Operaciones matriciales de bajo nivel

Las operaciones de matrices que involucran operaciones de transposición como `tr!*` son convertidas por el analizador de Julia en llamadas a funciones especialmente nombradas como `tr!*`. Si desea sobrecargar estas operaciones para sus propios tipos, será útil conocer los nombres de estas funciones.

Además, en muchos casos, hay versiones in situ de operaciones matriciales que le permiten suministrar un vector o matriz de salida preasignada. Esto es útil cuando se optimiza código crítico para evitar la sobrecarga de las asignaciones repetidas. Estas operaciones in situ tienen el sufijo `!` a continuación (por ejemplo, `tr!*`) de acuerdo con la convención habitual de Julia.

– Function.

Compute `in-place` and store the result in `out`, returning the result. If only two arguments are passed, then `out` overwrites `in` with the result.

The argument `out` should *not* be a matrix. Rather, instead of matrices it should be a factorization object (e.g. produced by `svd` or `qr`). 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., `inplace!`), and performance-critical situations requiring `inplace!` usually also require fine-grained control over the factorization of `in`.

– Function.

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

Calculates the matrix-matrix or matrix-vector product AB and stores the result in , overwriting the existing value of . Note that must not be aliased with either or .

Example

|

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates A/B .

– Function.

|

For matrices or vectors A and B , calculates A/B .

– Function.

|

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

Similar to , but return $A \setminus B$, computing the result in-place in (or overwriting if is not supplied).

– Function.

|

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates A/B .

– Function.

|

For matrices or vectors A and B , calculates A/B .

– Function.

|

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

Similar to , but return $A \setminus B$, computing the result in-place in (or overwriting if is not supplied).

– Function.

|

For matrices or vectors A and B , calculates $A \setminus B$.

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates AB .

– Function.

|

For matrices or vectors A and B , calculates A/B .

– Function.

|

For matrices or vectors A and B , calculates A/B .

52.3 Funciones BLAS

En Julia (como en gran parte de la computación científica), las operaciones de álgebra lineal densa se basan en la [biblioteca LAPACK](#), que a su vez se construye sobre bloques de construcción básicos de álgebra lineal conocidos como [BLAS](#). Hay implementaciones altamente optimizadas de BLAS disponibles para cada arquitectura de computadora, y algunas veces en rutinas de álgebra lineal de alto rendimiento, es útil llamar directamente a las funciones de BLAS.

proporciona envoltorios para algunas de las funciones de BLAS. Esas funciones de BLAS que sobrescriben una de las matrices de entrada tienen nombres que terminan en . Normalmente, una función BLAS tiene cuatro métodos definidos, para las arrays , , y .

BLAS Character Arguments

Muchas funciones BLAS aceptan argumentos que determinan si se debe transponer un argumento (), qué triángulo de una matriz referenciar (o), si se puede suponer que la diagonal de una matriz triangular está formada por unos () o a qué lado de una multiplicación de matrices pertenece el argumento de entrada (). Las posibilidades son:

Orden de Multiplicación

	Meaning
	El argumento va al lado <i>izquierdo</i> de una operación matriz-matriz.
	El argumento va al lado <i>derecho</i> de una operación matriz-matriz.

Referencia sobre el Triángulo

/	Meaning
	Sólo se usará el triángulo <i>superior</i> de la matriz.
	Sólo se usará el triángulo <i>inferior</i> de la matriz.

Operación de Transposición

/	Meaning
	La matriz de entrada no es transpuesta ni conjugada.
	La matriz de entrada será transpuesta.
	La matriz de entrada será conjugada y transpuesta.

Unidades en la Diagonal

/	Meaning
	Los valores diagonales de la matriz serán leídos.
	Se supone que los elementos de la diagonal de la matriz son todos unos.

– Function.

|

Dot function for two complex vectors consisting of elements of array with stride and elements of array with stride .

Example:

|

– Function.

|

Dot function for two complex vectors, consisting of elements of array with stride and elements of array with stride , conjugating the first vector.

Example:

|

– Function.

|

Copy elements of array with stride to array with stride . Returns .

– Function.

|

2-norm of a vector consisting of elements of array with stride .

Example:

|

– Function.

|

Sum of the absolute values of the first elements of array with stride .

Example:

|

– Function.

|

Overwrite with , where is a scalar. Returns .

Example:

|

– Function.

|

Overwrite with for the first elements of array with stride . Returns .

– Function.

|

Returns scaled by for the first elements of array with stride .

– Function.

|

Rank-1 update of the matrix with vectors and as .

– Function.

|

Rank-1 update of the symmetric matrix with vector as . controls which triangle of is updated. Returns .

– Function.

|

Rank-k update of the symmetric matrix as or according to . Only the triangle of is used. Returns .

– Function.

|

Returns either the upper triangle or the lower triangle of , according to , of or , according to .

– Function.

|

Methods for complex arrays only. Rank-1 update of the Hermitian matrix with vector as . controls which triangle of is updated. Returns .

– Function.

|

Methods for complex arrays only. Rank-k update of the Hermitian matrix A as $A = A + \alpha B B^H$ or $A = A - \alpha B B^H$ according to $UPLO$. Only the $UPLO$ triangle of A is updated. Returns A .

– Function.

|

Methods for complex arrays only. Returns the $UPLO$ triangle of A or B , according to $UPLO$.

– Function.

|

Update vector x as $x = x + \alpha A x$ or $x = x - \alpha A x$ according to $UPLO$. The matrix A is a general band matrix of dimension n by n with nb sub-diagonals and nb super-diagonals. α and x are scalars. Returns the updated x .

– Function.

|

Returns α or x according to $UPLO$. The matrix A is a general band matrix of dimension n by n with nb sub-diagonals and nb super-diagonals, and α is a scalar.

– Function.

|

Update vector x as $x = x + \alpha A x$ where A is a symmetric band matrix of order n with nb 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 x .

– Method.

|

Returns α where A is a symmetric band matrix of order n with nb super-diagonals stored in the argument A . Only the $UPLO$ triangle of A is used.

– Method.

|

Returns α where A is a symmetric band matrix of order n with nb super-diagonals stored in the argument A . Only the $UPLO$ triangle of A is used.

– Function.

|

Update a as a or the other three variants according to b and c . Returns the updated a .

– Method.

|

Returns a or the other three variants according to b and c .

– Method.

|

Returns a or the other three variants according to b and c .

– Function.

|

Update the vector a as a or b according to c . α and β are scalars. Returns the updated a .

– Method.

|

Returns a or b according to c . α is a scalar.

– Method.

|

Returns a or b according to c .

– Function.

|

Update a as a or b according to c . A is assumed to be symmetric. Only the n triangle of A is used. Returns the updated a .

– Method.

|

Returns a or b according to c . A is assumed to be symmetric. Only the n triangle of A is used.

– Method.

|

Returns or according to . is assumed to be symmetric. Only the triangle of is used.

– Function.

|

Update the vector as . is assumed to be symmetric. Only the triangle of is used. and are scalars. Returns the updated .

– Method.

|

Returns . is assumed to be symmetric. Only the triangle of is used. is a scalar.

– Method.

|

Returns . is assumed to be symmetric. Only the triangle of is used.

– Function.

|

Update as or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

|

Returns or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

|

Overwrite with the solution to or one of the other three variants determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

|

Returns the solution to or one of the other three variants determined by determined by and . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

|

Returns , where is determined by . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones. The multiplication occurs in-place on .

– Function.

|

Returns , where is determined by . Only the triangle of is used. determines if the diagonal values are read or are assumed to be all ones.

– Function.

|

Overwrite with the solution to or one of the other two variants determined by and . determines if the diagonal values are read or are assumed to be all ones. Returns the updated .

– Function.

|

Returns the solution to or one of the other two variants determined by and . determines if the diagonal values are read or are assumed to be all ones.

– Function.

|

Set the number of threads the BLAS library should use.

– Constant.

|

An object of type , representing an identity matrix of any size.

Example

52.4 LAPACK Functions

proporciona *wrappers* para algunas de las funciones LAPACK para álgebra lineal. Las funciones que sobrescriben una de las matrices de entrada tienen nombres que terminan en `_w`.

Por lo general, una función tiene 4 métodos definidos, uno para las arrays , , y .

Tenga en cuenta que la API LAPACK proporcionada por Julia puede y va a cambiar en el futuro. Dado que esta API no está orientada al usuario, no existe el compromiso de admitir/desaprobar este conjunto específico de funciones en futuras versiones.

- Function.

1

Compute the LU factorization of a banded matrix. `is` is the first subdiagonal containing a nonzero band, `isuper` is the last superdiagonal containing one, and `idim` is the first dimension of the matrix. Returns the LU factorization in-place and `ipiv`, the vector of pivots used.

- Function.

1

Solve the equation $Ax = b$. `perm` determines the orientation of A . It may be (no transpose), (transpose), or (conjugate transpose). `l` is the first subdiagonal containing a nonzero band, `u` is the last superdiagonal containing one, and `ldim` is the first dimension of the matrix A . `ipiv` is the vector of pivots returned from `gesv`. Returns the vector or matrix x , overwriting `b` in-place.

- Function.

1

Balance the matrix before computing its eigensystem or Schur factorization. can be one of (will not be permuted or scaled), (will only be permuted), (will only be scaled), or (will be both permuted and scaled). Modifies in-place and returns , , and . If permuting was turned on, if and or . contains information about the scaling/permutations performed.

- Function.

1

Transform the eigenvectors of a matrix balanced using to the unscaled/unpermuted eigenvectors of the original matrix. Modifies in-place. can be (left eigenvectors are transformed) or (right eigenvectors are transformed).

– Function.

|

Reduce in-place to bidiagonal form. Returns , containing the bidiagonal matrix ; , containing the diagonal elements of ; , containing the off-diagonal elements of ; , containing the elementary reflectors representing ; and , containing the elementary reflectors representing .

– Function.

|

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

|

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

|

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

|

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

|

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

|

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

|

Compute the pivoted factorization of , using BLAS level 3. is a pivoting matrix, represented by . stores the elementary reflectors. must have length greater than or equal to if is an matrix. must have length greater than or equal to the smallest dimension of .

, , and are modified in-place.

|

Compute the pivoted factorization of , using BLAS level 3. is a pivoting matrix, represented by . must have length greater than or equal to if is an matrix.

Returns and , modified in-place, and , which stores the elementary reflectors.

|

Compute the pivoted factorization of , using BLAS level 3.

Returns , modified in-place, , which represents the pivoting matrix , and , which stores the elementary reflectors.

– Function.

|

Compute the factorization of , . contains scalars which parameterize the elementary reflectors of the factorization. must have length greater than or equal to the smallest dimension of .

Returns and modified in-place.

|

Compute the factorization of , .

Returns , modified in-place, and , which contains scalars which parameterize the elementary reflectors of the factorization.

– Function.

|

Compute the blocked factorization of A . U contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of U sets the block size and it must be between 1 and n . The second dimension of U must equal the smallest dimension of A .

Returns U and A modified in-place.

|

Compute the blocked factorization of A . U sets the block size and it must be between 1 and n , the second dimension of U .

Returns U , modified in-place, and A , which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

– Function.

|

Recursively computes the blocked factorization of A . U contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization. The first dimension of U sets the block size and it must be between 1 and n . The second dimension of U must equal the smallest dimension of A .

Returns U and A modified in-place.

|

Recursively computes the blocked factorization of A .

Returns U , modified in-place, and A , which contains upper triangular block reflectors which parameterize the elementary reflectors of the factorization.

– Function.

|

Compute the pivoted factorization of A .

Returns U , modified in-place, P , the pivoting information, and an `info` code which indicates success (`0`), a singular value in U (in which case A is singular), or an error code (`< 0`).

– Function.

|

Transforms the upper trapezoidal matrix A to upper triangular form in-place. Returns U and P , the scalar parameters for the elementary reflectors of the transformation.

– Function.

|

Multiplies the matrix `by` by from the transformation supplied by `.` Depending on `or` the multiplication can be left-sided (`()`) or right-sided (`()`) and `can` be unmodified (`()`), transposed (`()`), or conjugate transposed (`()`). Returns matrix `which` is modified in-place with the result of the multiplication.

– Function.

|

Solves the linear equation `, , or` using a QR or LQ factorization. Modifies the matrix/vector `in` place with the solution. `is` overwritten with its `or` factorization. `may` be one of (no modification), (transpose), or (conjugate transpose). `searches` for the minimum norm/least squares solution. `may` be under or over determined. The solution is returned in `.`

– Function.

|

Solves the linear equation where `is` a square matrix using the factorization of `.` `is` overwritten with its factorization and `is` overwritten with the solution `.` contains the pivoting information for the factorization of `.`

– Function.

|

Solves the linear equation `, , or` for square `.` Modifies the matrix/vector `in` place with the solution. `is` the factorization from `, with` the pivoting information. `may` be one of (no modification), (transpose), or (conjugate transpose).

– Function.

|

Computes the inverse of `, using` its factorization found by `.` `is` the pivot information output and `contains` the factorization of `.` `is` overwritten with its inverse.

– Function.

|

Solves the linear equation `()`, `()`, or `()` using the factorization of `.` `may` be `, in` which case `will` be equilibrated and copied to `; , in` which case `and` from a previous factorization are inputs; or `, in` which case `will` be copied to `and` then factored. If `, may` be `, meaning` has not been equilibrated; `, meaning` was multiplied by `from` the left; `, meaning` was multiplied by `from` the right; or `, meaning` was multiplied by `from` the left and `from` the right. If `and` or the elements of `must` all be positive. If `and` or the elements of `must` all be positive.

Returns the solution `; , which` is an output if `is` not `, and` describes the equilibration that was performed; `, the` row equilibration diagonal; `, the` column equilibration diagonal; `, which` may be overwritten with its equilibrated form (if `and`) or (if `and`); `, the` reciprocal condition number of `after` equilibrating; `, the` forward error bound for each solution vector in `; , the` forward error bound for each solution vector in `; and` , the reciprocal pivot growth factor.

|

The no-equilibration, no-transpose simplification of .

– Function.

|

Computes the least norm solution of by finding the factorization of , then dividing-and-conquering the problem. is overwritten with the solution . Singular values below will be treated as zero. Returns the solution in and the effective rank of in .

– Function.

|

Computes the least norm solution of by finding the full factorization of , then dividing-and-conquering the problem. is overwritten with the solution . Singular values below will be treated as zero. Returns the solution in and the effective rank of in .

– Function.

|

Solves the equation where is subject to the equality constraint . Uses the formula to solve. Returns and the residual sum-of-squares.

– Function.

|

Finds the eigensystem of . If , the left eigenvectors of aren't computed. If , the right eigenvectors of aren't computed. If or , the corresponding eigenvectors are computed. Returns the eigenvalues in , the right eigenvectors in , and the left eigenvectors in .

– Function.

|

Finds the singular value decomposition of , , using a divide and conquer approach. If , all the columns of and the rows of are computed. If , no columns of or rows of are computed. If , is overwritten with the columns of (thin) and the rows of (thin) . If , the columns of (thin) and the rows of (thin) are computed and returned separately.

– Function.

|

Finds the singular value decomposition of A . If $computeU$, all the columns of U are computed. If $computeV$, all the rows of V are computed. If $computeS$, no columns of U are computed. If $computeUthin$, no rows of U are computed. If $computeVthin$, U is overwritten with the columns of (thin) U . If $computeSthin$, V is overwritten with the rows of (thin) V . If $computeS$, the columns of (thin) S are computed and returned separately. If the rows of (thin) S are computed and returned separately. $computeU$ and $computeV$ can't both be .

Returns U , S , and V , where S are the singular values of A .

– Function.

|

Finds the generalized singular value decomposition of A and B , and C . U has m on its diagonal and V has n on its diagonal. If $computeU$, the orthogonal/unitary matrix U is computed. If $computeV$, the orthogonal/unitary matrix V is computed. If $computeS$, the orthogonal/unitary matrix S is computed. If $computeUthin$ or $computeVthin$ is , that matrix is not computed. This function is only available in LAPACK versions prior to 3.6.0.

– Function.

|

Finds the generalized singular value decomposition of A and B , and C . U has m on its diagonal and V has n on its diagonal. If $computeU$, the orthogonal/unitary matrix U is computed. If $computeV$, the orthogonal/unitary matrix V is computed. If $computeS$, the orthogonal/unitary matrix S is computed. If $computeUthin$ or $computeVthin$ is , that matrix is not computed. This function requires LAPACK 3.6.0.

– Function.

|

Finds the eigensystem of A with matrix balancing. If $computeL$, the left eigenvectors of A aren't computed. If $computeR$, the right eigenvectors of A aren't computed. If $computeLR$ or $computeRL$, the corresponding eigenvectors are computed. If $computeLR$, no balancing is performed. If $computeLR$, A is permuted but not scaled. If $computeLR$, A is scaled but not permuted. If $computeLR$, A is permuted and scaled. If $computeLR$, no reciprocal condition numbers are computed. If $computeLR$, reciprocal condition numbers are computed for the eigenvalues only. If $computeLR$, reciprocal condition numbers are computed for the right eigenvectors only. If $computeLR$, reciprocal condition numbers are computed for the right eigenvectors and the eigenvectors. If $computeLR$, the right and left eigenvectors must be computed.

– Function.

|

Finds the generalized eigendecomposition of A and B . If $computeL$, the left eigenvectors aren't computed. If $computeR$, the right eigenvectors aren't computed. If $computeLR$ or $computeRL$, the corresponding eigenvectors are computed.

– Function.

|

Solves the equation $Ax = b$ where A is a tridiagonal matrix with a on the subdiagonal, d on the diagonal, and c on the super-diagonal.

Overwrites b with the solution and returns it.

– Function.

|

Finds the factorization of a tridiagonal matrix with a on the subdiagonal, d on the diagonal, and c on the superdiagonal.

Modifies a , d , and c in-place and returns them and the second superdiagonal e and the pivoting vector p .

– Function.

|

Solves the equation $Ax = b$, $Ax = b + c$, or $Ax = b + c + d$ using the factorization computed by `tridiagonal`. b is overwritten with the solution.

– Function.

|

Explicitly finds the matrix L of a factorization after calling `tridiagonal` on A . Uses the output of `tridiagonal`. A is overwritten by L .

– Function.

|

Explicitly finds the matrix L of a factorization after calling `tridiagonal` on A . Uses the output of `tridiagonal`. A is overwritten by L .

– Function.

|

Explicitly finds the matrix L of a factorization after calling `tridiagonal` on A . Uses the output of `tridiagonal`. A is overwritten by L .

– Function.

|

Explicitly finds the matrix L of a factorization after calling `tridiagonal` on A . Uses the output of `tridiagonal`. A is overwritten by L .

– Function.

|

Computes (\cdot) , (\cdot) , (\cdot) for or the equivalent right-sided multiplication for using from a factorization of computed using \cdot . \cdot is overwritten.

– Function.

|

Computes (\cdot) , (\cdot) , (\cdot) for or the equivalent right-sided multiplication for using from a factorization of computed using \cdot . \cdot is overwritten.

– Function.

|

Computes (\cdot) , (\cdot) , (\cdot) for or the equivalent right-sided multiplication for using from a factorization of computed using \cdot . \cdot is overwritten.

– Function.

|

Computes (\cdot) , (\cdot) , (\cdot) for or the equivalent right-sided multiplication for using from a factorization of computed using \cdot . \cdot is overwritten.

– Function.

|

Computes (\cdot) , (\cdot) , (\cdot) for or the equivalent right-sided multiplication for using from a factorization of computed using \cdot . \cdot is overwritten.

– Function.

|

Finds the solution to where \cdot is a symmetric or Hermitian positive definite matrix. If the upper Cholesky decomposition of \cdot is computed. If the lower Cholesky decomposition of \cdot is computed. \cdot is overwritten by its Cholesky decomposition. \cdot is overwritten with the solution \cdot .

– Function.

|

Computes the Cholesky (upper if \cdot , lower if \cdot) decomposition of positive-definite matrix \cdot . \cdot is overwritten and returned with an info code.

– Function.

|

Computes the inverse of positive-definite matrix `a` after calling `cholesky` to find its (upper if `u`, lower if `l`) Cholesky decomposition.

`a` is overwritten by its inverse and returned.

– Function.

|

Finds the solution to $Ax = b$ where A is a symmetric or Hermitian positive definite matrix whose Cholesky decomposition was computed by `cholesky`. If `u` the upper Cholesky decomposition of A was computed. If `l` the lower Cholesky decomposition of A was computed. `x` is overwritten with the solution.

– Function.

|

Computes the (upper if `u`, lower if `l`) pivoted Cholesky decomposition of positive-definite matrix `a` with a user-set tolerance `tol`. `a` is overwritten by its Cholesky decomposition.

Returns `p`, the pivots, the rank of `a`, and an `info` code. If `info` is 0, the factorization succeeded. If `info` is not 0, then `a` is indefinite or rank-deficient.

– Function.

|

Solves $Ax = b$ for positive-definite tridiagonal A . `d` is the diagonal of A and `e` is the off-diagonal. `x` is overwritten with the solution and returned.

– Function.

|

Computes the LDLt factorization of a positive-definite tridiagonal matrix with `d` as diagonal and `e` as off-diagonal. `d` and `e` are overwritten and returned.

– Function.

|

Solves $Ax = b$ for positive-definite tridiagonal A with diagonal `d` and off-diagonal `e` after computing A 's LDLt factorization using `ldlt`. `x` is overwritten with the solution.

– Function.

|

Finds the inverse of (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. is overwritten with its inverse.

– Function.

|

Solves (), (), or () for (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. is overwritten with the solution .

– Function.

|

Finds the reciprocal condition number of (upper if , lower if) triangular matrix . If , has non-unit diagonal elements. If , all diagonal elements of are one. If , the condition number is found in the infinity norm. If or , the condition number is found in the one norm.

– Function.

|

Finds the eigensystem of an upper triangular matrix . If , the right eigenvectors are computed. If , the left eigenvectors are computed. If , both sets are computed. If , all eigenvectors are found. If , all eigenvectors are found and backtransformed using and . If , only the eigenvectors corresponding to the values in are computed.

– Function.

|

Estimates the error in the solution to (), (), () for , or the equivalent equations a right-handed after computing using . If , is upper triangular. If , is lower triangular. If , has non-unit diagonal elements. If , all diagonal elements of are one. and are optional inputs. is the forward error and is the backward error, each component-wise.

– Function.

|

Computes the eigensystem for a symmetric tridiagonal matrix with as diagonal and as off-diagonal. If only the eigenvalues are found and returned in . If then the eigenvectors are also found and returned in .

– Function.

|

Computes the eigenvalues for a symmetric tridiagonal matrix with d as diagonal and e as off-diagonal. If all , all the eigenvalues are found. If $half$, the eigenvalues in the half-open interval $[a, b)$ are found. If $indices$, the eigenvalues with indices between low and $high$ are found. If $order$, eigvalues are ordered within a block. If $across$, they are ordered across all the blocks. tol can be set as a tolerance for convergence.

– Function.

|

Computes the eigenvalues (e) or eigenvalues and eigenvectors (v) for a symmetric tridiagonal matrix with d as diagonal and e as off-diagonal. If all , all the eigenvalues are found. If $half$, the eigenvalues in the half-open interval $[a, b)$ are found. If $indices$, the eigenvalues with indices between low and $high$ are found. The eigenvalues are returned in e and the eigenvectors in v .

– Function.

|

Computes the eigenvectors for a symmetric tridiagonal matrix with d as diagonal and e as off-diagonal. $eigen$ specifies the input eigenvalues for which to find corresponding eigenvectors. $submat$ specifies the submatrices corresponding to the eigenvalues in e . $split$ specifies the splitting points between the submatrix blocks.

– Function.

|

Converts a symmetric matrix A (which has been factorized into a triangular matrix) into two matrices L and U . If $upper$, U is upper triangular. If $lower$, it is lower triangular. p is the pivot vector from the triangular factorization. A is overwritten by L and U .

– Function.

|

Finds the solution to $Ax = b$ for symmetric matrix A . If $upper$, the upper half of A is stored. If $lower$, the lower half is stored. A is overwritten by the solution x . A is overwritten by its Bunch-Kaufman factorization. p contains pivoting information about the factorization.

– Function.

|

Computes the Bunch-Kaufman factorization of a symmetric matrix A . If $upper$, the upper half of A is stored. If $lower$, the lower half is stored.

Returns x , overwritten by the factorization, a pivot vector p , 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 $info$.

– Function.

|

Computes the inverse of a symmetric matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by its inverse.

– Function.

|

Solves the equation for a symmetric matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution .

– Function.

|

Finds the solution to for Hermitian matrix . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution . is overwritten by its Bunch-Kaufman factorization. contains pivoting information about the factorization.

– Function.

|

Computes the Bunch-Kaufman factorization of a Hermitian matrix . If , the upper half of is stored. If , the lower half is stored.

Returns , overwritten by the factorization, a pivot vector , and the error code which is a non-negative integer. If is positive the matrix is singular and the diagonal part of the factorization is exactly zero at position .

– Function.

|

Computes the inverse of a Hermitian matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by its inverse.

– Function.

|

Solves the equation for a Hermitian matrix using the results of . If , the upper half of is stored. If , the lower half is stored. is overwritten by the solution .

– Function.

|

Finds the eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix . If , the upper triangle of is used. If , the lower triangle of is used.

– Function.

|

Finds the eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix . If , the upper triangle of is used. If , the lower triangle of is used. If , all the eigenvalues are found. If , the eigenvalues in the half-open interval are found. If , the eigenvalues with indices between and are found. can be set as a tolerance for convergence.

The eigenvalues are returned in and the eigenvectors in .

– Function.

|

Finds the generalized eigenvalues () or eigenvalues and eigenvectors () of a symmetric matrix and symmetric positive-definite matrix . If , the upper triangles of and are used. If , the lower triangles of and are used. If , the problem to solve is . If , the problem to solve is . If , the problem to solve is .

– Function.

|

Computes the singular value decomposition of a bidiagonal matrix with on the diagonal and on the off-diagonal. If , is the superdiagonal. If , is the subdiagonal. Can optionally also compute the product .

Returns the singular values in , and the matrix overwritten with .

– Function.

|

Computes the singular value decomposition of a bidiagonal matrix with on the diagonal and on the off-diagonal using a divide and conquer method. If , is the superdiagonal. If , is the subdiagonal. If , only the singular values are found. If , the singular values and vectors are found. If , the singular values and vectors are found in compact form. Only works for real types.

Returns the singular values in , and if , the compact singular vectors in .

– Function.

|

Finds the reciprocal condition number of matrix . If , the condition number is found in the infinity norm. If or , the condition number is found in the one norm. must be the result of and is the norm of in the relevant norm.

– Function.

|

Converts a matrix to Hessenberg form. If is balanced with then and are the outputs of . Otherwise they should be and . contains the elementary reflectors of the factorization.

– Function.

|

Explicitly finds , the orthogonal/unitary matrix from . , , and must correspond to the input/output to .

– Function.

|

Computes the eigenvalues () or the eigenvalues and Schur vectors () of matrix . is overwritten by its Schur form. Returns , containing the Schur vectors, and , containing the eigenvalues.

– Function.

|

Computes the generalized eigenvalues, generalized Schur form, left Schur vectors (), or right Schur vectors () of and .

The generalized eigenvalues are returned in and . The left Schur vectors are returned in and the right Schur vectors are returned in .

– Function.

|

Reorder the Schur factorization of a matrix. If , the Schur vectors are reordered. If they are not modified. and specify the reordering of the vectors.

– Function.

|

Reorder the Schur factorization of a matrix and optionally finds reciprocal condition numbers. If , no condition numbers are found. If , only the condition number for this cluster of eigenvalues is found. If , only the condition number for the invariant subspace is found. If then the condition numbers for the cluster and subspace are found. If the Schur vectors are updated. If the Schur vectors are not modified. determines which eigenvalues are in the cluster.

Returns , , and reordered eigenvalues in .

– Function.

|

Reorders the vectors of a generalized Schur decomposition. specifies the eigenvalues in each cluster.

– Function.

|

Solves the Sylvester matrix equation where and are both quasi-upper triangular. If , is not modified. If , is transposed. If , is conjugate transposed. Similarly for and . If , the equation is solved. If , the equation is solved.

Returns (overwriting) and .

Chapter 53

Constantes

- Constant.

|

The singleton instance of type `Nothing`, used by convention when there is no value to return (as in a C `void` function). Can be converted to an empty `Array{Nothing,1}` value.

- Constant.

|

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 `ARGS`.

- Constant.

|

An array of the command line arguments passed to Julia, as strings.

- Constant.

|

The C null pointer constant, sometimes used when calling external code.

- Constant.

|

A `VersionNumber` object describing which version of Julia is in use. For details see [Version Number Literals](#).

- Constant.

|

An array of paths as strings or custom loader objects for the `function` and `and` statements to consider when loading code. To create a custom loader type, define the type and then add appropriate methods to the `function` with the following signature:

|

The `argument` is the current value in `,` `is` the name of the module to load, and `is` the path of any previously found code to provide `.` If no provider has been found earlier in `then` the value of `will` be `.` Custom loader functionality is experimental and may break or change in Julia 1.0.

- Constant.

|

A string containing the full path to the directory containing the `executable`.

- Constant.

|

Equivalent to `for` dispatch purposes, but signals the compiler to skip code generation specialization for that field.

- Constant.

|

The number of logical CPU cores available in the system.

See the `Hwloc.jl` package for extended information, including number of physical cores.

- Constant.

|

Standard word size on the current machine, in bits.

- Constant.

|

A symbol representing the name of the operating system, as returned by `of` the build configuration.

- Constant.

|

Chapter 54

Sistema de Ficheros

- Function.

|

Get the current working directory.

- Method.

|

Set the current working directory.

- Method.

|

Temporarily changes the current working directory and applies function before returning.

- Function.

|

Returns the files and directories in the directory (or the current working directory if not given).

- Function.

|

The method returns an iterator that walks the directory tree of a directory. The iterator returns a tuple containing . The directory tree can be traversed top-down or bottom-up. If encounters a it will rethrow the error by default. A custom error handling function can be provided through keyword argument. is called with a as argument.

|

|

– Function.

|

Make a new directory with name and permissions . defaults to , modified by the current file creation mask. This function never creates more than one directory. If the directory already exists, or some intermediate directories do not exist, this function throws an error. See for a function which creates all required intermediate directories.

– Function.

|

Create all directories in the given , with permissions . defaults to , modified by the current file creation mask.

– Function.

|

Creates a symbolic link to with the name .

Note

This function raises an error under operating systems that do not support soft symbolic links, such as Windows XP.

– Function.

|

Returns the target location a symbolic link points to.

– Function.

|

Change the permissions mode of to . Only integer s (e.g.) are currently supported. If and the path is a directory all permissions in that directory will be recursively changed.

– Function.

|

Change the owner and/or group of `to` and/or `.`. If the value entered for `or` is `.` the corresponding ID will not change. Only integer `s` and `s` are currently supported.

– Function.

|

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

– Function.

|

Like `.`, 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.

– Function.

|

Equivalent to

– Function.

|

Equivalent to `.`

– Function.

|

Equivalent to

– Function.

|

Equivalent to .

– Function.

|

Gets the permissions of the owner of the file as a bitfield of

Value	Description
01	Execute Permission
02	Write Permission
04	Read Permission

For allowed arguments, see .

– Function.

|

Like but gets the permissions of the group owning the file.

– Function.

|

Like but gets the permissions for people who neither own the file nor are a member of the group owning the file

– Function.

|

Copy the file, link, or directory from to . will first remove an existing .

If , and is a symbolic link, will be created as a symbolic link. If and is a symbolic link, will be a copy of the file or directory refers to.

– Function.

|

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 `wget` or `curl` 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.

– Function.

|

Move the file, link, or directory from `from` to `to`. `will` first remove an existing `to`.

– Function.

|

Delete the file, link, or empty directory at the given path. If `is` is passed, a non-existing path is not treated as error. If `is` is passed and the path is a directory, then all contents are removed recursively.

– Function.

|

Update the last-modified timestamp on a file to the current time.

– Function.

|

Generate a unique temporary file path.

– Function.

|

Obtain the path of a temporary directory (possibly shared with other processes).

– Method.

|

Returns `path`, where `path` is the path of a new temporary file in `dir` and `file` is an open file object for this path.

– Method.

|

Apply the function `remove` to the result of `mkstemp` and remove the temporary file upon completion.

- Method.

|

Create a temporary directory in the `dir` directory and return its path. If `dir` does not exist, throw an error.

- Method.

|

Apply the function `remove` to the result of `mkdtemp` and remove the temporary directory upon completion.

- Function.

|

Returns `1` if `fd` is a block device, otherwise, `0`.

- Function.

|

Returns `1` if `fd` is a character device, otherwise, `0`.

- Function.

|

Returns `1` if `fd` is a directory, otherwise, `0`.

- Function.

|

Returns `1` if `fd` is a FIFO, otherwise, `0`.

- Function.

|

Returns `1` if `fd` is a regular file, otherwise, `0`.

- Function.

|

Returns if is a symbolic link, otherwise.

– Function.

|

Returns if is a mount point, otherwise.

– Function.

|

Returns if is a valid filesystem path, otherwise.

– Function.

|

Returns if has the setgid flag set, otherwise.

– Function.

|

Returns if has the setuid flag set, otherwise.

– Function.

|

Returns if is a socket, otherwise.

– Function.

|

Returns if has the sticky bit set, otherwise.

– Function.

|

Return the current user's home directory.

Note

determines the home directory via `'s`. For details (for example on how to specify the home directory via environment variables), see the [documentation](#).

– Function.

|

Get the directory part of a path.

|

See also:

– Function.

|

Get the file name part of a path.

|

See also:

– Macro.

|

expands to a string with the absolute file path of the file containing the macro. Returns if run from a REPL or an empty string if evaluated by `.` Alternatively see `.`

– Macro.

|

expands to a string with the directory part of the absolute path of the file containing the macro. Returns if run from a REPL or an empty string if evaluated by `.`

– Macro.

|

expands to the line number of the call-site.

– Function.

|

Determines whether a path is absolute (begins at the root directory).

|

– Function.

|

Determines whether a path refers to a directory (for example, ends with a path separator).

|

– Function.

|

Join path components into a full path. If some argument is an absolute path, then prior components are dropped.

|

– Function.

|

Convert a path to an absolute path by adding the current directory if necessary.

|

Convert a set of paths to an absolute path by joining them together and adding the current directory if necessary.
Equivalent to .

– Function.

|

Normalize a path, removing "." and ".." entries.

|

- Function.

|

Canonicalize a path by expanding symbolic links and removing "." and ".." entries.

- Function.

|

Return a relative filepath to 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 or .

- Function.

|

On Unix systems, replace a tilde character at the start of a path with the current user's home directory.

- Function.

|

Split a path into a tuple of the directory name and file name.

|

- Function.

|

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.

- Function.

|

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.

|

Chapter 55

E/S y Redes

55.1 E/S General

- Constant.

|

Global variable referring to the standard out stream.

- Constant.

|

Global variable referring to the standard error stream.

- Constant.

|

Global variable referring to the standard input stream.

- Function.

|

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.

|

Alternate syntax for open, where a string-based mode specifier is used instead of the five booleans. The values of correspond to those from or Perl , 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

Apply the function `close` to the result of `open` and close the resulting file descriptor upon completion.

Example:

Start running `asynchronously`, and return a tuple `process, stream`. If `process` is `None`, then `stream` reads from the process's standard output and optionally specifies the process's standard input stream. If `process` is `None`, then `stream` writes to the process's standard input and optionally specifies the process's standard output stream.

Similar to `open`, but calls `close` on the resulting read or write stream, then closes the stream and waits for the process to complete. Returns the value returned by `close`.

– Type.

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.

Create an in-memory I/O stream.

Create a fixed size `IOBuffer`. The buffer will not grow dynamically.

Create a read-only `IOBuffer` on the data underlying the given string.

|

– Method.

|

Obtain the contents of an as an array, without copying. Afterwards, the is reset to its initial state.

– Function.

|

Create an object from an integer file descriptor. If is , closing this object will close the underlying descriptor. By default, an is closed when it is garbage collected. allows you to associate the descriptor with a named file.

– Function.

|

Commit all currently buffered writes to the given stream.

– Function.

|

Close an I/O stream. Performs a first.

– Function.

|

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 call. i.e. the following are equivalent:

|

– Function.

|

Open a file and read its contents. is passed to : this is equivalent to .

|

Read a series of values of type `from` , in canonical binary representation. `is` is either a tuple or a series of integer arguments specifying the size of the `to` to return.

|

Read at most `bytes` from `,` returning a `of` the bytes read.

|

Read at most `bytes` from `,` returning a `of` the bytes read.

If `is` (the default), this function will block repeatedly trying to read all requested bytes, until an error or end-of-file occurs. If `is` , at most one `call` is performed, and the amount of data returned is device-dependent. Note that not all stream types support the `option`.

|

Read a single value of type `from` , in canonical binary representation.

– Function.

|

Read binary data from an I/O stream or file, filling in `.`

– Function.

|

Read at most `bytes` from `into` , returning the number of bytes read. The size of `will` be increased if needed (i.e. if `is` greater than `and` enough bytes could be read), but it will never be decreased.

|

Read at most `bytes` from `into` , returning the number of bytes read. The size of `will` be increased if needed (i.e. if `is` greater than `and` enough bytes could be read), but it will never be decreased.

See `for` a description of the `option`.

– Function.

|

Copy from the stream object into (converted to a pointer).

It is recommended that subtypes override the following method signature to provide more efficient implementations:

– Function.

|

Copy from (converted to a pointer) into the object.

It is recommended that subtypes override the following method signature to provide more efficient implementations:

– Function.

|

Get the current position of a stream.

– Function.

|

Seek a stream to the given position.

– Function.

|

Seek a stream to its beginning.

– Function.

|

Seek a stream to its end.

– Function.

|

Seek a stream relative to the current position.

– Function.

|

Add a mark at the current position of stream . Returns the marked position.

See also , , .

– Function.

|

Remove a mark from stream . Returns if the stream was marked, otherwise.

See also , , .

– Function.

|

Reset a stream 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 , , .

– Function.

|

Returns if stream is marked.

See also , , .

– Function.

|

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 . Therefore it is always safe to read one byte after seeing return . will return as long as buffered data is still available, even if the remote end of a connection is closed.

– Function.

|

Determine whether a stream is read-only.

– Function.

|

Returns if the specified IO object is writable (if that can be determined).

- Function.

|

Returns if the specified IO object is readable (if that can be determined).

- Function.

|

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 to check for the ability to read data. Use to be notified when a stream might be writable or readable.

- Function.

|

Write an arbitrary value to a stream in an opaque format, such that it can be read back by . 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. values are serialized as all-zero bit patterns ().

- Function.

|

Read a value written by . assumes the binary data read from is correct and has been serialized by a compatible implementation of . It has been designed with simplicity and performance as a goal and does not validate the data read. Malformed data can result in process termination. The caller has to ensure the integrity and correctness of data read from .

- Function.

|

Print the shortest possible representation, with the minimum number of consecutive non-zero digits, of number , ensuring that it would parse to the exact same number.

- Function.

|

Returns the file descriptor backing the stream or file. Note that this function only applies to synchronous 's and 's not to any of the asynchronous streams.

- Function.

|

Create a pipe to which all C and Julia level output will be redirected. Returns a tuple representing the pipe ends. Data written to may now be read from the end of the pipe. The end is given for convenience in case the old object was cached by the user and needs to be replaced elsewhere.

Note

must be a , a , or a .

– Method.

|

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note

must be a , a , or a .

– Function.

|

Like , but for .

Note

must be a , a , or a .

– Method.

|

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note

must be a , a , or a .

– Function.

|

Like , but for . Note that the order of the return tuple is still , i.e. data to be read from may be written to .

Note

must be a , a , or a .

- Method.

|

Run the function while redirecting to . Upon completion, is restored to its prior setting.

Note

must be a , a , or a .

- Function.

|

Read the entirety of as a string and remove a single trailing newline. Equivalent to .

- Function.

|

Resize the file or buffer given by the first argument to exactly bytes, filling previously unallocated space with '\0' if the file or buffer is grown.

- Function.

|

Advance the stream until before the first character for which returns . For example will skip all whitespace. If keyword argument is specified, characters from that character through the end of a line will also be skipped.

- Function.

|

Read 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 are supported by passing them as the second argument.

- Function.

|

An that allows reading and performs writes by appending. Seeking and truncating are not supported. See for the available constructors. If is given, creates a to operate on a data vector, optionally specifying a size beyond which the underlying may not be grown.

- Function.

|

Read all available data on the stream, blocking the task only if no data is available. The result is a .

– Type.

|

provides a mechanism for passing output configuration settings among methods.

In short, it is an immutable dictionary that is a subclass of . It supports standard dictionary operations such as , and can also be used as an I/O stream.

– Method.

|

Create an that wraps a given stream, adding the specified pair to the properties of that stream (note that can itself be an).

- use to see if this particular combination is in the properties set
- use to retrieve the most recent value for a particular key

The following properties are in common use:

- : 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.
- : Boolean specifying that containers should be truncated, e.g. showing in place of most elements.
- : A 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 function.

|

– Method.

|

Create an `IO` that wraps an alternate `IO` but inherits the properties of `IO`.

55.2 E/S Textio

– Method.

|

Write an informative text representation of a value to the current output stream. New types should overload `show` where the first argument is a stream. The representation used by `show` generally includes Julia-specific formatting and type information.

– Function.

|

Show a compact representation of a value.

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 `iscompact` in its normal `show` method.

– Function.

|

Similar to `show`, except shows all elements of arrays.

– Function.

|

Return a string giving a brief description of a value. By default returns `typeof(x)`, e.g. `"{Int64}"`.

For arrays, returns a string of size and type info, e.g. `"{Array{Int64,1}}"`.

|

– Function.

|

|

Display a warning. `Argument` is a string describing the warning to be displayed.

|

– Function.

|

Stream output of informational, warning, and/or error messages to , overriding what was otherwise specified. Optionally, divert stream only for module , or specifically function within . can be (the default), , , or . See for the current set of redirections. Call with no arguments (or just the) to reset everything.

– Macro.

|

Print using C style format specification string, with some caveats: and are printed consistently as and for flags , , , , , and . 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 may be passed as the first argument to redirect output.

Examples

|

– Macro.

|

Return formatted output as string.

Examples

|

– Function.

|

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.

|

– Function.

|

Show a descriptive representation of an exception object.

– Function.

|

Show every part of the representation of a value.

– Function.

|

Read the entire contents of an I/O stream or a file as a string. The text is assumed to be encoded in UTF-8.

– Function.

|

Read a single line of text from the given I/O stream or file (defaults to `.`). When reading from a file, the text is assumed to be encoded in UTF-8. Lines in the input end with `\n` or the end of an input stream. When `trim` is true (as it is by default), these trailing newline characters are removed from the line before it is returned. When `trim` is false, they are returned as part of the line.

– Function.

|

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.

– Function.

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 repeatedly with the same arguments and saving the resulting lines as a vector of strings.

– Function.

Create an iterable object that will yield each line from an I/O stream or a file. Iteration calls on the stream argument repeatedly with 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 object is garbage collected.

– Method.

Read a matrix from the source where each line (separated by) 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 is a numeric type, the result is an array of that type, with any non-numeric elements as for floating-point types, or zero. Other useful values of include , , and .

If is , the first row of data will be read as header and the tuple is returned instead of only .

Specifying will ignore the corresponding number of initial lines from the input.

If is , blank lines in the input will be ignored.

If is , the file specified by is memory mapped for potential speedups. Default is except on Windows. On Windows, you may want to specify if the file is large, and is only read once and not written to.

If is , 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 as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files. If is , lines beginning with and text following in any line are ignored.

– Method.

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.

– Method.

The end of line delimiter is taken as `.`

– Method.

|

The end of line delimiter is taken as `.` 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.

– Method.

|

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `.`

– Method.

|

The columns are assumed to be separated by one or more whitespaces. The end of line delimiter is taken as `.` 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.

– Function.

|

Write (a vector, matrix, or an iterable collection of iterable rows) as text to (either a filename string or an `stream`) using the given delimiter (which defaults to tab, but can be any printable Julia object, typically a `Char` or `String`).

For example, two vectors `x` and `y` of the same length can be written as two columns of tab-delimited text to `io` by either `write(io, x, y)` or by `write(io, x, y, "\t")`.

– Function.

|

Equivalent to `write(io, x, y, "\t")` with `io` set to comma, and type optionally defined by `T`.

– Function.

|

Equivalent to `write(io, x, y, "\t")` with `io` set to comma.

– Type.

|

Returns a new write-only I/O stream, which converts any bytes written to it into base64-encoded ASCII bytes written to . Calling on the stream is necessary to complete the encoding (but does not close).

|

– Type.

|

Returns a new read-only I/O stream, which decodes base64-encoded data read from .

|

– Function.

|

Given a -like function , which takes an I/O stream as its first argument, calls to write to a base64-encoded string, and returns the string. is equivalent to : it converts its arguments into bytes using the standard functions and returns the base64-encoded string.

See also .

– Function.

|

Decodes the base64-encoded `data` and returns a `bytes` object of the decoded bytes.

See also

|

– Function.

|

Return the nominal size of the screen that may be used for rendering output to this io object

55.3 E/S Multimedia

Del mismo modo que la salida de texto se realiza mediante `print` y los tipos definidos por el usuario pueden indicar su representación textual sobrecargando `__str__`, Julia proporciona un mecanismo estandarizado para una salida multimedia enriquecida (como imágenes, texto formateado, o incluso audio y video) que consta de tres partes:

- Una función `show` para solicitar la visualización multimedia más completa disponible de un objeto Julia (con una reserva de texto sin formato).
- Sobrecargar `show` permite indicar representaciones multimedia arbitrarias (codificadas mediante tipos MIME estándar) de tipos definidos por el usuario.
- Pueden registrarse backends de visualización con capacidad multimedia subclasificando un tipo genérico de `Backend` y poniéndolos en una pila de backends de visualización mediante `push_backend!`.

El tiempo de ejecución base de Julia proporciona solo visualización de texto sin formato, pero las pantallas más ricas pueden habilitarse cargando módulos externos o utilizando entornos gráficos de Julia (como el *notebook* IJulia, basado en IPython).

– Function.

|

Display using the topmost applicable display in the display stack, typically using the richest supported multimedia output for `obj`, with plain-text output as a fallback. The `display` variant attempts to display `obj` on the given display `dis` only, throwing a `DisplayError` if `dis` cannot display objects of this type.

There are also two variants with a `mime` argument (a MIME type string, such as `"text/html"`), which attempt to display `obj` using the requested MIME type *only*, throwing a `DisplayError` if this type is not supported by either the display(s) or by `dis`. With these variants, one can also supply the "raw" data in the requested MIME type by passing `data` (for MIME types with text-based storage, such as `text/html` or `application/postscript`) or `data` (for binary MIME types).

– Function.

```
|
```

By default, the `display` functions simply call `show`. However, some display backends may override `show` to modify an existing display of `obj` (if any). Using `display` is also a hint to the backend that `obj` may be redisplayed several times, and the backend may choose to defer the display until (for example) the next interactive prompt.

– Function.

```
|
```

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 `dis` in the second variant.

– Method.

```
|
```

The `display` functions ultimately call `show` in order to write an object `obj` as a given `mime` type to a given I/O (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{T}(io::IOStream, obj::T, mime::String)`, where `mime` is a MIME-type string and the function body calls `write(io, obj)` (or similar) to write that representation of `obj` to `io`. (Note that the `show{T}` notation only supports literal strings; to construct `show{T}` types in a more flexible manner use `show{T}(io::IOStream, obj::T, mime::String)`.)

For example, if you define a `T` type and know how to write it to a PNG file, you could define a function `show{T}(io::IOStream, obj::T, mime::String)` to allow your images to be displayed on any PNG-capable (such as IJulia). As usual, be sure to `using Base.show{T}` in order to add new methods to the built-in Julia function `show`.

The default MIME type is `"text/html"`. There is a fallback definition for `show{T}(io::IOStream, obj::T, mime::String)` output that calls `show{T}(io::IOStream, obj::T)` with 2 arguments. Therefore, this case should be handled by defining a 2-argument `show{T}` method.

Technically, the `show{T}` 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{T}` can be an `IOStream` specifying output format properties. See `show{T}` for details.

– Function.

|

Returns a boolean value indicating whether or not the object can be written as the given type. (By default, this is determined automatically by the existence of the corresponding method for .)

– Function.

|

Returns an or containing the representation of in the requested type, as written by (throwing a if no appropriate is available). An is returned for MIME types with textual representations (such as or), whereas binary data is returned as . (The function returns whether or not Julia treats a given type as text.)

As a special case, if is an (for textual MIME types) or a (for binary MIME types), the function assumes that is already in the requested format and simply returns . This special case does not apply to the MIME type. This is useful so that raw data can be passed to .

– Function.

|

Returns an containing the representation of in the requested type. This is similar to except that binary data is base64-encoded as an ASCII string.

Como se mencionó anteriormente, también se pueden definir nuevos backends de pantalla. Por ejemplo, un módulo que puede mostrar imágenes PNG en una ventana puede registrar esta capacidad con Julia, de modo que llamar a en tipos con representaciones PNG mostrará automáticamente la imagen usando la ventana del módulo.

Para definir un nuevo backend de pantalla, primero se debe crear un subtipo de la clase abstracta . Luego, para cada tipo MIME (cadena) que se puede mostrar en , uno debe definir una función que muestra como ese tipo MIME, generalmente llamando a . Se debe lanzar si no se puede mostrar como ese tipo MIME; esto es automático si uno llama a . Finalmente, se debe definir una función que consulte para los tipos soportados por y muestre el "mejor"; debe lanzarse un si no se encuentran tipos MIME soportados para . Del mismo modo, algunos subtipos pueden sobrescribir . (De nuevo, uno debe hacer para agregar nuevos métodos a .) Los valores de retorno de estas funciones dependen de la implementación (ya que en algunos casos puede ser útil devolver un "manejador" *handle* de visualización de algunos tipo). Las funciones de visualización para se pueden llamar directamente, pero también se pueden invocar automáticamente desde simplemente presionando una nueva pantalla en la pila display-backend con:

– Function.

|

Pushes a new display on top of the global display-backend stack. Calling or will display on the topmost compatible backend in the stack (i.e., the topmost backend that does not throw a).

– Function.

|

Pop the topmost backend off of the display-backend stack, or the topmost copy of in the second variant.

– Type.

|

Returns a , which displays any object as the 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.)

– Function.

|

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).

55.4 E/S Mapeada en Memoria

– Type.

|

Create an -like object for creating zeroed-out mmapmed-memory that is not tied to a file for use in . Used by for creating shared memory arrays.

– Method.

|

Create an 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 with a bits-type element of and dimension that determines how the bytes of the array are interpreted. Note that the file must be stored in binary format, and no format conversions are possible (this is a limitation of operating systems, not Julia).

is a tuple or single specifying the size or length of the array.

The file is passed via the stream argument, either as an open or filename string. When you initialize the stream, use for a "read-only" array, and to create a new array used to write values to disk.

If no argument is specified, the default is .

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 .

The 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 keyword argument specifies whether the resulting and changes made to it will be visible to other processes mapping the same file.

For example, the following code

```

|

```

creates a `-by-` , linked to the file associated with stream .

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).

– Method.

```

|

```

Create a 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 , but the byte representation is different.

Example:

This would create a `25-by-30000` , linked to the file associated with stream .

– Function.

```

|

```

Forces synchronization between the in-memory version of a memory-mapped or and the on-disk version.

55.5 E/S por Red

– Method.

```

|

```

Connect to the host on port .

– Method.

```

|

```

Connect to the named pipe / UNIX domain socket at .

– Method.

|

Listen on port on the address specified by . By default this listens on only. To listen on all interfaces pass or as appropriate. determines how many connections can be pending (not having called) before the server will begin to reject them. The default value of is 511.

– Method.

|

Create and listen on a named pipe / UNIX domain socket.

– Function.

|

Gets the IP address of the (may have to do a DNS lookup)

– Function.

|

Get the IP address and the port that the given is connected to (or bound to, in the case of).

– Type.

|

Returns an IPv4 object from ip address formatted as an .

|

– Type.

|

Returns an IPv6 object from ip address formatted as an .

|

- Function.

|

Returns the number of bytes available for reading before a read from this stream or buffer will block.

- Function.

|

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.

- Function.

|

Create a 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.

- Function.

|

Monitor a file descriptor for changes in the read or write availability, and with a timeout given by seconds.

The keyword arguments determine which of read and/or write status should be monitored; at least one of them must be set to .

The returned value is an object with boolean fields , , and , giving the result of the polling.

- Function.

|

Monitor a file for changes by polling every seconds until a change occurs or seconds have elapsed. The should be a long period; the default is 5.007 seconds.

Returns a pair of objects when a change is detected.

To determine when a file was modified, compare to detect notification of changes. However, using for this operation is preferred, since it is more reliable and efficient, although in some situations it may not be available.

- Function.

|

Watch file or directory for changes until a change occurs or seconds have elapsed.

The returned value is an object with boolean fields , , and , 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 for more detailed information.

– Function.

|

Bind to the given . Note that will listen on all devices.

- The parameter disables dual stack mode. If , only an IPv6 stack is created.
- If , multiple threads or processes can bind to the same address without error if they all set , but only the last to bind will receive any traffic.

|

Associates the lifetime of with a task. Channel is automatically closed when the task terminates. Any uncaught exception in the task is propagated to all waiters on .

The 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.

|

|

– Function.

|

Send over to .

– Function.

|

Read a UDP packet from the specified socket, and return the bytes received. This call blocks.

– Function.

|

Read a UDP packet from the specified socket, returning a tuple of , where will be either IPv4 or IPv6 as appropriate.

– Function.

|

Set UDP socket options.

- : loopback for multicast packets (default:).
- : TTL for multicast packets (default:).
- : flag must be set to if socket will be used for broadcast messages, or else the UDP system will return an access error (default:).
- : Time-to-live of packets sent on the socket (default:).

– Function.

|

Converts the endianness of a value from Network byte order (big-endian) to that used by the Host.

– Function.

|

Converts the endianness of a value from that used by the Host to Network byte order (big-endian).

– Function.

|

Converts the endianness of a value from Little-endian to that used by the Host.

– Function.

|

Converts the endianness of a value from that used by the Host to Little-endian.

– Constant.

|

The 32-bit byte-order-mark indicates the native byte order of the host machine. Little-endian machines will contain the value . Big-endian machines will contain the value .

Chapter 56

Puntuación

Puede encontrar documentación extendida sobre símbolos y funciones matemáticas [aquí](#).

symbol	meaning
	Invoca la macro ; seguido de expresiones separadas por espacios
	Operador "not" prefijo
	Al final de un nombre de función, indica que la función modifica su(s) argumento(s)
	Inicio de un comentario de una sola línea
	Inicio de un comentario multilínea (ellos son anidables)
	Final de un comentario multilínea
	Interpolación de cadena y expresión
	Operador resto
	Operador exponente
	Operador and bit-a-bit
	Operador and booleano (en corto-circuito)
	Operador or bit-a-bit
	Operador or booleano (en corto-circuito)
	Operador xor bit-a-bit
	Multiplicación o producto matricial
	Tupla vacía
	Operador not bit-a-bit
	Operador backslash
	Operador transpuesto complejo A^H
	Indexación de array
	Concatenación vertical
	Concatenación vertical (también)
	Con expresiones separadas por espacios, concatenación horizontal
	Instanciación de tipo paramétrico
	Separador de instrucciones
	Separador de argumentos de función o de componentes de una tupla
	Operador condicional ternario (conditional ? if_true : if_false)
	Delimitador de literales cadena
	Delimitador de literales carácter
	Delimitador de especificaciones de proceso externo (mandato)
	splice arguments into a function call or declare a varargs function or type
	Acceso nombrado a campos en objetos/módulos, también llamadas a operator/function vectorizadas
	Rango a, a+1, a+2, ..., b
	Rango a, a+s, a+2s, ..., b
	Indexa una dimensión entera (1:end)
	Anotación de tipo ,dependiendo del contexto
	Expresión citada
	símbolo a
	(reverse of subtype operator)

Chapter 57

Ordenación y Funciones Relacionadas

Julia tiene una API amplia y flexible para ordenar e interactuar con matrices de valores ya ordenados. Por defecto, Julia selecciona algoritmos y ordenaciones razonables en orden ascendente estándar:

```
|
```

Uno también puede ordenar en orden inverso:

```
|
```

Para ordenar un array en el lugar, use la versión con admiración de la función de ordenación:

```
|
```

En lugar de ordenar un array directamente, podemos computar una premutación de los índices del array que ponen el array en un orden determinado:

```
|
```

|

Los arrays pueden ser ordenados fácilmente de acuerdo a una transformacin arbitraria de sus valores:

|

O en orden reverso mediante una transformación

|

Si es necesario, puede elegirse el algoritmo de ordenación:

|

Todas las funciones de ordenación y relacionadas con orden se basan en una relación "menor que" que define un orden total sobre los valores que van a manipularse. La función `sort` es la invocada por defecto, pero la relación puede ser especificada mediante la palabra clave `key`.

57.1 Funciones de Ordenación

– Function.

|

Sort the vector `x` in place. `sort` is used by default for numeric arrays while `sortrows` is used for other arrays. You can specify an algorithm to use via the `method` keyword (see [Sorting Algorithms](#) for available algorithms). The `key` keyword lets you provide a function that will be applied to each element before comparison; the `keyfunc` keyword allows providing a custom "less than" function; use `reverse` to reverse the sorting order. These options are independent and can be used together in all possible combinations: if both `key` and `keyfunc` are specified, the `keyfunc` function is applied to the result of the `key` function; `reverse` reverses whatever ordering specified via the `key` and `keyfunc` keywords.

Examples

|

– Function.

|

Variant of `sort` that returns a sorted copy of `x` leaving `x` itself unmodified.

Examples

|

|

|

Sort a multidimensional array along the given dimension. See for a description of possible keyword arguments.

Examples

|

– Function.

|

Return a permutation vector of indices of that puts it in sorted order. Specify to choose a particular sorting algorithm (see Sorting Algorithms). 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 , a different permutation that puts the array into order may be returned. The order is specified using the same keywords as .

See also .

Examples

|

– Function.

|

Like , but accepts a preallocated index vector . If is (the default), is initialized to contain the values .

Examples

|

– Function.

|

Sort the rows of matrix lexicographically. See for a description of possible keyword arguments.

Examples

|

– Function.

|

Sort the columns of matrix `x` lexicographically. See `sort` for a description of possible keyword arguments.

Examples

```
|
```

57.2 Funciones relacionadas con Orden

– Function.

```
|
```

Test whether a vector is in sorted order. The `by`, `order`, and `method` keywords modify what order is considered to be sorted just as they do for `sort`.

Examples

```
|
```

– Function.

```
|
```

Returns the range of indices of `x` which compare as equal to `val` (using binary search) according to the order specified by the `by`, `order`, and `method` keywords, assuming that `x` is already sorted in that order. Returns an empty range located at the insertion point if `x` does not contain values equal to `val`.

Examples

|

– Function.

|

Returns the index of the first value in greater than or equal to , according to the specified order. Returns if is greater than all values in . is assumed to be sorted.

Examples

|

– Function.

|

Returns the index of the last value in less than or equal to , according to the specified order. Returns if is less than all values in . is assumed to be sorted.

Examples

|

– Function.

|

Partially sort the vector `v` in place, according to the order specified by `comp`, and so that the value at index `i` (or range of adjacent values if `i` is a range) occurs at the position where it would appear if the array were fully sorted via a non-stable algorithm. If `i` is a single index, that value is returned; if `i` is a range, an array of values at those indices is returned. Note that `nth_element` does not fully sort the input array.

Examples

|

– Function.

|

Variant of `nth_element` which copies `v` before partially sorting it, thereby returning the same thing as `nth_element` but leaving `v` unmodified.

– Function.

|

Return a partial permutation of the vector `v`, according to the order specified by `ix`, and `l`, so that `sortperm(v, l, ix)` returns the first (or range of adjacent values if `l` is a range) values of a fully sorted version of `v`. If `ix` is a single index (Integer), an array of the first `l` indices is returned; if `l` is a range, an array of those indices is returned. Note that the handling of integer values for `l` is different from `sort` in that it returns a vector of `l` elements instead of just the `l`th element. Also note that this is equivalent to, but more efficient than, calling `sortperm(v, l)`.

– Function.

|

Like `sortperm`, but accepts a preallocated index vector `ix`. If `ix` is `nothing` (the default), `ix` is initialized to contain the values `1:n`.

57.3 Algoritmos de Ordenación

Actualmente hay cuatro algoritmos de ordenación disponibles en Julia base:

-
-
-
-

`sort` es un algoritmo de ordenación estable cuyo coste es $O(n^2)$. Es eficiente para `Array{<T>,1}` muy pequeños, y es usado internamente por `sortperm`.

`sort!::T{<T>}` es un algoritmo de ordenación que es *in-place* muy rápido pero no estable (es decir, los elementos que son considerados iguales no permanecerán en el mismo orden en que se encontraban originalmente en el array antes de ser ordenados. Su coste computacional es $O(n \log n)$. `sort!::T{<T>}` es el algoritmo por defecto para valores numéricos, incluyendo enteros y punto flotante.

`sort!::T{<T>}` es similar a `sort!::T{<T>}`, pero el array de salida es sólo ordenado hasta el índice `l` si `l` es un entero, o en el rango de `l` si `l` es un `Range{<T>}`. Por ejemplo:

|

es un algoritmo de ordenación estable, pero no *in-place* (requiere un array temporal de la mitad del tamaño del array de entrada), de coste $O(n \log n)$ y no suele ser tan rápido como `sort`. Es el algoritmo por defecto para datos no numéricos.

Los algoritmos de clasificación por defecto se eligen sobre la base de que son rápidos y estables, o *parezcan* serlo. Para los tipos numéricos, de hecho, se selecciona `sort` ya que es más rápido e indistinguible en este caso de un tipo estable (a menos que la matriz registre sus mutaciones de alguna manera). La propiedad de estabilidad tiene un costo no despreciable, por lo que si no la necesita, puede especificar explícitamente su algoritmo preferido, `sort!{f}`.

El mecanismo por el cual Julia selecciona los algoritmos de clasificación predeterminados se implementa a través de la función `sortkey`. Permite que un algoritmo particular se registre como el predeterminado en todas las funciones de ordenación para arrays específicos. Por ejemplo, aquí están los dos métodos predeterminados de `sortkey`:

```
|
```

En cuanto a los arrays numéricos, la elección de un algoritmo predeterminado no estable para los tipos de array para los cuales la noción de ordenación estable no tiene sentido (es decir, cuando dos valores que comparan iguales no se pueden distinguir) puede tener sentido.

Chapter 58

Funciones del Administrador de Paquetes

Todas las funciones del administrador de paquetes están definidas en el módulo `.` Ninguna de las funciones del módulo están exportadas. Por tanto, para usarlas, necesotamps prefijar cada llamada a función con un `.` explícito, por ejemplo `.` or `.`.

Las funciones para desarrollo de de paquetes (por ejemplo, `.`, `.`, etc.) se han movido al paquete [PkgDev](#). Ver [PkgDev README](#) para la documentación de estas funciones.

– Function.

|

Returns the absolute path of the package directory. This defaults to `.` on all platforms (i.e. `.` in UNIX shell syntax). If the `.` environment variable is set, then that path is used in the returned value as `.`. If `.` is a relative path, it is interpreted relative to whatever the current working directory is.

|

Equivalent to `.` – i.e. it appends path components to the package directory and normalizes the resulting path. In particular, `.` returns the path to the package `.`.

– Function.

|

Initialize `.` as a package directory. This will be done automatically when the `.` is not set and `.` 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.

– Function.

|

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 `.` and the state of installed packages in `.`, Packages that are no longer required are moved into `.`.

– Function.

|

Opens in the editor specified by the or environment variables; when the editor command returns, it runs to determine and install a new optimal set of installed package versions.

– Function.

|

Add a requirement entry for to and call . If are given, they must be objects and they specify acceptable version intervals for .

– Function.

|

Remove all requirement entries for from and call .

– Function.

|

If has a URL registered in , clone it from that URL on the default branch. The package does not need to have any registered versions.

|

Clone a package directly from the git URL . The package does not need to be registered in . The package repo is cloned by the name if provided; if not provided, is determined automatically from .

– Function.

|

Set the protocol used to access GitHub-hosted packages. Defaults to 'https', with a blank delegating the choice to the package developer.

– Function.

|

Returns the names of available packages.

|

Returns the version numbers available for package .

– Function.

|

Returns a dictionary mapping installed package names to the installed version number of each package.

|

If is installed, return the installed version number. If is registered, but not installed, return .

– Function.

|

Prints out a summary of what packages are installed and what version and state they're in.

|

Prints out a summary of what version and state , specifically, is in.

– Function.

|

Update the metadata repo – kept in – then update any fixed packages that can safely be pulled from their origin; then call 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.

– Function.

|

Checkout the repo to the branch . Defaults to checking out the "master" branch. To go back to using the newest compatible released version, use . Changes are merged (fast-forward only) if the keyword argument , and the latest version is pulled from the upstream repo if .

– Function.

|

Pin at the current version. To go back to using the newest compatible released version, use

|

Pin at registered version .

– Function.

|

Free the package to be managed by the package manager again. It calls to determine optimal package versions after. This is an inverse for both and .

You can also supply an iterable collection of package names, e.g., to free multiple packages at once.

– Function.

|

Run the build scripts for all installed packages in depth-first recursive order.

|

Run the build script in for each package in and all of their dependencies in depth-first recursive order. This is called automatically by on all installed or updated packages.

– Function.

|

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 file and test dependencies are specified in . Coverage statistics for the packages may be generated by passing . The default behavior is not to run coverage.

|

Run the tests for each package in ensuring that each package's test dependencies are installed for the duration of the test. A package is tested by running its file and test dependencies are specified in . Coverage statistics for the packages may be generated by passing . The default behavior is not to run coverage.

– Function.

|

List the packages that have as a dependency.

Chapter 59

Fechas y Tiempo

59.1 Tipos para Fechas y Tiempo

– Type.

|

types represent discrete, human representations of time.

– Type.

|

A 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 . In fact, a is automatically generated by addition of different period types, e.g. produces a result.

– Type.

|

types represent integer-based, machine representations of time as continuous timelines starting from an epoch.

– Type.

|

The `Instant` represents a machine timeline based on UT time (1 day = one revolution of the earth). The `precision` is a parameter that indicates the resolution or precision of the instant.

– Type.

```
|
```

`Instant` types wrap `machine` instances to provide human representations of the machine instant. `OffsetDateTime`, `LocalDateTime` and `LocalDate` are subtypes of `Instant`.

– Type.

```
|
```

`OffsetDateTime` wraps a `Instant` and interprets it according to the proleptic Gregorian calendar.

– Type.

```
|
```

`LocalDateTime` wraps a `Instant` and interprets it according to the proleptic Gregorian calendar.

– Type.

```
|
```

`LocalDate` wraps a `Instant` and represents a specific moment in a 24-hour day.

59.2 Funciones para Fechas

Todas las funciones para fechas están definidas en el módulo `java.time`; nótese que sólo se han exportado las funciones `now`, `nowUTC` y `nowInstant`; para usar todas las demás funciones de `java.time`, es necesario prefijar cada llamada a función con `java.time.`, por ejemplo, `java.time.LocalDate.now()`. Alternativamente, uno puede escribir `import java.time.*` para llevar todas las funciones exportadas en `java.time` para ser usadas sin el prefijo.

– Method.

```
|
```

Construct a `LocalDateTime` type by parts. Arguments must be convertible to `Instant`.

– Method.

```
|
```

Construct a `LocalDateTime` type by `type` parts. Arguments may be in any order. `DateTime` parts not provided will default to the value of `LocalDateTime`.

– Method.

|

Create a through the adjuster API. The starting point will be constructed from the provided arguments, and will be adjusted until returns . The step size in adjusting can be provided manually through the keyword. provides a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that is never satisfied).

– Method.

|

Converts a to a . The hour, minute, second, and millisecond parts of the new are assumed to be zero.

– Method.

|

Construct a by parsing the date string following the pattern given in the string.

This method creates a object each time it is called. If you are parsing many date strings of the same format, consider creating a object once and using that as the second argument instead.

– Method.

|

Construct a string by using a object and applying the provided . The following character codes can be used to construct the string:

Code	Examples	Comment
	6	Numeric year with a fixed width
	1996	Numeric year with a minimum width
	1, 12	Numeric month with a minimum width
	Jan	Month name shortened to 3-chars according to the
	January	Full month name according to the keyword
	1, 31	Day of the month with a minimum width
	0, 23	Hour (24-hour clock) with a minimum width
	0, 59	Minute with a minimum width
	0, 59	Second with a minimum width
	000, 500	Millisecond with a minimum width of 3
	Mon, Tue	Abbreviated days of the week
	Monday	Full day of week name

The number of sequential code characters indicate the width of the code. A format of specifies that the code should have a width of four while a width of two. Codes that yield numeric digits have an associated mode: fixed-width or minimum-width. The fixed-width mode left-pads the value with zeros when it is shorter than the specified width and truncates the value when longer. Minimum-width mode works the same as fixed-width except that it does not truncate values longer than the width.

When creating a `DateTimeFormat` you can use any non-code characters as a separator. For example to generate the string "1996-01-15T00:00:00" you could use : "yyyy-mm-ddTHH:MM:SS". Note that if you need to use a code character as a literal you can use the escape character backslash. The string "1996y01m" can be produced with the format "yyyy\ymm\m".

– Type.

|

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 `DateTimeFormat` string:

Code	Matches	Comment
	1996, 96	Returns year of 1996, 0096
	1996, 96	Returns year of 1996, 0096. Equivalent to
	1, 01	Matches 1 or 2-digit months
	Jan	Matches abbreviated months according to the keyword
	January	Matches full month names according to the keyword
	1, 01	Matches 1 or 2-digit days
	00	Matches hours
	00	Matches minutes
	00	Matches seconds
	.500	Matches milliseconds
	Mon, Tues	Matches abbreviated days of the week
	Monday	Matches full name days of the week
	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 string of "1996-01-15T00:00:00.0" would have a 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\ym\m".

Creating a `DateFormat` object is expensive. Whenever possible, create it once and use it many times or try the string macro. Using this macro creates the `DateFormat` object once at macro expansion time and reuses it later. see .

See `DateTimeFormat` and `DateTimeFormat` for how to use a `DateFormat` object to parse and write Date strings respectively.

– Macro.

|

Create a `DateTimeFormat` object. Similar to `DateTimeFormat` but creates the `DateFormat` object once during macro expansion.

See `DateTimeFormat` for details about format specifiers.

– Method.

|

Construct a `DateTimeFormat` by parsing the date string following the pattern given in the `DateTimeFormat` object. Similar to `DateTimeFormat` but more efficient when repeatedly parsing similarly formatted date strings with a pre-created `DateTimeFormat` object.

– Method.

|

Construct a `type` by parts. Arguments must be convertible to `.`

– Method.

|

Construct a `type` by `type` parts. Arguments may be in any order. `parts` not provided will default to the value of `.`

– Method.

|

Create a `through` the adjuster API. The starting point will be constructed from the provided `arguments`, and will be adjusted until `returns` `.` The step size in adjusting can be provided manually through the `keyword`. `provides` a limit to the max number of iterations the adjustment API will pursue before throwing an error (given that `is` never satisfied).

– Method.

|

Converts a `to` a `.` The hour, minute, second, and millisecond parts of the `are` truncated, so only the year, month and day parts are used in construction.

– Method.

|

Construct a `object` by parsing a date string following the pattern given in the `string`. Follows the same conventions as `.`

– Method.

|

Parse a date from a date string using a `object` `.`

– Method.

|

Construct a `type` by parts. Arguments must be convertible to `.`

– Method.

```
|
```

Construct a `type` by `type` parts. Arguments may be in any order. `parts` not provided will default to the value of `.`

– Method.

```
|
```

Create a `through` through the `adjuster` API. The starting point will be constructed from the provided `arguments`, and will be adjusted until `returns` `.` The step size in adjusting can be provided manually through the `keyword`. `provides` a limit to the max number of iterations the adjustment API will pursue before throwing an error (in the case that `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 `instead of` `.`

– Method.

```
|
```

Converts a `to a` `.` The hour, minute, second, and millisecond parts of the `are` used to create the new `.` Microsecond and nanoseconds are zero by default.

– Method.

```
|
```

Returns a `corresponding` to the user's system time including the system timezone locale.

– Method.

```
|
```

Returns a `corresponding` to the user's system time as UTC/GMT.

– Function.

```
|
```

Returns `for` values, `for` values, and `for` values.

Funciones Accesoras

– Function.

|

The year of a or as an .

– Function.

|

The month of a or as an .

– Function.

|

Return the **ISO week date** of a or as an . 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 is the 53rd week of 2004.

– Function.

|

The day of month of a or as an .

– Function.

|

The hour of day of a as an .

|

The hour of a as an .

– Function.

|

The minute of a as an .

|

The minute of a as an .

- Function.

|

The second of a as an .

|

The second of a as an .

- Function.

|

The millisecond of a as an .

|

The millisecond of a as an .

- Function.

|

The microsecond of a as an .

- Function.

|

The nanosecond of a as an .

- Method.

|

Construct a object with the given value. Input must be losslessly convertible to an .

- Method.

|

Construct a object with the given value. Input must be losslessly convertible to an .

- Method.

|

Construct a `OffsetDateTime` object with the given `value`. Input must be losslessly convertible to an `Instant`.

– Method.

|

Construct a `OffsetDateTime` object with the given `value`. Input must be losslessly convertible to an `Instant`.

– Method.

|

The hour part of a `DateTime` as a `int`.

– Method.

|

The minute part of a `DateTime` as a `int`.

– Method.

|

The second part of a `DateTime` as a `int`.

– Method.

|

The millisecond part of a `DateTime` as a `int`.

– Method.

|

The microsecond part of a `Time` as a `int`.

– Method.

|

The nanosecond part of a `Time` as a `int`.

- Function.

|

Simultaneously return the year and month parts of a or .

- Function.

|

Simultaneously return the month and day parts of a or .

- Function.

|

Simultaneously return the year, month and day parts of a or .

Funciones de Consulta

- Function.

|

Return the full day name corresponding to the day of the week of the or in the given .

- Function.

|

Return the abbreviated name corresponding to the day of the week of the or in the given .

- Function.

|

Returns the day of the week as an with .

- Function.

|

The day of month of a or as an .

- Function.

|

For the day of week of , returns which number it is in 's month. So if the day of the week of is Monday, then In the range 1:5.

– Function.

|

For the day of week of , returns the total number of that day of the week in 's month. Returns 4 or 5. Useful in temporal expressions for specifying the last day of a week in a month by including in the adjuster function.

– Function.

|

Return the full name of the month of the or in the given .

– Function.

|

Return the abbreviated month name of the or in the given .

– Function.

|

Returns the number of days in the month of . Value will be 28, 29, 30, or 31.

– Function.

|

Returns if the year of is a leap year.

– Function.

|

Returns the day of the year for with January 1st being day 1.

– Function.

|

Returns 366 if the year of is a leap year, otherwise returns 365.

- Function.

|

Returns the quarter that resides in. Range of value is 1:4.

- Function.

|

Returns the day of the current quarter of . Range of value is 1:92.

Funciones de Ajuste

- Method.

|

Truncates the value of according to the provided type. E.g. if is , then .

- Function.

|

Adjusts to the Monday of its week.

- Function.

|

Adjusts to the Sunday of its week.

- Function.

|

Adjusts to the first day of its month.

- Function.

|

Adjusts to the last day of its month.

- Function.

|

Adjusts to the first day of its year.

– Function.

|

Adjusts to the last day of its year.

– Function.

|

Adjusts to the first day of its quarter.

– Function.

|

Adjusts to the last day of its quarter.

– Method.

|

Adjusts to the next day of week corresponding to with . Setting allows the current to be considered as the next , allowing for no adjustment to occur.

– Method.

|

Adjusts to the previous day of week corresponding to with . Setting allows the current to be considered as the previous , allowing for no adjustment to occur.

– Function.

|

Adjusts to the first of its month. Alternatively, will adjust to the first of the year.

– Function.

|

Adjusts to the last of its month. Alternatively, will adjust to the last of the year.

– Method.

|

Adjusts by iterating at most iterations by increments until returns . must take a single argument and return a . allows to be considered in satisfying .

– Method.

|

Adjusts by iterating at most iterations by increments until returns . must take a single argument and return a . allows to be considered in satisfying .

Períodos

– Method.

|

Construct a type with the given value. Input must be losslessly convertible to an .

– Method.

|

Construct a from a of s. All s of the same type will be added together.

Examples

|

– Function.

|

Returns a sensible "default" value for the input Period by returning for Year, Month, and Day, and for Hour, Minute, Second, and Millisecond.

Funciones de Redondeo

and values can be rounded to a specified resolution (e.g., 1 month or 15 minutes) with , , or .

– Method.

|

Returns the nearest or less than or equal to at resolution .

For convenience, may be a type instead of a value: is a shortcut for .

|

– Method.

|

Returns the nearest or greater than or equal to at resolution .

For convenience, may be a type instead of a value: is a shortcut for .

|

– Method.

|

Returns the or nearest to at resolution . By default (), ties (e.g., rounding 9:30 to the nearest hour) will be rounded up.

For convenience, may be a type instead of a value: is a shortcut for .

|

Valid rounding modes for are (default), (), and ().

Las siguientes funciones no están exportadas:

– Function.

|

Simultaneously return the and of a or at resolution . More efficient than calling both and individually.

– Function.

|

Takes the number of days since the rounding epoch () and returns the corresponding .

– Function.

|

Takes the number of milliseconds since the rounding epoch () and returns the corresponding .

– Function.

|

Takes the given and returns the number of days since the rounding epoch () as an .

– Function.

|

Takes the given and returns the number of milliseconds since the rounding epoch () as an .

Funciones de Conversión

– Function.

|

Returns the date portion of .

– Function.

|

Takes the number of seconds since unix epoch and converts to the corresponding .

– Function.

|

Takes the given and returns the number of seconds since the unix epoch as a .

– Function.

|

Takes the number of Julian calendar days since epoch and returns the corresponding .

– Function.

|

Takes the given and returns the number of Julian calendar days since the julian epoch as a .

– Function.

|

Takes the number of Rata Die days since epoch and returns the corresponding .

– Function.

|

Returns the number of Rata Die days since epoch from the given or .

Constantes

Días de la semana:

Meses del Año:

Variable	Abbr.	Value (Int)
		1
		2
		3
		4
		5
		6
		7

Variable	Abbr.	Value (Int)
		1
		2
		3
		4
		5
		6
		7
		8
		9
		10
		11
		12

Chapter 60

Utilidades para Iteración

– Function.

|

For a set of iterable objects, returns an iterable of tuples, where the *th* tuple contains the *th* component of each input iterable.

Note that `zip` is its own inverse: `zip(*zip(*l, ...)) == l`.

Example

|

– Function.

|

An iterator that yields `(i, value)` where `i` is a counter starting at 1, and `value` is the *th* value from the given iterator. It's useful when you need not only the values over which you are iterating, but also the number of iterations so far. Note that `i` may not be valid for indexing; it's also possible that `iter` has indices that do not start at 1. See the `enumerate` method if you want to ensure that `i` is an index.

Example

|

|

An iterator that accesses each element of the array `array`, returning `array[index]`, where `index` is the index for the element and `start`. This is similar to `array.iter_index()`, except `start` will always be a valid index for `array`.

Specifying `start` ensures that `index` will be an integer; specifying `stop` ensures that `index` will be a `slice`; specifying `mode` chooses whichever has been defined as the native indexing style for array `array`.

Examples

|

Note that `iter_index()` returns `index` as a *counter* (always starting at 1), whereas `iter_indexed()` returns `index` as an *index* (starting at the first linear index of `array`, which may or may not be 1).

See also: `array.iter_indexed()`, `array.iter_indexed_with_mask()`.

– Function.

|

An iterator that yields the same elements as , but starting at the given .

– Function.

|

An iterator that counts forever, starting at and incrementing by .

– Function.

|

An iterator that generates at most the first elements of .

Example

|

– Function.

|

An iterator that generates all but the first elements of .

Example

|

```
|
```

- Function.

```
|
```

An iterator that cycles through forever.

- Function.

```
|
```

An iterator that generates the value forever. If is specified, generates that many times (equivalent to).

Example

```
|
```

- Function.

```
|
```

Returns an iterator over the product of several iterators. Each generated element is a tuple whose th element comes from the th argument iterator. The first iterator changes the fastest. Example:

Example

```
|
```

- Function.

```
|
```

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.

Example

|

– Function.

|

Iterate over a collection elements at a time.

Example

|

Chapter 61

Haciendo Pruebas Unitarias

61.1 Probando Julia Base

Julia está en rápido desarrollo y cuenta con un amplio conjunto de pruebas para verificar su funcionalidad en múltiples plataformas. Si compila Julia desde el origen, puede ejecutar este conjunto de pruebas con `runtests.jl`. En una instalación binaria, puede ejecutar el conjunto de pruebas utilizando `runtests.jl`.

– `Function`.

```
|
```

Run the Julia unit tests listed in `tests`, which can be either a string or an array of strings, using `processors`. (not exported)

61.2 Pruebas Unitarias Básicas

El módulo `Test` proporciona una funcionalidad simple de *realización de pruebas unitarias*. Las pruebas unitarias son una forma de ver si su código es correcto al verificar que los resultados sean los esperados. Puede ser útil asegurarse de que su código aún funcione después de realizar los cambios, y se puede usar al desarrollarlo como una forma de especificar los comportamientos que su código debería tener cuando se complete.

Se pueden realizar pruebas unitarias simples con las macros `@test` y `@testset`:

– `Macro`.

```
|
```

Tests that the expression `expr` evaluates to `val`. Returns a `Bool` if it does, a `String` if it is `nothing`, and an `Exception` if it could not be evaluated.

The `Test` form is equivalent to writing `Test.@test` which can be useful when the expression is a call using infix syntax such as approximate comparisons:

```
|
```

This is equivalent to the uglier test `Test.@test`. It is an error to supply more than one expression unless the first is a call expression and the rest are assignments `(;)`.

– `Macro`.

|

Tests that the expression throws an exception of type . Note that does not support a trailing keyword form.

Por ejemplo, supongamos que queremos comprobar que nuestra nueva función funciona como se esperaba:

|

Si la condición es cierta, se devuelve un :

|

Si la condición es falsa, se devuelve un y se lanza una excepción:

|

Si la condición no pudo ser evaluada porque se lanzó una excepción, lo que ocurre en este caso porque no está definido para símbolos, se devuelve un objeto y se lanza una excepción:

|

Si esperamos que al evaluar una expresión *deberían* lanzarse una excepción, entonces podemos usar para comprobar que esto es lo que ocurre:

|

61.3 Trabajando con Conjuntos de Test

Normalmente, se utiliza una gran cantidad de pruebas para garantizar que las funciones trabajan correctamente sobre distintas entradas. En el caso de que una prueba falle, el comportamiento predeterminado es lanzar una excepción de inmediato. Sin embargo, normalmente es preferible ejecutar el resto de las pruebas primero para obtener una mejor idea de cuántos errores hay en el código que se prueba.

La macro `testset` se puede usar para agrupar las pruebas en *conjuntos*. En un conjunto de pruebas, se ejecutarán varias y al final de su realización se imprimirá un resumen. Si alguna de las pruebas falla o no se puede evaluar debido a un error, el conjunto de prueba arrojará una `TestFailedException`.

– Macro.

```
|
```

Starts a new test set, or multiple test sets if a `loop` is provided.

If no custom testset type is given it defaults to creating a `TestSet`. `testset` records all the results and, if there are any `failures` 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 `TestSet`) 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 `loop` is used then the macro collects and returns a list of the return values of the `test` method, which by default will return a list of the testset objects used in each iteration.

Podemos poner nuestros tests para la función `add` en un conjuntos de tests:

```
|
```

Los conjuntos de pruebas pueden también anidarse:

```
|
```

En el caso de que un conjunto de pruebas anidado no tenga fallos, como pasa aquí, ello se ocultará en el resumen. Si tenemos un test que falle, sólo se mostrarán los detalles para este conjunto de tests que ha fallado:

|

61.4 Otras Macros para Tests

Como los cálculos sobre valores en punto flotante pueden ser imprecisos, podemos realizar comprobaciones de igualdad aproximada usando (donde , se obtiene mediante terminación con tabulador de , es la función) o usar directamente .

|

– Macro.

|

Tests that the call expression returns a value of the same type inferred by the compiler. It is useful to check for type stability.

can be any call expression. Returns the result of if the types match, and an if it finds different types.

|

– Macro.

|

Test whether evaluating results in output that contains the string or matches the regular expression. If is a boolean function, tests whether returns . If is a tuple or array, checks that the error output contains/matches each item in . Returns the result of evaluating .

See also to check for the absence of error output.

– Macro.

```
|
```

Test whether evaluating results in empty output (no warnings or other messages). Returns the result of evaluating .

61.5 Tests Rotos

Si un test falla consistentemente puede ser cambiado para utilizar la macro . Esto denotará el test como Roto () si el test continua fallando y alerta al usuario a través de un si el test tiene éxito.

– Macro.

```
|
```

Indicates a test that should pass but currently consistently fails. Tests that the expression evaluates to or causes an exception. Returns a if it does, or an if the expression evaluates to .

The form works as for the macro.

está también disponible para saltar un test sin evaluación, pero contando el test que se ha saltado en el informe del conjunto de tests. El test no se ejecutará pero da un .

– Macro.

```
|
```

Marks a test that should not be executed but should be included in test summary reporting as . This can be useful for tests that intermittently fail, or tests of not-yet-implemented functionality.

The form works as for the macro.

61.6 Creando Tipos Personalizados

Los paquetes pueden crear sus propios subtipos implementando los métodos y . El subtipo debe tener un constructor de un argumento que tome una cadena de descripción, con todas las opciones pasadas como argumentos palabra clave.

– Function.

```
|
```

Record a result to a testset. This function is called by the infrastructure each time a contained macro completes, and is given the test result (which could be an). This will also be called with an if an exception is thrown inside the test block but outside of a context.

– Function.

```
|
```

Do any final processing necessary for the given testset. This is called by the infrastructure after a test block executes. One common use for this function is to record the testset to the parent's results list, using .

asume la responsabilidad de mantener una pila de conjuntos de pruebas anidados a medida que se ejecutan, pero cualquier acumulación de resultados es responsabilidad del subtipo. Puede acceder a esta pila con los métodos y. Tenga en cuenta que estas funciones no se exportan.

– Function.

|

Retrieve the active test set from the task's local storage. If no test set is active, use the fallback default test set.

– Function.

|

Returns the number of active test sets, not including the default test set

también se asegura de que las invocaciones anidadas utilicen el mismo subtipo que sus padres a menos que se establezca explícitamente. Él no propaga ninguna propiedad del conjunto de pruebas. El comportamiento de herencia de opciones se puede implementar mediante paquetes que usan la infraestructura de pila que proporciona .

La definición de un subtipo básico de 'AbstractTestSet' podría verse así:

|

Y usar este conjunto de test tiene el siguiente aspecto:

|

|

Chapter 62

Interfaz C

– Keyword.

|

Call function in C-exported shared library, specified by tuple, where each component is a string or symbol.

Note that the argument type tuple must be a literal tuple, and not a tuple-valued variable or expression. Alternatively, may also be used to call a function pointer, such as one returned by .

Each to the will be converted to the corresponding , by automatic insertion of calls to . (See also the documentation for each of these functions for further details.) In most cases, this simply results in a call to .

– Function.

|

Obtain a pointer to a global variable in a C-exported shared library, specified exactly as in . Returns a , defaulting to if no argument is supplied. The values can be read or written by or , respectively.

– Function.

|

Generate C-callable function pointer from Julia function. Type annotation of the return value in the callback function is a must for situations where Julia cannot infer the return type automatically.

Examples

|

– Function.

|

Convert to a value of type

In cases where would need to take a Julia object and turn it into a , this function should be used to define and perform that conversion.

Be careful to ensure that a Julia reference to exists as long as the result of this function will be used. Accordingly, the argument to this function should never be an expression, only a variable name or field reference. For example, is acceptable, but is not.

The prefix on this function indicates that using the result of this function after the argument to this function is no longer accessible to the program may cause undefined behavior, including program corruption or segfaults, at any later time.

– Function.

|

Convert to a value of type , typically by calling

In cases where cannot be safely converted to , unlike , may return an object of a type different from , which however is suitable for to handle.

Neither nor should take a Julia object and turn it into a .

– Function.

|

Load a value of type from the address of the th element (1-indexed) starting at . This is equivalent to the C expression .

The prefix on this function indicates that no validation is performed on the pointer to ensure that it is valid. Incorrect usage may segfault your program or return garbage answers, in the same manner as C.

– Function.

|

Store a value of type to the address of the th element (1-indexed) starting at . This is equivalent to the C expression .

The prefix on this function indicates that no validation is performed on the pointer to ensure that it is valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

– Method.

|

Copy 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 prefix on this function indicates that no validation is performed on the pointers and to ensure that they are valid. Incorrect usage may corrupt or segfault your program, in the same manner as C.

– Method.

|

Copy elements from a source array to a destination, starting at offset in the source and in the destination (1-indexed).

The 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.

– Method.

|

Copy all elements from collection to array .

– Method.

|

Copy elements from collection starting at offset , to array starting at offset . Returns .

– Function.

|

Get the native address of an array or string element. Be careful to ensure that a Julia reference to exists as long as this pointer will be used. This function is "unsafe" like .

Calling is generally preferable to this function.

– Method.

|

Wrap a Julia object around the data at the address given by , without making a copy. The pointer element type determines the array element type. is either an integer (for a 1d array) or a tuple of the array dimensions. optionally specifies whether Julia should take ownership of the memory, calling on the pointer when the array is no longer referenced.

This function is labelled "unsafe" because it will crash if is not a valid memory address to data of the requested length.

- Function.

```
|
```

Get the memory address of a Julia object as a `Ptr{Cvoid}`. 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.

- Function.

```
|
```

Convert a `Ptr{Cvoid}` 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.

- 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 `block` syntax as follows:

```
|
|
|
```

This is not needed on worker threads (`Task`) since the `Ptr` 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.

- Function.

```
|
```

Re-enable Ctrl-C handler during execution of a function. Temporarily reverses the effect of `disable_sigint`.

- Function.

```
|
```

Raises a `JuliaError` for `Ptr` with the descriptive string `if` is `!isvalid{Ptr{Cvoid}}(ptr)`

- Type.

```
|
```

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.

– Type.

|

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 `Ptr{T}` itself is referenced.

When passed as a argument (either as a `Ptr{T}` or `Ptr{Cvoid}` type), a `Ptr{T}` object will be converted to a native pointer to the data it references.

There is no invalid (NULL) `Ptr{T}`.

– Type.

|

Equivalent to the native `c-type`.

– Type.

|

Equivalent to the native `c-type` `()`.

– Type.

|

Equivalent to the native `c-type` `()`.

– Type.

|

Equivalent to the native `c-type` `()`.

– Type.

|

Equivalent to the native `c-type` `()`.

– Type.

|

Equivalent to the native `c-type` `()`.

- Type.

|

Equivalent to the native c-type.

- Type.

|

Equivalent to the native c-type.

- Type.

|

Equivalent to the native c-type ().

- Type.

|

Equivalent to the native c-type ().

- Type.

|

Equivalent to the native c-type ().

- Type.

|

Equivalent to the native c-type ().

- Type.

|

Equivalent to the native c-type ().

- Type.

|

Equivalent to the native c-type.

– Type.

|

Equivalent to the native c-type ().

– Type.

|

Equivalent to the native c-type ().

– Type.

|

Equivalent to the native c-type ().

– Type.

|

Equivalent to the native c-type ().

Chapter 63

Interfaz LLVM

– Function.

|

Call LLVM IR string in the first argument. Similar to an LLVM function 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 to will be converted to the corresponding , by automatic insertion of calls to . (see also the documentation for each of these functions for further details). In most cases, this simply results in a call to .

See for usage examples.

Chapter 64

Librería Estándar C

– Function.

|

Call from the C standard library.

– Function.

|

Call from the C standard library.

– Function.

|

Call from the C standard library.

See warning in the documentation for regarding only using this on memory originally obtained from .

– Function.

|

Call from the C standard library. Only use this on memory obtained from , not on pointers retrieved from other C libraries. objects obtained from C libraries should be freed by the free functions defined in that library, to avoid assertion failures if multiple libraries exist on the system.

– Function.

|

Get the value of the C library's . If an argument is specified, it is used to set the value of .

The value of is only valid immediately after a to a C library routine that sets it. Specifically, you cannot call at the next prompt in a REPL, because lots of code is executed between prompts.

- Function.

|

Convert a system call error code to a descriptive string

- Function.

|

Call the Win32 function [only available on Windows].

- Function.

|

Convert a Win32 system call error code to a descriptive string [only available on Windows].

- Method.

|

Converts a struct to a number of seconds since the epoch.

- Function.

|

Convert time, given as a number of seconds since the epoch or a , to a formatted string using the given format. Supported formats are the same as those in the standard C library.

- Function.

|

Parse a formatted time string into a 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 to convert it to seconds since the epoch, the field should be filled in manually. Setting it to will tell the C library to use the current system settings to determine the timezone.

- Type.

|

Convert a number of seconds since the epoch to broken-down format, with fields , , , , , , and .

- Function.

|

Flushes the C and streams (which may have been written to by external C code).

Chapter 65

Enlazador Dinámico

The names in are not exported and need to be called e.g. as .

– Function.

|

Load a shared library, returning an opaque handle.

The extension given by the constant (, , or) can be omitted from the string, as it is automatically appended if needed. If is not an absolute path name, then the paths in the array are searched for , followed by the system load path.

The optional flags argument is a bitwise-or of zero or more of , , , , , , and . 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 flags are while on other platforms the defaults are . An important usage of these flags is to specify non 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 allows the library's symbols to be available for usage in other shared libraries, addressing situations where there are dependencies between shared libraries.

– Function.

|

Similar to , except returns a pointer instead of raising errors.

– Constant.

|

Enum constant for . See your platform man page for details, if applicable.

- Function.

|

Look up a symbol from a shared library handle, return callable function pointer on success.

- Function.

|

Look up a symbol from a shared library handle, silently return pointer on lookup failure.

- Function.

|

Close shared library referenced by handle.

- Constant.

|

File extension for dynamic libraries (e.g. dll, dylib, so) on the current platform.

- Function.

|

Searches for the first library in the paths in the list, 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 and used as the library name in future 's. On failure, it returns the empty string.

- Constant.

|

When calling , the paths in this list will be searched first, in order, before searching the system locations for a valid library handle.

Chapter 66

Profiling

- Macro.

|

runs your expression while taking periodic backtraces. These are appended to an internal buffer of backtraces.

The methods in are not exported and need to be called e.g. as .

- Function.

|

Clear any existing backtraces from the internal buffer.

- Function.

|

Prints profiling results to (by default,). If you do not supply a vector, the internal buffer of accumulated backtraces will be used.

The keyword arguments can be any combination of:

- - Determines whether backtraces are printed with (default,) or without () indentation indicating tree structure.
- - If , backtraces from C and Fortran code are shown (normally they are excluded).
- - If (default), instruction pointers are merged that correspond to the same line of code.
- - Limits the depth higher than in the format.
- - Controls the order in format. (default) sorts by the source line, whereas sorts in order of number of collected samples.
- - Limits frames that exceed the heuristic noise floor of the sample (only applies to format). A suggested value to try for this is 2.0 (the default is 0). This parameter hides samples for which , where is the number of samples on this line, and is the number of samples for the callee.
- - Limits the printout to only those lines with at least occurrences.

|

Prints profiling results to `io.stdout`. This variant is used to examine results exported by a previous call to `Profile.print`. Supply the vector of backtraces and a dictionary of line information.

See `Profile.print` for an explanation of the valid keyword arguments.

– `Profile.set_backtraces` Function.

|

Configure the `backtraces` between backtraces (measured in seconds), and the number 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 `[:backtraces, :instruction_pointers]`.

– `Profile.backtraces` Function.

|

Returns a reference to the internal buffer of backtraces. Note that subsequent operations, like `Profile.print`, can affect `backtraces` unless you first make a copy. Note that the values in `backtraces` 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; `Profile.print` may be a better choice for most users.

– `Profile.export` Function.

|

"Exports" profiling results in a portable format, returning the set of all backtraces (`Profile.backtraces`) and a dictionary that maps the (session-specific) instruction pointers in `backtraces` to values that store the file name, function name, and line number. This function allows you to save profiling results for future analysis.

– `Profile.callers` Function.

|

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` obtained from `Profile.backtraces`; otherwise, the current internal profile buffer is used.

– `Profile.clear` Function.

|

Clears any stored memory allocation data when running Julia with `Profile.clear`. Execute the command(s) you want to test (to force JIT-compilation), then call `Profile.clear`. Then execute your command(s) again, quit Julia, and examine the resulting files.

Chapter 67

StackTraces

– Type.

|

Stack information representing execution context, with the following fields:

- The name of the function containing the execution context.
- The MethodInstance containing the execution context (if it could be found).
- The path to the file containing the execution context.
- The line number in the file containing the execution context.
- True if the code is from C.
- True if the code is from an inlined frame.
- Representation of the pointer to the execution context as returned by .

– Type.

|

An alias for provided for convenience; returned by calls to and .

– Function.

|

Returns a stack trace in the form of a vector of s. (By default stacktrace doesn't return C functions, but this can be enabled.) When called without specifying a trace, first calls .

– Function.

|

Returns the stack trace for the most recent error thrown, rather than the current execution context.

Los siguientes métodos y tipos de no son exportados y, por tanto, deben ser prefijados en sus invocaciones. Por ejemplo, .

– Function.

|

Given a pointer to an execution context (usually generated by a call to), looks up stack frame context information. Returns an array of frame information for all functions inlined at that point, innermost function first.

– Function.

|

Takes a (a vector of) and a function name (a) and removes the specified by the function name from the (also removing all frames above the specified function). Primarily used to remove functions from the prior to returning it.

|

Returns the with all s from the provided removed.

Chapter 68

Soporte SIMD

El tipo `__m128` está pensado para construir librerías de operaciones SIMD operations. El uso práctico de él requiere usar `<immintrin.h>`. El tipo está definido como:

```
|
```

Él tiene una regla de compilación especial: una tupla homogénea de `__m128` se corresponde con un tipo LLVM cuando `__m128` es un tipo de bits primitivo y la longitud de la tupla está en el conjunto {2,4,8,16}.

En `clang`, el compilador *podría* automáticamente vectorizar operaciones sobre tales tuplas. Por ejemplo, el siguiente programa, cuando se compila con `clang -x86_64 -O3` genera dos instrucciones de adición SIMD (`__m128`) sobre los sistemas x86:

```
|
```

Sin embargo, dado que no se puede confiar en la vectorización automática, el uso futuro se realizará principalmente a través de bibliotecas que usen `<immintrin.h>`.

Part V

Developer Documentation

Chapter 69

Reflection and introspection

Julia provides a variety of runtime reflection capabilities.

69.1 Module bindings

The exported names for a module are available using `exports`, which will return an array of `Symbol` elements representing the exported bindings. `names` returns symbols for all bindings in a module, regardless of export status.

69.2 `DataType` fields

The names of `Field{...}` fields may be interrogated using `fieldnames`. For example, given the following type, `fieldnames` returns an array of `String` elements representing the field names:

```
struct Point
    x::Float64
    y::Float64
end
```

The type of each field in a `Point` object is stored in the `field_type` field of the `Point` variable itself:

```
Point{Float64,Float64}.field_type
```

While `Point` is annotated as an `Abstract{...}`, `Point` was unannotated in the type definition, therefore `field_type` defaults to the `Any` type.

Types are themselves represented as a structure called `Field{...}`:

```
Field{Point{Float64,Float64},:x,Float64}
```

Note that `fieldnames` gives the names for each field of `Field{...}` itself, and one of these fields is the `field_type` field observed in the example above.

69.3 Subtypes

The *direct* subtypes of any type may be listed using `subtypes`. For example, the abstract `Abstract{...}` has four (concrete) subtypes:

Any abstract subtype will also be included in this list, but further subtypes thereof will not; recursive application of `inspect` may be used to inspect the full type tree.

69.4 `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. `isCCompatible` returns true if `DataType` is stored with C-compatible alignment. `offsetOf` returns the (byte) offset for field `i` relative to the start of the type.

69.5 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 `findMethod`.

69.6 Expansion and lowering

As discussed in the [Metaprogramming](#) section, the `unquote` function gives the unquoted and interpolated expression `()` form for a given macro. To use `unquote`, the expression block itself (otherwise, the macro will be evaluated and the result will be passed instead!). For example:

The functions `display` and `depth-nested` are used to display S-expr style views and depth-nested detail views for any expression.

Finally, the `inspect` function gives the `inspect` form of any expression and is of particular interest for understanding both macros and top-level statements such as function declarations and variable assignments:

69.7 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 `lower`, and the type-inferred form is available using `lowerTypeInferred`. `highlight` adds highlighting to the output of `lower` (see [highlight](#)).

Closer to the machine, the LLVM intermediate representation of a function may be printed using `printLLVM`, and finally the compiled machine code is available using `printMachineCode` (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:

|

(likewise , , and)

Chapter 70

Documentation of Julia's Internals

70.1 Initialization of the Julia runtime

How does the Julia runtime execute ?

`main()`

Execution starts at `in` .

`calls` to set the C library locale and to initialize the "ios" library (see and [Legacy library](#)).

Next `is` called to process command line options. Note that `only` deals with options that affect code generation or early initialization. Other options are handled later by `in` .

`stores` command line options in the `global struct`.

`julia_init()`

`in` is called by `and` calls `in` .

`begins` by calling `again` (it does nothing the second time).

`is` called to zero the signal handler mask.

`searches` configured paths for the base system image. See [Building the Julia system image](#).

`sets` up allocation pools and lists for weak refs, preserved values and finalization.

`loads` and initializes a pre-compiled femtolisp image containing the scanner/parser.

`creates` type description objects for the `built-in types defined in` . e.g.

`creates` the `object`; initializes the `global struct`; and sets `to` the root task.

`initializes` the `LLVM library`.

`initializes` 8-bit serialization tags for 256 frequently used `values`. The serialization mechanism uses these tags as shorthand (in lieu of storing whole objects) to save storage space.

If there is no sysimg file () then the and modules are created and is evaluated:

creates the Julia module.

creates a new Julia module containing constant symbols. These define an integer code for each *intrinsic function*. translates these symbols into LLVM instructions during code generation.

hooks C functions up to Julia function symbols. e.g. the symbol is bound to C function pointer by calling .

creates the global "Main" module and sets .

Note: *then sets* . is an alias of at this point, so the set by above is overwritten.

calls which repeatedly calls to execute . <!-- TODO – drill down into eval? -->

initializes global C pointers to Julia globals defined in .

pre-allocates global boxed integer value objects for values up to 1024. This speeds up allocation of boxed ints later on. e.g.:

```
|
```

iterates over the looking for values and sets the type name's module prefix to .

does "using Base" in the "Main" module.

Note: now reverts to as it was before being set to above.

Platform specific signal handlers are initialized for (OSX, Linux), and (Windows).

Other signals (and) are hooked up to which prints a backtrace.

calls for each deserialized module to run the function.

Finally is hooked up to and calls .

then returns *back to in* and calls .

sysimg

If there is a sysimg file, it contains a pre-cooked image of the and modules (and whatever else is created by). See [Building the Julia system image](#).

deserializes the saved sysimg into the current Julia runtime environment and initialization continues after below...

Note: (and in general) uses the [Legacy library](#).

true_main()

loads the contents of into .

If a "program" file was supplied on the command line, then calls which calls which repeatedly calls to execute the program.

However, in our example (), looks up and executes it.

Base._start

calls which calls to create an expression object and to execute it.

julia_save()

Finally, `julia_save()` calls `save_image()`, which if requested on the command line, saves the runtime state to a new system image. See `save_image()` and `save_image!`.

70.2 Julia ASTs

Julia has two representations of code. First there is a surface syntax AST returned by the parser (e.g. the `AST` function), and manipulated by macros. It is a structured representation of code as it is written, constructed by `parse()` 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 `lowered_ast()`.

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 `node_type` field, and an `args` field which is a `Vector{Expr}` of subexpressions.
- `Slot`
Identifies arguments and local variables by consecutive numbering. `Slot` is an abstract type with subtypes `ArgSlot` and `LocalSlot`. Both types have an integer-valued `slot_index` field giving the slot index. Most slots have the same type at all uses, and so are represented with `Slot`. The types of these slots are found in the `slot_type` field of their `parent` object. Slots that require per-use type annotations are represented with `TypedSlot`, which has a `slot_type` field.
- `MethodIR`
Wraps the IR of a method.
- `Line`
Contains a single number, specifying the line number the next statement came from.
- `BranchTarget`
Branch target, a consecutively-numbered integer starting at 0.
- `UnconditionalBranch`
Unconditional branch.
- `SymbolRef`
Wraps an arbitrary value to reference as data. For example, the function `getfield` contains a `SymbolRef` whose `value` field is the symbol `getfield`, in order to return the symbol itself instead of evaluating it.
- `GlobalRef`
Refers to global variable `name` in module `module`.
- `StaticSingleAssignment`
Refers to a consecutively-numbered (starting at 0) static single assignment (SSA) variable inserted by the compiler.
- `Reset`
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 field of `s` in lowered form.

- - Function call (dynamic dispatch). `is` is the function to call, `args` are the arguments.
- - Function call (static dispatch). `is` is the `MethodInstance` to call, `args` are the arguments (including the function that is being called, at `at`).
- - Reference a static parameter by index.
- - Line number and file name metadata. Unlike a `Location`, can also contain a file name.
- - Conditional branch. If `is` is false, goes to label identified in `else_label`.
- - Assignment.
- - 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 `eval(f::Function, s::Symbol)`. 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 is created and 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:
 - `name::Symbol`
 A function name, or `nothing` 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 `nothing`, it means a method is being added strictly by type, `eval(f::Function, s::Symbol, t::Type)`.
 - `argtypes::Vector{DataType}`
 A vector of argument type data. `is` is a vector of the argument types, and `vars` is a vector of type variables corresponding to the method's static parameters.
 - `body::Expr`
 A vector 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 `Expr` expression.
 - `staged::Bool`
`is` or `vars`, identifying whether the method is staged (`is`).
- - Declares a (global) variable as constant.

- Has no arguments; simply yields the value .
- Allocates a new struct-like object. First argument is the type. The 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 expressions can easily segfault.
- Returns its argument as the value of the enclosing function.
- Yields the caught exception inside a block. This is the value of the run time system variable .
- Enters an exception handler (). is the label of the catch block to jump to on error.
- Pop exception handlers. is the number of handlers to pop.
- Controls turning bounds checks on or off. A stack is maintained; if the first argument of this expression is true or false (means bounds checks are disabled), it is pushed onto the stack. If the first argument is , the stack is popped.
- Indicates the beginning or end of a section of code that performs a bounds check. Like , a stack is maintained, and the second argument can be one of: , , or .
- Part of the implementation of quasi-quote. The argument is a surface syntax AST that is simply copied recursively and returned at run time.
- Metadata. 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:
 - and : Inlining hints.
 - : enters a sequence of statements from a specified source location.
 - * specifies a filename, as a symbol.
 - * optionally specifies the name of an (inlined) function that originally contained the code.
 - : returns to the source location before the matching .

Method

A unique'd container describing the shared metadata for a single method.

- , , , ,
Metadata to uniquely identify the method for the computer and the human.

- Cache of other methods that may be ambiguous with this one.
- Cache of all `MethodInstance` ever created for this `Method`, used to ensure uniqueness. Uniqueness is required for efficiency, especially for incremental precompile and tracking of method invalidation.
- The original source code (usually compressed).
- 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.
- `,,,`
Descriptive bit-fields for the source code of this `Method`.
- `/`
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.

- The primary key for this `MethodInstance`. Uniqueness is guaranteed through a lookup.
- The `parent` that this function describes a specialization of. Or, if this is a top-level `Lambda` that is not part of a `Method`.
- The values of the static parameters in `parameters` indexed by `static_parameter_index`. For the `parent` at `static_parameter_index`, this is the empty `Dict{Symbol, Any}`. But for a runtime `MethodInstance` from the cache, this will always be defined and indexable.
- The inferred return type for the `return_type` field, which (in most cases) is also the computed return type for the function in general.
- 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 `return_type`), or it could be set to `nothing` to just indicate `return_type` is inferred.
- The generic `jlcall` entry point.
- The ABI to use when calling `parent`. Some significant ones include:
 - 0 - Not compiled yet
 - 1 - `JL_CALLABLE`

- 2 - Constant (value stored in)
 - 3 - With Static-parameters forwarded
 - 4 - Run in interpreter
- /
- The range of world ages for which this method instance is valid to be called.

CodeInfo

A temporary container for holding lowered source code.

-
- An array of statements
-
- An array of symbols giving the name of each slot (argument or local variable).
-
- An array of types for the slots.
-
- A 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.
-
- Either an array or an .
- If an , it gives the number of compiler-inserted temporary locations in the function. If an array, specifies a type for each location.

Boolean properties:

-
- Whether this has been produced by type inference.
-
- Whether this should be inlined.
-
- Whether this should propagate when inlined for the purpose of eliding blocks.
-
- 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 entirely of s 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 corresponds to in Julia.

Calls

Input	AST

syntax:

parses as .

Operators

Most uses of operators are just function calls, so they are parsed with the head . 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.

Input	AST

Bracketed forms

Macros

Strings

Doc string syntax:

Input	AST

Input	AST

Input	AST

|

parses as .

Imports and such

Input	AST

Numbers

Julia supports more number types than many scheme implementations, so not all numbers are represented directly as scheme numbers in the AST.

Input	AST

Block forms

A block of statements is parsed as .

If statement:

```
|
```

parses as:

```
|
```

A loop parses as .

A loop parses as . If there is more than one iteration specification, they are parsed as a block: .

and are parsed as 0-argument expressions and .

is parsed as .

A basic function definition is parsed as . A more complex example:

```
|
```

parses as:

```
|
```

Type definition:

```
|
```

parses as:

```
|
```

The first argument is a boolean telling whether the type is mutable.

blocks parse as . If no variable is present after , is . If there is no clause, then the last argument is not present.

70.3 Más sobre tipos

Si ha usado Julia durante un tiempo, comprenderá el papel fundamental que juegan los tipos. Aquí intentamos meterlos debajo del capó, centrándonos fundamentalmente en los [Tipos Paramétricos](#).

Tipos y conjuntos (y y /)

Tal vez sea más sencillo concebir el sistema de tipos de Julia en términos de conjuntos. Aunque los programas pueden manipular los valores individuales, un tipo se refiere a un conjunto de valores. Esto no es la misma cosa que una colección; por ejemplo, un `Vector{Int}` de valores es en sí mismo un solo valor de tipo `Vector{Int}`. En lugar de ello, un tipo describe un conjunto de *posible* valores, expresando incertidumbre sobre qué valor tenemos.

Un tipo *concreto* describe el conjunto de valores cuya etiqueta directa, tal y como es obtenida por la función `typeof` es . Un tipo *abstracto* describe un conjunto de valores posiblemente más grande.

`Any` describe el universo completo de valores posibles. `Any` es un subconjunto de `Any` que incluye `Any`, `Any`, y otros tipos concretos. Internamente, Julia también hace un uso intensivo de otro tipo conocido como `Nothing`, que puede también ser escrito como `Nothing`. Esto corresponde al conjunto vacío.

Los tipos de Julia soportan las operaciones estándar de la teoría de conjuntos: uno puede preguntar si `A` es un "subconjunto" de `B` con `issubset(A, B)`. Análogamente, uno intersecta dos tipos usando `intersect(A, B)`, realiza su unión con `union(A, B)`, y calcula un tipo que contiene su unión con `Union{A, B}`.

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 methods be 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 `isless`, 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` has two parameters as in `obj{T, S}`. If we did not know the element type, we could write `UnionAll{T, S}`, which is the union of `obj{T, S}` for all values of `T` and `S`.

Such a type is represented by a `UnionAll` object, which contains a variable (in this example, of type `Any`), and a wrapped type (in this example, `obj{T, S}`).

Consider the following methods:

```

function Base.show(io::IO, t::UnionAll)
    print(io, "UnionAll{")
    show(io, t.var)
    print(io, ", ")
    show(io, t.wrapped)
    print(io, "}")
end

```

The signature of `show` is a `UnionAll` type wrapping a tuple type. All but `show` can be called with `io`; all but `show` can be called with `t`.

Let's look at these types a little more closely:

```

struct UnionAll{T, S}
    var::T
    wrapped::S
end

```

This indicates that `UnionAll` actually names a `UnionAll` type. There is one `UnionAll` type for each parameter, nested. The syntax `UnionAll{T, S}` is equivalent to `UnionAll{T, UnionAll{S}}`; 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; `UnionAll{T}` gives a type equivalent to `UnionAll{T, Any}`.

A `UnionAll` 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 `lower` and `upper`). The symbol `UnionAll` is purely cosmetic. Internally, `UnionAll`s 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 `UnionAll`s manually:

```

UnionAll{Any, obj{Int, String}}

```

There are convenience versions that allow you to omit any of these arguments except the `Symbol`.

The syntax `UnionAll{T}` is lowered to

```

UnionAll{Any, obj{Int, String}}

```

so it is seldom necessary to construct a `UnionAll` manually (indeed, this is to be avoided).

Free variables

The concept of a *free* type variable is extremely important in the type system. We say that a variable `x` is free in type `T` if `T` does not contain the `x` that introduces variable `x`. For example, the type `Array{T, N}` has no free variables, but the `T` part inside of it does have a free variable, `T`.

A type with free variables is, in some sense, not really a type at all. Consider the type `Array{Array{T, N}, N}`, which refers to all homogeneous arrays of arrays. The inner type `Array{T, N}`, 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, N}` cannot refer to just any old array. One could say that `Array{T, N}` effectively "occurs" multiple times, and `T` must have the same value each "time".

For this reason, the function `isfree` 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 types are functionally equivalent, yet print differently:

```

Array{Array{T, N}, N}
Array{Array{T, N}, N}

```

These can be distinguished by examining the `name` field of the type, which is an object of type `TypeName`:

```

Array{Array{T, N}, N}.name
Array{Array{T, N}, N}.name

```

In this case, the relevant field is `name`, which holds a reference to the top-level type used to make new `Array` types.

The `field` of `points` points to itself, but for `it` points back to the original definition of the type.

What about the other fields? `assigns` an integer to each type. To examine the `field`, it's helpful to pick a type that is less heavily used than `Array`. Let's first create our own type:

(The cache is pre-allocated to have length 8, but only the first two entries 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 `,` the type has to be able to accommodate any tuple. Let's check the parameters:

Unlike other types, tuple types are covariant in their parameters, so this definition permits `to match` any type of tuple:

However, if a variadic `()` tuple type has free variables it can describe different kinds of tuples:

Notice that when `T` is free with respect to the `U` type (i.e. its binding `U` type is outside the `T` type), only one `U` value must work over the whole type. Therefore a heterogeneous tuple does not match.

Finally, it's worth noting that `T` is distinct:

What is the "primary" tuple-type?

so `T` is indeed the primary type.

Diagonal types

Consider the type `T`. A method with this signature would look like:

According to the usual interpretation of a `T` type, this `T` ranges over all types, including `T`, so this type should be equivalent to `T`. 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(T)`, it's not clear what `T` should be. It could be `T`, or perhaps `T`. Intuitively, we expect the declaration `f::T` to mean `f::T`. To make sure that invariant holds, we need `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 `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 `T` and `U` types occur between an occurrence of a variable and the type that introduces it.) Such variables are called "diagonal variables" or "concrete variables".

So for example, `forall T. T -> T` can be seen as `forall T. T -> T`, where `T` ranges over all concrete types. This gives rise to some interesting subtyping results. For example `forall T. T -> T` is not a subtype of `forall T. T -> T`, because it includes some types like `forall T. T -> T` where the two elements have different types. `forall T. T -> T` and `forall T. T -> T` have the non-trivial intersection `forall T. T -> T`. However, `forall T. T -> T` is a subtype of `forall T. T -> T`, because in that case `T` occurs only once and so is not diagonal.

Next consider a signature like the following:

```
|
```

In this case, `T` occurs in invariant position inside `forall T. T -> T`. That means whatever type of array is passed unambiguously determines the value of `T` -- we say `forall T. T -> 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 `T` and `U` 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

```
|
```

to clarify whether `T` and `U` 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 `T` if `T` and `U` can have different types.

The next complication is the interaction of unions and diagonal variables, e.g.

```
|
```

Consider what this declaration means. `T` has type `T`. `U` then can have either the same type `T`, or else be of type `U`. So all of the following calls should match:

```
|
```

These examples are telling us something: when `T` is `T`, there are no extra constraints on `T`. It is as if the method signature had `T`. 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 `T` has type `T`, we don't need to use the `T` in `T`, so `T` does not "occur". Indeed, we have the following type equivalence:

```
|
```

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 `in` and `cov` (in `Env`) for each variable in the environment, tracking the number of invariant and covariant occurrences, respectively. A variable is diagonal when `in` is 0.

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 type whose variable is diagonal, we look at the final values of the bounds. Since the variable must be concrete, a contradiction occurs if its lower bound could not be a subtype of a concrete type. For example, an abstract type like `Abstract{A}` cannot be a subtype of a concrete type, but a concrete type like `Concrete{A}` can be, and the empty type `{}{A}` can be as well. If a lower bound fails this test the algorithm stops with the answer `Union{}`.

For example, in the problem `isconcretetype{A} <: Concrete{A}`, we derive that this would be true if `Abstract{A}` were a supertype of `Concrete{A}`. However, `Abstract{A}` is an abstract type, so the relation does not hold.

This concreteness test is done by the function `isconcretetype`. Note that this test is slightly different from `isconcretetype{A}`, since it also returns `Union{}` for `Abstract{A}`. 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, `isconcretetype{A} <: Concrete{A}` is equivalent to the concrete type `Concrete{A}` only if both the upper and lower bounds of `A` equal `Concrete{A}`. 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 `src/typemap.c` and `src/typemap.h`. 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:

```
|
```

and then set a breakpoint in `isconcretetype`. Once this breakpoint gets triggered, you can set breakpoints in other functions.

As a warm-up, try the following:

```
|
```

We can make it more interesting by trying a more complex case:

```
|
```

Subtyping and method sorting

The functions `isconcretetype` and `isabstracttype` are used for imposing a partial order on functions in method tables (from most-to-least specific). Specificity is strict; if `f` is more specific than `g`, then `f` does not equal `g` and `g` is not more specific than `f`.

If `f` is a strict subtype of `g`, then it is automatically considered more specific. From there, `isconcretetype` employs some less formal rules. For example, `isconcretetype{A}` is sensitive to the number of arguments, but `isconcretetype` may not be. In particular, `isconcretetype{A}` is more specific than `isconcretetype`, even though it is not a subtype. (Of `isconcretetype{A}` and `isconcretetype`, neither is more specific than the other.) Likewise, `isconcretetype{A}` is not a subtype of `isconcretetype`, but it is considered more specific. However, `isconcretetype{A}` does get a bonus for length: in particular, `isconcretetype{A}` is more specific than `isconcretetype`.

If you're debugging how methods get sorted, it can be convenient to define the function:

which allows you to test whether `tuple type` is more specific than `tuple type`.

70.4 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:

Each `jl_value_t` struct is contained in a `jl_object_t` struct that contains metadata information about the Julia object, such as its type and garbage collector (gc) reachability:

The type of any Julia object is an instance of a `jl_object_t` object. The `jl_typeof` function can be used to query for it:

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 `jl_get_field` methods:

If the field types are known, a priori, to be all pointers, the values can also be extracted directly as an array access:

As an example, a "boxed" `jl_value_t` is stored as follows:

This object is created by `jl_boxed`. 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 `jl_egal` function in Julia), should instead be used to compare two unknown objects for equivalence:

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 `Ptr{...}`, the Julia garbage collector must be notified also:

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](#) `src/abi.h`. The corresponding global objects are created by `init_globals` in `src/init.c`.

Garbage collector mark bits

The garbage collector uses several bits from the metadata portion of the `Object` to track each object in the system. Further details about this algorithm can be found in the comments of the [garbage collector implementation in](#) `src/gc.c`.

Object allocation

Most new objects are allocated by `new_obj`:

Although, objects can be also constructed directly from memory:

And some objects have special constructors that must be used instead of the above functions:

Types:

While these are the most commonly used options, there are more low-level constructors too, which you can find declared in `src/abi.h`. These are used in `src/init.c` to create the initial types needed to bootstrap the creation of the Julia system image.

Tuples:

The representation of tuples is highly unique in the Julia object representation ecosystem. In some cases, a `Tuple{...}` object may be an array of pointers to the objects contained by the tuple equivalent to:

However, in other cases, the tuple may be converted to an anonymous `Union{...}` type and stored unboxed, or it may not stored at all (if it is not being used in a generic context as a `Union{...}`).

Symbols:

Functions and MethodInstance:

Arrays:

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 [header file](#).

Internal to Julia, storage is typically allocated by (or for the special types):

And at the lowest level, memory is getting allocated by a call to the garbage collector (in), then tagged with its 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 for large objects.

Singleton Types

Singleton types have only one instance and no data fields. Singleton instances have a size of 0 bytes, and consist only of their metadata. e.g. .

See [Singleton Types](#) and [Nothingness and missing values](#)

70.5 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, , , 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 .
2. The C function `main` gets called. This function processes the command line arguments, filling in the `argv` struct and setting the variable `argc`. It then initializes Julia (by calling `jl_init`, which may load a previously compiled `sysimg`). Finally, it passes off control to Julia by calling `jl_main`.
3. When `jl_main` 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` gets invoked to load the file and `jl_parse` it. Each fragment of code is then passed to `jl_eval` to execute.
6. Each fragment of code (or AST), is handed off to `jl_eval` to turn into results.
7. `jl_eval` takes each code fragment and tries to run it in `jl_eval`.
8. `jl_eval` decides whether the code is a "toplevel" action (such as `println` or `exit`), which would be invalid inside a function. If so, it passes off the code to the toplevel interpreter.
9. `jl_eval` then `expands` the code to eliminate any macros and to "lower" the AST to make it simpler to execute.
10. `jl_eval` then uses some simple heuristics to decide whether to JIT compile the AST or to interpret it directly.
11. The bulk of the work to interpret code is handled by `jl_interpret`.
12. If instead, the code is compiled, the bulk of the work is handled by `jl_compile`. 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 `main` method returns.
14. Just before exiting, `jl_main` calls `jl_atexit_hook`. This calls `jl_atexit_hook` (which calls any functions registered to `jl_atexit_hook` inside Julia). Then it calls `jl_atexit_hook`. Finally, it gracefully cleans up all `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 `src/flisp`. The code in `src/flisp` handles this handoff on the Julia side.

The other relevant files at this stage are `src/ast`, which handles tokenizing Julia code and turning it into an AST, and `src/ast`, which handles transforming complex AST representations into simpler, "lowered" AST representations which are more suitable for analysis and execution.

Macro Expansion

When `jl_eval` encounters a macro, it expands that AST node before attempting to evaluate the expression. Macro expansion involves a handoff from `jl_eval` (in Julia), to the parser function `jl_parse` (written in `femtolisp`) to the Julia macro itself (written in `femtolisp` - what else - Julia) via `jl_parse`, and back.

Typically, macro expansion is invoked as a first step during a call to `jl_eval`, although it can also be invoked directly by a call to `jl_parse`.

Type Inference

Type inference is implemented in Julia by `in`. 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.
- **box**
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 `to` 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 `in`.

On demand, a Julia method is converted into a native function by the function `jitcall`. (note, when using the MCJIT (in LLVM v3.4+), each function must be JIT into a new module.) This function recursively calls `jitcall` 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, `ccall` knows how to inline many of the primitive functions (defined in `ccall.jl`) for various combinations of argument types.

Other parts of codegen are handled by various helper files:

- `ccall.jl`
Handles backtraces for JIT functions
- `ccall.jl`
Handles the `ccall` and `llvmcall` FFI, along with various `ccall` files
- `ccall.jl`
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.

System Image

The system image is a precompiled archive of a set of Julia files. The `sysimage.jl` file distributed with Julia is one such system image, generated by executing the file `sysimage.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 `Base`, `Core`, and `Libc` modules (and whatever else was in the environment at the end of bootstrapping). This serializer/deserializer is implemented by `Serialization.jl` / `in`.

If there is no `sysimage.jl` file, this also implies that `sysimage.jl` was given on the command line, so the final result should be a new `sysimage.jl` file. During Julia initialization, minimal `Base` and `Core` modules are created. Then a file named `sysimage.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 "sysimage" file for use as a starting point for a future Julia run.

70.6 Calling Conventions

Julia uses three calling conventions for four distinct purposes:

Name	Prefix	Purpose
Native		Speed via specialized signatures
JL Call		Wrapper for generic calls
JL Call		Builtins
C ABI		Wrapper callable from C

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 small return values is returned as LLVM return values. A large return values is returned via the "structure return" () convention, where the caller provides a pointer to a return slot.

An argument or return values thta 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 . The convention uses exactly 3 parameters:

- - Julia representation of function that is being applied
- - pointer to array of pointers to boxes
- - 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.

70.7 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.

Otherwise, they will be emitted as literal constants.

To emit one of these objects, call . 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 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 table. Like globals, this allows the deserializer to reference them by index.

Note that functions are handled separately, with names, via the usual symbol resolution mechanism in the linker.

Note too that functions are also handled separately, via a manual GOT and Procedure Linkage Table (PLT).

Representation of Intermediate Values

Values are passed around in a 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: (for immediate values) and (for pointers to values).

The function `convert` can transform between any two types. It returns an R-value with `set` to `0`. 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.

By contrast `unbox` will change `to`, only if it can 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 `isbits`. The selector is fixed-size as `126`, 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 `is` is actually a tagged heap-allocated `is`, and needs to be treated as normal for a boxed object rather than as a tagged union.

The high bit of the selector (`126`) can be tested to determine if the `is` is actually a heap-allocated (`is`) 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 `is` is an exact test for the type, if the value can be represented by a tag – it will never be marked `is`. It is not necessary to also test the type-tag when testing `is`.

The `memory` region may be allocated at *any* size. The only constraint is that it is big enough to contain the data currently specified by `memory`. 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 `cc` 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 `args` and `ret` 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.

The total logic for this is implemented by `jl_pass_struct` and `jl_pass_vector`.

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.

70.8 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 `s`. `A` describes a family of parameterized types. For example `Array{T, N}` and `Array{T, 1}` share the same `type` name object.

All objects in Julia are potentially callable, because every object has a `type`, which in turn has a `method_table`.

Function calls

Given the call `f(x, y)`, the following steps are performed: first, the method table to use is accessed as `f.method_table`. Second, an argument tuple type is formed, `typeof(f, x, y)`. Note that the type of the function itself is the first element. This is because the type might have parameters, and so needs to take part in dispatch. This tuple type is looked up in the method table.

This dispatch process is performed by `jl_call_method`, 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:

```
|
```

This entry point for the same functionality accepts the function separately, so the `args` array does not contain the function:

```
|
```

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_add_method` implements this operation. `jl_add_method` 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 `gfun` 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 `gfun` is `gfun`. By convention, the name of the created *type* is the same as the function name, with a `gfun` prepended. So `gfun` is `gfun_gfun`.

Closures

A closure is simply a callable object with field names corresponding to captured variables. For example, the following code:

```
|
```

is lowered to (roughly):

```
|
```

Constructors

A constructor call is just a call to a type. The type of most types is `Function`, so the method table for `Function` contains most constructor definitions. One wrinkle is the fallback definition that makes all types callable via :

```
|
```

In this definition the function type is abstract, which is not normally supported. To make this work, all subtypes of `Function` (`Function`, `Function{...}`, and `Function{...}`) currently share a method table via special arrangement.

Builtins

The "builtin" functions, defined in the `Base` module, are:

```
|
```

These are all singleton objects whose types are subtypes of `Function`, which is a subtype of `Function`. Their purpose is to expose entry points in the run time that use the "jllcall" calling convention:

```
|
```

The method tables of builtins are empty. Instead, they have a single catch-all method cache entry `(jllcall)` whose `jllcall` `fp` points to the correct function. This is kind of a hack but works reasonably well.

Keyword arguments

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 `kw` 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 `kw` 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 substitute 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:

```
|
```

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:

```
|
```

The second method is an ordinary definition for the original `function`, which handles the case where no keyword arguments are passed:

```
|
```

This simply dispatches to the first method, passing along default values. Finally there is the `kwsorter` definition:

```
|
```

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 `kwsort` fetches (and creates, if necessary) the field `kw`.

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:

```
|
```

is lowered to:

The unpacking procedure represented here as `unpack` actually further unpacks each *element* of `args`, expecting each one to contain two values (a symbol and a value). `unpack` (also in `Base`) fetches the `kwsorter` for the called function. Notice that the original `func` 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 `call` 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 pass in the front end. When `call` sees an argument in the type hierarchy passed to a slot declared as `Function` or `Symbol`, it pretends the slot was declared as `Any` (the "don't specialize" hint). This heuristic seems to be extremely effective in practice.

The next issue concerns the structure of method cache hash tables. Empirical studies show that the vast majority of dynamically-dispatched calls involve 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 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 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.

However this caused a new problem. When many tests are grouped together in a single function, e.g. a single top level expression, or some other test grouping function, that function could have a very large number of exception handlers. This triggered a kind of dataflow analysis worst case, where type inference spun around for minutes enumerating possible paths through the forest of handlers. This was fixed by simply bailing out of type inference when it encounters more than some number of handlers (currently 25). Presumably no performance-critical function will have more than 25 exception handlers. If one ever does, I'm willing to raise the limit to 26.

A minor issue occurs during the bootstrap process due to storing all constructors in a single method table. In the second bootstrap step, where `Base` is compiled using `Base`, constructors for `Base`'s types remain in the table, so there are still references to the old inference module and `Base` is 2x the size it should be. This was fixed in `Base` by filtering definitions from "replaced

modules" out of method tables and caches before saving a system image. A "replaced module" is one that satisfies the condition – in other words, some newer module has taken its name and place.

Another type inference worst case was triggered by the following code from the [QuadGK.jl package](#), formerly part of Base:

```

function f(x)
    f(x)
end

```

This code has a 3-way tail recursion, where each call wraps the current function argument in a different new closure. Inference must consider 3^n (where n is the call depth) possible signatures. This blows up way too quickly, so logic was added to to immediately widen any argument that is a subtype of and that grows in depth down the stack.

70.9 Base.Cartesian

El módulo (no exportado) proporciona macros que facilitan escribir algoritmos multidimensionales. Es deseable que, a largo plazo, este módulo no sea necesario; sin embargo, en la actualidad es una de las pocas formas de escribir código multidimensional compacto y con rendimiento.

Principios de uso

Un ejemplo de uso simple podría ser:

```

@nloops 2 (i,j) begin
    # do something
end

```

que genera el siguiente código:

```

for i = 1:2
    for j = 1:2
        # do something
    end
end

```

En general, permitirá escribir código que contiene elementos repetitivos, como los bucles anidados de este ejemplo. Otras aplicaciones incluyen expresiones repetidas (por ejemplo, desenrollado de bucles) o crear llamadas a función con números variables de argumentos sin usar la construcción "splat" ().

Sintaxis Básica

La sintaxis básica de es la siguiente:

* El primer argumento debe ser un entero (*no* una variable) que especifica el número de bucles. * El segundo argumento es el prefijo simbólico que se utilizará para la variable iteradora. De este modo, en el ejemplo anterior usamos , y se generaron las variables . * El tercer argumento especifica el rango para cada variable iteradora. Si se usa una variable (símbolo) aquí, es considerado como . De forma más flexible, se puede usar la sintaxis de expresiones basadas en funciones anónimas que se describe más adelante. * El último argumento es el cuerpo del bucle. En el ejemplo anterior, lo que aparece entre .

Hay otras características adicionales de descritas en la [sección de referencia](#).

sigue un patrón similar, generando a partir de . La práctica general es leer de izquierda a derecha, por lo que es (como en el bucle , donde está a la izquierda y el rango a la derecha) mientras que es (como en , donde el array va primero).

Si estás desarrollando código con Cartesian, puedes encontrar que depurar es más sencillo cuando examinas el código generado, usando :

Proporcionando el número de expresiones

El primer argumento de estas dos macros es el número de expresiones, que debe ser un entero. Cuando estás escribiendo una función que pretendes que trabaje en múltiples dimensiones, esto puede no ser algo que desees codificar. Si estás escribiendo código que necesitas que trabaje con versiones antiguas de Julia, deberías usar la macro descrita en [una versión más antigua de esta documentación](#).

Empezando en Julia 0.4-pre, el enfoque recomendado es usar una . He aquí un ejemplo:

Naturalmente, también podemos preparar expresiones o realizar cálculos antes del bloque .

Expresiones función anónima como argumentos de macros

Quizás la característica más potente de es la capacidad de proporcionar expresiones función-anónima que son evaluadas en tiempo de análisis sintáctico. Consideremos un ejemplo sencillo:

```
|
```

genera expresiones que siguen un patrón. Este código generaría las siguientes instrucciones:

```
|
```

En cada instrucción generada un aislado (la variable de la función anónima) es reemplazada por valores en el rango . Hablando de forma general, Cartesian emplea una sintaxis parecida a LaTeX. Esto te permite hacer operaciones sobre el índice . He aquí un ejemplo que calcula los pasos de un array:

```
|
```

generará las expresiones

```
|
```

Las expresiones función anónima tienen muchos usos en la práctica.

Referencia de las Macros – Macro.

```
|
```

Generate nested loops, using as the prefix for the iteration variables. may be an anonymous-function expression, or a simple symbol in which case the range is for dimension .

Optionally, you can provide "pre" and "post" expressions. These get executed first and last, respectively, in the body of each loop. For example:

```
|
```

would generate:

```
|
```

If you want just a post-expression, supply `post` for the pre-expression. Using parentheses and semicolons, you can supply multi-statement expressions.

– Macro.

|

Generate expressions like `for`. `for` can either be an iteration-symbol prefix, or an anonymous-function expression.

|

– Macro.

|

Generate variables `x`, `y`, ..., to extract values from `expr`. `expr` can be either a `for` or anonymous-function expression.

would generate

|

while `yield` yields

|

– Macro.

|

Generate expressions. `body` should be an anonymous-function expression.

|

– Macro.

|

Generate a function call expression. `args` represents any number of function arguments, the last of which may be an anonymous-function expression and is expanded into `args` arguments.

For example `call` generates

|

`while` yields

|

– Macro.

|

Generates an `-tuple`. would generate `,` and would generate `.`

– Macro.

|

Check whether all of the expressions generated by the anonymous-function expression evaluate to `.`
 would generate the expression `.` This can be convenient for bounds-checking.

– Macro.

|

Check whether any of the expressions generated by the anonymous-function expression evaluate to `.`
 would generate the expression `.`

– Macro.

|

Generates a sequence of statements. For example:

|

would generate:

|

70.10 Talking to the compiler (the 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.

expressions are created with macros. As an example, consider the implementation of the `__meta__` macro:

```
|
```

Here, `__meta__` is expected to be an expression defining a function. A statement like this:

```
|
```

gets turned into an expression like this:

```
|
```

`__meta__` appends to the end of the `__meta__` expression, creating a new `__meta__` expression if necessary. If `__meta__` is specified, a nested expression containing `__meta__` 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, `__meta__` is very handy: `__meta__` will scan a function *body* expression (one without the function signature) for the first `__meta__` expression containing `__meta__`, extract any arguments, and return a tuple. If the metadata did not have any arguments, or `__meta__` was not found, the `__meta__` array will be empty.

Not yet provided is a convenient infrastructure for parsing `__meta__` expressions from C++.

70.11 SubArrays

El tipo `SubArray{T, N, P, I}` de Julia es un contenedor que codifica una "vista" de un `P` padre. Esta pagina documenta algunos de los principios de diseño e implementación de `SubArray`.

Indexación: indexación cartesiana vs. lineal

Ampliamente hablando, hay dos formas principales de acceder a los datos en un array. La primera, frecuentemente llamada indexación cartesiana, usa índices para un `N`-dimensional. Por ejemplo, una matriz (bidimensional) puede ser indexada en estilo cartesiano como `A[i, j]`. El segundo método de indexación, denominado indexación lineal, usa un solo índice incluso para objetos de mayor dimensión. Por ejemplo si `A`, la expresión `A[i]` devuelve el valor 5. Julia nos permite combinar estos estilos de indexación: por ejemplo, un array 3d puede ser indexado como `A[i, j, k]`, en cuyo caso `A[i, j, k]` es interpretado como un índice cartesiano para la primera dimensión, y `A[i, j, k]` es un índice lineal sobre las dimensiones 2 y 3.

Para los `s`, la indexación lineal apela al formato subyacente de almacenamiento: un array se presenta como un bloque contiguo de memoria, y por tanto el índice lineal es justo el desplazamiento (+1) de la correspondiente entrada relativa al principio del array. Sin embargo, esto no es cierto para muchos otros tipos: ejemplos de ello incluyen `Complex`, unos arrays que requieren alguna clase de cálculo (tal como interpolación), y el tipo bajo discusión aquí, `SubArray`. Para estos tipos, la información subyacente es descrita más naturalmente en términos de índices cartesianos.

Uno puede convertir manualmente un índice cartesiano a uno lineal con `LinearIndex`, y viceversa usando `CartesianIndex`. Los `AbstractArray` pueden incluir operaciones similares.

Aunque convertir de un índice cartesiano a uno lineal es rápido (es justo una multiplicación y una suma), convertir de un índice lineal a uno cartesiano es muy lento: se basa en la operación `LinearIndices`, que es una de las operaciones de bajo nivel más lentas que uno puede realizar con una CPU. Por esta razón, cualquier código que trate con tipos `SubArray` está mejor diseñado en términos de indexación cartesiana en lugar de lineal.

Reemplazo de Índices

Considere hacer rebanadas bidimensionales de un array tridimensional:

```
|
```

`SubArray{A, N, T, I}` elimina las dimensiones "singleton" (las que están especificadas por un `1`), por lo que tanto `SubArray{A, 2, T, I}` como `SubArray{A, 3, T, I}` son `s` bidimensionales. En consecuencia, el camino natural para indexar esto es con `CartesianIndices`. Para extraer el valor del array padre, el enfoque natural es reemplazar `CartesianIndices` con `CartesianIndices{A, N, T, I}`.

La característica clave del diseño de `SubArray`s es que este reemplazo de índices puede realizarse sin ninguna sobrecarga en tiempo de ejecución.

Diseño de SubArray's

Parámetros de Tipo y Campos

La estrategia adoptada está expresada en la definición del tipo:

```
|
```

`SubArray{A, N, T, I}` tiene cinco parámetros de tipo. Los dos primeros son el tipo de elemento estándar y la dimensionalidad. La siguiente es el tipo del array padre. El usado más intensamente es el cuarto parámetro, una tupla de los tipos de los índices para cada dimensión. El final, `isindexing`, es sólo proporcionado como una conveniencia para el despacho; es un valor booleano que representa si los tipos del índice soportan indexación lineal rápida. Más sobre este tema después.

Si en nuestro ejemplo de arriba `A` es un `Array{Float64, 3}`, nuestro caso sería un `SubArray{Float64, 2, Array{Float64, 3}, Tuple{Int{1}, Int{2}}, true}`. Note en particular el parámetro tupla, que almacena los tipos de los índices usados para crear `SubArray`. Igualmente,

```
|
```

Almacenar estos valores permite el reemplazo de índices, y tener los tipos codificados como parámetros permite a uno despachar a eficientes algoritmos.

Traducción de Índices

Realizar la traducción de índices requiere que uno haga diferentes cosas para diferentes tipos concretos de `Index{N}`. Por ejemplo, para `Index{1}` necesita aplicar los índices a las dimensiones primera y tercera del array padre, mientras que para `Index{2}` necesita aplicarlas a la segunda y la tercera. El enfoque más sencillo a indexar sería hacer el análisis de tipos en tiempo de ejecución:

```

function translate_index{T, N}(parent::Array{T, N}, index::Index{N})
    # This function would handle the translation of indices for different
    # types of Index{N}. For example, for Index{1}, it would map the first
    # and third dimensions of the parent array to the index.
end

```

Desgraciadamente, esto sería desastroso en términos de rendimiento: cada acceso a elemento asignaría memoria, e implicaría la ejecución de un montón de código pobremente tipado.

El mejor enfoque es despachar a métodos específicos para manejar cada tipo de índice almacenado. Esto es lo que hace `Base.IndexedArray`: él despacha sobre el tipo del primer índice almacenado y consume el número apropiado de índices de entrada, y entonces recurre sobre los índices restantes. En el caso de `Index{1}`, esto expande a

```

function translate_index{T, N}(parent::Array{T, N}, index::Index{1})
    # This function would handle the translation of indices for Index{1}.
end

```

para cualquier par de índices (excepto `s` and arrays de este tipo, ver abajo).

Este es el núcleo de un `Index{N}`; los métodos de indexación se basan en `translate_index` para hacer esta traducción de índices. Sin embargo, algunas veces, podemos evitar la indirección y hacerlo incluso más rápido.

Indexación Lineal

La indexación lineal puede implementarse de forma eficiente cuando el array completo tiene un solo paso que separe elementos sucesivos, empezando desde cierto desplazamiento. Esto significa que nosotros pre-computamos estos valores y representamos la indexación lineal simplemente como una adición y multiplicación, evitando la indirección de `Index{N}` y (lo que es más importante) la computación lenta de las coordenadas cartesianas por completo.

Para los tipos `Index{1}`, la disponibilidad de una indexación lineal eficiente está basada puramente en los tipos de los índices, y no depende de valores como el tamaño de array padre. Uno puede preguntar si un conjunto de índices dado soporta indexación lineal rápida con la función interna:

```

function is_linear_indexable{T, N}(indices::Index{N})
    # This function would check if a set of indices supports fast linear
    # indexing.
end

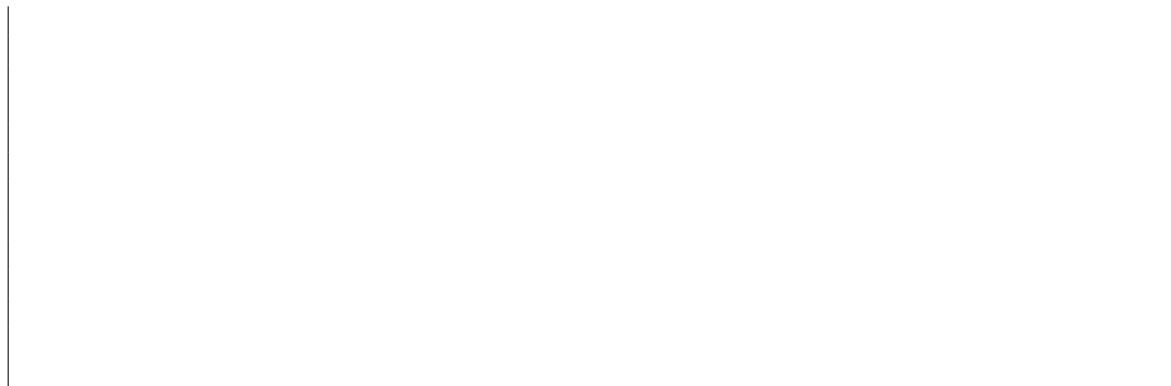
```


Esto se calcula durante la construcción del `view` y se almacena en el parámetro de tipo `bool` como un boolean que codifica soporte de indexación lineal rápido. Aunque no es estrictamente necesario, esto significa que podemos definir despacho directamente sobre `view` sin intermediarios.

Como esta computación no depende de valores en tiempo de ejecución, puede perder algunos casos en los que el paso sea uniforme:



Una vista construida como `view(array, start, end)` tiene un paso uniforme y, por tanto la indexación lineal podría llevarse a cabo eficientemente. Sin embargo, el éxito en este caso depende del tamaño del array: Si, a diferencia del caso anterior, la primera dimensión fuera impar,



entonces `view` no tiene un paso uniforme, por lo que no podemos garantizar indexación lineal eficiente. Como tenemos que basar esta decisión puramente en los tipos codificados en los parámetros del `view`, no puede implementar una indexación lineal eficiente.

Unos pocos detalles

- Note que la función `view` es agnóstica a los tipos de los índices de entrada; ella simplemente determina como y donde deberían reindexarse los índices almacenados. Ella no solo soporta índices enteros,

sino que también soporta indexación no escalar. Esto significa que las vistas de vistas no necesitan dos niveles de indirección; ellas pueden simplemente recomputar los índices en el array padre original.

- Es de esperar que a estas alturas esté bastante claro que soportar rebanadas en arrays significa que la dimensionalidad, dada por el parámetro `axes`, no es necesariamente igual a la dimensionalidad del array padre o la longitud

de la tupla . Tampoco los índices proporcionados por el usuario se alinean necesariamente con las entradas en la tupla (por ejemplo, el segundo índice proporcionado por el usuario puede corresponder a la tercera dimensión de la matriz padre, y el tercer elemento en la tupla).

Lo que podría ser menos obvio es que la dimensionalidad del array padre almacenado sea igual al número de índices efectivos en la tupla . Algunos ejemplos:

Ingenuamente, uno pensaría que podría simplemente establecer `dims`, pero el hecho de soportar esto complica dramáticamente el proceso de reindexación, especialmente para vistas de vistas. No solo se necesita despachar los tipos de los índices almacenados, sino que se debe examinar si un índice dado es el último y "fusionar" los índices almacenados restantes. Esto no es una tarea fácil, y aún peor: es lenta ya que depende implícitamente de la indexación lineal.

Afortunadamente, este es precisamente el cálculo que `'ReshapedArray'` realiza, y lo hace linealmente si es posible. En consecuencia, asegura que el array padre es la dimensionalidad adecuada para los índices dados mediante reformato (*reshaping*) si es necesario. El constructor interno asegura que este invariante esté satisfecha.

- y sus matrices retuerquen de una forma desagradable el esquema . Recuerde que simplemente despacha sobre el tipo de índices almacenados para determinar cuántos índices pasados deberían usarse y a dónde deberían ir. Pero con `dims`, ya no hay una correspondencia uno a uno entre la cantidad de argumentos pasados y la cantidad de dimensiones en las que indexan. Si volvemos al ejemplo anterior de `dims`, puede ver que la expansión es incorrecta para `dims`. Él debería *salta* el `dims` por completo y devolver:

Y si embargo, lo que devuelve es:

Hacer esto correctamente requeriría el envío *combinado* en los índices almacenados y pasados en todas las combinaciones de dimensionalidades de una manera intratable. Como tal, nunca debe invocarse con índices . Afortunadamente, el caso escalar se maneja fácilmente aplanando primero los argumentos a enteros simples. Sin embargo, las matrices de `dims` no se pueden dividir en piezas ortogonales tan fácilmente. Antes de intentar usar `dims`, debe garantizar que no haya matrices de `dims` en la lista de argumentos. Si los hay, simplemente puede "puntualizar" evitando por completo el cálculo de `'reindex'`, construyendo un anidado con dos niveles de indirección en su lugar.

70.12 System Image Building

Building the Julia system image

Julia ships with a preparsed system image containing the contents of the `base` module, named `base.ji`. This file is also precompiled into a shared library called `libjulia.so` 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 `src` 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.ji`, rebuild the system image and use the new `base.ji` next time Julia is started.

- Include a file that includes packages into the system image, 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 `buildsystemimage.jl` that lives in `src/systemimage/`. That is, to include it into a current Julia session, type:

```
|
```

This will include a `function`:

– `Function`.

```
|
```

Rebuild the system image. Store it in `sysimage.jl`, which defaults to a file named `sysimage.jl` that sits in the same folder as `src/systemimage/`, except on Windows where it defaults to `sysimage.exe`. Use the `cpu` instruction set given by `cpu`. Valid CPU targets are the same as for the `cpu` option to `llvmcc1`, or the `cpu` option to `llvmcc1`. Defaults to `native`, which means to use all CPU instructions available on the current processor. Include the user image file given by `userimage`, which should contain directives such as `include_package` to include that package in the new system image. New system image will not replace an older image unless `replace` is set to true.

Note that this file can also be run as a script itself, with command line arguments taking the place of arguments passed to the `function`. For example, to build a system image in `sysimage.jl`, with the `native` CPU instruction set, a user image of `userimage.jl` and `replace` set to `true`, one would execute:

```
|
```

70.13 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 `dynamic` to link dynamically.

The code for lowering Julia AST to LLVM IR or interpreting it directly is in directory `src/llvm`.

File	Description
<code>builtins.jl</code>	Builtin functions
<code>lowering.jl</code>	Lowering
<code>loweringutils.jl</code>	Lowering utilities, notably for array and tuple accesses
<code>codegen.jl</code>	Top-level of code generation, pass list, lowering builtins
<code>debuginfo.jl</code>	Tracks debug information for JIT code
<code>native.jl</code>	Handles native object file and JIT code disassembly
<code>generic.jl</code>	Generic functions
<code>intrinsic.jl</code>	Lowering intrinsics
<code>custom.jl</code>	Custom LLVM pass for
<code>io.jl</code>	I/O and operating system utility functions

Some of the `src/llvm` 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.

Alias Analysis

Julia currently uses LLVM's [Type Based Alias Analysis](#). To find the comments that document the inclusion relationships, look for `in` in `.`

The `option` enables LLVM's [Basic Alias Analysis](#).

Building Julia with a different version of LLVM

The default version of LLVM is specified in `.` You can override it by creating a file called `in` in the top-level directory and adding a line to it such as:

```
|
```

Besides the LLVM release numerals, you can also use `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 `.` The resulting executable is `.` You can pass LLVM options to this executable via the environment variable `.` Here are example settings using `syntax`:

- `dump IR` after each pass.
- `dump LLVM diagnostics` for loop vectorizer *if* you built Julia with `.` Otherwise you will get warnings about "Unknown command line argument". Counter-intuitively, building Julia with `is not` enough to dump `diagnostics` from a pass.

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 `tool` to study it and the pass of interest in isolation.

1. Create an example Julia code of interest.
2. Use `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.

70.14 `printf()` and `stdio` in the Julia runtime

Libuv wrappers for `stdio`

defines [libuv](#) wrappers for the `streams`:

```
|
```

... and corresponding output functions:

These functions are used by the files in the `libuv` and `stdio` 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, `stdio` can be used to directly to :

Interface between `JL_STD*` and Julia code

`jl_stdin`, `jl_stdout` and `jl_stderr` are bound to the `libuv` streams defined in the runtime.

Julia's `jl_init_stdio` function (in `libuv`) calls `jl_stdin` (in `stdio`) to create Julia objects for `jl_stdin`, `jl_stdout` and `jl_stderr`.

`jl_stdin` uses `jl_stdin` to retrieve pointers to `jl_stdin` and `jl_stdout` and calls `jl_stdin` to inspect the type of each stream. It then creates a Julia `IOStream`, `IOStream` or `IOStream` object to represent each stream, e.g.:

The `jl_stdin` and `jl_stdout` methods for these streams use `jl_stdin` to call `libuv` wrappers in `stdio`, e.g.:

`printf()` during initialization

The `libuv` streams relied upon by `jl_stdin`, `jl_stdout` and `jl_stderr`, are not available until midway through initialization of the runtime (see [70.14.1](#)). Error messages or warnings that need to be printed before this are routed to the standard C library `printf` function by the following mechanism:

In `stdio`, the stream pointers are statically initialized to integer constants: `jl_stdin`. In the `jl_stdin` function checks its `argument` and calls `jl_stdin` if stream is set to `jl_stdin` or `jl_stdout`.

This allows for uniform use of `printf` throughout the runtime regardless of whether or not any particular piece of code is reachable before initialization is complete.

Legacy library

The `stdio` library is inherited from `femtolisp`. It provides cross-platform buffered file IO and in-memory temporary buffers. `stdio` is still used by:

- `jl_stdin`
- `jl_stdout` – for serialization file IO and for memory buffers.
- `jl_stderr` – for file IO (see `stdio` for `libuv` equivalent).

Use of `in` 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 `with` with a legacy `stream`.

There is a hack in `in` that makes the `field` line up with the `in` and ensures that the values used for `in` to not overlap with valid values. This allows `pointers` to point to `streams`.

This is needed because `caller` is passed an `stream` by `femtolisp`'s `function`. Julia's `function` has special handling for this:

```
|
```

70.15 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. For the built-in `Array` type, the magic happens inside the `inbounds` and `unsafe_at` intrinsics. User-defined array types instead use the `inbounds` macro to achieve context-sensitive code selection.

Eliding bounds checks

The `inbounds` macro marks blocks of code that perform bounds checking. When such blocks appear inside of an `inbounds` block, the compiler removes these blocks. When the `inbounds` is nested inside of a calling function containing an `inbounds`, the compiler will remove the `inbounds` block *only if it is inlined* into the calling function. For example, you might write the method `inbounds` as:

```
|
```

With a custom array-like type `MyArray` having:

```
|
```

Then when `inbounds` is inlined into `MyArray`, the call to `inbounds` will be elided. If your function contains multiple layers of inlining, only `inbounds` 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 declarations. For instance, the default `inbounds` methods have the chain `inbounds` calls `inbounds`.

To override the "one layer of inlining" rule, a function may be marked 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:

- which calls
 - which calls
 - * which recursively calls
 - for each dimension

Here `arr` is the array, and `indices` contains the "requested" indices. `check_bounds` returns a tuple of "permitted" indices of `arr`.

`check_bounds` throws an error if the indices are invalid, whereas `check_bounds_safe` returns `None` in that circumstance. `check_bounds` discards any information about the array other than its 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, `check_index`: typically,

so `check_index` checks a single dimension. All of these functions, including the unexported `check_index` have docstrings accessible with `help()`.

If you have to customize bounds checking for a specific array type, you should specialize `check_index`. However, in most cases you should be able to rely on `check_index` as long as you supply useful `check_index` for your array type.

If you have novel index types, first consider specializing `check_index`, which handles a single index for a particular dimension of an array. If you have a custom multidimensional index type (similar to `Index`), then you may have to consider specializing `check_index`.

Note this hierarchy has been designed to reduce the likelihood of method ambiguities. We try to make `check_index` the place to specialize on array type, and try to avoid specializations on index types; conversely, `check_index` is intended to be specialized only on index type (especially, the last argument).

70.16 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 here):

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 `lock` and `unlock`. use the `lock_level` 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 `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 4 lock, which can only recurse to acquire level 1, 2, or 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

Broken Locks

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->writelock

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 (while holding codegen lock):
 - rettype
 - inferred
 - these can also be reset, see for that logic as it needs to keep in sync
- flag:
 - optimization to quickly avoid recurring into while it is already running
 - actual state (of setting , then) is protected by codegen lock
- Function pointers (and ,):
 - these transition once, from to a value, while the codegen lock is held
- Code-generator cache (the contents of):
 - 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) and assume it is coordinated, unless also holding the codegen lock
- flag:
 - unknown

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

70.17 Arrays with custom indices

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 (and not just , 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 `with` with `with`
- replace `with` with `,` and `with` with `with`
- replace explicit allocations like `with` with `with`

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 are likely to experience errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```

function f(x)
    return x[1]
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 `-s`.)

To ensure that such errors are caught, in Julia 0.5 both `Base.getindex` and `Base.setindex!` **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 `ndrange` for bounds checks and loop iteration

(reminiscent of `range`) returns a tuple of `range` 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 `i`, there is `a.ndrange(i)`.

`Base` implements a custom range type, `ndrange`, where `ndrange` means the same thing as `range` but in a form that guarantees (via the type system) that the lower index is 1. For any new `ndarray` type, this is the default returned by `a.ndrange`, 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:

```
|
```

at the top of any function.

For bounds checking, note that there are dedicated functions `ndim` and `ndrange` which can sometimes simplify such tests.

Linear indexing (`flat`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `flat`, even if `a` is multi-dimensional. In "true" linear indexing, the indices always range from 0 to `a.size - 1`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `1D`): does `flat` mean linear indexing, or Cartesian indexing with the array's native indices?

For this reason, if you want to use linear indexing in an algorithm, your best option is to get the index range by calling `a.flat`. This will return `flat` if `a` is an `ndarray`, and the equivalent of `range(a.size)` otherwise.

In a sense, one can say that 1-dimensional arrays always use Cartesian indexing. To help enforce this, it's worth noting that `flat` and `flatiter` will throw an error if `a` indicates a 1-dimensional array with unconventional indexing (i.e., `a` is a `1D` rather than a tuple of `1D`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `flat` and `flatiter`, here is one way you could rewrite :

```
|
```

Allocating storage using generalizations of `zeros`

Storage is often allocated with `zeros` or `ones`. 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 `zeros_like`. `zeros_like` indicates the kind of underlying "conventional" behavior you'd like, e.g., `zeros_like(a)` or `zeros_like(b)` or even `zeros_like(a, dtype=0)` (which would allocate an all-zeros array). `zeros_like` is a tuple of `ndim` or `shape` values, specifying the indices that you want the result to use.

Let's walk through a couple of explicit examples. First, if `a` has conventional indices, then `zeros_like(a)` would end up calling `zeros(a.shape)`, and thus return an array. If `a` is an `ndarray` type with unconventional indexing, then `zeros_like(a)` should return something that "behaves like" an `ndarray` but with a shape (including indices) that matches `a`. (The most obvious implementation is to allocate an `ndarray` and then "wrap" it in a type that shifts the indices.)

Note also that `zeros_like(a, dtype=0)` would allocate an `ndarray` (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 `Base.getindex{A, B}(a::A, b::B)`, meaning that the first index along a dimension indexed by `b` 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 `Base.getindex{A, B}` are really about computing an index offset; when that is the case, a candidate replacement is:

```
|
```

In other words, while `Base.getindex{A, B}` 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 `Base.getindex{A, B}` or `Base.setindex{A, B}`

Perhaps the majority of pre-existing code that uses `Base.getindex{A, B}` will not work properly for arrays with non-1 indices. For that reason, it is much better to avoid implementing these methods, and use the resulting `Base.getindex{A}` to identify code that needs to be audited and perhaps generalized.

Do not annotate bounds checks

Julia 0.5 includes `@noinbounds` to annotate code that can be removed for callers that exploit `@noinbounds`. Initially, it seems far preferable to run with bounds checking always enabled (i.e., omit the `@noinbounds` annotation so the check always runs).

Custom types

If you're writing a non-1 indexed array type, you will want to specialize `Base.getindex{A}` so it returns a `Base.getindex{A}`, or (perhaps better) a custom `Base.getindex{A}`. The advantage of a custom type is that it "signals" the allocation type for functions like `Base.getindex{A}`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `Base.getindex{A}`, where `Base.getindex{A}` is equivalent to `Base.getindex{A}`.

In general, you should probably *not* export `Base.getindex{A}` from your package: there may be other packages that implement their own `Base.getindex{A}`, and having multiple distinct `Base.getindex{A}` types is (perhaps counterintuitively) an advantage: `Base.getindex{A}` indicates that `Base.getindex{A}` should create a `Base.getindex{A}`, whereas `Base.getindex{A}` indicates a `Base.getindex{A}` 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 `Base.getindex{A}` type.

Specializing

Once you have your `Base.getindex{A}` type, then specialize :

```
|
```

where here we imagine that `Base.getindex{A}` has a field called `Base.getindex{A}` (there would be other ways to implement this).

In some cases, the fallback definition for :

```
|
```

may not be what you want: you may need to specialize it to return something other than `when`. Likewise, in `there` is a dedicated function `which` which is equivalent to `but` but which avoids checking (at runtime) whether `.` (This is purely a performance optimization.) It is defined as:

```
|
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing

Given your custom `type`, then you should also add the following two specializations for :

```
|
```

Both of these should allocate your custom array type.

Specializing

Optionally, define a method

```
|
```

and you can `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.

70.18 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 are currently used to power Julia's package manager. It is expected that this 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](#).

– Type.

Abstract credentials payload

– Type.

|

A data buffer for exporting data from libgit2. Matches the struct.

When fetching data from LibGit2, a typical usage would look like:

|

In particular, note that should be called afterward on the object.

– Type.

Credentials that support caching

– Type.

|

Matches the struct.

– Type.

|

Matches the struct.

– Type.

|

Description of changes to one entry. Matches the struct.

The fields represent:

- : One of , indicating whether the file has been added/modified/deleted.
- : 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.

- : Used to indicate if a file has been renamed or copied.
- : 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).
- : A containing information about the file(s) before the changes.
- : A containing information about the file(s) after the changes.

– Type.

|

Description of one side of a delta. Matches the struct.

– Type.

|

Matches the struct.

– Type.

|

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.

– Type.

|

Matches the struct.

– Type.

|

Return a object from specified by /.

- is a full () or partial () hash.
- is a textual specification: see [the git docs](#) for a full list.

– Type.

|

Return a `GitObject` from `repo` specified by `ref`.

- `ref` is a full `()` or partial `()` hash.
- `ref` is a textual specification: see [the git docs](#) for a full list.

– Type.

|

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.

– Type.

|

Return the specified object (`obj`, `ref`, or `refspec`) from `repo` specified by `ref`.

- `ref` is a full `()` or partial `()` hash.
- `ref` is a textual specification: see [the git docs](#) for a full list.

– Type.

|

Look up a remote git repository using its name and URL. Uses the default fetch refspec.

Example

|

|

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).

Example

|

– Function.

|

Look up a remote git repository using only its URL, not its name.

Example

|

– Type.

|

Opens a git repository at .

– Function.

|

Opens a git repository at with extended controls (for instance, if the current user must be a member of a special access group to read).

– Type.

|

This is a shortened form of , which can be used to identify a git object when it is unique.

Internally it is stored as two fields: a full-size () and a length (). Only the initial hex digits of are used.

– Type.

|

This is a Julia wrapper around a pointer to a object.

– Type.

|

Collect information about the status of each file in the git repository (e.g. is the file modified, staged, etc.). can be used to set various options, for instance whether or not to look at untracked files or whether to include submodules or not.

– Type.

|

Return a `object` from `specified` by `/`.

- is a full `()` or partial `()` hash.
- is a textual specification: see [the git docs](#) for a full list.

– Type.

|

Return a `object` from `specified` by `/`.

- is a full `()` or partial `()` hash.
- is a textual specification: see [the git docs](#) for a full list.

– Type.

|

In-memory representation of a file entry in the index. Matches the `struct`.

– Type.

|

Matches the `struct`.

– Type.

|

Matches the `struct`.

– Type.

|

Options for connecting through a proxy.

Matches the `struct`.

– Type.

|

Matches the `struct`.

– Type.

|

Describes a single instruction/operation to be performed during the rebase. Matches the struct.

– Type.

|

Matches the struct.

– Type.

|

Callback settings. Matches the struct.

– Type.

SSH credentials type

– Type.

|

An action signature (e.g. for committers, taggers, etc). Matches the struct.

– Type.

|

Providing the differences between the file as it exists in HEAD and the index, and providing the differences between the index and the working directory. Matches the struct.

– Type.

|

Options to control how will issue callbacks. Matches the struct.

– Type.

|

A LibGit2 representation of an array of strings. Matches the struct.

When fetching data from LibGit2, a typical usage would look like:

In particular, note that `set` should be called afterward on the `object`.

Conversely, when passing a vector of strings to `LibGit2`, it is generally simplest to rely on implicit conversion:

Note that no call to `is` is required as the data is allocated by Julia.

– Type.

Time in a signature. Matches the `struct`.

– Type.

Credentials that support only `and` parameters

– Function.

Add a *fetch* refspec for the specified `.` This refspec will contain information about which branch(es) to fetch from.

Example

– Function.

Add a *push* refspec for the specified `.` This refspec will contain information about which branch(es) to push to.

Example

Note

You may need to `close()` and reopen the `Repository` in question after updating its push refsspecs in order for the change to take effect and for calls to `push()` to work.

– Function.

```
|
```

Reads the file at `path` and adds it to the object database of `repo` as a loose blob. Returns the `ObjectId` of the resulting blob.

Example

```
|
```

– Function.

```
|
```

Returns all authors of commits to the `Repository`.

Example

```
|
```

– Function.

```
|
```

Equivalent to `git checkout -b`. Create a new branch from the current HEAD.

– Function.

```
|
```

Checkout a new git branch in the repository. is the , in string form, which will be the start of the new branch. If is an empty string, the current HEAD will be used.

The keyword arguments are:

- : the name of the remote branch this new branch should track, if any. If empty (the default), no remote branch will be tracked.
- : if , branch creation will be forced.
- : if , after the branch creation finishes the branch head will be set as the HEAD of .

Equivalent to .

Example

```
|
```

– Function.

```
|
```

Equivalent to . Checkout the git commit (a in string form) in . If is , force the checkout and discard any current changes. Note that this detaches the current HEAD.

Example

```
|
```

– Function.

Checks if credentials were used

Checks if credentials were used or failed authentication, see

– Function.

```
|
```

Clone a remote repository located at `url` to the local filesystem location `path`.

The keyword arguments are:

- `branch`: which branch of the remote to clone, if not the default repository branch (usually `master`).
- `bare`: if `True`, clone the remote as a bare repository, which will make `path` itself the git directory instead of `path/.git`. This means that a working tree cannot be checked out. Plays the role of the git CLI argument `--bare`.
- `clone`: a callback which will be used to create the remote before it is cloned. If `clone` (the default), no attempt will be made to create the remote - it will be assumed to already exist.
- `credentials`: provides credentials if necessary, for instance if the remote is a private repository.

Equivalent to `git clone`.

Examples

```
|
```

– Function.

Wrapper around

Commit changes to repository

```
|
```

Commits the current patch to the rebase, using `author` as the committer. Is silent if the commit has already been applied.

– Function.

```
|
```

Create a new branch in the repository with name `branch`, which points to commit `commit` (which has to be part of `branch`). If `branch` is `True`, overwrite an existing branch named `branch` if it exists. If `branch` is `False` and a branch already exists named `branch`, this function will throw an error.

– Function.

Credentials callback function

Function provides different credential acquisition functionality w.r.t. a connection protocol. If a payload is provided then `payload` should contain a `dict` object.

For `ssh` type, if the payload contains fields: `username` & `password`, they are used to create authentication credentials. Empty `username` and `password` trigger an authentication error.

For `https` type, if the payload contains fields: `username`, `password` & `cert`, they are used to create authentication credentials. Empty `username` triggers an authentication error.

Credentials are checked in the following order (if supported):

- ssh key pair (if specified in payload's field)
- plain text

Note: Due to the specifics of the 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, the function can be called. This function returns if the credentials were used. Using credentials triggers a user prompt for (re)entering required information. and are implemented using a call counting strategy that prevents repeated usage of faulty credentials.

– Function.

C function pointer for

– Function.

Return signature object. Free it after use.

– Function.

|

Delete the branch pointed to by .

– Function.

|

Show which files have changed in the git repository between branches and .

The keyword argument is:

- , 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.

Example

|

Equivalent to .

– Function.

|

Fetch from the specified remote git repository, using to determine which remote branch(es) to fetch. The keyword arguments are:

- : determines the options for the fetch, e.g. whether to prune afterwards.
- : a message to insert into the reflogs.

|

Fetches updates from an upstream of the repository .

The keyword arguments are:

- : which remote, specified by name, of to fetch from. If this is empty, the URL will be used to construct an anonymous remote.
- : the URL of . If not specified, will be assumed based on the given name of .
- : determines properties of the fetch.
- : provides credentials, if necessary, for instance if is a private repository.

Equivalent to .

– Function.

|

Get the *fetch* refsspecs for the specified . These refsspecs contain information about which branch(es) to fetch from.

– Function.

C function pointer for

– Function.

Fastforward merge changes into current head

– Function.

|

Return the name of the reference pointed to by the symbolic reference . If is not a symbolic reference, returns an empty string.

– Function.

Obtain the cached credentials for the given host+protocol (credid), or return and store the default if not found

– Function.

|

Returns the location of the "git" files of :

- for normal repositories, this is the location of the folder.
- for bare repositories, this is the location of the repository itself.

See also , .

– Function.

|

Returns a to the current HEAD of .

|

Return current HEAD of the repo as a string.

– Function.

|

Set the HEAD of to the object pointed to by .

– Function.

|

Lookup the object id of the current HEAD of git repository .

– Function.

|

Lookup the name of the current HEAD of git repository . If is currently detached, returns the name of the HEAD it's detached from.

– Function.

|

Opens a new git repository at . If is , the working tree will be created in . If is , no working directory will be created.

– Function.

|

Returns if , a in string form, is an ancestor of , a in string form.

Example

|

– Function.

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.

– Function.

|

Checks if commit (which is a in string form) is in the repository.

Example

|

– Function.

|

Checks if there are any differences between the tree specified by `tree` and the tracked files in the working tree (if `working`) or the index (if `index`). `options` are the specifications for options for the diff.

Example

|

Equivalent to `git diff`.

– Function.

|

Checks if there have been any changes to tracked files in the working tree (if `working`) or the index (if `index`). `options` are the specifications for options for the diff.

Example

|

Equivalent to `git diff`.

– Function.

|

Checks if the current branch is an "orphan" branch, i.e. has no commits. The first commit to this branch will have no parents.

– Function.

|

Determine if the branch specified by `branch` exists in the repository. If `remote` is `1`, is assumed to be a remote git repository. Otherwise, it is part of the local filesystem.

`git_has_branch` returns a `Bool`, which will be `false` if the requested branch does not exist yet. If the branch does exist, the `git_has_branch` contains a `String` to the branch.

– Function.

Mirror callback function

Function sets `refspecs` and `flag` for remote reference.

– Function.

C function pointer for

– Function.

|

Return the full name of .

|

Get the name of a remote repository, for instance . If the remote is anonymous (see) the name will be an empty string .

Example

|

|

The name of (e.g.).

– Function.

|

Equivalent to . Returns if needs updating.

– Function.

|

Returns the type corresponding to the enum value.

– Function.

|

The base file path of the repository .

- for normal repositories, this will typically be the parent directory of the ".git" directory (note: this may be different than the working directory, see for more details).
- for bare repositories, this is the location of the "git" files.

See also , .

– Function.

|

Recursively peel until an object of type is obtained. If no is provided, then will be peeled until an object other than a is obtained.

- A will be peeled to the object it references.
- A will be peeled to a .

Note

Only annotated tags can be peeled to objects. Lightweight tags (the default) are references under which point directly to objects.

|

Recursively peel until an object of type is obtained. If no is provided, then will be peeled until the type changes.

- A will be peeled to the object it references.
- A will be peeled to a .

– Function.

|

Standardise the path string to use POSIX separators.

– Function.

|

Push to the specified remote git repository, using to determine which remote branch(es) to push to. The keyword arguments are:

- : if , a force-push will occur, disregarding conflicts.

- : determines the options for the push, e.g. which proxy headers to use.

Note

You can add information about the push refsspecs in two other ways: by setting an option in the repository's (with as the key) or by calling . Otherwise you will need to explicitly specify a push refspect in the call to for it to have any effect, like so: .

|

Pushes updates to an upstream of .

The keyword arguments are:

- : the name of the upstream remote to push to.
- : the URL of .
- : determines properties of the push.
- : determines if the push will be a force push, overwriting the remote branch.
- : provides credentials, if necessary, for instance if is a private repository.

Equivalent to .

– Function.

|

Get the *push* refsspecs for the specified . These refsspecs contain information about which branch(es) to push to.

– Function.

|

Read the tree (or the tree pointed to by in the repository owned by) into the index . The current index contents will be replaced.

– Function.

|

Attempt an automatic merge rebase of the current branch, from if provided, or otherwise from the upstream tracking branch. is the branch to rebase onto. By default this is .

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 . This is roughly equivalent to the following command line statement:

|

– Function.

|

Get a list of all reference names in the repository.

– Function.

|

Returns a corresponding to the type of :

- if the reference is invalid
- if the reference is an object id
- if the reference is symbolic

– Function.

|

Returns a vector of the names of the remotes of .

– Function.

Resets credentials for another use

Updates some entries, determined by the , in the index from the target commit tree.

Sets the current head to the specified commit oid and optionally resets the index and working tree to match.

`git reset [<committish>] [-] <pathspecs>...`

|

Reset the repository to its state at , using one of three modes set by :

1. - move HEAD to .
2. - default, move HEAD to and reset the index to .
3. - move HEAD to , reset the index to , and discard all working changes.

Equivalent to .

Example

|

- Function.

|

Return a repository to a previous , for example the HEAD of a branch before a merge attempt. can be generated using the function.

- Function.

|

List the number of revisions between and (committish OIDs in string form). Since and may be on different branches, performs a "left-right" revision list (and count), returning a tuple of s - the number of left and right commits, respectively. A left (or right) commit refers to which side of a symmetric difference in a tree the commit is reachable from.

Equivalent to .

- Function.

|

Set the for for the git repository . The default name of the remote is .

Examples

|

|

Set the for for the git repository located at . The default name of the remote is .

– Function.

|

Returns a shortened version of the name of that's "human-readable".

|

– Function.

|

Take a snapshot of the current state of the repository , storing the current HEAD, index, and any uncommitted work. The output can be used later during a call to to return the repository to the snapshotted state.

– Function.

|

Lookup the status of the file at in the git repository . For instance, this can be used to check if the file at has been modified and needs to be staged and committed.

– Function.

|

Create a new git tag (e.g.) in the repository , at the commit .

The keyword arguments are:

- : the message for the tag.
- : if , existing references will be overwritten.
- : the tagger's signature.

– Function.

|

Remove the git tag from the repository .

– Function.

|

Get a list of all tags in the git repository .

– Function.

|

The of the target object of .

– Function.

Traverse the entries in a tree and its subtrees in post or pre order.

Function parameter should have following signature:

|

– Function.

|

Determine if the branch containing has a specified upstream branch.

returns a , which will be null if the requested branch does not have an upstream counterpart. If the upstream branch does exist, the contains a to the upstream branch.

– Function.

|

Get the fetch URL of a remote git repository.

Example

|

– Function.

Resource management helper function

– Function.

|

The location of the working directory of . This will throw an error for bare repositories.

Note

This will typically be the parent directory of , but can be different in some cases: e.g. if either the configuration variable or the environment variable is set.

See also , .

70.19 Module loading

[@ref] is responsible for loading modules and it also manages the precompilation cache. It is the implementation of the statement.

Experimental features

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 , by registering a 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:

|

Chapter 71

Developing/debugging Julia's C code

71.1 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 \(\)](#)
- [Segfaults when running a script](#)
- [Errors during Julia startup](#)

Version/Environment info

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. If your version is or higher, please include the output of in any report you create:



Segfaults during bootstrap ()

Segfaults toward the end of the process of building Julia are a common symptom of something going wrong while Julia is preparing the corpus of code in the folder. Many factors can contribute toward this process dying unexpectedly,

however it is as often as not due to an error in the C-code portion of Julia, and as such must typically be debugged with a debug build inside of . Explicitly:

Create a debug build of Julia:

```
|
```

Note that this process will likely fail with the same error as a normal incantation, however this will create a debug executable that will offer the debugging symbols needed to get accurate backtraces. Next, manually run the bootstrap process inside of :

```
|
```

This will start , 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 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 \(\)](#). Create a debug build of Julia, and run your script inside of a debugged Julia process:

```
|
```

Note that will sit there, waiting for instructions. Type to run the process, and to generate a backtrace once it segfaults:

```
|
```

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:

```
|
```

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 process:

- On Linux, use :

```
|
```

- On OSX, use :

```
|
```

Create a [gist](#) with the / ouput, the [version info](#), and any other pertinent information and open a new [issue](#) on Github with a link to the gist.

Glossary

A few terms have been used as shorthand in this guide:

- refers to the root directory of the Julia source tree; e.g. it should contain folders such as , , , etc.....

71.2 gdb debugging tips

Displaying Julia variables

Within , any object can be displayed using

```
|
```

The object will be displayed in the 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.,), you can print using

```
|
```

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 objects; these can be displayed with .

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 for a complete list) that are even more useful.

- (when in) and :: for figuring out a bit about the call-stack
- and :: for figuring out what line in a test to go start debugging from (or figure out how far into a file has been parsed)
- :: not really a variable, but still a useful shorthand for referring to the result of the last gdb command (such as)
- :: sometimes useful, since it lists all of the command line options that were successfully parsed
- :: because who doesn't like to be able to interact with stdio

Useful Julia functions for Inspecting those variables

- :: For looking up the current function and line. (use on i686 platforms)
- :: For dumping the current Julia backtrace stack to stderr. Only usable after has been called.
- :: For invoking in gdb, where it doesn't work natively. For example, , , and .
- :: only works in lldb. Note: add something like to prevent lldb from printing its prompt over the output
- :: for invoking side-effects to modify the current state or to lookup symbols
- :: for extracting the type tag of a Julia value (in gdb, call first, or pick something short like for the first arg to define a shorthand)

Inserting breakpoints for inspection from gdb

In your session, set a breakpoint in like so:

```
|
```

Then within your Julia code, insert a call to by adding

```
|
```

where can be any variable or tuple you want to be accessible in the breakpoint.

It's particularly helpful to back up to the frame, from which you can display the arguments to a function using, e.g.,

```
|
```

Another useful frame is . The 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 and then pass the result to :

```
|
```

Inserting breakpoints upon certain conditions

Loading a particular file

Let's say the file is :

```
|
```

Calling a particular method

```
|
```

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 properly. The profiler uses for sampling and the garbage collector uses 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 with or other signals you want to ignore):

```
|
```

The corresponding LLDB command is (after the process is started):

```
|
```

If you are debugging a segfault with threaded code, you can set a breakpoint on (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 `bootstrap` need special handling. Julia is built in two stages, constructing `libjulia.so` and `libjulia.dylib`. To see what commands are running at the time of failure, use `bootstrap --verbose`.

At the time of this writing, you can debug build errors during the `bootstrap` phase from the `src` directory using:

```
|
```

You might need to delete all the files in `src` to get this to work.

You can debug the `bootstrap` phase using:

```
|
```

By default, any errors will cause Julia to exit, even under `gdb`. To catch an error "in the act", set a breakpoint in `src` (there are several other useful spots, for specific kinds of failures, including: `src`, `src`, and `src`).

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_call_julia`. To take a real-world example:

```
|
```

The most recent `jl_call_julia` is at frame #3, so we can go back there and look at the AST for the function `jl_call_julia`. This is the unique name for some method of `jl_call_julia`. In this frame is a `jl_call_julia`, so we can look at the type signature, if any, from the `jl_call_julia` field:

```
|
```

Then, we can look at the AST for this function:

```
|
```

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 `jl_call_julia` from the `jl_call_julia`:

```
|
```

Then, set a breakpoint somewhere useful (e.g. `, , ,` etc.), and run `codegen`:

```
|
```

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:

```
|
```

or:

```
|
```

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.

71.3 Usando Valgrind con Julia

`Valgrind` es una herramienta para depuración de memoria, detección de fallos de página y creación de perfiles. Esta sección describe cosas a tener en cuenta cuando se usa Valgrind para depurar cuestiones de memoria con Julia.

Consideraciones Generales

Por defecto, Valgrind asume que no hay código automodificador en el programa que está ejecutando. Esta suposición trabaja bien en la mayoría de las instancias, pero falla miserablemente para un compilador JIT como `Julia`. Por esta razón es crucial pasar `--no-instrument`, si no el código puede bloquearse o comportarse de forma inesperada (frecuentemente de una forma sutil).

En algunos casos, para detectar mejor errores de memoria usando Valgrind puede ayudar compilar `Julia` con los *pools* de memoria deshabilitados. El flag de tiempo de compilación `--no-jit` deshabilita los *pools* de memoria en Julia y el flag `--no-jit` deshabilita los *pools* de memoria en FemtoLisp. Para construir `Julia` con ambos flags, añada la siguiente línea a `Makefile`:

```
|
```

Otra cosa a notar: si nuestro programa usa múltiples procesos workers, es probable que queramos que todos esos procesos worker se ejecuten bajo Valgrind, no sólo bajo el proceso padre. Para hacer esto, pasaremos `--no-jit` a `Julia`.

Supresiones

Valgrind típicamente mostrará avisos falsos mientras se ejecuta. Para reducir el número de tales avisos, ayuda proporcionar un [fichero supresiones](#) a Valgrind. Un fichero supresiones de ejemplo se incluye en la distribución fuente de Julia en `src/valgrind/`.

El fichero de supresiones puede ser usado desde el directorio fuente de la siguiente manera:

```
|
Any memory errors that are displayed should either be reported as bugs or 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 `src/valgrind/` directory:

```
|
If you would like to see a report of "definite" memory leaks, pass the flags --leak-check=full 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 `JULIA_NUM_THREADS=1` you find that your program behaves differently when run under Valgrind, it may help to pass `--enable-minimal` 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.

71.4 Sanitizer support

General considerations

Using Clang's sanitizers obviously require you to use Clang (`clang`), but there's another catch: most sanitizers require a runtime library, provided by the host compiler, while the instrumented code generated by Julia's JIT relies 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 `clang` and overriding `LLVM_CONFIG_PATH` (Autotool-based builds are incompatible with ASAN). You can then refer to this toolchain from another build folder by specifying `LLVM_CONFIG_PATH` while overriding the `LLVM_CONFIG_PATH` and `LLVM_CONFIG_PATH` variables.

Address Sanitizer (ASAN)

For detecting or debugging memory bugs, you can use Clang's [address sanitizer \(ASAN\)](#). By compiling with `clang -fsanitize=address` you enable ASAN for the Julia compiler and its generated code. In addition, you can specify `clang -fsanitize=address,llvm` 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 `make test` 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 `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 `ASAN_OPTIONS=detect_leaks=0` and `ASAN_OPTIONS=abort_on_error=0`, at the cost of backtrace accuracy. For now, Julia also sets `ASAN_OPTIONS=abort_on_error=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 .