Programming in C++

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Version 18 February 2002

Programming in C++ **Contents**

Programming in C++

☆☆☆ Introduction ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Introduction Hello World

- ❑ Kindergarden (BASIC)

```
10 PRINT "Hello World"
20 END
```

- ❑ School (PASCAL)

```
program Hello(input, output);
begin
   writeln ('Hello World')
end.
```

- ❑ University (LISP)

```
(defun hello
   (print (cons 'Hello (list 'World))))
```

- ❑ Professional (C)

```
#include <stdio.h>
```

or

```
perl -e 'print "Hello World\n"'
```

```
int main (int argc, char *argv[]) {
   printf ("Hello World\n");
   return 0;
}
```

- Seasoned pro (C++)

```
#include <iostream>
   using std::cout;
   using std::endl;
#include <string>
   using std::string;

int main(int argc, char *argv[]) {
  string str = "Hello World";
  cout << str << endl;
  return(0);
}
```


- Manager

Dr. Mohr, for tomorrow I need a program to output the words "Hello World"!

1979	May	Bjarne Stroustrup at AT&T Bell Labs starts working on C with Classes
1982	Jan	1st external paper on C with Classes
1983	Dec	C++ named
1984	Jan	1st C++ manual
1985	Oct	Cfront Release 1.0 (first commercial release)
	Oct	*The C++ Programming Language* [Stroustrup]
1987	Feb	Cfront Release 1.2
	Dec	1st GNU C++ release (1.13)
1988	Jun	1st Zortech C++ release
1989	Jun	Cfront Release 2.0
	Dec	ANSI X3J16 organizational meeting (Washington, DC)
1990	Mar	1st ANSI X3J16 technical meeting (Somerset, NJ)
	May	1st Borland C++ release
	May	*The Annotated C++ Reference Manual* (ARM) [Ellis, Stroustrup]
	Jul	Templates accepted (Seattle, WA)
	Nov	Exceptions accepted (Palo Alto, CA)

1991	Jun	*The C++ Programming Language* (2nd edition) [Stroustrup]
	Jun	1st ISO WG21 meeting (Lund, Schweden)
	Oct	Cfront Release 3.0 (including templates)
1992	Feb	1st DEC C++ release (including templates and exceptions)
	Mar	1st Microsoft C++ release
	May	1st IBM C++ release (including templates and exceptions)
1993	Mar	Run-time type identification accepted (Portland, OR)
	July	Namespaces accepted (Munich, Germany)
1995	Apr	1st Public-Comment Draft ANSI/ISO standard
1996	Dec	2nd Public-Comment Draft ANSI/ISO standard
1997	Nov	Final Draft International Standard (FDIS) for C++
1998	Jul	International Standard (ISO/IEC 14882:1998, "*Programming Language -- C++*")

➠ expect changes in compilers in the next years

➠ buy only up-to-date (reference) books!

- Up-to-date books on C++ should:
  - have examples using `bool` and `namespace`
  - include (at least) a chapter on the C++ Standard Library and STL
    - ➠ use `string` and `vector` in examples
  - mention RTTI and new-style casts
- Even better they (especially reference guides) include
  - member templates
  - partial specialization
  - `operator new[]` and `operator delete[]`
- Even more better if they contain / explain the new keyword `export`

C++ legality guides – what you can and can’t do in C++

- Stroustrup, *The C++ Programming Language*, **Third Edition** or **Special Edition (**14th printing), Addison-Wesley, 1997 or 2000, ISBN 0-201-88954-4 or ISBN 0-201-70073-5.

  Stroustrup, *Die C++ Programmiersprache*, **Dritte Auflage or Vierte Auflage**, Addison-Wesley, 1997 or 2000, ISBN 3-8273-1296-5 or 3-8273-1660-X.

  ➠ Covers a lot of ground; Reference style; Better if you know C
- Lippman and Lajoie, *C++ Primer*, **Third Edition**, Addison-Wesley, 1998, ISBN 0-201-82470-1.

  ➠ Tutorial style; Better for novices
- Koenig and Moo, *Accelerated C++*, Addison-Wesley, 2000, ISBN 0-201-70353-X.

  ➠ First-rate introductory book that takes a practical approach to solving problems using C++
- Josuttis, *Objektorientiertes Programmieren in C++*, **2. Auflage,** Addison-Wesley, 2001, ISBN 3-8273-1771-1.

C++ morality guides – what you should and shouldn't do in C++

- Meyers, *Effective C++*,
  Addison-Wesley, 1992, ISBN 0-201-56364-9.
  - Covers 50 topics in a short essay format; a *must* for anyone programming C++
- Cline and Lomow, *C++ FAQs*,
  Addison-Wesley, 1995, ISBN 0-201-58958-3.
  - Covers 470 topics in a FAQ-like Q&A format (**see also on-line FAQ**)
    Examples are complete, working programs rather than code fragments or stand-alone classes
- Murray, *C++ Strategies and Tactics*,
  Addison-Wesley, 1993, ISBN 0-201-5638-2.
  - Lots of tips and tricks in an easy to read style

C++ legality guides

- Stroustrup, *The Design and Evolution of C++*,
  Addison-Wesley, 1994, ISBN 0-201-54330-3.
  - Explains the rationale behind the design of C++ and its history plus new features
- Ellis and Stroustrup, *The Annotated C++ Reference Manual*, (ARM)
  Addison-Wesley, 1990, ISBN 0-201-51459-1.
  - The former unofficial “official” standard on C++

C++ morality guides

- Meyers, *More Effective C++*,
  Addison-Wesley, 1996, ISBN 0-201-63371-X.
  - Covers 35 advanced topics: exceptions, efficiency, often used techniques (patterns)
- Sutter, *Exceptional C++* and *More Exceptional C++*,
  Addison-Wesley, 2000 and 2002, ISBN ISBN 0-201-61562-2 and 0-201-70434-X.
  - Provides successful strategies for solving real-world problems in C++

- FZJ C++ WWW Information Index

  `http://www.fz-juelich.de/zam/PT/lang/cplusplus.html`

  - WWW C++ Information

    `http://www.fz-juelich.de/zam/cxx/`

  - Parallel Programming with C++
  - Forschungszentrum Jülich Local C++ Information

- Official C++ On-line FAQ

  `http://www.parashift.com/c++-faq-lite/`

- The Association of C & C++ Users
  `http://www.accu.org/`

  - Book reviews section with over 2400 books
    `http://www.accu.org/bookreviews/public/`

This C++ Course is based on the following sources:

- Dr. Aaron Naiman, Jerusalem College of Technology, Object Oriented Programming with C++

  `http://hobbes.jct.ac.il/%7Enaiman/c++-oop/`

  - Classes, Pointer Data Members, More on Classes, Array Examples

- Dr. Roldan Pozo, Karin Remington, NIST, C++ Programming for Scientists

  `http://math.nist.gov/pozo/c++class/`

  - Motivation, From C to C++

- Sean A Corfield, OCS, C++ - Beyond the ARM

  `http://www.corfield.org/cplusplus.phtml/`

  - Advanced C++

- Meyers, *Effective C++* and *More Effective C++*

- Stroustrup, Lippman, Murray, ...

Name of the Compiler

- CC (Sun, HP, SGI, Cray)
- cxx (DEC)
- xlC (IBM)
- g++ (GNU, egcs)
- KCC (KAI)
- . . .

Typical Compiler Options (UNIX)

-O	Turn Optimization on
-g	Turn Debugging on
-o file	Specify output file name
-c	Create object file only
-D / -I / -U / -E	Standard cpp options
-L / -l	Standard linker options

Source File Names

.cc .cpp .C .cxx	C++ source files
.h .hh .H .hpp	C++ header files

Compiling and Linking (UNIX)

```
CC -c main.cpp
CC -o prog main.cpp sum.cpp -lm
```

Compiler/Programming Environments

- Visual Workshop (Sun)
- VisualAge (IBM)
- Softbench (HP)
- ObjectCenter (for Sun, HP)
- Energize (Lucid for Sun, HP)
- C++ on PCs (Borland, Microsoft, **...)**
- CodeWarrier (Macs, Windows, Unix)

"C++: The Cobol of the 90s"

- C++ is a very powerful and complex programming language
  - ➠ hard to learn and understand
  - ➠ can easily be mis-used, easy to make errors

but

- It is an illusion that you can solve complex, real-world problems with simple tools
- You need to know the dark sides / disadvantages so you can avoid them
  - ➠ this is why for C++ the Morality Books are important
- What you don’t use, you don’t pay for (zero-overhead rule)
- It is easy / possible just to use the parts of C++ you need or like
  - ❍ non object-oriented subset
  - ❍ only use (class / template) libraries
  - ❍ concrete data types ("simple classes") only . . .

- ❑ Procedural Programming [Fortran77, Pascal, **C]**

  "Decide what procedures you want; use the best algorithms you can find"

  ➠ focus on algorithm to perform desired computation

- ❑ Modular Programming (Data Hiding Principle) [Modula-2, **C++]**

  "Decide which modules you want; partition the program so that data is hidden in modules"

  ➠ Group data with related functions

- ❑ Abstract Data Types (ADT) [Ada, Clu, Fortran90, **C++]**

  "Decide which types you want; provide a full set of operations for each type"

  ➠ if more than one object of a type (module) is needed

- ❑ Object Oriented Programming [Simula, Eiffel, Java, **C++]**

  "Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance"

- ❑ Generic Programming [Eiffel, **C++]**

  "Decide which classes you want; provide a full set of operations for each class; make commonality of classes or methods explicit by using templates"

C++ is **NOT** an *object-oriented language*
but

C++ is a *multi-paradigm programming language* with a bias towards systems programming that

- ❑ supports *data abstraction*
- ❑ supports *object-oriented programming*
- ❑ supports *generic programming*

- ❑ is a better C

  "*as close to C as possible – but no closer*" [Stroustroup / Koenig, 1989]

- ❑ ANSI C89 is almost a subset of C++ (e.g. all examples of [K&R2] are C++!)
- ❑ This is not true for ANSI C99!

Programming in C++

☆☆☆ Basics: The C part of C++ ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

C++ is

❑ *format-free* (like Pascal; Fortran: line-oriented)

➠ `int *iptr[20];` the same as
```
int*    iptr [
                20
               ];
```

❑ is case-sensitive

➠ `foo` and `Foo` or `FOO` are all distinct identifiers!

❑ *keywords* are written *lower-case*

➠ `switch`, `if`, `while`, ...

❑ semicolon is *statement terminator* (Pascal and Fortran: statement separator)

➠
```
if ( expr ) {
  statement1;
  statement2;
}
```

Basic Data Types

	ANSI C	Pascal	Fortran
Boolean	`(int)`	**`boolean`**	**`logical`**
Character	**`char, wchar_t`**	**`char`**	**`character`***(n)*
Integer	**`short int`** **`int`** **`long int`**	**`integer`**	**`integer`**
FloatingPoint	**`float`** **`double`**	**`real`**	**`real`**
Complex	❖ `(in C99)`	❖	**`complex`**

- Size of data types in ANSI C is implementation defined but: `short` ≤ `int` ≤ `long` and `float` ≤ `double`
- ANSI C has also `signed` and `unsigned` qualifiers
- ANSI C has no special boolean type (uses `int` instead), but C++ now has: `bool`
- Fortran also supports different size for `integer` or `real`, e.g.,

```
integer,parameter :: short = selected_int_kind(4)
integer(short) :: i
```


Literals

	ANSI C	Pascal	Fortran
Boolean	`0, (`*nonzero*`)`	`false, true`	`.false., .true.`
Character	`'c'`	`'c'`	`'c'` *or* `"c"`
String	`"foo"`	`'foo'`	`'foo'` *or* `"foo"`
Integer	`456, -9`	`456, -9`	`456, -9`
Integer (octal)	`0177`	❖	❖
Integer (hexdecimal)	`0xFF, 0X7e`	❖	❖
FloatingPoint	`3.89, -0.4e7`	`3.89, -0.4e7`	`3.89, -0.4e7`
Complex	❖	❖	`(-1.0,0.0)`

- ANSI C has no special boolean type (uses `int` instead), but C++ now has: `bool` with constants `true` and `false`
- ANSI C characters and strings can contain escape sequences (e.g. `\n, \077, \xFF`)
- ANSI C also has suffix qualifiers for numerical literals: `F` or `f` (`float`), `L` or `l` (`double` / `long`), `U` or `u` (`unsigned`)

ANSI C	Pascal	Fortran
	const	
const double PI=3.1415;	PI = 3.1415;	**real**,**parameter**::PI=3.1415
const char SP=' ';	SP = ' ';	**character**,**parameter**::SP=' '
const double NEG_PI=-PI;	NEG_PI = -PI;	**real**,**parameter**::NEG_PI=-PI
const double TWO_PI=2*PI;	❖	**real**,**parameter**::TWO_PI=2*PI
	type	
typedef int Length;	Length = **integer**;	❖

enum State {
error, warn, ok
};
	State =
	(error, warn, ok);
		❖

	var	
int a, b;	a, b : **integer**;	**integer** a, b
double x;	x : **real**;	**real** x
enum State s;	s : State;	❖
int i = 396;	❖	**integer**::i = 396

- ❑ ANSI C and Fortran: declarations in any order
- ❑ ANSI C is case-sensitive: `foo` and `Foo` or `FOO` are all distinct identifiers!

	ANSI C	Pascal	Fortran
Numeric Ops	+, -, *	+, -, *	+, -, *
Division	/ (real)	/	/
	/ (int)	**div**	/
Modulus	%	**mod**	mod *or* modulo
Exponentation	❖	❖	**
Incr/Decrement	++, --	❖	❖
Bit Operators	~, ^, &, |	❖	not, ieor, iand, ior
Shift Operators	<<, >>	❖	ishft
Arith. Comparison	<, <=, >, >=	<, <=, >, >=	<, <=, >, >=
– Equality	==	=	==
– Unequality	!=	<>	/=
Logical Operators	&&, ||, !	**and**, **or**, **not**	.and., .or., .not. (.eqv. , .neqv.)

- ❑ Pascal also has *sets* and corresponding *set operators* (`+, -, *, in, =, <>, <=, >=`)
- ❑ ANSI C also has `?:` and `,` operators
- ❑ ANSI C also has short-cuts: `a = a` *`op`* `b;` can be written as `a` *`op`*`= b;`
- ❑ ANSI C Precedence rules complicated!
  Practical rule: `*` and `/` before `+` and `-`; put parenthesis, `()`, around anything else!

Basics — Arrays

ANSI C

```
typedef
   int Nums[20];

char c[10];

Nums p, q;

float a[2][3];

p[4] = -78;
a[0][1] = 2.3;
```

Pascal

```
type
  Nums = array [0..19]
         of integer;

var
  c : array [0..9]
      of char;
  p, q : Nums;
a : array [0..1,0..2]
    of real;

p[4] := -78;
a[0,2] := 2.3;
```

Fortran

❖

```
character,dimension(0:9):: c

integer,dimension(0:19):: p, q

real,dimension(0:1,0:2):: a

p(4) = -78
a(0,2) = 2.3
```

- ❑ ANSI C arrays always start with index `0`, Fortran with default index `1`
- ❑ Pascal and Fortran allow *array assignment* between arrays of same type
- ❑ Arrays cannot be returned by functions in ANSI C and Pascal
- ❑ ANSI C: `a[0,1]` is valid expression but `a[0,1]` ≠ `a[0][1]`!

Basics — Records

ANSI C

```
typedef struct {
   int level;
   float temp, press;
   int lights[5];

} Engine;

Engine m1, m2;

printf ("%d",m1.level);
m2.temp += 22.3;
m1.lights[2] = 0;
```

Pascal

```
type
  Engine = record
    level : integer;
    temp, press : real;
    lights : array [0..4]
             of integer;
  end;
var
  m1, m2 : Engine;

write(m1.level);
m2.temp := m2.temp+22.3;
m1.lights[2] := 0;
```

Fortran

```
type Engine
  integer:: level
  real:: temp, press
  integer:: lights(0:4)

end type Engine

type(Engine):: m1, m2

write(*,*) m1%level
m2%temp = m2%temp+22.3
m1%lights(2) = 0
```

- ❑ Pascal `records` cannot be returned by functions

`typedef` of `struct`s

- `struct` names form a separate namespace:

```
struct Engine {
   int level;
   float temp, press;
   int lights[5];
};
```

  - ➠ usage in declarations and definitions requires keyword `struct`:

```
struct Engine m1, m2;
```

- typedef can be used to shorten declarations and definitions:

```
typedef struct Engine Engine;

Engine m1, m2;
```

➠ `typedef` and `struct` declarations can be combined into one construct:

```
typedef
   struct Engine {
      int level;
      float temp, press;
      int lights[5];
   }
Engine;
```

➠ Typically, inner `struct` declaration needs no name:

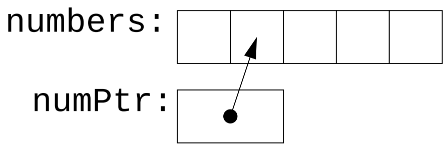
```
typedef struct {
   int level;
   float temp, press;
   int lights[5];
} Engine;
```


Pointers

ANSI C	Pascal	Fortran
`int i;`		`integer,target :: i`
	`var`	
`int *a;`	`a : ^integer;`	`integer,pointer :: a`
`char *b, *c;`	`b, c : ^char;`	`character,pointer :: b, c`
`Engine *mp;`	`mp : ^Engine;`	`type(Engine),pointer :: mp`
`a = &i;`	❖	`a => i`
`*a = 3;`	`a^ := 3;`	`a = 3`
`b = 0;`	`b := nil;`	`nullify(b)`
`mp = malloc(`	`new(mp);`	`allocate(mp)`
`sizeof(Engine));`		
`(*mp).level = 0;`	`mp^.level := 0;`	`mp%level = 0`
`mp->level = 0;`		
`free(mp);`	`dispose(mp);`	`deallocate(mp)`

- ANSI C uses constant `0` as null pointer (often with `#define NULL 0`)
- ANSI C provides `->` short-cut because precedence of the dot is higher than that of `*`
- Fortran 95 has: `b => null()`

```
long numbers[5];
long *numPtr = &(numbers[1]);
```



- ❑ Dereferencing and pointer arithmetic:

```
*numPtr += 2;                /* numbers[1] += 2;        */
 numPtr += 2;                /* numPtr = &(numbers[3]); */
```

- ❑ Similarities
  - ❍ The array name by itself stands for a constant pointer to its first element

    `if (numbers == numPtr) { /*...*/ }`
  - ❍ Can use array (`array[i]`) and pointer syntax (`*(array + i)`) for both

    `numPtr[1] = *(numbers + 2);          /* == 2[numbers] :-) */`
- ❑ Differences
  - ❍ `numbers` only has an *rvalue*: refers to address of beginning of array and cannot be changed
  - ❍ `numPtr` also has an *lvalue*: a `(long *)` is allocated and can be set to address of a `long`

ANSI C	Pascal	Fortran
#include "*file*"	❖	**include** '*file*'
		module *Global*
global constant, type, variable, function and procedure declarations		*global constant, variable decls* **contains** *function, procedure decls*
	(* Pascal comment1 *)	**end module**
int main() /* C comment */	**program** AnyName; { Pascal comment2 } *global constant, type, variable, function and procedure declarations*	**program** *AnyName* ! Fortran comment **use** *Global*
{	**begin**	
local declarations *statements*	*statements*	*local declarations* *statements*
}	**end.**	**end [program]**

- ❑ ANSI C and Fortran: declarations in any order

Functions

ANSI C

```
int F(double x,
      int i)

{
   local decls
   statements incl.
   return expr;
}
int j;
j = 3 * F(2.0, 6);
```

Pascal

```
function F(x:real;
       n:integer):integer;

   local decls
begin
   statements incl.
   F := expr;
end;
var j:integer;
j := 3 * F(2.0, 6);
```

Fortran

```
function F(x,n)
integer F
integer n
real x
local decls

statements incl.
F = expr
return
end
integer j
j = 3 * F(2.0, 6)
```

- Pascal allows the definition of *local* functions, Fortran too with `contains` (but 1 level only)
- Default parameter passing: C and Pascal: by value Fortran: by reference
- Output parameters: C: use pointers Pascal: **`var`**
- Fortran allows additional attributes for parameters: `intent` and `optional`

Procedures

ANSI C

```
void F(int i)

{
   local decls
   statements
}

F(6);
```

Pascal

```
procedure F(i:integer);

   local decls
begin
   statements

end;

F(6);
```

Fortran

```
subroutine F(i)
integer i
local decls

statements
return
end

call F(6)
```

- Pascal allows the definition of *local* procedures, Fortran too with `contains` (but 1 level only)
- Default parameter passing: C and Pascal: by value Fortran: by reference
- Output parameters: C: use pointers Pascal: **`var`**
- Fortran allows additional attributes for parameters: `intent` and `optional`

ANSI C

```
if (a<0)
   negs = negs + 1;

if (x*y < 0)
{
   x = -x;
   y = -y;
}

if (t == 0)
   printf("zero");
else if (t > 0)
   printf("greater");
else
   printf("smaller");
```

Pascal

```
if a < 0 then
   negs := negs + 1;

if x*y < 0 then
begin
   x := -x;
   y := -y
end;

if t = 0 then
   write('zero')
else if t > 0 then
   write('greater')
else
   write('smaller');
```

Fortran

```
if (a < 0) [then]
   negs = negs + 1

if (x*y < 0)
then
   x = -x
   y = -y
end if

if (t == 0) then
   write(*,*) 'zero'
else if (t > 0) then
   write(*,*) 'greater'
else
   write(*,*) 'smaller'
```

- Semicolon is *statement terminator* in ANSI C, *statement separator* in Pascal
- Don't mix up assignment (=) and equality test (==) in ANSI C, as assignment is an *expression* (not a *statement*) and therefore, `if (a = b) {...}` is valid syntax!

ANSI C

```
switch (ch)
{
case 'Y': /*NOBREAK*/
case 'y': doit = 1;
          break;

case 'N': /*NOBREAK*/
case 'n': doit = 0;
          break;

default : error();
          break;
}
```

Pascal

```
case ch of

'Y', 'y':
   doit := true;

'N', 'n':
   doit := false;

otherwise error()

end;
```

Fortran

```
select case (ch)

case ('Y','y')
   doit = .true.

case ('N','n')
   doit = .false

case default
   call error()
end select
```

- `otherwise` (also: `else`) not part of standard Pascal but common extension
- ANSI C only doesn't allow multiple case labels but this can be implemented by "*fall-trough*" property
- Fortran allows *ranges* in case labels: `case ('a' : 'z', 'A' : 'Z')`

Indexed Iteration

ANSI C

```
for (i=0; i<n; i++)
{
    statements
}
for (i=n-1; i>=0; i--)
{
    statements
}
for (i=b; i<e; i+=s)
{
    statements
}
```

Pascal

```
for i:=0 to n-1 do
begin
    statements
end;
for i:=n-1 downto 0 do
begin
    statements
end;
❖
```

Fortran

```
do i=0,n-1

    statements
end do
do i=n-1,0,-1

    statements
end do
do i=b,e-1,s

    statements
end do
```

- ❑ In Pascal and Fortran, the control variable (e.g. `i`) cannot be changed in the loop body
- ❑ Pascal has no support for `step != 1` or `-1`, has to be written as `while` loop
- ❑ The ANSI C `for` loop is actually more powerful as shown here

Controlled-Exit Iteration

ANSI C

```
while (expr)
{
    statements
}

do {
    statements

} while (!expr);

for / while / do {
    continue;
    statements
    break;
}
```

Pascal

```
while expr do
begin
    statements
end;

repeat
    statements

until expr;

❖
```

Fortran

```
do while (expr)

    statements
end do

do
    statements
    if (expr) exit
end do

do
    cycle
    statements
    exit
end do
```

- ❑ *expr* in ANSI C `do` loop must be the negated *expr* of the corresponding Pascal `repeat` loop

Basics — Style

As C and C++ are (like many others) *format-free* programming languages, it is important to program in a ***consistent style*** in order to maintain readable source code.

- One statement per line!
- ***Useful*** comments and ***meaningful*** variable/function names

```
double length;  /* measured in inch */
```

- Indention style (two major styles popular):

```
int func(int i) {
   if (i > 0) {
      return 1;
   } else {
      return 0;
   }
}
```

```
int func(int i)
{
   if (i > 0)
   {
      return 1;
   }
   else ...
```

- Naming of programming language objects

```
const int MAX_BUF = 256;
int my_buffer[MAX_BUF];
int get_buffer();
```

```
const int MAXBUF = 256;
int myBuf[MAXBUF];
int getBuffer();
```


Basics — Access to Command Line Arguments

- Programmer has access to command line arguments through two parameters to `main()`
  - `int argc`: number of command line arguments (including name of executable)
  - `char *argv[]`: array of command line arguments (as character strings)

    `argv[0]`: name of executable
    `argv[1]`: first command line argument
    `...`
    `argv[argc]`: always `(char *) 0`

- Command line parsing should be done via UNIX system function `getopt()`

```
int c, a_flag = 0;

while ((c = getopt(argc, argv, "ab:")) != EOF)
   switch (c) {
   case ’a’: aflag = 1; break;
   case ’b’: b_arg = optarg; break;
   case ’?’: /* error ... */ break;
   }

for ( ; optind < argc; optind++) next_arg = argv[optind];
```

Programming in C++

☆☆☆ Motivation ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

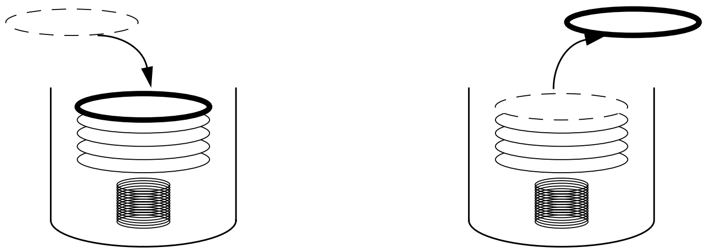
Motivation — Features of ANSI C

- ❑ a small, simple language (by design)
  - ➠ boils down to macros, pointers, structs, and functions
- ❑ ideal for short-to-medium size programs and applications
- ❑ lots of code and libraries written in C
- ❑ good efficiency (a close mapping to machine architecture)
- ❑ very stable (ANSI/ISO C)
- ❑ available for pretty much every computer
- ❑ writing of portable programs possible
  - ❍ ANSI C and basic libraries (e.g. stdio) are portable
  - ❍ however, operating system dependencies require careful design
- ❑ C preprocessor (cpp) is a good, close friend
- ❑ poor type-checking of K&R C (especially function parameters) addressed by ANSI C

➠ **so, what’s the problem? Why C++?**

Motivation — Example: a Stack

Goal: create some C code to manage a stack of numbers



- a *stack* is a simple first-in/last-out data structure resembling a stack of plates:
  - elements are removed or added only at the top
  - elements are added to the stack via a function `push()`
  - elements are removed from the stack via `pop()`
- stacks occur in many software applications: from compilers and language parsing, to numerical software
- one of the simplest container data structures

➠ **sounds easy enough...**

Motivation — Simple Stack in C

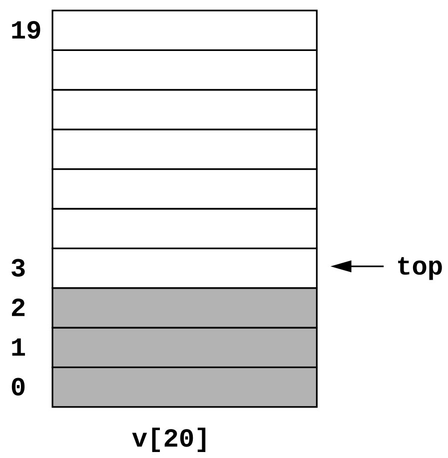
```
typedef struct {
   float v[20];
   int top;
} Stack;

Stack *init(void) {
   Stack *r = malloc(sizeof(Stack));
   r->top = 0;
   return r;
}

void push(Stack *S, float val) {
   S->v[(S->top)++] = val;
}

float pop(Stack *S) {
   return(S->v[--(S->top)]);
}

int empty(Stack *S) {
   return (S->top <= 0);
}
```




```
void finish(Stack *S) {
   free(S);
}
```

Using the Stack data structure in C programs

```
Stack *S;

S = init();                             /* initialize                 */

push(S, 2.31);                          /* push a few elements        */
push(S, 1.19);                          /* on the stack...            */

printf("%g\n", pop(S));                 /* use return value in        */
                                        /* expressions...             */

push(S, 6.7);
push(S, pop(S) + pop(S));               /* replace top 2 elements */
                                        /* by their sum               */

void MyStackPrint(Stack *A) {
   int i;
   if (A->top == 0) printf("[empty]");
   else for (i=0; i<A->top; i++) printf(" %g", A->v[i]);
}
```

➠ **so what’s wrong with this?**

A few gotcha’s...

```
Stack *A, *B;
float x, y;

push(A, 3.1415);                /* oops! forgot to initialize A   */

A = init();
x = pop(A);                     /* error: A is empty!             */
                                /* stack is now in corrupt state  */
                                /* x’s value is undefined...      */

A->v[3] = 2.13;                 /* don’t do this! */
A->top = -42;

push(A, 0.9);                   /* OK, assuming A’s state is valid*/
push(A, 6.1);
B = init();
B = A;                          /* lost old B (memory leak)       */

finish(A);                      /* oops! just wiped out A and B   */
```

- NOT VERY FLEXIBLE
  - fixed stack size of 20
  - fixed stack type of `float`

- NOT VERY PORTABLE
  - function names like `full()` and `init()` likely to cause naming conflicts

- biggest problem: NOT VERY SAFE
  - internal variables of `Stack` are exposed to outside world (`top`, `v`)
  - their semantics are directly connected to the internal state
  - can be easily corrupted by external programs, causing difficult-to-track bugs
  - no error handling
    - ✰ initializing a stack more than once or not at all
    - ✰ pushing a full stack / popping an empty stack
  - assignment of stacks (`A=B`) leads to reference semantics and dangerous dangling pointers


```
typedef struct {
   float* vals;
   int top, size;
} DStack;

DStack *DStack_init(int size) {
   DStack *r = malloc(sizeof(DStack));
   assert (r != 0);
   r->top  = 0;
   r->size = size;
   r->vals = malloc(size*sizeof(float));
   assert (r->vals != 0);
   return r;
}

void DStack_finish(DStack* S) {
   assert (S != 0);
   free(S->vals);
   free(S);
   S = 0;
}
```

```
void DStack_assign(DStack* dst, DStack* src) {
   int i;

   assert (dst != 0 && src != 0);
   free(dst->vals);
   dst->top  = src->top;
   dst->size = src->size;
   dst->vals  = malloc(dst->size*sizeof(float));
   assert (dst->vals != 0);
   for (i=0; i<dst->top; ++i) dst->vals[i] = src->vals[i];
}

void DStack_push(DStack* S, float val) {
   assert (S != 0 && S->top <= S->size);
   S->vals[S->top++] = val;
}

float DStack_pop(DStack* S) {
   assert (S != 0 && S->top > 0);
   return S->vals[--S->top];
}
```



```
DStack *DStack_copy(DStack* src) {
   int i;
   DStack *r;

   assert (src != 0);
   r = malloc(sizeof(DStack));
   assert (r != 0);
   r->top  = src->top;
   r->size = src->size;
   r->vals  = malloc(r->size*sizeof(float));
   assert (r->vals != 0);
   for (i=0; i<r->top; ++i) r->vals[i] = src->vals[i];
   return r;
}

int DStack_empty(DStack* S) {
   assert (S != 0);
   return (S->top == 0);
}
```

Improvements

- ❑ dynamic size
- ❑ primitive error handling (using `assert()`)
- ❑ function names like `DStack_init()` less likely to cause naming conflicts

Remaining Problems

- ❑ not flexible regarding to base type of stack
- ❑ dynamic memory allocation requires `DStack_finish()`or causes memory leak
- ❑ still unsafe

New Problems

- ❑ old application code that used `S->v` no longer works!!
- ❑ copying and assigning `DStack`s requires `DStack_copy()` and `DStack_assign()` or pointer alias problems


```
typedef struct {
   TYPE *vals;
   int size, top;
} GDStack;                    /* Generic(?) Dynamic Stack       */

void GDStack_push(GDStack *S, TYPE val) {...}
```

How to use:

- ❑ put all source into file `GDStack.h`
- ❑ in application code do

```
#define TYPE float
#include "GDStack.h"
GDStack S;                    /* Whoa! a stack of floats        */

#define TYPE int              /* oops! preprocessor warning!    */
                              /* macro TYPE redefined           */
#include "GDStack.h"          /* error: functions redefined!    */
GDStack S2;                   /* nice try, but won’t work       */
```

➠ **only works if only *one* type of stack is used in *one* source file, but that is no good solution...**

```
typedef struct {
   TYPE *vals;
   int size, top;
} GDStack_TYPE;

void GDStack_TYPE_push(GDStack_TYPE *S, TYPE val) {...}
```

How to use:

- put all source into base files `GDStack.h` and `GDStack.c`
- use editor's global search&replace to convert "`TYPE`" into "`float`" or "`int`" and store in new files `GDStack_float.*` and `GDStack_int.*`
- in application code do

```
#include "GDStack_float.h"
#include "GDStack_int.h"
GDStack_float S;           /* hey! a stack of floats!        */
GDStack_int S2;            /* finally! a stack of ints!      */
GDStack_String T;          /* oops! need some more files...  */
```

➠ **works, but is extremely ugly and still has problems...**

- software is constantly being modified
  - better ways of doing things
  - bug fixes
  - algorithm improvements
  - platform (move from Sun to HP) and environment (new random number lib) changes
  - customer or user has new needs and demands
- real applications are very large and complex typically involving more than one programmer
- you can never anticipate how your data structures and methods will be utilized by application programmers
- ad-hoc solutions OK for tiny programs, but don't work for large software projects
- software maintenance and development costs keep rising, and we know it's much cheaper to *reuse* rather than to *redevelop* code

What have we learned from years of software development?

➠ the major defect of the data-structure problem solving paradigm is the *scope* and *visibility* that the key data structures have with respect to the surrounding software system

So, we would like to have ...

❑ **DATA HIDING:**

the inaccessibility of the internal structure of the underlying data type

❑ **ENCAPSULATION:**

the binding on an underlying data type with the associated set of procedures and functions that can be used to manipulate the data (*abstract data type*)

❑ **INHERITANCE:**

the ability to re-use code by referring to existing data types in the definition of new ones. The new data type *inherits* data objects and functionality of the already existing one.

➠ **OBJECTS**

There are many object-oriented languages out there, so why C++?

❑ compromise between elegance and usefulness

❑ Superset of C

❍ C in widespread use

❍ all C libraries easily usable from C++

❑ Cross-platform availability

❑ Mass-market compilers

❑ Wide usage on all platforms

❑ Popular with programmers

❑ Only pay for what you use

❑ New ANSI/OSI standard

So how does C++ help solve our Stack problems?

- ❑ provides a mechanism to describe abstract data types by packaging `C struct` and corresponding member functions together (*classes*)
- ❑ protects internal data structure variables and functions from the outside world (`private` and `protected` *keywords*)
- ❑ provides a mechanism for automatically initializing and destroying user-defined data structures (*constructors* and *destructors*)
- ❑ provides a mechanism for generalizing argument *types* in functions and data structures (*templates*)
- ❑ provides mechanism for gracefully handling program errors and anomalies (*exceptions*)
- ❑ provides mechanism for code reuse (*inheritance*)

Programming in C++

☆☆☆ From C to C++ ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

New Keywords (to C)	New (to ARM)	Used for:
`delete, new`	-	Memory Management
`class, this, private, public, protected`	-	Classes
`catch, try, throw`	-	Exceptions
`friend, inline, operator, virtual`	`mutable`	Class member type qualifier
`template`	`typename, export`	Templates
-	`bool, true, false`	Boolean datatype
-	`const_cast, static_cast, reinterpret_cast`	New style casts
-	`using, namespace`	Namespaces
-	`typeid, dynamic_cast`	RunTime Type Identification
	`explicit`	Constructor qualifier
-	`wchar_t`	Wide character datatype

From C to C++ — Alternative Tokens

C99 [iso646.h]

- Furthermore, *alternative representations* are reserved and shall not be used otherwise:

Alternative:	Primary:	Alternative:	Primary:
`and`	`&&`	`and_eq`	`&=`
`bitor`	`|`	`or_eq`	`|=`
`or`	`||`	`xor_eq`	`^=`
`xor`	`^`	`not`	`!`
`compl`	`~`	`not_eq`	`!=`
`bitand`	`&`		

C99

- Also, in addition to the *trigraphs* of ANSI C, C++ supports the following *digraphs*:

Alternative:	Primary:	Alternative:	Primary:
`<%`	`{`	`<:`	`[`
`%>`	`}`	`:>`	`]`
`%:`	`#`	`%:%:`	`##`

From C to C++ — Comments

C99

New "`//`" symbol can occur anywhere and signifies a comment until the end of line

```
float r, theta;              // this is a comment
```

New "`//`" comment can be nested

```
// int i = 42;               // so is this
```

Of course, there are the still two other ways to denote comments:

- with the familiar `/* */` pair

  ```
  /* nothing new here...      */
  ```

  but careful, these do not nest!

- using the preprocessor via `#ifdef`, `#endif`.
  This is the best method for commenting-out large sections of code.

  ```
  #ifdef CURRENTLY_NOT_NEEDED
    a = b + c;
    x = u * v;
  #endif
  ```

  Remember that the # must be the first non-whitespace character in that line.

- C++ doesn't require that a type name is prefixed with the keywords `struct` or `union` (or `class`) when used in object declarations or type casts.

```
struct Complex { double real, imag; };

Complex c;      // has type "struct Complex"
```

- A C++ `typedef` name must be different from any class type name declared in the same scope, except if the typedef is a synonym of the class name with the same name

```
struct Bar { ... };
typedef struct Foo { ... } Bar;      // OK in C, but not C++
typedef struct Foo { ... } Foo;      // OK in C + C++
```

- In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope.
  In C, an inner scope declaration of a struct tag name never hides an object in an outer scope

```
int x[99];
void f(){
   struct x { int a; };
   sizeof(x);   // sizeof the array in C, sizeof the struct in C++
}
```


Local variable declarations in C++ need **not** be congregated at the beginning of functions:

C99

```
double sum(const double x[], int N) {
   printf("entered function sum().\n");

   double s = 0;              // note the declarations here...
   int i = 42;

   for (int i=0; i<N; i++) {  // also notice the declaration of
      s += x[i];              // loop variable "i"
   }
   int j = i;                 // j == 42!!

   return s;
}
```

- ➠ declare variables close to the code where they are used and where they can be initialized
- ➠ particularly useful in large procedures with many local variables
- ➠ this improves code readability and helps avoid type-mismatch errors
- ➠ Note: scope of `i` in `for` loop changed in C++ standard!
  (was: until end of block: now: end of loop) !

C++ allows to specify default arguments for user-defined functions

- ❑ default value can either specified in function declaration *or* definition, but not in both
  - ➠ (by convention) this is specified in the function declaration in a header file

```
//myfile.h
extern double my_log(double x=1.0, double base=2.71828182845904);

//myfile.cpp
#include "myfile.h"
double my_log(double x, double base) {...}

//main.cpp
#include "myfile.h"

double y = my_log(x);        // defaults to log_e
double z = my_log(x, 10);    // computes log_10
```

- ❑ arguments to the call are resolved *positionally*
- ❑ initialization expression needs not to be constant
- ❑ use only if default values are intuitively obvious and are documented well!
- ❑ the order of evaluation of function arguments (and so their initialization) is unspecified!

C++ introduces a new type: *reference types.* A reference type (sometimes also called an *alias*) serves as an alternative name for the object with which it has been initialized.

- ❑ a reference type acts like a special kind of pointer. Differences to pointer types:
  - ❍ a reference must be initialized (must always refer to some object, i.e., no null reference)
    - ➠ no need to test against 0
  - ❍ once initialized, it cannot changed
  - ❍ syntax to access reference object is the same as for "normal" objects

```
/* references */

int i = 5, val = 10;
int &refVal = val;

int &otherRef;  //error!
refVal++;       //val/refVal now 11
refVal = i;     //val/refval now 5
refVal++;       //val/refVal now 6
```

```
/* pointers */

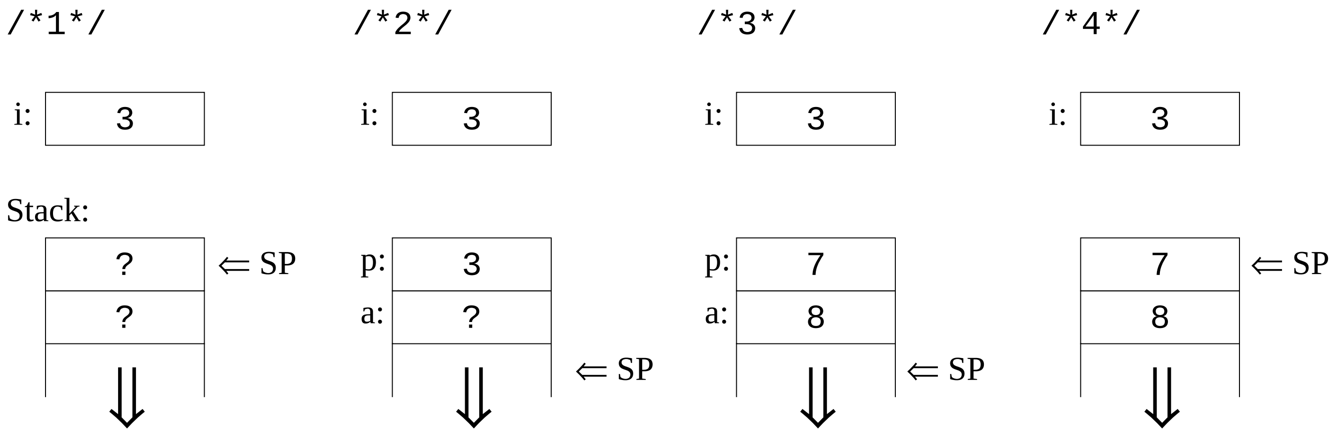
int i = 5, val = 10;
int *ptrVal = &val;

int *otherPtr;          //OK
(*ptrVal)++;
ptrVal = &i;  //val still 11
(*ptrVal)++;  // i now 6!
```

- ❑ The primary use of a reference type is as an *parameter type* or *return type* of a function.
  They are rarely used directly.

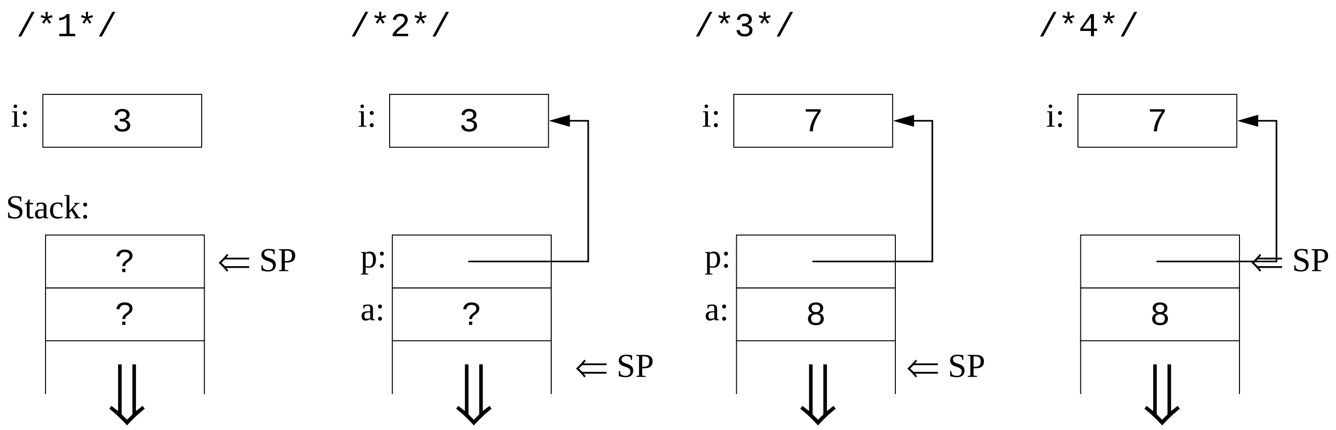
Passing Arguments by Value

```
          void func(int p) {
            int a = p + 5;
/*3*/       p = 7;
          }
/*1*/     int i = 3;
/*2*/     func(i);
/*4*/     ...
```



Passing Arguments by Reference

```
          void func(int* p) {            void func(int& p) {
            int a = *p + 5;                int a = p + 5;
/*3*/       *p = 7;                        p = 7;
          }                              }
/*1*/     int i = 3;                     int i = 3;
/*2*/     func(&i);                      func(i);
/*4*/     ...                            ...
```



Function parameters in C (and C++) are always passed *by value*

- therefore, *output* parameters (and structures/arrays for efficiency) have to be passed as *pointers*
- caller has to remember to pass the *address* of the argument (error-prone!)
- callee has to use `*` and `->` to access parameters in function body (clumsy!)
- *Reference parameters* are denoted by & before function argument (like VAR in PASCAL)

```
void print_complex(const Complex& x) {
   printf("(%g+%gi)", x.real, x.imag);
}

Complex c;
c.real = 1.2; c.imag = 3.4;
print_complex(c);        // note missing &, prints "(1.2+3.4i)"
```

- Side note: `const` is necessary to allow passing constants/literals to the function

- *reference parameters* are only a special case of *reference types*
- references cannot be uninitialized
  - if you need NULL values as parameters, you have to use pointers

C99

Aim of *inline* functions is to reduce the overhead associated with a function call. The effect is to substitute *inline* each occurrence of the function call with the text of the function body.

```
inline double max(double a, double b) {
   return (a > b ? a : b );
}

inline int square(int i) {
   return (i*i);
}
```

They are used like regular functions.

- **Advantages**: removes overhead; provides better opportunities for further compiler optimization
- **Disadvantages**: increases code size; reduces efficiency if abused (e.g. code no longer fits cache)
- The `inline` modifier is simply a *hint*, not a *mandate* to the C++ compiler. Functions which
  - define arrays
  - are recursive or from with address is taken
  - contain statics, switches, (gotos)

  are typically not inlined.

In C++, function *argument types*, as well as the *function name*, are used for identification. This means it is possible to define the same function with different argument types. For example,

```
void swap(Complex& i, Complex& j) {
   Complex t;
   t = i; i = j; j = t;
}

void swap(int& i, int& j) {
   int t;
   t = i; i = j; j = t;
}

void swap(int *i, int *j) {  // possible, but should be avoided
   int t;                    // why? what happens if you pass 0?
   t = *i; *i = *j; *j = t;
}

Complex u, v;  int i, j;

swap(u, v);                  // calls Complex version
swap(i, j);                  // calls integer reference version
swap(&i, &j);                // calls integer pointer version
```


- overloading cannot be based on the return type! (because it is allowed to ignore returned value)
- use overloaded functions instead of `#define func(a,b) ...` because
  - type-safety
  - avoids problems when body uses parameters more than once or has more than one statement

```
#define min(a,b) (a<b?a:b)
int i=3, j=5, k=min(i++, j);         // i now 5!!!
```

- unfortunately, overloaded functions are not as generic as macros, but *function templates* (more on that later) fix that problem nicely:

```
template<class T>
inline void swap(T& i, T& j) {
   T t;
   t = i; i = j; j = t;
}

double x, y; int i, j;

swap(x, y);  // compiler generates double version automatically
swap(i, j);  // again for int
swap(x, i);  // error!
```

- ❑ Choose carefully between function overloading and parameter defaulting

```
void f();                              void g(int x=0);
void f(int);
f();      // calls f()                 g();        // calls  g(0);
f(10);    // calls f(int);             g(10);      // calls g(10);
```

Use argument defaulting when

- ❍ Same algorithm is used for all overloaded functions
- ❍ Appropriate (natural) default argument values exist
- ➠ use function overloading in all other cases

- ❑ Avoid overloading on a pointer and a numerical type
  - ❍ The value `0` used as an actual function argument means `int 0`
  - ❍ You cannot use the value `0` to represent a null pointer passed as an actual argument
  - ❍ A named constant representing "null pointer" (`NULL`) must be typed `void *` and must be cast explicitly to a specific pointer type in most contexts (e.g., as function argument)
  - ❍ You can declare different named constants to represent "null pointers" for different types

Functions aren't the only thing that can be overloaded in C++; operators (such as `+`, `*`, `%`, `[]`, ...) are fair game too. Given `Complex` numbers `u`, `v`, `w`, and `z`, which is easier to read?

```
w = z * (u + v);          or        Complex t;
                                    C_add(&t, u, v);
                                    C_mult(&w, z, t);
```

How did we do it?

```
Complex operator*(const Complex& a, const Complex& b) {
   Complex t;

   t.real = a.real * b.real - a.imag * b.imag;
   t.imag = a.real * b.imag + a.imag * b.real;
   return t;
}

Complex operator+(const Complex& a, const Complex& b) {...}
```

- ❑ only existing operators can be overloaded; creating new ones (e.g., `**`) not possible
- ❑ operator precedence, associativity, and "ary-ness" cannot be changed
- ❑ only overload operator if it is "natural" for the given data type (**avoid surprises for users!**)

C99
[stdbool.h]

- The C++ Standard now includes a `bool` type with the constants `true` and `false`
- Conditionals (`if`, `while`, `for`, `?:`, &&, `||`, `!`) now require a value that converts to `bool`
- Comparison and logical operators (==, `!=`, <, <=, >, >=, &&, `||`, `!`) now return `bool`
- Integral and pointer values convert to `bool` by implicit comparison against zero
- `bool` converts to `int` with `false` becoming zero and `true` becoming one
- Because `bool` is a distinct type, you can now overload on it

```
void foo(bool);
void foo(int);  // error on older compilers! now OK
```


C++ introduces a new concept for handling I/O: *file streams*.

- include the C++ header file `<iostream.h>` instead of `<stdio.h>`
- use the << operator to write data to a stream
- use the >> operator to read data from a stream
- use predefined streams `cin, cout, cerr` instead of `stdin, stdout, stderr`

```
#include <iostream.h>

int main(int argc, char *argv[]) {
   int birth = 1642;
   char *name = "Issac Newton";

   cout << name << " was born " << birth << endl;

   cout << "What is your birth year? ";

   cin >> birth;
}
```

Advantages of C++ stream I/O

❑ **type safety**

```
int i = 5;
float f = 3.4;
```

```
printf("%d %f\n", f, i);           // few compilers catch this!
cout << f << " " << i << endl;       // automatically right
```

❑ **extensibility**: can be extended for user-defined types

```
ostream& operator<<(ostream& s, const Complex& x) {
   s << "(" << x.real << "+" << x.imag << "i)";   // what’s wrong?
   return s;
}
```

creates a new way to print `Complex` numbers in C++:

```
Complex c;
```

```
c.real=2.1; c.imag=3.6;
cout << "c = " << c << endl;         // prints "c = (2.1+3.6i)"
```


❑ Use "`#include <fstream.h>`"

❑ Instantiate object of correct stream class

`ifstream`	Read-only files
`ofstream`	Write-only files
`fstream`	Read/Write files

❑ Typical usage

```
ifstream foofile("foo");      // open existing file "foo" readonly
```

```
ofstream foofile("foo");      // create new file "foo" for writing
                              // overwrite if already existing
```

```
if (!foofile) cerr << "unable to open file ’foo’" << endl;
```

❑ use normal stream operations to read from and write to file

```
foofile << somevalue << anothervalue << endl;
```

```
#include <fstream.h>                  // includes <iostream.h>
#include <stdlib.h>                   // needed for ’exit()’

int main(int, char**) {
  char filename[512];
  int birth = 1642;
  char *name = "Issac Newton";

  cout << "Filename? ";
  if ( !(cin >> filename) ) return 1;

  ofstream out(filename);
  if (!out) {
    cerr << "unable to open output file '"
         << filename << "'" << endl;
    exit(1);
  }
  out << name << " was born " << birth << endl;

  return 0;
}
```


- The C way to request dynamic memory from the heap:

```
int *i = (int *) malloc(sizeof(int));
double *d = (double *) malloc(sizeof(double)*NUM);
free(d);
free(i);
```

- The new C++ way:

```
int *i = new int;
double *d = new double[NUM];
delete [] d;
delete i;
```

- Advantages:
  - type-safe; no type casts needed
  - can be extended for user-defined types
  - handles `new int[0];` and `delete 0;`
  - (takes care of object construction / destruction)
- don’t mix `new/delete` with `malloc/free` [watch out for `strdup()`! (string duplication)]

- Because of a C++ feature called *name mangling* (to support type-safe linking and name overloading), you need a a special declaration to call external functions written in C:

```
extern "C" size_t strlen(const char *s1);
```

- It is also possible to declare several functions as extern "C" at once:

```
extern "C" {
   char *strcpy(char *s1, const char *s2);
   size_t strlen(const char *s1);
}
```

➠ can also be used to "make" a C++ function callable from C

- How to write header files which can be used for C and C++?

```
#ifdef __cplusplus
extern "C" {
#endif

   /* all C declarations come here...                          */

#ifdef __cplusplus
}
#endif
```


- *Include guards* prevent double declaration errors (no longer allowed in C++) and speed up the compilation
- Example:

```
lib1.h:     #include "util.h"
            ...

lib2.h:     #include "util.h"
            ...

main.cpp    #include "lib1.h"
            #include "lib2.h"   // ERROR: double declaration
                                // errors for util.h
```

➠ Use include guards for header files, e.g. `util.h`:

```
#ifndef UTIL_H
#define UTIL_H
   ... contents of util.h here ...
#endif
```

- Typically, system header files already use `extern "C"` and include guards.

- Typical C code used the C preprocessor to define symbolic constants:

```
#define PI 3.1415
#define BUFSIZ 1024
```

- Better approach is to use `const` declarations because
  - it is type-safe
  - compiler knows about it
  - it shows up in the symbol table (debugger knows it too)

```
const float PI = 3.1415;
const int BUFSIZ = 1024;

static char myBuffer[BUFSIZ];        // allowed in C++, not in C
```

- Be careful when defining constant strings (note the **two** `const`)

```
const char* const programName = "fancy_name_here";

const char* programName = "fancy_name_here";  //ptr to const char
char* const programName = "fancy_name_here";  //const pointer
```


- In C, union variables or fields have to be named

```
struct foo {
  union {
    short i;
    char ch[2];
  } buf;
  int n;
} f;

f.buf.i = 0x0001;

if (f.buf.ch[0]==0 &&
   f.buf.ch[1]==1)
  printf("big endian");
else
  printf("little endian");
```

- In C++, union variables or fields can be anonymous

```
struct foo {
  union {
    short i;
    char ch[2];
  };
  int n;
} f;

f.i = 0x0001;

if (f.ch[0]==0 && f.ch[1]==1)
  cout << "big endian";
else
  cout << "little endian";
```

- In C++, a function declared with an empty parameter list takes no arguments.
  In C, an empty parameter list means that the number and type of the function arguments are "unknown"

```
int f();   // means int f(void) in C++
           //       int f(unknown) in C
```

- In C++, the syntax for function definition excludes the "old-style" C function.
  In C, "old-style" syntax is allowed, but deprecated as "obsolescent."

```
void func(i)      instead of      void func(int i)
int i;                            { ... }
{ ... }
```

- In C++, types may not be defined in return or parameter types.
- `int i; int i;` ("tentative definitions") in the same file is not allowed on C++

C99

- Implicit declaration of functions is not allowed
- Banning implicit `int`

- general expressions are allowed as initializers for static objects (C: constant expressions)
- `sizeof('x') == sizeof(int)` in C but not in C++ (typeof `’x’` is `char`)
- Main cannot be called recursively and cannot have its address taken
- Converting `void*` to a pointer-to-object type requires casting
- It is invalid to jump past a declaration with explicit or implicit initializer
- `static` or `extern` specifiers can only be applied to names of objects or functions (not to types)
- `const` objects must be initialized in C++ but can be left uninitialized in C
- In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.
- C++ objects of enumeration type can only be assigned values of the same enumeration type.

```
enum color { red, blue, green } c = 1;  // valid C, invalid C++
```

```
static const int FStack_def_size = 7;

struct FStack {
   float* vals;
   int top, size;
};

FStack *init(int size = FStack_def_size) {
   FStack *r = new FStack;
   assert (r != 0);
   r->top  = 0;
   r->size = size;
   r->vals = new float[size];
   assert (r->vals != 0);
   return r;
}

void finish(FStack* S) {
   assert (S != 0);
   delete [] S->vals;  delete S;
   S = 0;
}
```



```
void assign(FStack* dst, FStack* src) {
   assert (dst != 0 && src != 0);
   delete [] dst->vals;
   dst->top  = src->top;
   dst->size = src->size;
   dst->vals = new float[dst->size];
   assert (dst->vals != 0);
   for (int i=0; i<dst->top; ++i) dst->vals[i] = src->vals[i];
}

void push(FStack* S, float val) {
   assert (S != 0 && S->top <= S->size);
   S->vals[S->top++] = val;
}

float pop(FStack* S) {
   assert (S != 0 && S->top > 0);
   return S->vals[--S->top];
}
```

```
FStack *copy(FStack* src) {
   assert (src != 0);
   FStack *r = new FStack;
   assert (r != 0);
   r->top  = src->top;
   r->size = src->size;
   r->vals = new float[r->size];
   assert (r->vals != 0);
   for (int i=0; i<r->top; ++i) r->vals[i] = src->vals[i];
   return r;
}
bool empty(FStack* S) {
   assert (S != 0);
   return (S->top == 0);
}
```

Programming in C++

☆☆☆ Classes ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

Classes — Introduction

- a *class* can be characterized by three features:
  - Classes provide method for logical grouping;
    Groupings are of both data and functionality (C `struct` + associated functions)
  - A class defines a new, *user-defined type*
  - Defines *access rights* for members (allowing data hiding, protection, **...**)
- *New defined class* `:=` C++ specification of *abstract data type*
- instance of class `:=` *object*

➠ A class should describe a *set* of (related) objects (e.g., complex number**s**)

- *Data members* `:=` variables (also fields) of any type, perhaps themselves of other classes
- *Member functions* `:=` actions or operations
  - usually applied to data members
  - important: called through objects

➠ Use data members for variation in *value*, reserve member functions for variation in *behavior*

- Normally build by class library programmers

- Explicit aim of C++ to support *definition* and *efficient use* of such user-defined types very well
- *Concrete Data Types* `:=` *user-defined types* which are as "*concrete*" as builtin C types like `int`, `char`, or `double`
  - ➠ should well behave anywhere a built-in C type is well behaved
- Typical examples:

❍ Complex numbers	❍ Points	❍ Coordinates
❍ (*pointer*, *offset*) pairs	❍ Dates	❍ Times
❍ Ranges	❍ Links	❍ Associations
❍ Nodes	❍ (*value*, *unit*) pairs	❍ Disc locations
❍ Source code locations	❍ BCD characters	❍ Currencies
❍ Lines	❍ Rectangles	❍ Rationals
❍ Strings	❍ Vectors	❍ Arrays . . .

➠ A typical application uses a few directly and many more indirectly from libraries

This all happens to a programming language object (note the symmetry):

① allocation of memory to hold object data

② construction (initialization)

③ usage

④ destruction

⑤ deallocation of memory

Example: Lecture:

① reservation: reserve/occupy lecture room

② setup: clean blackboard, switch on projector, ...

③ usage: give lecture, take and answer questions, ...

④ breakdown: switch off projector, sort slides, ...

⑤ departure: leave lecture room

Example: an object of type `int`:

```
{
   int i;          ① allocation of object i of type int
   i = 5;            ② initialization
   /*...*/
   j = 5 * i;          ③ usage
   /*...*/           ④ [destruction not needed for built-in types]
}                  ⑤ leaves scope: deallocation of memory
```

For classes: special member functions

❑ *constructor*: automatically called after allocation for class specific initialization ②

❑ *destructor*: automatically called prior to deallocation for class specific clean-up ④

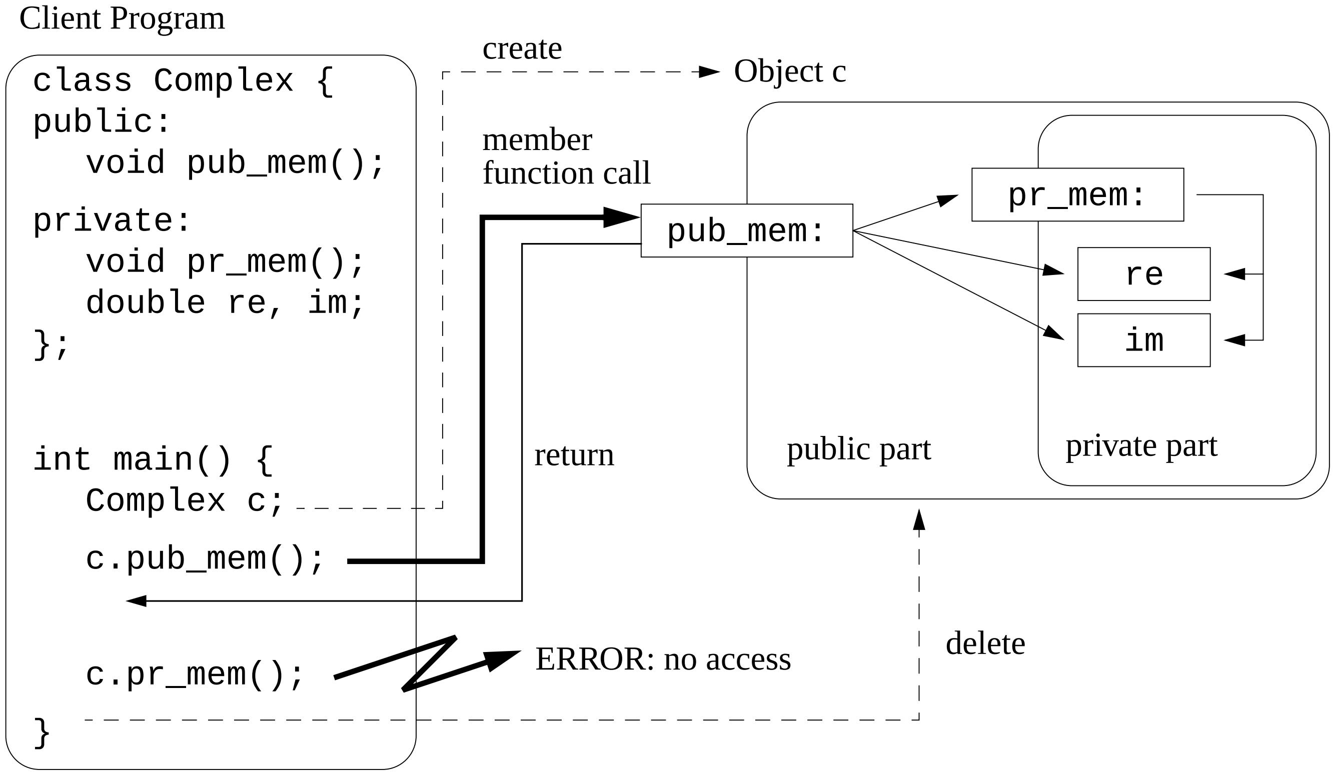

```
class Complex {              // new type with name "Complex"
public:                      // public interface, can be
     /*...*/                 // accessed by all functions
private:                     // hidden data/functions for use
     double re;              // by member functions only
     double im;
};                           // note ";" versus end of function
```

❑ `public` / `protected` / `private` is *mainly* a permission issue

members of base class which are	can be accessed in client code	can be accessed inside class (member + friend functions)
`public`	✔	✔
`private`	✘	✔
`protected`	✘	✔

❑ `public` part describes *interface* to the new type clients can / have to use
`protected`/`private` determine the *implementation*

Classes — Conceptual Organization of a C++ Program



Classes — Constructors

- Responsibility of *all* constructors (often abbreviated ctor)
  - initialization of *all* data members (put object in well-defined state)
  - perhaps: set globals, e.g., number of objects
  - perhaps: special I/O – for debugging only (why?)
- Important: constructor is "called" by an object and effects the object (like all member functions)
- Has the same name as the class
- Does not specify a return type or return a value (not even `void`)
- Cannot be called explicitly, but automatically (Stage ②) after creation of new object (Stage ①)
- Different constructors possible through overloading
- Two special constructors: *default* and *copy*
  - ➠ if not specified, compiler generates them automatically if needed
  - default ctor: generated as no-op if no other ctor (including copy ctor) exists otherwise error
  - copy ctor: calls recursively copy ctor for each non-static data member of class type

Default constructor `:=` willing to take no arguments

```
class Complex {

public:
   Complex(void) {            // Complex default constructor
      re = 0.0; im = 0.0;
   }

private:
   /* ... */
};

// ... usage in an application
Complex c1, *cp = &c1;        // just as with int
```

- ❑ Rule: member function definitions included in class definitions are automatically inline!
- ❑ `re` and `im`: declared by and belong to calling object (`c1` above)
- ❑ Note: constructor *not* called for `cp`!

➠ **How about constructors with client initialization?**

"regular" constructor `:=` initialize object with user supplied arguments

```
class Complex {
public:
   Complex(double r) {              // Construct Complex out of
      re = r; im = 0.0;             // real number
   }

   Complex(double r, double i) {    // Construct Complex out of
      re = r; im = i;               // real and imaginary part
   }

};

// ... usage in an application
Complex c3(3.5), c4 = -1.0,
                 c5(-0.7, 2.0);
```

- ❑ Any constructor call with a single argument can use "`=`" form (e.g., `c4`)
  - ➠ Such a constructor is also used by the compiler to automatically convert types if necessary!

➠ **These three constructors can be combined by using default arguments!**

```
class Complex {
public:
   Complex(double r = 0.0, double i = 0.0) {
      re = r; im = i;
   }
private:
   /* ... */
};
// ... usage in an application
Complex c2,                  // NOT: c2()!
        c3(3.5),             // c3 calls ctor with argument 3.5
        c4 = -1.0, c5(-0.7, 2.0);
```

- ❑ Supply default arguments for all data members ➟ can be used as *default constructor*
- ❑ Note: `c2()` declares a function which returns a `Complex`

➠ **How about same-class, object-to-object initialization, e.g.,** `int i, j = i;` **?**

- ❑ Purpose: to initialize with another (existing) `Complex` object ("cloning")

```
Complex(const Complex& rhs) {        // why reference type?
   re = rhs.re; im = rhs.im;
}
// ... usage in an application
Complex c6(c3), c7 = c2;     // just like: int i(-4), j = i;
complex_sin(c7);    // copy ctor called if pass/return by value
```

- ❑ Has the same name as the class (as every constructor) and has exactly one argument of type "reference to const classtype"
- ❑ Argument passed by reference
  - ❍ for effiency and mimic built-in type syntax
  - ❍ to avoid calling constructor/destructor for temporary `Complex` object (on stack)
  - ❍ even worse: to avoid recursive copy constructor calling!!
- ❑ `rhs.re` and `rhs.im` (rhs `:=` right hand side)
  - ❍ since constructor is member function access even to different object’s `private` members
  - ❍ but `rhs.` necessary to specify other object

```
class Complex {
public:
   ~Complex(void) {}         // Complex dtor
};
```

- ❑ Placed in `public:` section
- ❑ Has the same name as class prepended with ~
- ❑ Does not specify a return type or return a value (not even `void`)
- ❑ Does not take arguments
- ❑ Cannot be overloaded!!!
- ❑ Is *not* called explicitly but automatically (Stage ④) prior to deallocation of an `Complex` object (Stage ⑤)
- ❑ Primary purpose/responsibility: cleanup (nothing needed for `Complex`)
- ❑ Default compiler-generated (no-op) destructor

- ❑ What do we have so far?

```
class Complex {
public:
   // default constructor
   Complex(double r = 0.0, double i = 0.0) {
      re = r; im = i;
   }
   // copy constructor
   Complex(const Complex& rhs) {
      re = rhs.re; im = rhs.im;
   }
   // destructor
   ~Complex(void) {}
private:
      double re;
      double im;
};
```

➠ **But this is kind of boring...**

- To access (i.e., read from) data members

```
class Complex {

public:
   double real(void) {return re;}
   double imag(void) {return im;}
};

// ... called by
double d = c2.real(), e = cp->imag();
```

- Required, as client has no access to `private` fields
- Access syntax is the same as for C `struct`: ("`.`" for member, "`->`" for pointer member access)
- Why functions?
  - Consistency
  - Flexibility (check parameter validity; implement no access / read-only, read-write access)
  - can be replaced by computation (e.g. calculate polar coordinates out of `(re,im)`)
- Choose user-friendly, meaningful names for functions (moreso than for data members)

- To modify (i.e, write to) data members

```
class Complex {

public:
   void real(double r) {re = r;}
   void imag(double i) {im = i;}
};

// ... called by
c5.imag(c1.real());
```

- Exploiting function overloading
- How about object-to-object assignment? Involves:

```
c1.real(c6.real());

c1.imag(c6.imag());
```

  - ➠ 2 access member function, 2 modify member function calls

```
class Complex {
public:
   Complex& operator=(const Complex& rhs) {
      if (this == &rhs) return *this;   // time-saving self-test
      re = rhs.re; im = rhs.im;         // copy data
      return *this;                     // return "myself"
   }
};
```

```
// ... called by
c1 = c6;        // c1 calls operator with argument c6
                // can also be written as:  c1.operator=(c6);
```

- Function name: `operator=`
- if not defined, compiler automatically defines one (doing memberwise copying)!
- Reference (i.e., lvalue) returned for speed and daisy-chaining:

  ```
  c6 = c5 = c3;   // c6.operator=(c5.operator=(c3));
  ```

  ➠ **match return value and argument type!**

- reference argument
  - for speed and to avoid constructor/destructor calls
  - to enable self-test
- New keyword `this` is
  - defined for every body of member functions
  - a short-cut for "*pointer to myself*" i.e., "*pointer to calling object*"
- Reasons for self-test
  - speed
  - another reason later
- Possible self-assignment situations

  ```
  Complex& c8 = c3, *pc = &c2;
  ```

  ```
  // later ...
  c3 = c8;     // or vice versa
  *pc = c2;    // ditto
  ```

➠ **Don’t forget the `operator=()` self-test!**

- Construction: `Complex c9 = c2;`
  or: `complex_sin(c2);`

- Assignment: `c9 = c2;`

	create new object	can daisy-chain	self-assignment check	returns reference to `*this`
copy constructor	✔	–	–	–
`operator=()`	–	✔	✔	✔

- To test equality to another `Complex` object

- For now, just a field-wise "and" test

- User can define == operator

```
class Complex {
public:
   bool operator==(const Complex& rhs) {
      return ((real() == rhs.real()) && (imag() == rhs.imag()));
   }
};
```

```
// ... called by
if (c7 == c2)
   cout << "Yup, they’re equal!" << endl;
```

- `real()` will be inline, so no slowdown, but makes maintenance easier
  - ➠ **Use public interface when possible and not harmful!**

- e.g., Addition: `c6 = c1 + c8;`

```
class Complex {
public:
   const Complex operator+(const Complex& rhs) {
      return Complex(real()+rhs.real(), imag()+rhs.imag());
   }
};
```

- Why return const object?
  - disallow expressions like `c6 = (c1 + c8)++; c6++++;`
- Why call constructor?
  - result of addition is new `Complex` object; unlikely that this already exists and it is `const`
- And why return by value (and not by reference)?
  - Local value is deallocated before leaving function scope
  - returned reference would be undefined!


```
class Complex {   // comments missing for space conservation
public:
   Complex(double r = 0.0, double i = 0.0) {re = r; im = i;}
   Complex(const Complex& rhs) {re = rhs.re; im = rhs.im;}
   ~Complex(void) {}
   double real(void) {return re;}
   double imag(void) {return im;}
   void real(double r) {re = r;}
   void imag(double i) {im = i;}
   Complex& operator=(const Complex& rhs);
   bool operator==(const Complex& rhs) {
      return ((real()==rhs.real()) && (imag()==rhs.imag()));
   }
   const Complex operator+(const Complex& rhs) {
      return Complex(real()+rhs.real(), imag()+rhs.imag());
   }
   // define all other missing operators and functions here...
private:
      double re, im;
};
```

Programming in C++

☆☆☆ Pointer Data Members ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

Pointer Data Members — Pointer vs. non-Pointers

What are the issues if a class includes data members which are of pointer type?

- ❑ Free store allocation (and deallocation)
  - ❍ Pointers usually indicate this
  - ❍ Who should do this and keep track? (class or user?)
  - ❍ How about the constructors and destructor?
- ❑ Pointer value vs. what it points to
  - ❍ For object assignment, do we want memberwise assignment of pointer fields?
    - ➠ No!
  - ❍ What should `operator==()` mean?
    - ➠ *identity*? ➠ equal if same object (i.e. same address)
    - ➠ *equality*? ➠ equal if same contents (i.e. same value or state)

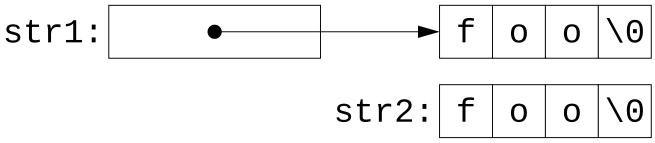
➠ Let’s look at the popular `String` class

Pointer Data Member — Motivation `String` Class

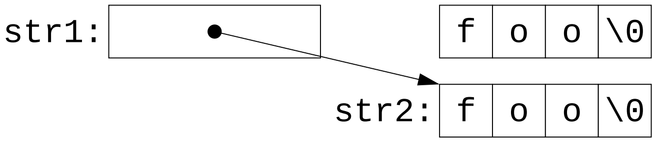
- C++ character strings are really *NUL terminated arrays of char* (as in C)
  - ➠ Beware of difference between arrays and pointers!

```
char *str1   = "foo";
char str2[4] = "foo";           // = {'f', 'o', 'o', '\0'};
```

```
// sizeof(str1) != sizeof(str2) sometimes!
```




```
if (str1 != str2) str1 = str2;   // compares/copies pointer!
```



  - ➠ arrays cannot be assigned

```
char str3[4] = "bar";
str2 = str3;                         // ERROR!
```


Pointer Data Members — More Pointer Assignment Problems

```
char *str1 = new char[42];   // areas are allocated
char *str2 = new char[7];

str2 = str1;                 // address is copied over
                             // old contents of str2 lost

delete [] str1;              // area is deallocated

cout << str2[7] << endl;     // area is accessed, but undefined
```

- General problem/hazard: two pointers to same free store allocation
  - Usually, on assignment, we want copy of *info* pointed to (*"deep" copy*), not copy of *address* (bitwise *"shallow" copy*)
  - memory leaks
- Deep copy:
  - ➠ deallocate old info if necessary (to avoid memory leak)
  - ➠ new allocation
  - ➠ assertion of success
  - ➠ copy info over

➠ So, to work with C++ character strings, need to use `new/delete` and *ANSI C String library*!

❑ Frequently used string functions:

- ❍ `strcmp` compares strings, returns `<0, 0, >0` if `s1<s2, s1==s2, s1>s2`

  `int strcmp(char *s1, char *s2);`
- ❍ `strcpy` copies strings, `dst=src, dst` must have enough space

  `char *strcpy(char *dst, char *src);`
- ❍ `strcat` appends a copy of `src` to the end of `dst, dst` must have enough space

  `char *strcat(char *dst, char *src);`
- ❍ `strlen` returns the number of bytes in `s` (without the terminating `NUL` character)

  `size_t strlen(char *s);`
- ❍ `strdup` returns a pointer to a new string which is duplicate of `s`

  ➠ uses `malloc`, user responsible for `free`ing space

  `char *strdup(char *s);`
- ❍ ...

❑ But even with ANSI C String library functions, our examples still not work as expected:

```
char *str1 = "foo";
str2[4] = "foo";
str3[4] = "bar";

if (strcmp(str1, str2) != 0)
  strcpy(str1, str2);       //ERROR: typically core dumps

strcpy(str2, str3);
```

❑ Even more problems:

```
char *str1;
char str2[3];

strcpy(str1, "foo");         //ERROR: no space allocated for str1

strcpy(str2, "foo");         //ERROR: not enough space
                             //       (forgot terminating NUL)
```

➠ **need better, automatically managed `String` objects!**

The Life of a `String` Object

What memory allocation issues occur during the lifetime of a `String` object?

- ① allocation of memory for hold pointer to `String` data (`sizeof(char *)`)
  - ② construction: perhaps `new` to allocate space for data
    - ③ usage: perhaps other free storing (`operator=()`, ...)
  - ④ destruction: perhaps `delete` to deallocate data
- ⑤ deallocation of memory for pointer

➠ **Goal: conceal all free store business from client (user)**

`String` Class Skeleton

```
class String {
public:
   // ...        // constructors, destructor, ...
private:
   char *str;    // place to store string value
};
```

❑ For information hiding, don’t let client know value of `str`

❑ What functions should be in `public`?

➠ **A good start: what do constructors look like?**

- Default constructor

```
String() {                     // Default: empty string
   str = new char[1];
   assert(str != 0);           // checking
   str[0] = '\0';              // = ""
}
```

- Char* constructor

```
String(const char *s) {
   if (s) {                           // safety
      str = new char[strlen(s) + 1];  // allocating
      assert(str != 0);               // checking
      strcpy (str, s);                // copying
   } else {
      str = new char[1];
      assert(str != 0);               // checking
      str[0] = '\0';                  // = ""
   }
}
```

➠ Use of default argument would save extra default constructor code

- Copy Constructor

```
String(const String& rhs) {
   int len = strlen(rhs.str);
   str = new char [len + 1]; // allocating
   assert(str != 0);         // checking
   strcpy (str, rhs.str);    // copying
}
```

- Recall:
  - Signature: *classname*(const *classname*&);
  - Copy constructor initializes from another (existing) object
    - ➠ no need for null pointer check as `rhs.str != 0` always

➠ Default constructor, `char*` constructor, and copy constructor share a lot of code

➠ Use private helper function (e.g., `set_str`)

```
#include <assert.h>
#include <string.h>

// safe (checking s, not str) and sound, all in one place

void set_str(const char *s) {

   if (s) {                          // safety
      int len = strlen(s);
      str = new char [len + 1];      // allocating
      assert(str != 0);              // checking
      strcpy (str, s);               // copying

   } else {
      str = new char[1];
      assert(str != 0);              // checking
      str[0] = '\0';                 // = ""

   }

}
```

➠ Can now be used by constructors and others....

❑ Default and copy constructor

```
class String {
public:
   String(const char *s = 0) { set_str(s); }
   String(const String& rhs) { set_str(rhs.str); }
private:
   char *str;
   void set_str(const char *) { /* ... */ }
};
```

Recall: constructors responsible for initialization

➠ pointer initialization: allocate memory from free store, or set to 0

❑ Destructor

```
~String(void) {delete [] str;}
```

When `String` object leaves scope, destructor (responsibility: cleanup) is called

➠ Rule: call delete for each pointer data member

Pointer Data Members — Access Member Function

```
class String {
public:
   const char *c_str(void) {
      return (const char *) str;
   }
};
```

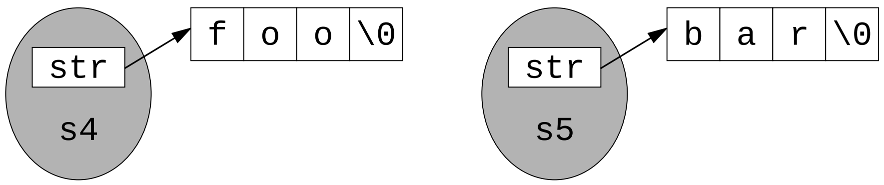
- With `Complex real()`, a *copy* of `re` is returned
- So to with `c_str()`, a *copy* of `str` pointer is returned
- *But*, copy refers to address of character array itself
- Client could change character array indirectly! Ouch!
- `const` ensures that this will not happen:

```
String s1, s2(s1), s3 = "I am s3";
char *cs1 = s2.c_str();                  // error!
const char *cs2 = s1.c_str();            // OK
cout << s3.c_str() << endl;              // OK
```

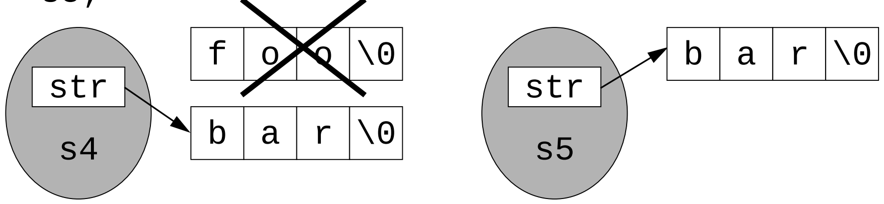

Pointer Data Members — `String` Assignment Operator

```
String& operator=(const String& rhs) {      //poor implementation
   delete [] str;
   set_str (rhs.str);
   return *this;
}
```

```
String s4("foo"), s5("bar");
```




```
s4 = s5;
```



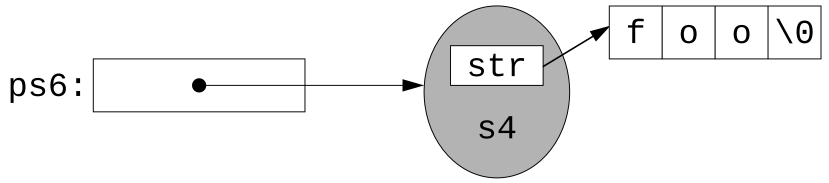
- Recall: reference returned for daisy-chaining

```
s4 = s3 = s2 = "And now something completely different...";
```

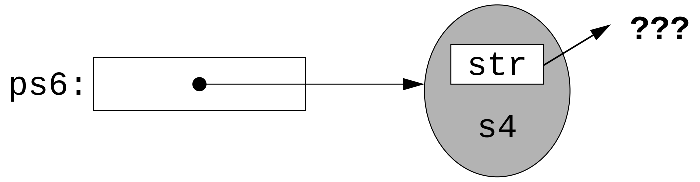
```
String& operator=(const String& rhs) {     //poor implementation
   delete [] str;
   set_str (rhs.str);
   return *this;
}
```

- What if

```
String* ps6 = &s4;
```




```
// and later
*ps6 = s4;
```



➠ **We need to check for self assignment!**


```
class String {
public:
   String& operator=(const String& rhs) {
      if (this != &rhs) {
         delete [] str;
         set_str(rhs.str);
      }
      return *this;
   }
};
// ... later
s1 = s2;
```

- Note: Self-test performed on object addresses, not `(char *)` value
- Earlier reason for self-test: speed
- Now also: avoid catastrophic delete before `set_str()`

Pointer Data Members (char *) to String Assignment

- We can initialize a `String` with a character string; how about assignment?

```
s4 = "zap";     // works!!
```

- How does this work? ➠ Compiler automatically generates code along the lines of:

```
String tmp("zap");
s4.operator=(tmp);
tmp.~String();
```

- For efficiency, we can do better by implementing a character string assignment operator:

```
class String { public:
   String& operator=(const char *rhs) {
      if (str == rhs) return *this;
      delete [] str;
      set_str(rhs);
      return *this;
   }
};
s4 = "baz";
```

- Modify member function `c_str(char*)` not necessary because of `operator=(char*)`

Pointer Data Members operator=() Lessons

- There can be several overloaded versions of the assigment operator.
  `T& operator=(const T&)` is sometimes called *copy assigment operator*
- Do not forget the self-test!
- Have `operator=()` assign all data members (just like constructors)
- Another difference to copy constructor (recall earlier table):
  - ➠ to `delete` old free store allocation
- After `operator=()` invocation
  - ❍ corresponding pointer fields are *not* the same, but
  - ❍ typically point to copies of the same data (deep copy)
- Without copy constructor and `operator=()` definitions
  - ❍ compiler *will* perform memberwise copying (bitwise)
  - ❍ leads to incorrect multiple pointers to same data
  - ➠ $\exists$ **pointer data fields** $\Rightarrow$ **define copy constructor and `operator=()` !!!!**

Other `String` Capabilities

- Obvious one: `strlen()`:

```
class String {
public:
   int length(void) { return strlen(str); }
};
```

- Equality testing (`operator==()`)
- String concatenation (`operator+()`, `operator+=()`)
- Finding a `char` in a `String`
- Finding a `String` (or `(char *)`) in a `String`
- Change to lower/upper case
- ...

➠ **Let’s define the first two operators**

Equality Test Operator

- `operator=()` is different for classes with pointer fields
  - ➠ What about `operator==()`?
- Definition: equality if dereferenced data is the same

```
class String {
public:
   bool operator==(const String& rhs) {
      if (this == &rhs)                              // time saver
         return true;

      return strcmp(str, rhs.str) == 0;

   }

};
```

```
class String {
public:
   const String operator+(const String& rhs) {
      char* r = new char [length() + rhs.length() + 1];    // temp
      assert(r != 0);

      strcpy(r, str);                      // init with lhs
      strcat(r, rhs.str);                  // add rhs

      String result(r);                    // construct String object

      delete [] r;                         // free temporary

      return result;
   }
};

String a = "Hi, ", b = "mom!", c = a + b;
```

- Recall: cannot return reference!
- Using `operator+()` results in 2 calls each to copy constructor and destructor!!


```
#include <assert.h>
#include <string.h>

class String {                 // bad! missing documentation

public:
   String(const char *s = 0) { set_str(s); }
   String(const String& rhs) { set_str(rhs.str); }
   ~String(void) { delete [] str; }
   const char *c_str(void) { return (const char *) str; }
   int length(void) { return strlen(str); }
   String& operator=(const char *rhs) { /* ... */ }
   String& operator=(const String& rhs) { /* ... */ }
   bool operator==(const String& rhs) { /* ... */ }
   const String operator+(const String& rhs) { /* ... */ }

   /* ... */

private:
   void set_str(const char *) { /* ... */ }
   char *str;

};
```

```
#include <iostream.h>
#include "String.h"

int main(int argc, char *argv[]) {
   String red = "red", blue = "blue", purple = red, clear;

   // oops, fixing
   purple = "purple";

   if ((red + blue) == purple)
      cout << "It’s a MIRACLE! How did you get "
           << purple.c_str() << "?" << endl;

   return 0;
}
```


- ❑ Use `String` class instead of C++ character strings
  - ➠ fortunately, new C++ standard library includes `string` class
  - ➠ get public domain standard C++ `string` class emulation
    if your compiler doesn’t have one yet!
  - ➠ Learn about it and *use* it!

  [see also `string` class description in appendix]

- ❑ Possible enhancements/optimizations:
  - ❍ store length
  - ❍ copy into old space if possible (and avoid `delete/new`)
  - ❍ reference counting (see chapter "*More Class Examples*")

Programming in C++

☆☆☆ More on Classes ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

More on Classes — Scope and Related Global Functions

❑ Member function bodies in class definition violate principle of "*separation of implementation and interface*" and make it hard to read

```
class String {
   int length(void) {
      return strlen(str);
   }
   ...
};
```

➠ Move member functions bodies outside of class definition

➠ Class scope: members associated to class with scope operator "`::`"

```
class String {
   int length(void);
   ...
};

int String::length(void) {          // this is String::length()
   return strlen(str);              // not a global function
}                                   // length()!
```

- Scope operator without class name refers to global scope

```
int x[99];
void foo() {                    // compare to example
   struct x {int a;};           // on page 55
   sizeof(::x);
}
```

- Recall: building a class library for others to use
  - `.h` `#included` for interface
  - `.cpp` `#includes` `.h` and is compiled away for linking

- Therefore, other related global functions, e.g.,
  - declared in `String.h`

    `bool sound_same(const String&, const String&);`

  - defined in `String.cpp`

    `bool sound_same(const String& s1, const String& s2) { ... }`

- Class interface `foo.h`:
  - `class foo` definition and definitions of `inline` functions
  - declarations of global functions
- Class implementation `foo.cpp`:
  - `#include "foo.h"`
  - (missing) definitions of member functions
  - definitions of global functions
- User program `prog.cpp:`
  - `#include "foo.h"`
  - Implementation of user code (including `main()`) that uses `class foo`

- Compilation (on UNIX):
  - Compile class implementation: `CC -O -c foo.cpp`
  - Later compile prog.cpp: `CC -O -c prog.cpp`
  - And finally link: `CC -O -o prog prog.o foo.o`

- `const` allows the compiler to help enforce the "read-only" constraint

```
const Complex cplx_pi(3.14);
cplx_pi = 3.0;              // error! good!
double pi = cplx_pi.real();  // error! oops!
```

- Problem: compiler doesn't know whether member function changes object
  - ➠ doesn't allow to call member function on `const` object
- Solution: declare member function to be `const`
  - ❍ Member function *declaration*: add `const` after parameter list

```
class Complex {
   double real(void) const;
};
```

  - ❍ Member function *definition*: add `const` between parameter list and function body

```
double Complex::real(void) const { return re; }
const char *String::c_str(void) const {
        return (const char*) str; }
```


- `const` member function can also be invoked on non-`const` objects, but not vice-versa

```
String s1 = "bim";
const String s2 = "bam";
const char *str1 = s1.c_str();      // OK!
const char *str2 = s2.c_str();      // OK!
s1 = s2;                            // OK!
s2 = s1;                            // error!
```

- It is possible to overload a member function based on its `const`ness

```
class String {
public:
   do_special(void) { /*will be called for non-const strings*/ }
   do_special(void) const { /*will be called for const strings*/ }
   ...
};
```

- *Global* and *static* functions cannot be declared `const`. Why?

```
#include <assert.h>
#include <string.h>

class String {              // bad! missing documentation

public:
   String(const char *s = 0) { set_str(s); }
   String(const String& rhs) { set_str(rhs.str); }
   ~String(void) { delete [] str; }
   const char *c_str(void) const { return (const char *) str; }
   int length(void) const;
   String& operator=(const char *rhs);
   String& operator=(const String& rhs);
   bool operator==(const String& rhs) const;
   const String operator+(const String& rhs) const;

   /* ... */

private:
   void set_str(const char *);
   char *str;

};
```



```
#include "String.h"

int String::length(void) const {
   return strlen(str);
}

String& String::operator=(const String& rhs) {
   if (this != &rhs) {
      delete [] str;
      set_str(rhs.str);
   }
   return *this;
}

String& String::operator=(const char *rhs) {
   /* ... */
}

/* ... */
```

Literals

- ❑ Not possible to define literals (constants) of a user–defined class

```
Complex c1 = 1 + 2i;          // NOT POSSIBLE!!!
```

- ❑ Literals of the basic types can be used if conversion constructor provided

```
Complex c2 = 5.0;             // calls Complex(double,double);
```

Pointer to Class Objects

- ❑ Possible to dynamically allocate objects of class type

```
Complex *cp1 = new Complex;          // uses default constructor
Complex *cp2 = new Complex(1.0,4.5); // uses "normal" constructor
```

- ❑ Possible to dynamically allocate arrays of objects of class type

```
Complex *carray1 = new Complex[3];   // array of 3 Complex
Complex *carray2 = new Complex[3](1.0,4.5);   // NOT POSSIBLE !!!
```

  ➠ if initialization of different values needed, use **array of pointers**:

```
Complex *carray[3];
for (int i=0; i<3; i++) carray[i] = new Complex(5*i*i);
```


- ❑ Fixed-length arrays of class objects can be declared and initialized like arrays of built-in types:

```
#include "Complex.h"

int main (int argc, char *argv) {
   int i=1, ia1[3], ia2[] = { 5, i, ia1[2] };

   Complex c=1.0, ca1[3], ca2[] = {
               5.0,                  // default constructor
               Complex(3.4, 4.5),    // default constructor
               c,                    // copy constructor
               ca1[2]                // copy constructor
            };
}
```

  - ❍ Use braces as usual for array initialization
  - ❍ Individual element initialization (any constructor)
    - ✰ single argument: as is
    - ✰ multiple arguments: use constructor form

- Situation: designing new function related to class
- Question: make the function global or a member?
- Answer in general:
  - make it a member
  - keep things object-oriented and neat
- E.g., for matrix multiplication: $C_{l \times n} = A_{l \times m} \times B_{m \times n}$

```
Matrix A(3,2), B(2,7);
// ... later
Matrix C(A.rows(), B.cols()) = A * B;
```

- Neatness: object *calls* instead of being an argument

➠ **But in two situations this fails...**

- We currently have

```
const Complex operator+(const Complex& rhs) {
   return Complex(real()+rhs.real(), imag()+rhs.imag());
}
// ... called by
c6 = c1 + c8;   // fine and dandy
```

- What if we want mixed-type addition?

```
c6 = c1 + 19;   // still OK, or
c6 = 19 + c1;   // error! Why isn't addition commutative?
```

- To understand, look at explicit function call

```
c6 = c1.operator+(19);   // implicit int -> Complex conversion
c6 = 19.operator+(c1);   // no int class, so no member function
```

  - Rule: no implicit conversions on invoking object

➠ Solution: define global function for `Complex` addition

- Global form for addition

```
const Complex operator+(const Complex& lhs, const Complex& rhs) {
   return Complex(lhs.real()+rhs.real(), lhs.imag()+rhs.imag());
}
```

- Note single argument of `Complex::operator+()`, and two here
- Now both arguments of `operator+()` can be converted
- Reminder: even though global, declare in `Complex.h`, define in `Complex.cpp`

- Don't forget to define related operators: e.g., define `operator+=()` out of `operator+()`

```
const Complex& operator+=(const Complex& rhs) {//very bad idea!!!
   return *this = *this + rhs;
}
```

- Can we do even better? ➠ yes, start with `operator+=()`

```
inline const Complex& operator+=(const Complex& rhs) {
   re += rhs.re; im += rhs.im;
   return *this;
}
```


- Define `operator+()` out of `operator+=()`

```
const Complex operator+(const Complex& lhs, const Complex& rhs) {
   return Complex(lhs) += rhs;
}
```

- Also, define `operator++()` out of `operator+=()`

```
const Complex& operator++() { // prefix form: increment and fetch
   *this += 1;                // should be better: this->re += 1;
   return *this;
}
```

- Then, define `operator++(int)` out of `operator++()`
  - ➠ to distinguish postfix from prefix form artificial (not-used) `int` argument is used

```
const Complex operator++(int) { // postfix form: fetch and incr.
   Complex oldValue = *this;
   ++(*this);
   return oldValue;
}
```

  - ➠ **always define both operators otherwise old compilers use operator++() for both forms!**

➠ Do the same for `-`, `*`, `/`, ...

- ❑ What if there is no `Complex::real()` and `Complex::imag()`?
- ❑ Alternative: use `Complex::re` and `Complex::im`
- ❑ Problem
  - ❍ data members are (properly) hidden in `private:`
  - ❍ our function is now global
  - ➠ **no access!**
- ❑ Solution: declare our global function to be a `friend`

➠ How do I apply for `friend`ship?

- ❑ Remain global functions (or member function in other class)
- ❑ Have access to members in `private:` (and `protected:`) of `friend` class
- ❑ Declared "`friend`" in class (e.g., in `Complex`)

```
friend const Complex operator+(const Complex&, const Complex&);
```

  - ➠ usually all friends together at beginning
- ❑ "`friend class foo;`" within "`class bar`" definition
  - ➠ befriends all `foo`'s member functions (but *not* `foo`'s friends) to `bar`
- ❑ Opposite is not true
- ❑ "`friend`" is not transitive!

`foo` friend-of `bar` $\wedge$ `bar` friend-of `zap` =|⇒ `foo` friend-of `zap`

➠ **And the second reason for a non-member function...**

- Goal: print objects
  - to appear in a natural format
  - in an easy-to-use fashion
- Appearance: depends on object
  - `Complex`: parenthesized, comma-separated list: `(-4.3, 94.3i)`
  - `String`: just the pointer field (`str`)
- Ease-of-use: can we get something like:

```
cout << "this is c3: " << c3 << endl;
cout << "    and s2: " << s2 << endl;  // no c_str()
```

➠ **Let's try a member function**

- First try: here is the prototype (i.e., declaration)

```
ostream& Complex::operator<<(ostream& output);
```

- Recall: as a member function, `Complex` object invokes (calls) the function
- Therefore: function invocation

```
c3 << cout;     // most unnatural!
```

- Again we want object to be an argument (and `cout` the invoker)
  - ➠ Therefore, make `operator<<()` global for `Complex` objects

```
ostream& operator<<(ostream& output, const Complex& rhs) {
   return output << "(" << rhs.real() <<
                    "," << rhs.imag() << "i)";
}
```

- Again, if no `Complex::real()` and `Complex::imag()`
  - ➠ need to make this a `friend`

➠ **So, in summary...**

- In general: keep functions as members if possible
- Reason for non-member function
  - do not want object to invoke, rather be an argument
- Reason for global function to be a `friend`
  - access to hidden data
- For defining function `func` for objects of class `foo`

```
if (  (func needs type conversion on left most argument) ||
      (func is operator>> || operator<<) ) {
   make func global;
   if (func needs access to non-public members of foo)
      make func a friend of foo;
} else
   make func a member;
```

➠ **And why is it so important to hide the data?**

Client access only through library-programmer-supplied member functions:

- **Simplicity**: client needs only know member functions, and not data implementation details
- **Uniformity**: client *always* accesses members (object) via function
  - `Complex::real()`, and not `Complex::re`
- **Protection**: to disallow client access (none, reading, writing, both)
  - writing to `String::str` without proper allocation
  - keeping data members in sync (e.g., a `String::len` data member for speed optimization)
- **Correctness:** only correctness of the interface functions need to be proven
- **Forward compatibility**: future class library changes localized to member functions
  - will not need to change client code (as long we change the interface)
  - errors("well, not in your code, of course")
  - data member name changes (`Complex::re` → `Complex::_re`)
  - Algorithm changes
  - Underlying data structure changes (`Stack<vector>` → `Stack<linked_list>`)

- e.g., Addition: `c6 = c1 + c8;`

```
const Complex operator+(const Complex& lhs, const Complex& rhs) {
   return Complex(lhs) += rhs;
}
```

- Why return const object?
  - ➠ disallow expressions like `c6 = (c1 + c8)++; c6++++;`

- Why call constructor?

➠ result of addition is new `Complex` object; very unlikely that this already exists and it is `const`

- And why return by value (and not by reference)?

```
//first wrong way with allocating object on the stack
const Complex& operator+(const Complex& lhs, const Complex& rhs) {
   Complex temp(lhs); temp += rhs;
   return temp;
}
```

➠ `temp` is deallocated before leaving function scope, so returned reference is undefined!


```
//second wrong way with allocating object on the heap
const Complex& operator+(const Complex& lhs, const Complex& rhs) {
   Complex *result = new Complex(lhs);
   result += rhs;
   return *result;
}
```

➠ new problem: who will call `delete` for the return object?

- ➠ memory leak!

- Even if caller would be willing to take the address of the function’s result and call `delete` on it (astronomically unlikely), complicated expressions yield unnamed temporaries that programmers would never be able to get at. Example:

```
Complex w, x, y, z;

w = x + y + z;                  // how to get at the result of +’s?
```

➠ **Don’t try to return a reference when you must return an object!**

Optimized `String` Concatenation Operator

❑ `String` concatenation operator `String::operator+()` very inefficient!

➠ add *private* helper constructor (dummy argument necessary for distinction)

```
class String {
   ...
private:
   String(const char *s, bool) { str = s; }
};
```

➠ use new constructor for avoiding extra copying in temporary `result` variable

```
const String String::operator+(const String& rhs) {
   char *r = new char [length() + rhs.length() + 1];
   assert(r != 0);
   strcpy(r, str);
   strcat(r, rhs.str);
   return String(r, true);
}
```


Summary Operator Overloading

❑ Operators that CAN be overloaded:

Operators(@)	Expression	As member function	As global function
`+ - * & ! ~ ++ --`	`@a`	`(a).operator@()`	`operator@(a)`
`+ - * / % ^ & | < > ==` `!= <= >= << >> && || ,`	`a@b`	`(a).operator@(b)`	`operator@(a, b)`
`+= -= *= /= %=` `^= &= |= <<= >>=`	`a@b`	`(a).operator@(b)`	`operator@(a, b)`
`=`	`a=b`	`(a).operator=(b)`	
`[]`	`a[b]`	`(a).operator[](b)`	
`()`	`a(b, ...)`	`(a).operator()(b, ...)`	
`-> ->*`	`a@b`	`(a.operator@())@b`	
`++ --`	`a@`	`(a).operator@(0)`	`operator@(a, 0)`
`new delete` `new[] delete[]`	➠ see extra slides		

❑ Operators that SHOULD NOT be overloaded: `&&` `||` `,` and global `@=` operators

❑ Operators that CANNOT be overloaded: `.` `.*` `::` `?:` `sizeof` `throw` `typeid`

Besides the usual implicit conversions (`int`→`double`, `char`→`int`, ...), C++ compilers perform implicit conversions using the following member functions:

- single (non-default) argument constructors with argument type ≠ class type
  - ➠ *to* class type *from* another type: `classname::classname(anothertype) {}`

```
void foo(const String& s);
foo("I am not a String");    // calls String::String(const char*)
```

- implicit type conversion operators
  - ➠ *from* class type *to* another type: `classname::operator anothertype () {}`

```
Complex::operator String() const {   // no return type!!!
   char *s = new char[32];
   sprintf(s, "(%f,%fi)", real(), imag());
   String res(s);
   delete [] s;
   return res;
}
Complex cx(1.2,3.4);
foo(cx);
```


- Guideline: if it is necessary to convert a `T` into some other class `U`,
  the conversion should be handled by class `U` (through conversion constructor)

  Exceptions:
  - ✰ class `U` source not available
  - ✰ `U` is built-in C type

- Problem: unwanted conversions / ambiguities likely
  - ➠ define explicit conversion function, e.g., `toType()`

    `String Complex::toString() const;`
  - ➠ define conversion constructor `explicit` (recent addition to C++ Standard)

    `explicit Complex::Complex(double, double);`
  - ➠ **use conversion operators sparingly!**

- Another example: we can now re-write `String::c_str()` as conversion function:

```
String::operator const char *() { return (const char *)str; }
```

- Often con/destructors call `new` / `delete` (e.g., for `String`)
- Opposite is true as well: for classes, `new` / `delete` call con/destructors of the class

```
// allocates memory and calls default constructor
String *sp1 = new String;

// allocates memory and calls appropriate constructor
String *sp2 = new String("hello");

// allocates memory for 10 strings and
// calls default constructor on each of them
String *sp3 = new String[10];

// call destructor as well as deallocate memory
delete sp1;
delete sp2;
delete [] sp3;   // call destructor for every element in array
```

- Additional benefits over C's `malloc()` and `free()`

- Passing an object by value (default), to or from a function, invokes copy constructor.
  And a constructor call will be followed by a destructor invocation

```
class Coord { ... double x, y, z; };
class Triangle { ... Coord v1, v2, v3; };
Triangle return_triangle (Triangle t) { return t; }
```

➠ Eight (copy) constructors and eight destructors called

➠ Solution: pass by reference (0 additional invocations)

```
Triangle& return_triangle (const Triangle& t) { return t; }
```

- If possible, pass parameters by `const` reference to enable
  - processing of `const` objects
  - automatic parameter conversions (non-`const` would change generated temporary only)
- Recall: do not ***return*** references to
  - local objects
  - object allocated in function from free store
  - ➠ when *new* object is needed, return it by value

```
class Empty {};
```

is actually (some member functions are automatically generated by the compiler if necessary):

```
class Empty {
public:
  Empty() {}   // constructor (only if there is no other ctor)
  ~Empty() {}  // only in derived classes if base class has dtor
  Empty(const Empty& rhs) {         // copy constructor
    foreach non-static member m: m(rhs.m); }
  Empty& operator=(const Empty& rhs) {    // assignment operator
    foreach non-static member m: m = rhs.m; }
  Empty* operator&() { return this; }    // address-of operators
  const Empty* operator&() const { return this; }
};
```

- What if you do not want to allow the use of this functions (e.g., assignment)?
  - ➠ declare them `private` and do *not* define them!

```
private:
  Empty& operator=(const Empty& rhs);
```



```
class NamedData {
private:  String name;
          void *data;
public:   NamedData(const String& initName, void *dataPtr);
};
```

- Version *Assignment*: uses `String::String()` + `String::operator=()`

```
NamedData::NamedData(const String& initName, void *dataPtr) {
            name = initName; data = dataPtr;
}
```

- Version *Initialization*: uses `String::String(const String &)`

```
NamedData::NamedData(const String& initName, void *dataPtr) :
            name(initName), data(dataPtr) {}
```

- Rule: Prefer initialization to assignment in constructors
- Also, initialization form must be used for reference or const members!
- Note, initializations done in order of definition in class, not in order of the initialization list

Naming of Members

- Approach 1:
  - member *mem* / argument *mem*
  - access function `get`*mem*`()` / `set`*mem*`()`

```
class Point {
private:
  int x, y;
public:
  Point(int x=0, int y=0) :
                x(x), y(y) {}
  void setx(int x) {Point::x=x;}
  void sety(int y) {this->y=y; }
  int getx() { return x; }
  int gety() { return y; }
};
```

- Approach 2:
  - member *mem*`_` / argument *mem*
  - access function *mem*`()` overloading

```
class Point {
private:
  int x_, y_;
public:
  Point(int x=0, int y=0) :
            x_(x), y_(y) {}
  void x(int x) { x_=x; }
  void y(int y) { y_=y; }
  int x() { return x_; }
  int y() { return y_; }
};
```

- Approach 3: **???**

➠ **Do it consistently!**

Inline Member Functions – Revisited

- Inline for short and sweet functions (no loops, ...)
- Member function definition in
  - `.h`: function `inline` if inside class definition *or* declared `inline`
  - `.cpp`: function not `inline`; even if declared `inline` in `.h`
    (no expansion definition for compilation)
- Recall and beware: `inline` is only a suggestion to the compiler!
- If compiler cannot `inline` a function
  - copy of function in every module that `#includes .h`
  - makes function static to avoid linkage problem
- Sometimes, you cannot set breakpoints on `inline` functions in debuggers
  - ➠ make it easy to switch between `inline` / not `inline`
  - ➠ put `inline` definitions in separate file `.inl` which is included in `.h`
  - ➠ doesn’t clutter up the class definition

- `foo.inl`:
  - definitions of `inline` functions
- `foo.h`:
  - `class foo` definition
  - declarations of non-`inline` and global functions
  - `#include "foo.inl"` (in normal case)
- `foo.cpp`:
  - `#include "foo.h"`
  - `#include "foo.inl"` (during debugging)
  - definitions of non-`inline` and global functions
- `foo.test.cpp` [optional, but recommended]
  - `#include "foo.h"`
  - contains `main()` with code which tests all the functionality of `foo`


```
#ifndef COMPLEX_H
#define COMPLEX_H

class Complex {
public:
   Complex(double r = 0.0, double i = 0.0);
   Complex(const Complex& rhs);
   ~Complex(void);
   double real(void);
   double imag(void);
   void real(double r);
   void imag(double i);
   Complex& operator=(const Complex& rhs);
   bool operator==(const Complex& rhs);
   // ...
};

#ifndef NO_INLINE
#  include "Complex.inl"
#endif
#endif
```

```
#ifndef COMPLEX_INL
#define COMPLEX_INL


inline Complex::Complex(double r, double i) {re = r; im = i;}
inline Complex::Complex(const Complex& rhs) {
   re = rhs.re; im = rhs.im;
}

inline Complex::~Complex(void) {}

inline double Complex::real(void) {return re;}

inline double Complex::imag(void) {return im;}

inline void Complex::real(double r) {re = r;}

inline void Complex::imag(double i) {im = i;}


#endif
```



```
#include "Complex.h"
#ifdef NO_INLINE
#  define inline
#  include "Complex.inl"
#endif

bool Complex::operator==(const Complex& rhs) {
  return ((real()==rhs.real()) && (imag()==rhs.imag()));
}

const Complex Complex::operator+(const Complex& rhs) {
  return Complex(real()+rhs.real(), imag()+rhs.imag());
}

Complex& operator=(const Complex& rhs) {
  if (this == &rhs) return *this;
  re = rhs.re; im = rhs.im;
  return *this;
}
```

- Normal compiling: `CC -c Complex.cpp`
- Compiling for debugging: `CC -c -g -DNO_INLINE Complex.cpp`

- ❑ What if one wants a global variable for a class, e.g.,
  - ❍ `num_numbers` for `Complex` (probably in/decremented in con/destructors)
  - ❍ `max_length` for `String` (probably set once at beginning of `main()`)
- ❑ How does one associate it with the class?
- ❑ And making sure to only have one copy? otherwise wasteful and error prone
- ➠ **static data members** (also called "*class variables*")

- ❑ New (third) type of `static`
- ❑ *Declaration*: in class definition (in `.h`)

```
static int num_numbers;
```

- ❑ *Definition*: (only once) needed elsewhere (in `.cpp`, ***not*** in `.h`)

```
int Complex::num_numbers = 0;
```

- ❑ Is just a global variable (i.e., could be accessed without any class instance)
- ❑ But has protection like any other data member
  - ➠ keep it out of `public:`

- ❑ Like `static` data members, it is possible to declare `static` member functions

```
class Complex : {
public:
   static void print_num() { cout << num_numbers; }
   // ...
};
```

- ❑ Associated with their class as a whole
- ❑ Is like global function, **but**
  - ❍ can be `private` or `protected`
  - ❍ doesn’t pollute global namespace
- ❑ Invoked any time class declaration in scope
  - ➠ can be accessed without any class instance

```
Complex::print_num();
```

- ❑ `this` pointer not defined
  - ➠ can only refer to static members of its class
  - ➠ `const static` member functions not possible

- To create an object on the heap, use the `new` operator
  - `new` operator is built into the language
  - cannot be overloaded
  - if invoked,
    1.) allocates memory for the object using operator `new`
    2.) calls constructor of the object

- operator `new`
  - can be used to allocate raw memory

    ```
    void *raw_mem = operator new(50*sizeof(char));
    ```
  - can be overloaded
    - ➠ never overload *global* operator `new`
    - ➠ should only be done on a per class basis (e.g., for efficiency)

    ```
    void *operator new(size_t);
    ```

- Same rules apply to `delete`

- Overloading operator `new` and `delete` on per class basis
- Example:

```
class Complex {
private:
   union {                      // private data used either for
      double re, im;            // - old data members
      Complex *next;            // - pointer in freelist
   };
   static Complex* headOfFreelist;   // class wide freelist
   static const int BSIZ = 256;      // allocation block size

public:
   // overloaded new / delete declarations
   static void *operator new(size_t size);
   static void operator delete(void *deadobj, size_t size);

   // ...
};

Complex* Complex::headOfFreelist = 0;
const int Complex::BSIZ;
```

```
void *Complex::operator new(size_t size) {
   // use global new if called from derived class
   if (size != sizeof(Complex)) return ::operator new(size);
   Complex *p = headOfFreelist;

   // if p is valid, return next element from freelist
   if (p) {
      headOfFreelist = p->next;
   } else {
      // allocate next block
      Complex *newBlk = ::operator new(BSIZ * sizeof(Complex));
      if (newBlk == 0) return 0;

      // link memory chunks together for free list
      for (int i=1; i<BSIZ-1; ++i) newBlk[i].next = &newBlk[i+1];
      newBlk[BSIZ-1].next = 0;

      // return first block; point freelist to second
      p = newBlk; headOfFreelist = &newBlk[1];
   }
   return p;
}
```



```
void Complex::operator delete(void* deadObj, size_t size) {
   // allow null pointers
   if (deadObj == 0) return;

   // use global delete if called from derived class
   if (size != sizeof(Complex)) {
      ::operator delete(deadObj);
      return;
   }

   // add to front of freelist
   Complex* dead = (Complex *) deadObj;
   dead->next = headOfFreelist;
   headOfFreelist = dead;
}
```

- Both the keywords `class` and `struct` can be used to declare new user-defined types:
  - C++ `struct`s *can* have member functions, constructors, statics, overloaded operators, ...
- The only difference is the default permission:
  - `class` members are by default `private`
  - `struct` members are by default `public`

```
class foo {
  int i;   // private
public:
  int j;   // public
};
```

```
struct foo {
  int i;   // public
private:
  int j;   // private
};
```

- Use `class` for defining new data types
- Use `struct`
  - to define plain data records (especially if they must be compatible with C code)
  - for defining small, auxiliary helper types (to express that they are no full-blown types)

- A *declaration* tells compilers about the *name* and *type* of an object, function, class, or template, but nothing more. These are declarations:

```
extern int x;                    // object declaration

int numDigits(int number);       // function declaration

class String;                    // class declaration
```

- A *definition* provides compilers with further details. For an object, the definition is where compilers *allocate memory* for the object. For a function or a function template, the definition provides the code body. For a class or a class template, the definition lists the members of the class or template:

```
int x;                           // object definition

int numDigits(int number) {      // function definition
   ...
}

class Clock {                    // class definition
public:
   ...
};
```

- ❑ Regular pointers

```
int i;                          void foo(int i) {...}
int *ptr = &i;                  void (* f_ptr)(int) = &foo;
*ptr = 5;                       (*f_ptr)(7);
```

- ❑ Pointer to class members

```
class A {
public:
  int i;
  void foo(int i) {...}
};

int A::*m_ptr = &A::i;          void (A::* mf_ptr)(int) = &A::foo;

A a, *a_ptr = new A;

a.*m_ptr = 5;                   (a.*mf_ptr)(7);
a_ptr->*m_ptr = 5;              (a_ptr->*mf_ptr)(7);
```


- ❑ For a `class Foo` we typically need the following members:
  - ❍ Default constructor (only if reasonable default exists!):
    `Foo(void);` or `Foo(type = default);`
  - ❍ (Conversion) constructors:
    `Foo(builtin_type);` or `Foo(const another_type&);`
  - ❍ Equality and in-equality operators (<, >, <=, >= too if Foo has total order)
    global functions if conversion on left-most argument is needed
    ```
    bool operator==(const Foo&) const;
    bool operator!=(const Foo&) const;
    ```
  - ❍ Access functions to class members:
    ```
    builtin_type getmem1() const;
    const builtin_type* getmem2() const;
    const another_type& getmem3() const;
    ```
- ❑ Global Input and Output operators:
  ```
  ostream& operator<<(ostream&, const Foo&);
  istream& operator>>(istream&, Foo&);
  ```

- If `class Foo` has pointer data members, we also need:
  - Copy constructor: `Foo(const Foo&);`
  - Assignment operator (self-test!) `Foo& operator=(const Foo&);`
  - Destructor: `~Foo();`

- For mathematical classes we also define (the same for `-`, `*`, `/`):

```
const Foo& Foo::operator+=(const Foo&) { /*...*/ }

const Foo operator+(const Foo& lhs, const Foo& rhs) {
   return Foo(lhs) += rhs;
}

const Foo& Foo::operator++() {
   *this += 1; return *this;
}

const Foo Foo::operator++(int) {
   Foo old(*this); ++(*this); return old;
}
```

Programming in C++

☆☆☆ More Class Examples ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

`FStack` Class Interface (.h)

```
#ifndef FSTACK_H
#define FSTACK_H

class FStack {
public:
   FStack(int sz = FStack_def_size);
   FStack(const FStack& src);
   ~FStack();
   FStack& operator=(const FStack& src);
   void push(float val);
   float pop();
   bool empty();

private:
   static const int FStack_def_size = 7;

   float* vals;
   int top, size;

   void init(int tp, int sz, float* vs);
};

#endif
```

- Default constructor, helper function and standard member functions

```
FStack::FStack(int sz) { init(0, sz, 0); }

void FStack::init(int tp, int sz, float* vs) {
   top  = tp;
   size = sz;
   vals = new float[size];
   assert (vals != 0);
   for (int i=0; i<top; ++i) vals[i] = vs[i];
}

void FStack::push(float val) {
  assert (top <= size);
  vals[top++] = val;
}

float FStack::pop() {
  assert (top > 0);
  return vals[--top];
}

bool FStack::empty() { return (top == 0); }
```


- `FStack` has pointer data member
  - ➠ need copy constructor, destructor, and assignment operator

```
FStack::FStack(const FStack& src) {
   init(src.top, src.size, src.vals);
}

FStack::~FStack() { delete [] vals; }
```

  - ➠ assignment operator: don’t forget self test, deep copy, and to return `*this`!

```
FStack& FStack::operator=(const FStack& src) {
   if ( this != &src ) {
      delete [] vals;
      init(src.top, src.size, src.vals);
   }
   return *this;
}
```

- Don’t forget to *define* static data members!

```
const int FStack::FStack_def_size;
```

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include "String.h"

class Employee {
public:
   Employee(const String& name, int id);
   String name() const;
   int ident() const;
   const Employee* manager() const;
   void setManager(const Employee* mgr);
   bool isDirector() const;

private:
   Employee& operator=(const Employee&);
   Employee(const Employee&);
   String ename;
   int eid;
   const Employee* emgr;
};

#endif
```


- Employee has pointer data member: why no copy constructor, destructor?
  - ➠ Pointer "used as" pointer / link / reference (not for implementation of dynamic storage)!
  - ➠ initialize to 0 or address of other object
  - ➠ make copy constructor and assignment operator private if it makes sense

- Constructor: use member initialization lists for efficiency!

```
Employee::Employee(const String& name, int id)
     : ename(name), eid(id), emgr(0) {}
```

- Access functions: `const`

```
String Employee::name() const { return ename; }
int Employee::ident() const { return eid; }
const Employee* Employee::manager() const { return emgr; }
```

- Other members:

```
void Employee::setManager(const Employee* mgr) { emgr = mgr; }
bool Employee::isDirector() const { return (emgr != 0); }
```

- Allow multiple inclusion of class header file; Include necessary system headers

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream.h>

class Rational {
```

- private part: class specific data + any necessary helper functions

```
private:
   long num;    // numerator
   long den;    // denominator (keep > 0!)

   long gcd(long, long);     // helper function for normalization

public:
```

- Define standard member functions: constructors

```
   Rational() : num(0), den(1) {}
   Rational(long n, long d = 1);
   // compiler generated:
   // Rational(const Rational& rhs) : num(rhs.num),den(rhs.den){}
```


- Standard member functions: destructor, assignment operators

```
   // ~Rational(void) {}                          // compiler generated

   // Rational& operator=(const Rational& rhs);  // compiler gen.
   Rational& operator=(long rhs);
```

- Access functions: `const` + typically inline!

```
   long numerator(void) const { return num; }
   long denominator(void) const { return den; }
```

- Unary operators: `const` + return value!

```
   Rational operator+(void) const { return *this; }
   Rational operator-(void) const { return Rational(-num, den); }
   Rational invert(void) const { return Rational(den, num); }
```

- Binary short–cut operators: not `const` + return `const` reference + take `const` reference!

```
   const Rational& operator+=(const Rational& rhs);
   const Rational& operator-=(const Rational& rhs);
   const Rational& operator*=(const Rational& rhs);
   const Rational& operator/=(const Rational& rhs);
```

- Binary short–cut operators for mixed mode arithmetic

```
const Rational& operator+=(long rhs);
const Rational& operator-=(long rhs);
const Rational& operator*=(long rhs);
const Rational& operator/=(long rhs);
```

- Increment / decrement iterators: not `const` + return `const`

```
const Rational& operator++();
const Rational  operator++(int);
const Rational& operator--();
const Rational  operator--(int);
```

- Conversion operator: `const` + no return type!

```
  // -- better implemented as explicit conversion
  // -- function toDouble (see below)
  // operator double(void) const { return double(num)/den; }
};
```


- Binary operators: global to get conversion on both arguments + return `const` value!

```
const Rational operator+(const Rational& l, const Rational& r);
const Rational operator-(const Rational& l, const Rational& r);
const Rational operator*(const Rational& l, const Rational& r);
const Rational operator/(const Rational& l, const Rational& r);
```

- Boolean operators: global to get conversion on both arguments + take `const` reference!

```
bool operator==(const Rational& lhs, const Rational& rhs);
bool operator!=(const Rational& lhs, const Rational& rhs);
bool operator<=(const Rational& lhs, const Rational& rhs);
bool operator>=(const Rational& lhs, const Rational& rhs);
bool operator<(const Rational& lhs, const Rational& rhs);
bool operator>(const Rational& lhs, const Rational& rhs);
```

- Output operator: global + take `const` reference!

```
ostream& operator<< (ostream& s, const Rational& r);
```

- Other global functions: take `const` reference

```
Rational rabs(const Rational& rhs);
```

- Multi-line inline function definitions (should really be in `Rational.inl`)
  - Assignment operators
    (no self–test as there are no pointer members and probably no speed improvement)

```
// compiler generated:
// inline Rational& Rational::operator=(const Rational& rhs) {
//    num = rhs.num; den = rhs.den;
//    return *this;
// }
```

```
inline Rational& Rational::operator=(long rhs) {
   num = rhs; den = 1;
   return *this;
}
```

  - Explicit conversion function `Rational` → `double`: take `const` reference

```
inline double toDouble (const Rational& r) {
   return double(r.numerator())/r.denominator();
}
```


  - Explicit conversion functions `Rational` → `long`: take `const` reference

```
inline long trunc(const Rational& r) {
   return r.numerator() / r.denominator();
}
```

```
inline long floor(const Rational& r) {
   long q = r.numerator() / r.denominator();
   return (r.numerator() < 0 && r.denominator() != 1) ? --q : q;
}
```

```
inline long ceil(const Rational& r) {
  long q = r.numerator() / r.denominator();
  return (r.numerator() >= 0 && r.denominator() != 1) ? ++q : q;
}
```

- Explicit conversion function `double` → `Rational`

```
Rational toRational(double x, int iterations = 5);
```

- Dont't forget this! ;-)

```
#endif   // RATIONAL_H
```

❑ Greatest common divisor: euclid’s algorithm

➠ keep class `Rational` local

```
long Rational::gcd(long u, long v) {
   long a = labs(u), b = labs(v);
   long tmp;

   if (b > a) {
      tmp = a; a = b; b = tmp;
   }
   for(;;) {
      if (b == 0L)
         return a;
      else if (b == 1L)
         return b;
      else {
         tmp = b; b = a % b; a = tmp;
      }
   }
}
```


❑ Put object in well–defined state

```
#include <stdlib.h>

Rational::Rational(long n, long d) { // default values in header!
   if (d == 0L) {
      cerr << “Division by Zero” << endl;
      exit(1);
   }
   // always keep sign in numerator
   if (d < 0L) { n = -n; d = -d; }

   if (n == 0L) {
      num = 0L;  den = 1L;
   } else {
      // always keep normalized form
      long g = gcd(n, d);
      num = n/g; den = d/g;
   }
}
```

- Start with `operator+=()`
  - To keep operators consistent
  - More efficient this way
- Take `const` reference + return `const` reference!
- Avoid overflow!

```
const Rational& Rational::operator+=(const Rational& rhs) {
   long g1 = gcd(den, rhs.den);
   if (g1 == 1L) {           // 61% probability!
      num = num*rhs.den + den*rhs.num;
      den = den*rhs.den;
   } else {
      long t = num * (rhs.den/g1) + (den/g1)*rhs.num;
      long g2 = gcd(t, g1);
      num = t/g2;
      den = (den/g1) * (rhs.den/g2);
   }
   return *this;
}
```


- Binary short–cut operators for mixed mode arithmetic
  - `g1 == 1` and `rhs.den == 1` always: simplifies considerably!

```
const Rational& Rational::operator+=(long rhs) {
   num = num + den*rhs;
   return *this;
}
```

- Binary addition operator: define out of `operator+=()`
  - global to get conversion on both arguments
  - Take `const` references
  - return `const` value!

```
const Rational operator+(const Rational& l, const Rational& r) {
   return Rational(l) += r;
}
```

```
const Rational& Rational::operator-=(const Rational& rhs) {
   long g1 = gcd(den, rhs.den);
   if (g1 == 1L) {            // 61% probability!
      num = num*rhs.den - den*rhs.num;
      den = den*rhs.den;
   } else {
      long t = num * (rhs.den/g1) - (den/g1)*rhs.num;
      long g2 = gcd(t, g1);
      num = t/g2;
      den = (den/g1) * (rhs.den/g2);
   }
   return *this;
}
const Rational& Rational::operator-=(long rhs) {
   num = num - den*rhs;  return *this;
}
const Rational operator-(const Rational& l, const Rational& r) {
   return Rational(l) -= r;
}
```


- Apply the same guidelines to multiplication operators

```
const Rational& Rational::operator*=(const Rational& rhs) {
   long g1 = gcd(num, rhs.den);
   long g2 = gcd(den, rhs.num);
   num = (num/g1) * (rhs.num/g2);
   den = (den/g2) * (rhs.den/g1);
   return *this;
}
const Rational& Rational::operator*=(long rhs) {
   long g = gcd(den, rhs);
   num *= rhs/g;
   den /= g;
   return *this;
}
const Rational operator*(const Rational& l, const Rational& r) {
   return Rational(l) *= r;
}
```

- ❑ Apply the same guidelines to division operators
  - ➠ but do not forget special case of division by zero!

```
const Rational& Rational::operator/=(const Rational& rhs) {
   if (rhs == 0) {
      cerr << “Division by Zero” << endl;
      exit(1);
   }
   long g1 = gcd(num, rhs.num);
   long g2 = gcd(den, rhs.den);
   num = (num/g1) * (rhs.den/g2);
   den = (den/g2) * (rhs.num/g1);
   if (den < 0L) { num = -num; den = -den; }
   return *this;
}
```



```
const Rational& Rational::operator/=(long rhs) {
   if (rhs == 0L) {
      cerr << “Division by Zero” << endl;
      exit(1);
   }
   long g = gcd(num, rhs);
   num /= g;
   den *= rhs/g;
   if (den < 0L) { num = -num; den = -den; }
   return *this;
}

const Rational operator/(const Rational& l, const Rational& r) {
   return Rational(l) /= r;
}
```

- Prefix operators: define out of binary shortcut operator + return `const` reference

```
const Rational& Rational::operator++() {
   return (*this += 1);
}

const Rational& Rational::operator--() {
   return (*this -= 1);
}
```

- Postfix operators: define out of prefix operators + return `const` value

```
const Rational Rational::operator++(int) {
   Rational oldVal = *this;
   ++(*this);
   return oldVal;
}

const Rational Rational::operator--(int) {
   Rational oldVal = *this;
   --(*this);
   return oldVal;
}
```


- Boolean operators
  - ➠ global to get conversion on both arguments
  - ➠ Take `const` references
  - ➠ return type `bool`

```
bool operator==(const Rational& lhs, const Rational& rhs) {
   return (lhs.numerator() == rhs.numerator() &&
           lhs.denominator() == rhs.denominator());
}

bool operator!=(const Rational& lhs, const Rational& rhs) {
   return (lhs.numerator() != rhs.numerator() ||
           lhs.denominator() != rhs.denominator());
}
```

- In a sense, this is a cheat `:-)`
  but simple and efficient implementation

```
bool operator<(const Rational& lhs, const Rational& rhs) {
   return (toDouble(lhs) < toDouble(rhs));
}
bool operator>(const Rational& lhs, const Rational& rhs) {
   return (toDouble(lhs) > toDouble(rhs));
}
```

- Define `<=` and `>=` out of `<` and `==` or `>` and `==` repectively

```
bool operator<=(const Rational& lhs, const Rational& rhs) {
   return ((lhs < rhs) || (lhs == rhs));
}
bool operator>=(const Rational& lhs, const Rational& rhs) {
   return ((lhs > rhs) || (lhs == rhs));
}
```


- Example of related global function: take `const` reference + return value!
- Could overload `abs`, but named `rabs` in consistency with `abs`, `labs`, and `fabs`

```
Rational rabs(const Rational& r) {
   if (r.numerator() < 0) return -r; else return r;
}
```

- Output operator: take `const` reference + take/return reference to `ostream`

```
ostream& operator<< (ostream& s, const Rational& r) {
   if (r.denominator() == 1L)
      s << r.numerator();
   else {
      s << r.numerator();
      s << "/";
      s << r.denominator();
   }
   return s;
}
```

- Explicit conversion function `double` → `Rational`
- Uses method of continued fractions:
  repeatedly replace fractional part of number to convert x with $\frac{1}{1/x}$
- Example:

$$0.12765 = \frac{1}{7.8333686} = \frac{1}{7+\frac{1}{1.199949}} = \frac{1}{7+\frac{1}{1+\frac{1}{5.00126}}} = \frac{1}{7+\frac{1}{1+\frac{1}{5+\frac{1}{787}}}}$$

1/787 is very small, so approximate it with 0, then simplify:

$$\frac{1}{7+\frac{1}{1+\frac{1}{5+0}}} = \frac{1}{7+\frac{1}{1+\frac{1}{5}}} = \frac{1}{7+\frac{1}{\left(\frac{6}{5}\right)}} = \frac{1}{7+\frac{5}{6}} = \frac{1}{\left(\frac{47}{6}\right)} = \frac{6}{47}$$

- Only Problem left: how to implement the termination rule?

- Recursive implementation of method of continued fractions

```
static Rational toRational(double x,
                           double limit,
                           int iterations) {
   double intpart;
   double fractpart = modf(x, &intpart);
   double d = 1.0 / fractpart;
   long left = long(intpart);

   if ( d > limit || iterations == 0 ) {
      // approximation good enough, stop recursion
      return Rational(left);
   } else {
      // remember left part and add inverted approximation
      // of fractional part
      return  Rational(left) +
              toRational(d, limit * 0.1, --iterations).invert();
   }
}
```

- Wrap internal recursive function for general usage:

```
Rational toRational(double x, int iterations) {
   if ( x == 0.0 ||
        x < numeric_limits<long>::min() ||
        x > numeric_limits<long>::max() ) {
      // x==0 or x too small or too large to represent in a long
      return Rational(0,1);
   } else {
      // setup recursive call
      // take care of negative numbers!
      int sign = x < 0.0 ? -1 : 1;
      return sign * toRational(sign * x, 1.0e12, iterations);
   }
}
```


- Harmonic Number defined as $H_n = 1/1 + 1/2 + 1/3 + ... + 1/n$

```
Rational harmonic(int n) {
   Rational r = 1;
   for (int i=2; i<=n; ++i) r += Rational(1,i);
   return r;
}
```

- Approximate Euler’s constant γ=0.5772156 out of $H_n = log\ n + \gamma + (1/2n) - (1/12n^2) + O(1/4n^4)$

```
int main() {
   cout << “n\tEuler Approx.\tHarmonic(n)” << endl;
   cout << “========================================” << endl;
   for (int n=1; n<25; ++n) {
      Rational r = harmonic(n);
      double g = toDouble(r) - log(n);
      g -= (1.0/(2*n)) * (1.0 - (1.0/(6*n)));
      cout << n << ‘\t’ << g << ‘\t’ << r << endl;
   }
}
```

```
n       Euler Approx.   Harmonic(n)
========================================
1       0.58333333      1
2       0.57768615      3/2
3       0.57731364      11/6
4       0.57724731      25/12
5       0.57722875      137/60
6       0.57722201      49/20
7       0.5772191       363/140
8       0.57721768      761/280
9       0.57721693      7129/2520
10      0.57721649      7381/2520
11      0.57721623      83711/27720
12      0.57721607      86021/27720
13      0.57721596      1145993/360360
14      0.57721588      1171733/360360
15      0.57721583      1195757/360360
...
23      0.57721569      444316699/118982864
24      0.57721569      1347822955/356948592
```


- Bernoulli Numbers defined as $B_n = \left(-\sum_{k=0}^{n-1} \binom{n+1}{k} \cdot B_k\right) / (n+1)$ and $B_0 = 1$
- great importance for the construction of asymptotic series

```
Rational bernoulli(int n) {
   if (n < 0) { cerr << “index out of range” << endl; exit(1); }
   else if (n == 0) return 1;
   else if (n == 1) return Rational(-1,2);
   if (n % 2) return 0;
   Rational r = 0;
   for (int k=0; k<n; ++k) {
      r -= binomial(n+1, k) * bernoulli(k);
   }
   r /= n+1;
   return r;
}
```

➠ need routine for *binomial coefficients*

- Use equation $\binom{n}{k} = [(n-k+1)/k] \cdot \binom{n}{k-1}$ and use binomials in the form of rationals

```
Rational binomial(int n, int k) {
   if (n < 0) { cerr << “1st out of range” << endl; exit(1); }
   if (k < 0 || k > n) {
       cerr << “2nd out of range” << endl; exit(1); }
   if (k > n-k) k = n-k;
   if (k == 0) return 1;
   return Rational(n-k+1, k) * binomial(n, k-1);
}
```

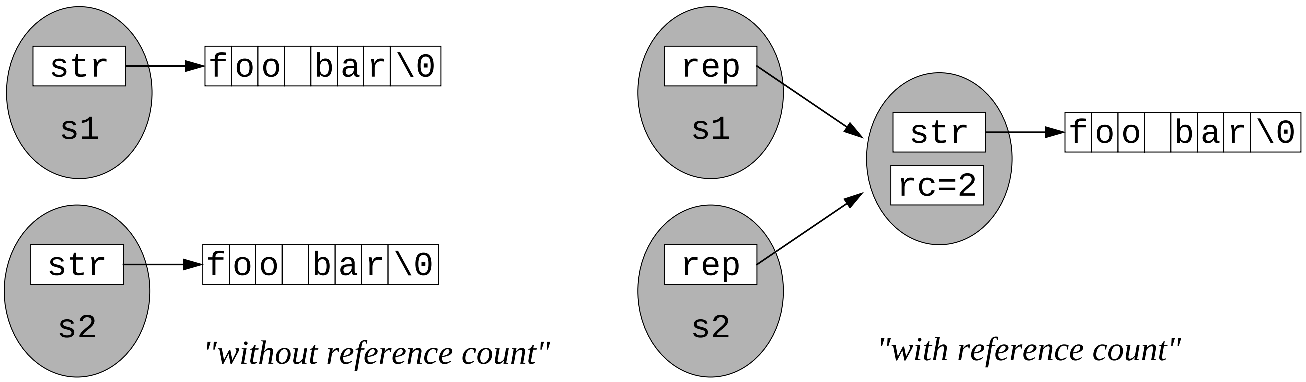
- Tabulate Bernoulli numbers from `0` to `23` (for all but `n=1` the odd numbers are zero)

```
int main() {
   cout << endl << “n\tBernoulli(n)” << endl;
   cout << “========================================” << endl;
   Rational b;
   for (int n=0; n<23; ++n) {
      if ((b=bernoulli(n)) != 0) cout << n << ‘\t’ << b << endl;
   }
}
```



```
n       Bernoulli(n)
========================================
0       1
1       -1/2
2       1/6
4       -1/30
6       1/42
8       -1/30
10      5/66
12      -691/2730
14      7/6
16      -3617/510
18      43867/798
20      -174611/330
22      854513/138
```

- *Reference Counting* `:=` instead of *deep copy* (for assignment or copy initialization) do *shallow copy* but count how many objects share the data
  - ➠ use if
    - ❍ objects are often copied (through assignment or parameter passing!) and
    - ❍ copying is expensive because objects are large / complex
- Example: `String s1 = "foo bar", s2 = s1;`



- Reference Counting is special case of *handle/body class idiom*
- *Handle class* `:=` user visible class
- *Body class* `:=` Helper class for data representation
  - ➠ handle class is `friend` of body class
  - ➠ all members of body class are `private`
  - ➠ contains (typically) only ctor/dtor and necessary data members

```
class StringRep {
friend class String;
private:
  StringRep(const char *s = 0) { set_str(s); }    // constructor
  ~StringRep(void) { delete [] str; }              // destructor
  void set_str(const char *);       // auxiliary help function
  char *str;                        // pointer to string value
  int rc;                           // reference counter
};
```

- Handle class `String`
  - implements extra intelligence to do the reference counting
  - forwards / uses the body class `StringRep`
- Private data now pointer to body class `StringRep`

```
class String {
private:
   StringRep *rep;
```

- Default constructor allocates `StringRep` object and sets reference count to `1`

```
public:
   String(const char *s = 0) {
      rep = new StringRep(s); rep->rc = 1;
   }
```

- Copy constructor just copies `StringRep` object and increments reference count

```
   String(const String& rhs) { rep = rhs.rep; rep->rc++; }
```


- Destructor deletes `StringRep` object if it is no longer referenced

```
   ~String(void) { if (--rep->rc <= 0) delete rep; }
```

- Access functions "forward" operation to `StringRep` object

```
   const char *c_str(void) { return (const char *) rep->str; }

   int length(void) const { return strlen(rep->str); }
```

- Other member functions ...

```
   // assignment operators
   String& operator=(const char *rhs);
   String& operator=(const String& rhs);

   // equality test operator
   bool operator==(const String& rhs);

   // concatenation operator
   const String operator+(const String& rhs);
};
```

- `String` assignment operator
  - deletes old `StringRep` object if no longer referenced
  - just copies `StringRep` object and increments reference count
  - in addition to the usual self-test and returning a reference for daisy-chaining

```
String& String::operator=(const String& rhs) {
   if (rep == rhs.rep)               // includes (this == &rhs)!
      return *this;
   if (--rep->rc <= 0) delete rep;
   rep = rhs.rep;
   rhs.rep->rc++;
   return *this;
}
```

- `(char *)` to `String` Assignment

```
String& String::operator=(const char *rhs) {
   if (--rep->rc <= 0) delete rep;
   rep = new StringRep(rhs); rep->rc = 1;
   return *this;
}
```


- Equality test operator
  - compares pointers first for speed
  - "forwards" operation to `StringRep` object

```
bool String::operator==(const String& rhs) {
   if (rep == rhs.rep)       // time saver
      return true;
   return !strcmp(rep->str, rhs.rep->str);
}
```

- (Inefficient) `String` concatenation operator
  - ➠ new dynamically generated character string is copied again in `StringRep` constructor
  - ➠ could be avoided by providing additional `StringRep` constructor that either
    - ✰ takes two character strings as parameters *or*
    - ✰ doesn't copy its argument

```
const String String::operator+(const String& rhs) {
   // construct new value
   char *buf = new char [length() + rhs.length() +1];
   assert(buf != 0);
   strcpy(buf, rep->str);
   strcpy(buf + length(), rhs.rep->str);

   // construct result String and StringRep objects
   String result(buf);
   delete [] buf;
   return result;
}
```

Programming in C++

☆☆☆ Advanced I/O ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Advanced I/O — Basics

- C++ introduces a new concept for handling I/O: *streams*
- include the C++ header file `<iostream.h>` instead of `<stdio.h>`
- each stream has some *source* or *sink* which can be
  - standard input
  - standard output
  - a file
  - an array of characters
- Advantages of C++ stream I/O
  - type safety
  - runtime efficiency
  - extensibility

- Streams used for output are objects of this class
- The insertion `operator<<()` writes data to an `ostream`
- `put()` member function writes one `char` to an `ostream`

  `ostream& put(char);`
- `write()` member function writes `char`s to an `ostream`

  `ostream& write(const char *buf, int nchars);`
- Predefined `ostream`
  - `cout` connected to standard output
  - `cerr` and `clog` connected to standard error
- **Hint:** use parenthesis to guarantee order of action when printing expressions

```
cout << a + b;              // OK: + has higher precedence
cout << (a + b);            //      than <<

cout << (a & b);            // << has higher precedence than &
cout << a & b;              // probably error: (cout << a) & b
```


- Output to `ostream` objects is buffered by default (like C stdio).
- Generally, buffers flush only when:
  - Full
  - Program terminates
  - `flush()` function
  - A `flush` manipulator is inserted into the stream (explained in a second)
- Predefined `ostream`
  - `cout` and `clog` is buffered
  - `cerr` is unit-buffered (i.e., its buffer is flushed after every insertion operation)
  - `cout` is also flushed whenever `cin` or `cerr` streams are accessed

- Recall: `operator<<()` can be overloaded to allow writing of user-defined types
- takes `ostream&` as its first argument
- returns the same `ostream` to allow daisy-chaining

```
cout << somevalue << anothervalue;
```

- Typical definition

```
ostream& operator<<(ostream& ostr, const T& val) {
   // write components of T
   return ostr;
}
```

- Example: `Complex::operator<<()`

```
ostream& operator<<(ostream& ostr, const Complex& rhs) {
   return ostr  << "(" << rhs.real() <<
                   "," << rhs.imag() << "i)";
}
```


- Streams used for input are objects of this class
- The extraction `operator>>()` reads data from an `istream`
- `get()` member function reads one `char` from an `istream`

```
istream& get(char&);
```

- `getline()` and `read()` member functions read `char`s from an `istream`

```
istream& getline(char *buf, int nchars, char delim='\n');
istream& read(char *buf, int nchars);
```

- Predefined `cin` connected to standard input
- In general, leading whitespace characters (spaces, newlines, tabs, form-feeds, `...`) are ignored
- `istream` can be tested whether extraction was successful

```
if ( cin >> somevalue ) { // reading somevalue OK ... }
```

- Recall: `operator>>()` can be overloaded to allow reading of user-defined types
- takes `istream&` as its first argument
- converts characters from `istream`, stores them in second argument
- returns the same `istream` to allow daisy-chaining

```
cin >> somevalue >> anothervalue;
```

- Typical definition

```
istream& operator>>(istream& istr, T& val) {
   // read components of T
   // check validity !!!
   return istr;
}
```

- **Checking validity of istream and input characters is non-trivial!**
  - ➠ "robust" `operator>>()` for compound types hard to write

Some useful `iostream` functionality for implementing `operator>>()`

- Reading arbitrary text: `string` extractor (skips leading whitespace, reads until next whitespace)

```
string buffer; cin >> buffer;
```

- If `string` not available, use the `(char *)` extractor

```
char buffer[80]; cin >> setw(80) >> buffer;
```

- Reading any single character (e.g., if skipping leading whitespace is a problem)

```
char c1;   cin.get(c1);     // leaves c1 unchanged if input fails
int c2;    cin.get(c2);     // sets c2 to EOF if input fails
           c2 = cin.get(); // same for new C++ standard
```

- Peeking at input (look at next character without extracting it)

```
if (cin.peek() != ch)  ...
```

- Pushing back a character into the stream

```
cin.putback(ch);
```

- Ignore the next `count` characters or until delimiter `delim` is read (whatever comes first)

```
cin.ignore(count, delim);
```

- Conventions for implementing `operator>>()`
  - After calling `operator>>()`, `istream` should "point" to the first illegal character
  - In case of format errors, set `ios::failbit` and do not change output parameter

- Example: `Rational::operator>>()`
  accepted formats: # and #/# where # is integer number

```
istream& operator>> (istream& istr, Rational& r) {
   long n = 0, d = 1;

   istr >> n;

   if ( istr.peek() == '/' ) {
      istr.ignore(1);
      istr >> d;
   }

   if ( istr ) r = Rational(n,d);

   return istr;
}
```


- Example: `Complex::operator>>()`
  accepted formats: `#`, `(#)`, `(#,#)`, and `(#,#i)` where # is floating-point number

```
istream& operator>>(istream& istr, Complex& rhs) {
   double re = 0.0, im = 0.0;
   char c = 0;

   istr >> c;
   if (c == '(') {
      istr >> re >> c;
      if (c == ',') istr >> im >> c;
      if (c == 'i') istr >> c;
      if (c != ')') istr.clear(ios::failbit);        // set state
   } else {
      istr.putback(c);
      istr >> re;
   }
   if (istr) rhs = Complex(re, im);
   return istr;
}
```

➠ still not complete: doesn’t handle `#i`, `(#i)`, `...`

- Use "`#include <fstream.h>`"
- Instantiate object of correct stream class

`ifstream`	Read-only files
`ofstream`	Write-only files
`fstream`	Read/Write files

- Associate stream object with external file name by using `open()` function or initializer list
- Typical usage

```
ifstream foofile("foo");     // open existing file "foo" readonly
ofstream foofile("foo");     // create new file "foo" for writing
                             // overwrite if already existing
if (!foofile) cerr << "unable to open file 'foo'" << endl;
```

- use normal stream operations to read from and write to file

```
foofile << somevalue << anothervalue << endl;
```


- Set *file mode* by using optional second argument
  - `ios::in` open for reading
  - `ios::out` open for writing
  - `ios::ate` start position at end of file
  - `ios::app` append mode: all additions at end of file
  - `ios::trunc` delete old file contents at open
  - `ios::binary` open file in binary mode

  Additional *file modes* (**non-standard!**):
  - `ios::nocreate` do not create file (must exist)
  - `ios::noreplace` do not replace old contents (file must not exist)
- The mode is constructed by or-ing the predefined values

```
ofstream foofile("foo", ios::out|ios::app); // append to foo
```

- Comparison `open` mode flags with `stdio` equivalents (ignoring `ios::ate`)

ios_base Flag combination					stdio equivalent
binary	in	out	trunc	app	
	+				"r"
		+			"w"
		+	+		"w"
		+		+	"a"
	+	+			"r+"
	+	+	+		"w+"
+	+				"rb"
+		+			"wb"
+		+	+		"wb"
+		+		+	"ab"
+	+	+			"r+b"
+	+	+	+		"w+b"

➠ **Only the combination of flags shown in the table are allowed!**

- Declaring `istream` without connecting to a filename
  and opening the file later with `open()` (also has optional 2nd argument for *file mode*)

```
ofstream outfile;
outfile.open("foo");
```

- Closing files
  - `fstream` destructor closes file automatically
  - manually by using `close()`

```
ifstream infile;
for (char** f=&argv[1]; *f; ++f) {
   infile.open(*f, ios:in);
   ...
   infile.close();
}
```

- Complicated way to calculate harmonic number H_{100}

```
#include <fstream.h>
#include <stdlib.h>

int main(int, char**) {
   ofstream out("iotest.out");
   if (!out) {
      cerr << "unable to open output file 'iotest.out'" << endl;
      exit(1);
   }
   for (int n=1; n<=100; ++n) out << (1.0/n) << endl;

   double d, sum = 0.0;
   ifstream in("iotest.out");
   if (!in) {
      cerr << "unable to open input file 'iotest.out'" << endl;
      exit(1);
   }
   while (in >> d) sum += d;
  cout << "sum: " << sum << endl;
}
```


- Format flags describe the stream's current format state

Flag	Flag Group	Description
`skipws`		skip leading whitespace on input
`left / right` `internal`	`adjustfield`	left / right justification padding after sign or base
`dec / oct / hex`	`basefield`	decimal / octal / hexadecimal conversion

`fixed`
`scientific`
`0`
	`floatfield`	fixed point notation
		exponential notation
		general format (default)

`showbase`
`showpos`
`showpoint`
`uppercase`
`unitbuf`
`stdio`
`boolalpha`
		show base indicator on output (`int`)
		show + sign if positive
		always show decimal point (`float`)
		uppercase hex/exponent output
		flush all streams after insertion
		synchronize with C stdio (**non-std!**)
		insert/extract booleans as text (**new!**)

- Stream class member functions
  - to get format flags

    `long flags()`
  - to set format flags (flags combined by using "`|`" operator): `flags = f`

    `long flags(long f)`
  - to set or unset additional flags: `flags |= f` or `flags &= ~f`

    ```
    long setf(long f)                     long unsetf(long f)
    long setf(long f, long field)
    ```
  - to set minimum field width (**Only for next value!**)

    `int width(int)`
  - to set number of significant digits

    `int precision(int)`
  - to fill character (default: space)

    `char fill(char)`
- or alternatively: Stream class *manipulators* (inserted in stream instead of values)


```
inline void ifmt(ostream& s,int w,long b)
                          {s.width(w); s.setf(b,ios::basefield);}
inline void ffmt(ostream& s,long f) {s.setf(f,ios::floatfield);}

int i = 12345;
double d = 3.1415;

cout.fill('.');
cout.setf(ios::showbase);
ifmt(cout,10,ios::dec); cout << i;                     // "%#10d"
ifmt(cout,10,ios::oct); cout << i;                     // "%#10o"
ifmt(cout,10,ios::hex); cout << i;                     // "%#10x"
cout << endl;
cout.precision(3);
ffmt(cout,0);                   cout << d << "  ";      // "%.3g"
ffmt(cout,ios::scientific);     cout << d << "  ";      // "%.3e"
ffmt(cout,ios::fixed);          cout << d << endl;      // "%.3f"
```

prints:

```
.....12345....030071....0x3039

3.14  3.142e+00  3.142
```

- Predefined manipulators:

```
istream istr;  ostream ostr;
```

Predefined Manipulator	Description
`ostr << dec, istr >> dec`	makes the integer conversion base 10
`ostr << oct, istr >> oct`	makes the integer conversion base 8
`ostr << hex, istr >> hex`	makes the integer conversion base 16
`ostr << endl`	inserts a newline character (`'\n'`) and calls `ostream::flush()`
`ostr << ends`	inserts a null character (`'\0'`). useful when dealing with `strstream`
`ostr << flush`	calls `ostream::flush()`
`istr >> ws`	extracts whitespace characters (skips whitespace) until a non-white character is found

- New C++ standard now also has: `[no]boolalpha, [no]showbase, [no]skipws, [no]showpoint, [no]showpos, [no]uppercase, scientific, fixed, left, right, internal`

- Additional manipulators (those with arguments) defined in "`iomanip.h`"

```
long f;  int n;  char c;
istream istr;  ostream ostr;
```

Predefined Manipulator	Description
`ostr << setbase(n)` `istr >> setbase(n)`	set the integer conversion base to `n`
`ostr << setw(n), istr >> setw(n)`	sets the minimal field width to `n` **Only for next value!**
`ostr << resetiosflags(f)` `istr >> resetiosflags(f)`	clears the flags bitvector according to the bits set in `f`
`ostr << setioflags(f),` `istr >> setioflags(f)`	sets the flags bitvector according to the bits set in `f`
`ostr << setfill(c),` `istr >> setfill(c)`	sets the fill character to `c` (for padding a field)
`ostr << setprecision(n),` `istr >> setprecision(n)`	sets the floating-point precision to `n` digits

- A plain manipulator is a function that
  - takes a reference to a stream
  - operates on it in some way
  - returns its argument

- Example: a tab manipulator

```
ostream& tab(ostream& ostr) {
   return ostr << '\t';
}
...
cout << x << tab << y << endl;
```

This is just a simple example; for tabs better use

```
const char tab = '\t';
...
cout << x << tab << y << endl;
```


- A more complex example: switch floating-point format

```
#include <iostream.h>
#include <iomanip.h>

ostream& scientific(ostream& ostr) {
   ostr.setf(ios::scientific, ios::floatfield); return ostr;
}

ostream& fixed(ostream& ostr) {
   ostr.setf(ios::fixed,ios::floatfield); return ostr;
}

ostream& general(ostream& ostr) {
   ostr.setf(0,ios::floatfield); return ostr;
}

...

int main() {
   double pi = 3.1415;
   cout << scientific << pi << endl;
}
```

```
#include <iostream.h>
#include <iomanip.h>

// define manipulators scientific/fixed/general/showbase here

int main(int, char**) {
   int i = 12345;
   double d = 3.1415;

   cout << setfill('_') << showbase;
   cout << setw(10) << dec << i;                          // "%#10d"
   cout << setw(10) << oct << i;                          // "%#10o"
   cout << setw(10) << hex << i << endl;                  // "%#10x"
   cout << setprecision(3);
   cout << general     << d << "  ";                       // "%.3g"
   cout << scientific << d << "  ";                       // "%.3e"
   cout << fixed       << d << endl;                       // "%.3f"
}
```

prints:

```
_____12345____030071____0x3039

3.14  3.142e+00  3.142
```


- Usage

```
outfile.write((char *) &x, sizeof(x));
infile.read ((char *) &x, sizeof(x));
```

- Be careful if type of `x` is a class
  - with pointer fields
  - virtual member functions
  - which requires non-trivial constructor actions
  - ➠ restrict usage to input/output of built-in types

- Example: binary read and write of `double`:

```
double d;
outfile.write((char *) &d, sizeof(double));     // or: sizeof(d)
 infile.read ((char *) &d, sizeof(double));
```

- Source or sink are strings (character arrays)
- C++ equivalent to `sscanf` and `sprintf`
- Use `#include <strstream.h>`
- Has classes `ostrstream`, `istrstream`, and `strstream`
- `strstream` member function `str()` returns get constructed string (but: freezes stream!)
- Buffer of frozen streams are not free'd (or changed) ➠ unfreeze stream with freeze(0)
- Example

```
#include <strstream.h>

ostrstream os;
int i = 15, j, k;

os << i << " is a number, in hex: 0x" << hex << i << ends;
char *str = os.str();
os.freeze(0);           // some compilers require os.rdbuf()->freeze(0);

istrstream is("15 18 798");
is >> i >> j >> k;
```


- Problem: `operator<<()` for compound types does not obey `width` stream attribute

  ➠ use `ostrstream` for temporary buffering!

- Example: `Complex::operator<<()`

```
#include <strstream.h>

ostream& operator<<(ostream& ostr, const Complex& rhs) {
   ostrstream buf;
   buf.flags(ostr.flags());           // copy stream flags
   buf.precision(ostr.precision());   // copy precision

   buf << "(" << rhs.real() << "," << rhs.imag() << "i)" << ends;
   ostr << buf.str();
   buf.freeze(0);       // unfreeze buffer or delete [] buf.str();

   return ostr;
}
```

Advanced I/O — Overloading `operator<<()`

- Another example: `Rational::operator<<()`

```
#include <strstream.h>

ostream& operator<< (ostream& ostr, const Rational& r) {
   if (r.denominator() == 1L)
      ostr << r.numerator();

   else {
      ostrstream buf;
      buf.flags(ostr.flags());          // copy stream flags
      buf.fill(ostr.fill());            // copy fill character

      buf << r.numerator() << "/" << r.denominator() << ends;

      ostr << buf.str();
      buf.freeze(0);                    // unfreeze buffer
   }
   return ostr;
}
```


Advanced I/O — Random File Access

The `get` Pointer

- Identifies the current extraction point from an input stream
- Advances automatically when stream data is extracted
- Can be explicitly re-positioned using the `seekg()` function:

  `istream& seekg(streampos nbytes, seek_dir base);`
- Current value is returned by the `tellg()` function: `streampos tellg();`

The `put` Pointer

- Identifies the current insertion point into an output stream
- Advances automatically when stream data is inserted
- Can be explicitly re-positioned using the `seekp()` function:

  `ostream& seekp(streampos nbytes, seek_dir base);`
- Current value is returned by the `tellp()` function: `streampos tellp();`

Base Position Specification

- `base` can be `ios::beg` (beginning of file), `ios::cur` (current position) , `ios::end` (EOF)

- Stream State is represented internally for each stream by the flags
  - `eofbit`: EOF on input
  - `failbit`: format errors (e.g., number begins with letter)
  - `badbit`: no space left on device or read from `ostream` or write on `istream`
- Use stream class methods to determine current stream state

stream state function	**Description**
`int good() const`	no state flag set?
`int eof() const`	`eofbit` set?
`int fail() const`	`failbit` or `badbit` set?
`int bad() const`	`badbit` set?
`int operator! () const`	like `fail()`
`operator void*() const`	return 0 if `failbit` or `badbit` set otherwise `this`

- Check stream state after any stream operation that might produce error (especially input)
- There are also methods for clearing or setting stream state (`rdstate`, `clear`, `setstate`)

Books on C++ iostreams

- Josuttis, *The C++ Standard Library – A Tutorial and Reference*, Addison-Wesley, 1999, ISBN 0-201-37926-0.
  - ➠ Most up-to-date and complete book on whole C++ standard library (including iostream, string, complex, ...)
  - ➠ Covers also more advanced topics like streambufs and internationalization
- Langer and Kreft, IOStreams and Locales: Advanced Programmer's Guide and Reference, Addison-Wesley, Januar 2000, ISBN 0-201-18395-1.
  - ➠ Everything you want and do not want to know about iostreams

General C++ Books (but cover iostreams quite well)

- Stroustrup, *The C++ Programming Language*, **Third Edition**
- Lippman and Lajoie, *C++ Primer*, **Third Edition**

Programming in C++

☆☆☆ Array Redesign ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Array Redesign — Motivation

The problems of C and C++ built-in arrays

- ❑ The size must be a constant
  - ❍ sometimes hard to decide on in advance
  - ❍ certainly not changed later
- ❑ The array does not carry around its own size
  - ➠ when passing arrays, size has to explicitly passed, too
- ❑ Cannot assign:

```
array1 = array2;            // error!
```

- ❑ No range checking:

```
cout << array1[-3];         // compiles! bad!
                            // often no runtime error message
```

➠ Want what arrays can do, plus more...

- Default constructor

```
const int def_farray_size = 7;        // outside class definition
FArray(int def_size = def_farray_size);   // in public: section
int len = 23; FArray fa1(len);            // note: len not const!
```

  - Allocate additional memory for the array safely
  - Zero out or initialize the array
- Copy constructor

```
FArray(const FArray&);       // called by
FArray fa2(fa1), fa3 = fa2;
```

- Prototype to initialize with "normal" array

```
FArray(const float *, int);  // called by
float normal_c_array[22] = { -4.3, 5.7, /*...*/, 84.23 };
FArray fa4(normal_c_array, 22);
```

- Destructor: primarily to deallocate free store memory

```
~FArray(void);
```


- Assignment operator

```
FArray& operator=(const FArray&);     // called by
fa1 = fa4;
```

- Array index operator (name: `operator[]()`)

```
float& operator[](int);   // called twice by
fa4[2] = fa3[5];         // fa4.operator[](2) = fa3.operator[](5)
```

  - Return value: a *reference* to a float to allow usage on lhs of assignments
- `const` Array index operator

```
const float& operator[](int) const;
```

  - to allow indexing constant `FArray`’s
  - Note: overloading based on return type is *not* allowed, but it is on constness of function
- Getting size of an `FArray`

```
int size(void) const;                 // called by
cout << "size of fa2: " << fa2.size() << endl;
```

- What should internal representation look like?

```
private:
   float *fa;    // pointer to memory holding values
   int sz;       // size of array
```

- `private:` so access only allowed to member functions
- What have we paid?
  - Eight extra bytes
- What have we gained?
  - extra four bytes: `FArray` can be assigned and reassigned
  - another four bytes: retain size information

➠ **What do we have so far?**


```
const int def_farray_size = 7;
class FArray {                // float array
public:
   // constructors and destructor
   FArray(int def_size = def_farray_size);
   FArray(const FArray&);
   FArray(const float *, int);
   ~FArray(void);

   // other member functions
   FArray& operator=(const FArray&);
   float& operator[](int);
   const float& operator[](int) const;
   int size(void) const;
private:
   float *fa;    // pointer to memory holding values
   int sz;       // size of array
};
```

- Motivation: all three constructors do similar things
- For assistance in defining other constructors (and assignment operators)
- Therefore, declared in `private:` section of `FArray` definition:

```
private:
   void init(const float *, int);
```

- Arguments
  - optionally take an `float` array for initialization
  - target array size
- Responsibilities
  - Free store allocate memory for new array
  - Check for success
  - Optionally initialize


```
#include <assert.h>

void FArray::init(const float *array, int size) {
   // initialize all class data members
   sz = size;
   fa = new float [size];
   assert(fa != 0);                    // quit if new() failed
   // initialize array values if specified
   for (int index=0; index<size; index++) {
      // did we receive an initialization array?
      fa[index] = (array != (float *)0) ? array[index] : 0.0;
   }
}
```

- `FArray::` scope
  - ➠ otherwise `init()` would be global

```
class FArray {
public:
    FArray(int def_size = def_farray_size) {
        init((const float *) 0, def_size);
    }
    FArray(const FArray& rhs) {
        init(rhs.fa, rhs.sz);
    }
    FArray(const float *array, int size) {
        init(array, size);
    }

// ...
};
```

- Make constructor definitions (implicitly) `inline` for speed
- Note: `init()` itself cannot be `inline` as it includes a loop

- Recall: constructors called just after allocation of memory for data members, i.e., `fa` and `sz`
- Symmetrically, destructors called just *before deallocation* of data members
  - ➠ no need to reset the data members (`fa` and `sz`)
- Aside from these bytes, what else needs to be done?
  - ➠ freeing up memory of free store allocated array

```
class FArray {
public:
    // constructors here
    ~FArray(void) { delete [] fa; }   // note [] !
// ...
}
```

- Note: also made `inline` for speed

FArray::operator=() Definition

```
FArray& FArray::operator=(const FArray& rhs) {
   if (this != &rhs) {        // self-test
      delete [] fa;           // free up current memory
      init(rhs.fa, rhs.sz);   // copy rhs to lhs
   }
   return *this;              // ref returned for daisy-chaining
}
```

- Recall: `this` contains the address of calling object, e.g.,

  `fa4 = fa1;   // in FArray::operator=(), this == &fa4`
- If self-test is true, return early (as with `String`)
  - safe time (nothing to do)
  - avoid catastrophic self-assignment

    ```
    FArray &fa5 = fa3, *pfa = &fa2;
    ```

    ```
    //...later
    fa3 = fa5;   // or viceversa
    *pfa = fa2;  // ditto
    ```


operator[]() and size() Definitions

```
float& operator[](int index) { return fa[index]; }
const float& operator[](int index) const { return fa[index]; }
int size (void) const { return sz; }
```

- Recall: `operator[]()` returns a reference to allow for, e.g.,

  `fa3[4]--;`

  `fa2[7] = fa5[2];`
- Inlining:
  - `operator=()`: probably not
  - `operator[]()` and `size()`: yes

```
#include <iostream.h>
#include "farray.h"

void remove_hot(FArray&);

int main(int, char**) {   // for compiler: I know that main has
                          // arguments, but I don't use them
   const int num_days = 6;
   float temperature[num_days] =
      { 23.1, 28.5, 21.3, 33.2, 35.5, 27.5 };// probably read in
   FArray  all_temp(temperature, num_days),
           not_too_hot = all_temp;

   remove_hot(not_too_hot);

   cout << "Temperature of nice days: ";
   for (int day=0; day<not_too_hot.size(); day++)
      cout << " " << not_too_hot[day];
   cout << endl;
}
```



```
void remove_hot(FArray& day_temp) {
   const float too_hot = 30.0;
   int num_nice = 0;
   int day;

   for (day=0; day<day_temp.size(); day++) {
      if (day_temp[day] < too_hot)
         num_nice++;
   }

   FArray dummy(num_nice);    // works for num_nice==0 too:
                              // creates empty array!
   num_nice = 0;
   for (day=0; day<day_temp.size(); day++) {
      if (day_temp[day] < too_hot)
         dummy[num_nice++] = day_temp[day];
   }

   day_temp = dummy;
}
```

- Use `FArray` for dynamic storage

```
class FStack {
public:
   FStack(int size = FStack_def_size) : vals(size), top(0) {}

   void push(float val) {
      assert(top <= vals.size());  vals[top++] = val;
   }

   float pop() { assert(top > 0);  return vals[--top]; }

   bool empty() { return (top == 0); }

private:
   static const int FStack_def_size = 7;
   FArray vals;
   int top;
};

const int FStack::FStack_def_size;
```

➠ definition of copy constructor, assignment operator, and destructor no longer necessary!

➠ `size` member no longer necessary!

Programming in C++

☆☆☆ Templates ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Class Templates — Motivation

- `FArray` fine for `float`, but what about `int`, `short`, `char`, `double`, etc. ?
- Solution: use class templates, as with function templates
  - ➠ class templates describe a set of related classes much like
    classes describe a set of related objects
- Class definitions are generated (*instantiated*) by the compiler on the fly when needed:

```
Array<int> ia6(16);          // generates code for array of int
Array<char> ca1;             // generates code for array of char
```

➠ doesn't save code compared to manual copying
but saves programming time and is less error-prone!

- Some people might prefer more "natural" type names:

```
typedef Array<float> FArray;
FArray fa6(16);
```

- Note: use only when the type of the objects *does not* affect the implementation!

➠ **What does it look like?**

```
const int def_array_size = 7;

template<typename T>          // historically:  <class T>

class Array {                 // name of class, ctors, dtor: Array
public:                       // otherwise: Array<T>
   Array(int def_size = def_array_size) {
      init((const T *) 0, def_size); }
   Array(const Array<T>& rhs) { init(rhs.a, rhs.sz); }
   Array(const T* ay, int size) { init(ay, size); }
   ~Array(void) { delete [] a; }

   Array<T>& operator=(const Array<T>&);
   T& operator[](int index) { return a[index]; }
   const T& operator[](int index) const { return a[index]; }
   int size(void) const { return sz; }

private:
   T *a;
   int sz;
   void init(const T*, int);
};
```


- Definition of member functions outside the template class definition also requires the `template<...>` specification

```
template<typename T> class Array {
public:
   Array(int def_size = def_array_size);
   //...
};

template<typename T>
Array<T>::Array(int def_size) {
   init((const T *) 0, def_size);
}
```

- **Problem**: Template member function definitions need to be known to the compiler
  - ➠ can**not** compiled away in separate `.cpp` file
- **Solution**:
  - put everything in header file
  - `#include` corresponding `.cpp` file at the end of header file (recommended)

- We can finally write down a general implementation of our stack example:

```
template<typename T> class Stack {
public:
    Stack(int size = def_size) : vals(size), top(0) {}

    void push(const T& val) {
        assert(top <= vals.size());
        vals[top++] = val;
    }

    T pop() {
        assert(top > 0);  return vals[--top];
    }

    bool empty() { return (top == 0); }

private:
    static const int def_size = 7;
    Array<T> vals;
    int top;
};

template<typename T> const int Stack<T>::def_size;
```


- Main usage of class templates: *container types*
  - `Array<T>`
  - `stack<T>`
  - `queue<T>`
  - `deque<T>`
  - `list<T>`
  - `map<K,T>`
  - `set<T>`
  - `...`

- But also useful for specifying *base type size* or *precision*
  - `Complex<T>` `T = float | double | long double`
  - `Rational<T>` `T = short | int | long | long long`
  - `String<T>` `T = char | wchar_t | unicode | ...`
  - `...`

- C++ provides no direct support for specifying constraints on template arguments
  - ➠ e.g., template argument to `Rational<T>` must be `integer`-like type

- Best solution so far:
  - ❍ Write well-documented, special constraint template classes of the following form:

```
template<class T> struct Is_integer_like {
   static void constraints(T a) { T b(0), c(1); b=a%c; a<b; ... }
   Is_integer_like() { void(*p)(T) = constraints; }
};
```

  - ❍ Constraints can be checked (inside template source code, e.g., constructor) by instantiating a temporary constraint template class object:

```
Is_integer_like<T>();
```

  - ➠ Constraints can use full expressiveness of C++
  - ➠ No code is generated for a constraint using current compilers
  - ➠ Current compilers give acceptable error messages, including the word “constraints” (to give the reader a clue), the name of the constraints, and the specific error that caused the failure (e.g. “expression must have integral or enum type”)

- Define a family of related functions
- Function template argument list is used to specify different functions in family
- Normally, each function template argument must be used as a type of some function argument in the function argument list
- Different functions in the family have different function signatures (argument list profiles)

```
template<typename TYPE>
inline void swap(TYPE& v1, TYPE& v2) {
   TYPE tmp;
   tmp = v1; v1 = v2; v2 = tmp;
}

double x, y;
int i, j;

swap(x, y);  // compiler generates double version automatically
swap(i, j);  // again for int
swap(x, i);  // error! good!
```

Function Templates versus...

...Macros

- ❑ Function templates are easier to read because they look just like regular function definitions
- ❑ Function template instantiation is less prone to context-related errors than macro expansion
- ❑ Diagnostics for errors in template functions are better than for errors in macros
- ❑ Function templates have scope (e.g., they can be part of a `namespace` or `class`)
- ❑ Pointers to function template specializations are possible
- ❑ Function templates can be overloaded or specialized
- ❑ Function templates can be recursive

...Overloaded Functions

- ❑ Use overloaded functions when behaviour of function differs depending on type of function argument(s)
- ❑ Use function template when behaviour of function *does not* depend on type of function argument(s)

Function Templates Example: Defining Math Operators

Operators `+`, `*`, and prefix and postfix `++` can be defined out of `+=` and `*=` ➠ use function templates

```
template<class T> const T operator*(const T& lhs, const T& rhs) {
   return T(lhs) *= rhs;
}
template<class T> const T operator+(const T& lhs, const T& rhs) {
   return T(lhs) += rhs;
}
template<class T> T& operator++(T& t) {
   t += T(1); return t;
}
template<class T> const T operator++(T& t, int) {
   T old(t); ++t; return old;
}
class Complex { /* only need operator+=, operator*=, and Complex(1) */ };
```

- ❑ Disadvantage 1: function templates are global and might cause problems with other classes!
- ❑ Disadvantage 2: automatic conversion on arguments no longer works!

➠ will be fixed in Chapter Inheritance!

- Consider function template for a function to compute the absolute value of a number:

```
template<typename T>
inline T abs(T x) {
   if (x < 0)
      return -x;
   else
      return x;
}
```

- ➠ works for any type `T` that supports copying, unary `-`, and binary `<` (e.g. `int` or `double`)
- ➠ doesn't work for complex numbers (not totally ordered, so typically no `operator<` defined!)

```
Complex c1(2.3, 7.8), c2 = abs(c1);  // error!
```

- ➠ use *template specialization*

```
template<> inline double abs(Complex z) {
   return sqrt(z.real()*z.real() + z.imag()*z.imag());
}
```


- Class templates may also be specialized:

```
template<typename T>
class Array {
public:
   // definition of an array of any type ...
};

template<>
class Array<bool> {
public:
   // definition of an array of bool's => bitvector
   // e.g., pack 8 bits per byte
};

Array<int> x(10);            // uses Array<T>
Array<bool> bvec(64);        // uses Array<bool>
```

- Specialized class
  - has specialized implementation
  - *can* have specialized or different interface

❑ Template function overloading

```
template<class T> void foo(T);        // #1
template<class T> void foo(T*);       // #2
template<class T> void foo(T**);      // #3

int i;
int *pi = &i;
int **ppi = &pi;

foo(i);                               // calls #1
foo(pi);                              // calls #2
foo(ppi);                             // calls #3
```

❑ Explicit qualification of template functions (if type cannot be deduced from supplied arguments)

```
template<class T> inline T min(T x, T y) { return(x<y ? x : y); }
int i; long l;
int a = min<int>(i, l);

template<class T, class U> T make(U u);
Thing t = make<Thing>(1.23);
```


❑ **Problem**: implicit type conversions are *not* inherited by template container classes!

```
template <typename T> class Array { /*...*/ }

Array<int> v1;
Array<short> v2;
v1 = v2;   // Error!
```

➠ use *member templates* to define generic assignment operator

```
template<typename T> class Array {   // old
public:
   Array<T>& operator=(const Array<T>&);
   // ...
};
```

⇓

```
template<typename T> class Array {   // new
public:
   template<typename T2>
   Array<T>& operator=(const Array<T2>&);
   // ...
};
```

- Implementation of member templates outside class definition requires the usual prefixes:

```
template<typename T>
template<typename T2>
Array<T>& Array<T>::operator=(const Array<T2>&) {
  // ...
}
```

- Of course, member templates can be used for other member functions as well
  - e.g., generic constructors
- Member templates can implement generic member functions inside
  - class templates
  - "ordinary" classes

- Template parameters are not restricted to one type ("`template<typename Type>`") there can also be
  - more than one parameter
  - integral *constants* (`int`, `bool`, `char`, ...)
  - pointer types and pointer to functions

```
// Two type parameters
template<typename type1, typename type2> ...

// One type parameter, one integral parameter
template<typename T, int N> ...

// Pointer to function parameter
template<double Tfunc(double)> ...

// Monstrosity
template<typename T, T Tfunc(T), int N, bool Flag> ...
```

- **Note**: template parameter may not be *floating-point* constants because the result could differ between platforms

❑ Template parameters can also have *defaults* (very much like function default parameters)

```
template<int N = 10, typename T = char>
class FixedArray {
   T data[N];
   // ...
};

FixedArray<100, int> a100_ints;      // like int a[100];
FixedArray<256>  b256_char;          // char b[256];
FixedArray<>     c10_char;           // not: FixedArray c10_char; !
```

❑ **Note**: onlys *class* templates can have default template parameters!
(not function or member templates)

❑ templates `:=` intelligent macros that support C++ naming, typing, and scope rules

❑ Template Design

- ❍ write concrete example class or function first and test it
- ❍ possibly: use "pseudo" parameters like

  ```
  typedef T <concrete-type-like-int>;
  ```

  to write concrete example class or function

  ➠ will be easier to convert to template later
- ❍ then re-write as template(s)

❑ Templates allow so-called *Generic Programming* in C++ (more later)

Programming in C++

☆☆☆ Inheritance ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Inheritance — Motivation

- What has `FArray` bought us?
  - The size of the `FArray` need not be constant ✔
  - The `FArray` knows its own size ✔
  - `FArray` assignment: `fa4 = fa6;` ✔
  - Range checking: `cout << fa2[-3];  // still bad!` ✘
- What are our options for `FArray` range checking?
  - Add this capability to `FArray::operator[]()`
    - ➠ Problem: this may be slow, not always desired, e.g.:

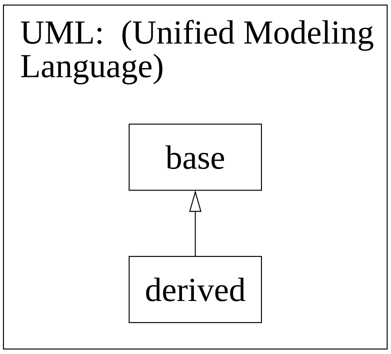
```
for(int i=0; i<fa3.size(); i++) fa3[i]=0.0;   // just fine
```

vs.

```
cin >> farray_index;                   // unsafe!
cout << fa1[farray_index];
```

  - Define a new class `safeFArray`
    - ➠ Problem: this will repeat lots of code, error prone
  - **Inheritance**: Create a subclass!

Inheritance — Basic Idea



- Goal: a new, *derived* class (*subclass*) whose objects are both
  - objects of the *base* class (*superclass*), and also
  - specific, specialized objects of the base class

- **single most important rule**: a derived object ***is-a*** base object, but not vice versa

- Inheritance is commonly used in two ways
  - *reuse* mechanism
    - ➠ a way to create a new class that strongly resembles an existing one (like `safeFArray`)
  - *abstraction* mechanism:
    - ➠ a tool to organize classes into hierarchies of specialization
    - ➠ describe *relationships* between user-defined types

Inheritance — The *is-a* Test

- Suppose our triangle class

```
class triangle { private: Coord v1, v2, v3; /*...*/ };
```

- Possible derived classes:
  - particular characteristic: `right_triangle`
  - additional field (`color`): `color_triangle`

➠ they pass the *is-a* test

- Not possible derived classes:
  - particular characteristic: `trianglar_pipe` (*is-a* `pipe`, but not a `triangle`)
  - additional field (`v4`): `rectangle`

➠ they fail the *is-a* test

- Note: `trianglar_pipe` will probably *include* `triangle` data member (*hasA*)

- Derived class *inherits* member functions and data members from base class
  - in addition to its own members
  - **exceptions:** constructors, destructors, `operator=()`, and friends are ***not*** inherited

- Access rules for derived class implementation:

members of base class which are	can be accessed in base class	can be accessed in derived class	compare to: client code
`public`	✔	✔	✔
`private`	✔	✘	✘
`protected`	✔	✔	✘

- General form of derived class definitions:

```
class DerivedClassName : AccessMethod BaseClassname {
   /* ... */
};
```

where *`AccessMethod`* is one of the following:

`public` inherited `public` base class members stay `public` in derived class

(`private` inherited `public` base class members become `private` in derived class)

➠ **does affect access rights of client code and classes which derive from derived classes**

➠ specifying AccessMethod `public` is important as the default is `private`!

- **Note**: the *is-a* test only applies to `public` inheritance!

Inheritance — Summary: Access Control

Public inheritance

```
class b {
private:   int prv;
protected: int prt;
public:    int pub;
   void f() {
   // access to pub, prt, prv
   }
};

class d: public b {
private:   int dprv;
public:    int dpub;
   void df() {
   // pub, prt, dpub, dprv, f
   }
};

void func() {   //b::pub, b::f
   //d::pub, d::dpub, d::f, d::df
}
```

Private inheritance

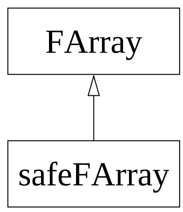
```
class b {
private:   int prv;
protected: int prt;
public:    int pub;
   void f() {
   // access to pub, prt, prv
   }
};

class d: private b {
private:   int dprv;
public:    int dpub;
   void df() {
   // pub, prt, dpub, dprv, f
   }
};

void func() {   //b::pub, b::f
   //d::pub, d::dpub, d::f, d::df
}
```


Inheritance — `safeFArray`

UML:



❑ how about our `safeFArray` class?

```
#include "farray.h"

class safeFArray : public FArray {
public:
   // new constructors needed (not inherited)

   // destructor and operator= not inherited but default OK

   // override definition of operator[]() and operator[]() const
};
```

❑ Placed in "`safefarray.h`"

- Constructors for the base class are always called first, destructors for base classes last. e.g.,

```
class base { public:
  base()  { cout << "constructing base" << endl; }
  ~base() { cout << "destructing base" << endl; }
};

class derived : public base {
public:
  derived(){ cout << "constructing derived" << endl; }
  ~derived(){ cout << "destructing derived" << endl; }
};

int main(int, char **) {
  derived d;
}
```

will print out

```
constructing base
constructing derived
destructing derived
destructing base
```


- Problem: if not otherwise specified, default constructor of base class is called
  - ➠ for `safeFArray`, we always would end up with `FArray(def_farray_size)`
- Solution: use the following syntax to specify the base constructor to call:
  (special case of general *member initialization list*)

```
class safeFArray : public FArray {

public:
  safeFArray(int size = def_FArray_size) : FArray(size) {}

  safeFArray(const FArray& rhs) : FArray(rhs) {}

  safeFArray(const float* fa, int size) : FArray(fa, size) {}

// ...
};
```

- Note: nothing else needs to be constructed
  - ➠ copy constructor can take `FArray`'s
- Note: only *direct* base class initialization possible

- we like and keep declaration/interface **and** must match return value as well, but
- definition needs reworking, e.g.,

```
#include <assert.h>
#include "safefarray.h"

float& safeFArray::operator[](int index) {
   assert(index >= 0 && index < size());      // exit on failure
   return FArray::operator[](index); // reuse base implementation
}

const float& safeFArray::operator[](int index) const {
   // same as above...
}
```

- both versions of `operator[]()`, const and non-const, must be overridden
- if possible, reuse base function definition
- Placed in "`safefarray.cpp`"
- `safeFArray::operator[]()` replaces `FArray::operator[]() !!!`
  (unlike constructors/destructors)

- Rule: derived objects, pointers to derived objects, and references to derived objects are automatically converted to corresponding base objects if necessary:

```
class base { /*...*/ };

class derived : public base {
   /*...*/
};

void base_func (base& b);
void devd_func (derived& d);

base b;
derived d;

base_func(b);
base_func(d);   // d is-a base
devd_func(d);
devd_func(b);   // error!
```

```
class car { /*...*/ };

class rv : public car {
   /*...*/
};

void drive (car& c);
void live_in (rv& r);

car c;
rv r;

drive(c);
drive(r);       // r is-a car
live_in(r);
live_in(c);     // error!
```

➠ C++ permissions follow *is-a* model.

- if a inherited member function is overridden by a derived class and called through a *pointer* or *reference*, the type of the *pointer/reference object itself* determines what function gets called:

```
class base {
  void foo(void);
};
class derived : public base {
  void foo(void);
};

derived d, *pd=&d, &rd2=d;
base b, *pb=&b, &rb=b, &rd1=d;

pb->foo();   // base::foo()
rb.foo();
pd->foo();   // derived::foo()
rd2.foo();

pb = pd;     // or: pb = &d;
pb->foo();   // base::foo()!!!
rd1.foo();
```

```
class car {
  void park(void);
};
class rv : public car {
  void park(void);
};

rv r, *pr=&r, &rr2=r;
car c, *pc=&c, &rc=c, &rr1=r;

pc->park();  // parks car
rc.park();
pr->park();  // parks rv
rr2.park();

pc = pr;     // rv is-a car
pc->park();  // parks rv like
rr1.park();  // a car (ouch!)
```

➠ *static binding*, i.e., type of member function determined at compile time

- if a **`virtual`** member function is overridden by a derived class and called through a *pointer* or *reference*, the type of the *pointed to/referenced object* determines what function gets called:

```
class base {
  virtual void foo(void);
};
class derived : public base {
  virtual void foo(void);
};

derived d, *pd=&d, &rd2=d;
base b, *pb=&b, &rb=b, &rd1=d;

pb->foo();   // base::foo()
rb.foo();
pd->foo();   // derived::foo()
rd2.foo();

pb = pd;     // or: pb = &d;
pb->foo();   // derived::foo()
rd1.foo();   // good!
```

```
class car {
  virtual void park(void);
};
class rv : public car {
  virtual void park(void);
};

rv r, *pr=&r, &rr2=r;
car c, *pc=&c, &rc=c, &rr1=r;

pc->park();  // parks car
rc.park();
pr->park();  // parks rv
rr2.park();

pc = pr;     // rv is-a car
pc->park();  // parks rv like
rr1.park();  // a rv (good!)
```

➠ *dynamic binding*, i.e., type of member function determined at run time

- ❑ Rule: Never redefine an inherited non-virtual function!
- ❑ Derived class by adding fields to subclass:
  - ➠ likely `operator=()` and `operator==()` (at least) will change
  - ➠ make them `virtual` in base class
- ❑ Destructor called through pointer when using `new` / `delete`
  - ➠ static / dynamic binding rules applies

```
base *pb; derived *pd = new derived;

// ... later
pb = pd; delete pb;  // only calls base::~base()!!!
```

  - ➠ make destructors `virtual` in base class (if base has >1 virtual member function; if not, shouldn't be base class in the first place)
- ❑ But: `virtual` functions cause *run-time* (indirect call) and *space* (virtual table) overhead
- ❑ If function needs to be `virtual`
  - ➠ another reason to use member vs. global function

➠ **Virtual functions supply subclasses with default definitions**

- ❑ What if there is no (useful) default definition?

```
class Shape {
public:
   virtual double area(void) const = 0;          // pure virtual
   // ...
};

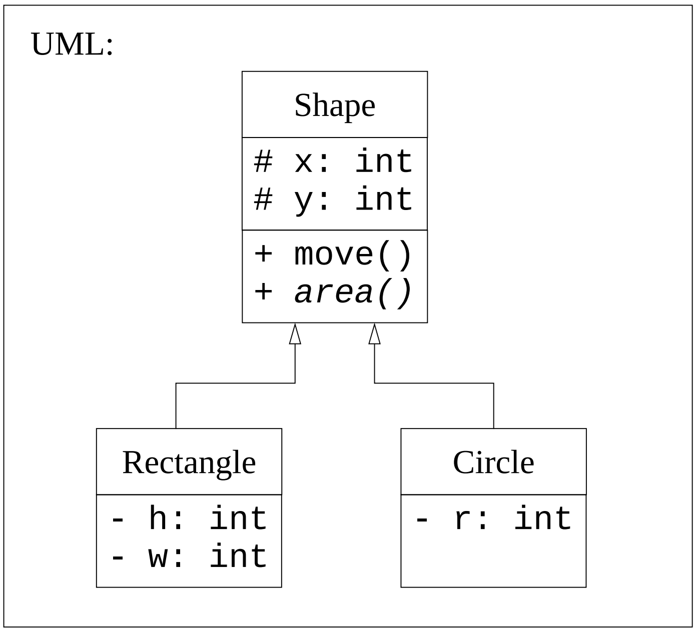
class Rectangle : public Shape { /* ... */ };

class Circle : public Shape { /* ... */ };
```

- ❑ effects of ≥ 1 pure virtual function(s)
  - ❍ `Shape` is an *abstract base class* (ABC)
  - ❍ cannot create objects of class `Shape`
  - ❍ but can build subclasses on top of `Shape`
  - ❍ subclasses (`Rectangle` and `Circle`) must
    - ✰ implement pure virtual functions (and become *concrete* subclasses), or
    - ✰ inherit them, and become an ABC themselves

Inheritance — Dynamic Binding Example

- Example:
  Define framework for geometrical shapes like rectangles and circles.
  Each shape has a center (x, y) and methods to move the object and to calculate its area.



\+ → public member
\# → protected member
\- → private member

abstract member

Inheritance — Dynamic Binding Example

- `shape.h:`

```
class Shape {
protected: int x_, y_;  // center
public:    Shape(int x, int y) : x_(x), y_(y) {}
           virtual double area() const = 0;
           void move(int dx, int dy) { x_ += dx; y_ += dy; }
};

class Rectangle : public Shape {
private:   int h_, w_;
public:    Rectangle(int x, int y, int h, int w)
                                : Shape(x, y), h_(h), w_(w) {}
           virtual double area() const { return h_*w_; }
};

class Circle : public Shape {
private:   int r_;
public:    Circle(int x, int y, int rad) : Shape(x, y), r_(rad) {}
           virtual double area() const { return r_*r_*3.1415926; }
};
```

- Main program:

```
#include <iostream.h>
#include "shape.h"

double area(Shape *s[], int n) {
   double sum = 0.0;
   for (int i=0; i<n; ++i) sum += s[i]->area();
   return sum;
}

int main(int, char**) {
   Shape *shapes[3];
   shapes[0] = new Rectangle(2,6,2,10);
   shapes[1] = new Circle(0,0,1);
   shapes[2] = new Rectangle(3,4,5,5);

   cout << "area: " << area(shapes, 3) << endl;
   for (int i=0; i<3; ++i) shapes[i]->move(10, -4);
}
```

- prints:

```
area: 48.1416
```



```
class Shape {
public:   enum kind { REC, CIR };
          Shape(int x, int y, int h, int w)
             : x_(x), y_(y), h_(h), w_(w), t_(REC), r_(0) {}
          Shape(int x, int y, int r)
             : x_(x), y_(y), h_(0), w_(0), t_(CIR), r_(r) {}
          void move(int dx, int dy) { x_ += dx; y_ += dy; }
          double area() {
             switch (t_) {
                case REC:  return h_*w_;
                case CIR:  return r_*r_*3.1415926;
                default:   return 0.0;
             }
          };
private:  int x_, y_, h_, w_, r_;
          kind t_;
};
```

- ➠ Adding new shape requires changes everywhere `kind` is used (don't forget one!)
- ➠ Recompilation of all source files necessary which depend on `Shape`!

Summary: for `public` inheritance (derived *is-a* base), the following rules apply:

- ❑ pure virtual function
  - ➠ function interface only is inherited
  - ➠ concrete subclasses ***must*** supply their own implementation
- ❑ simple (non-pure) virtual function
  - ➠ function interface and default implementation is inherited
  - ➠ concrete subclasses ***can*** supply their own implementation
- ❑ non-virtual function
  - ➠ function interface and mandatory implementation is inherited
  - ➠ concrete subclasses ***should not*** supply their own implementation

➠ **Function virtuality is an important C++ feature**

➠ **But: polymorphism is not the solution to every programming problem (KISS principle!)**

- ❑ Only member functions can be `virtual`
- ❑ Problem: how to make class related global functions (e.g., `operator<<()`) behave correctly for inheritance

➠ introduce `public` `virtual` helper method (e.g., `print()`)

➠ derived classes can redefine helper method if necessary

```
class foo {
public:
   virtual ostream& print(ostream& ostr) {
      // usual implementation of output operator here
      ...
   }
   ...
};

ostream& operator<<(ostream& ostr, const foo& f) {
   return f.print(ostr);
}
```

```
class base {
public:
  virtual void func() const { cout << "base::func" << endl; }
};
class drvd : public base {
public:
  virtual void func() const { cout << "drvd::func" << endl; }
};
void gen_func(base b) { b.func(); }
derived d; gen_func(d);              // prints: base::func !!!
```

- Problem: argument `b` of `gen_func` is passed by *value*
  - ➠ copy constructor is invoked
  - ➠ `b` is not a reference or pointer ➠ *static* binding ➠ copy constructor of `base` is invoked
  - ➠ copies only `base` part (*slicing*)

➠ **Another reason to pass class objects by** *reference:*

```
void gen_func(const base& b) { b.func(); }
```


- Recall rules:
  - ❍ Assign to all members
  - ❍ Check for assignment to self
  - ❍ Return a reference to `*this`
- New rule: derived class's assignment operator must also handle base class members!

```
class base {
private:
 int x;
public:
 base(int i) : x(i) {}
};
```

```
class derived : public base{
private:
 int *y;
public:
 derived(int i) : base(i), y(0) {}
 derived& operator=(const derived& rhs);
};
```

```
derived& operator=(const derived& rhs) {
        if (this == &rhs) return *this;
        base::operator=(rhs);      // don't forget this
        if (y) { *y = *(rhs.y); } else { y = new int(*(rhs.y)); }
        return *this;
}
```

Inheritance — More Gotchas

Rules for overloading base class functions in derived classes:

```
class base {
public:
   virtual void f(int) {}
   virtual void f(double) {}
   virtual void g(int i = 10) {}
};
```

```
class Derived: public Base {
public:
   void f(Complex) {}

   void g(int i = 20) {}
};
```

```
Base b;  Derived d;  Base *pb = new Derived;
```

- A derived function with the same name but *without* a matching signature **hides** all base class functions of the same name

```
d.f(1.0);                    // Derived::f(Complex) !!!
```

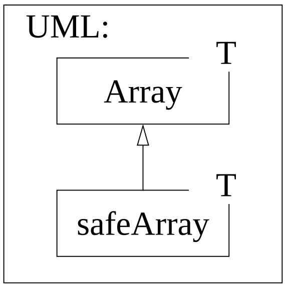
  ➠ use "using" declarations to bring them into scope

- Never change the default argument values of overridden inherited functions

```
b.g();                       // Base::g(int = 10)
d.g();                       // Derived::g(int = 20)
pb->g();                     // Derived::g(int = 10) !!!
```

  ➠ default taken from base class function because compiler does it at compile time

Inheritance — Derived Class Templates



- We had an `Array<T>` template; how about `safeArray<T>`?

```
#include <assert.h>
#include "array.h"

template<class T>
class safeArray : public Array<T> {

public:
   safeArray(int size = def_array_size) : Array<T>(size) {}
   safeArray(const Array<T>& rhs) : Array<T>(rhs) {}
   safeArray(const T* ay, int size) : Array<T>(ay, size) {}

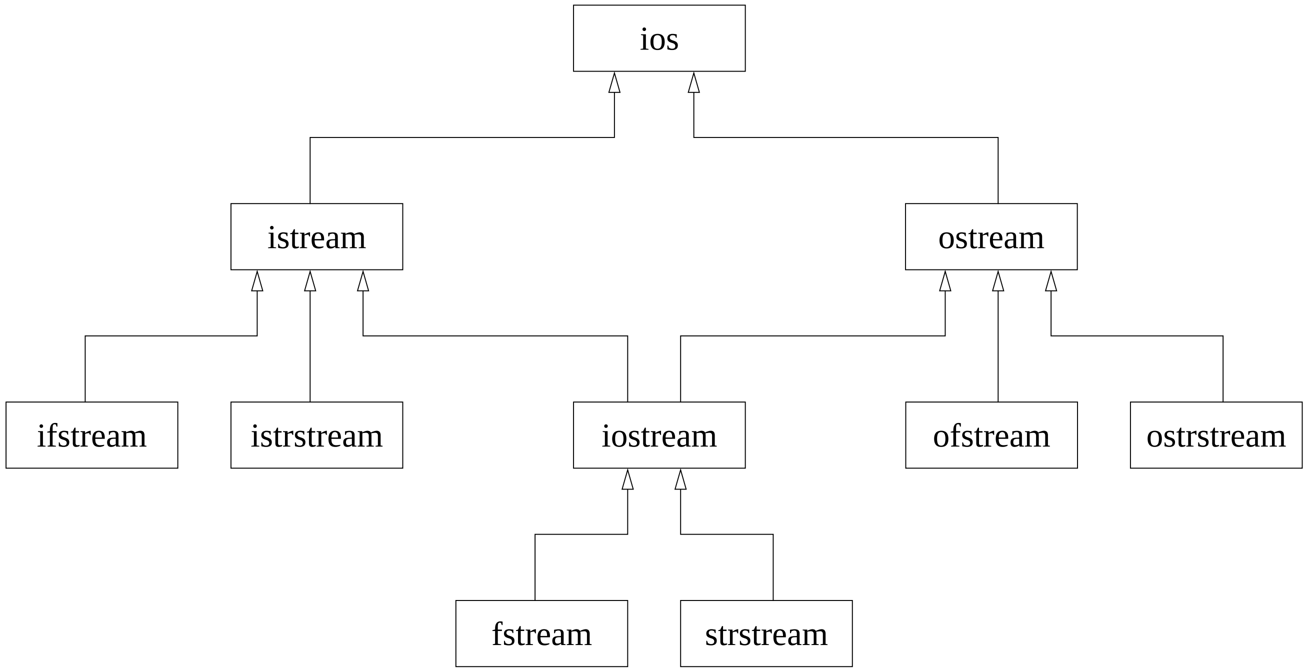
   T& operator[](int index) {
      assert(index >= 0 && index < size());
      return Array<T>::operator[](index);
   }
};
```

- Instantiation: `safeArray<float> ssa(14);` causes base and derived class generation
- Note: `remove_hot()` should also be re-written into a function template

```
template<class T> void remove_hot(Array<T>& day_temp);
```

Inheritance — Example: IOStreams

❑ User-defined input and output operators work for all IOStreams
because IOStream classes actually form inheritance hierarchy!



Inheritance — Example: Arrays with Arbitrary Bounds

❑ Allow arrays with arbitrary bounds (not just `0` to `n-1`):

```
template<class T>                                      // boundarray.h
class boundArray : public safeArray<T> {
public:
   boundArray(int lowIdx, int highIdx)
            : safeArray<T>(highIdx-lowIdx+1), low(lowIdx) {}
   boundArray(const boundArray<T>& rhs)
            : safeArray<T>(rhs), low(rhs.low) {}
   boundArray(int lowIdx, int highIdx, const T* ay)
            : safeArray<T>(ay, highIdx-lowIdx+1), low(lowIdx) {}

   boundArray<T>& operator=(const boundArray<T>& rhs);

   T& operator[](int index);
   const T& operator[](int index) const;

   int lowerBound() const { return low; }
   int upperBound() const { return low+size()-1; }

private:
   int low;
};
```

```
                                                  // boundarray.cpp
template<class T>
boundArray<T>& boundArray<T>::operator=(const boundArray<T>& rhs)
{
   if (this == &rhs) return *this;
   safeArray<T>::operator=(rhs);
   low = rhs.low;
   return *this;
}

template<class T>
T& boundArray<T>::operator[](int index) {
   return safeArray<T>::operator[](index - low);
}

template<class T>
const T& boundArray<T>::operator[](int index) const {
   return safeArray<T>::operator[](index - low);
}
```


- ❑ Better solution for automatically defining mathematical operators:

```
template<class T> class Ring {       // incomplete
public:

   // inline friend: defines global function within class
   friend const T operator*(const T& lhs, const T& rhs) {
      return T(lhs) *= rhs;
   }

   friend const T operator+(const T& lhs, const T& rhs) {
      return T(lhs) += rhs;
   }

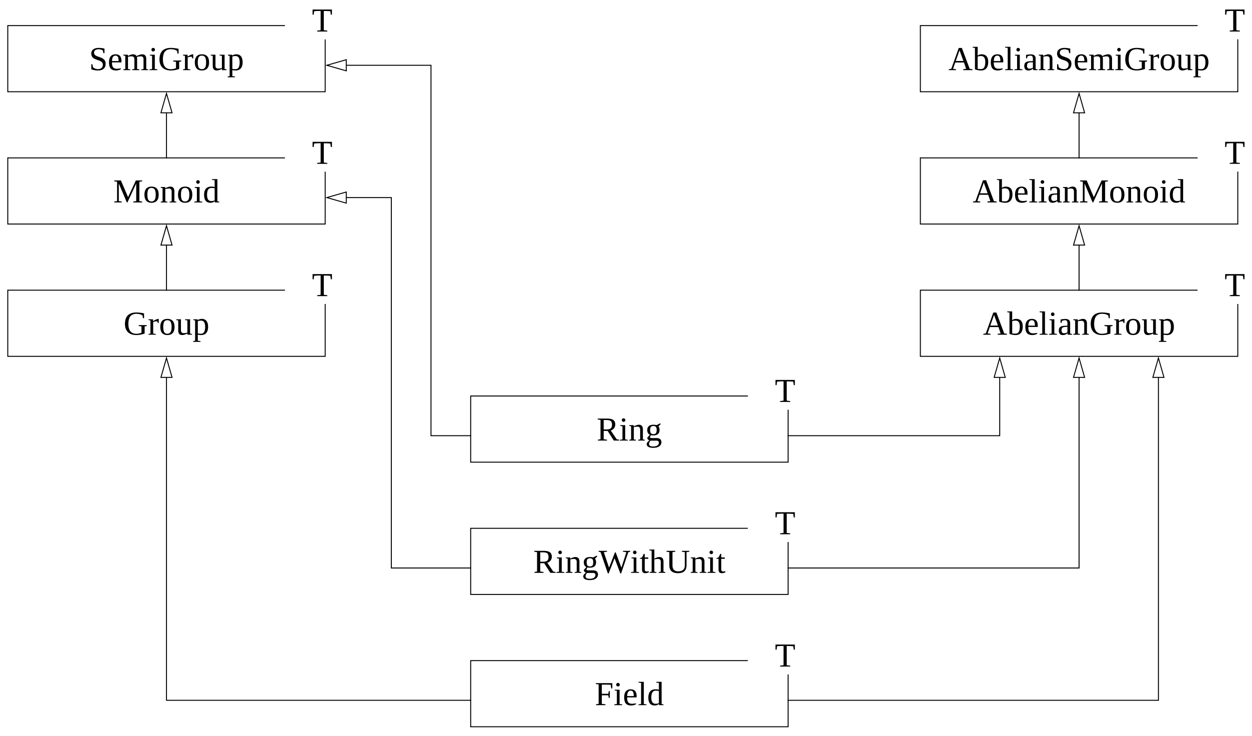
   T& operator++() { return ((T&)*this) += T(1); }

   const T operator++(int) {
      T old((T&)*this); ++(*this); return old;
   }
};
```

- ❑ Now, number classes only define base operators `+=`, `*=` and `T(1)`, then derive from `Ring<T>`:

```
class Complex : public Ring<Complex> { ... };
```

- Why stop with `Ring<T>`? ➠ define classes for other algebraic structures as well



- **Not** every *is-a* relationship in our natural environment is a candidate for inheritance
- Do **not** use inheritance
  - if not all methods of the base class make sense for the derived class
  - if methods would have a different semantic in the derived class
  - if the meaning of components would change in the derived class
  - if values or properties of components would be restricted in the derived class

  Example: A `Square` is a `Rectangle` but the method `change_width()` or `change_height()` would not make sense for square!

- For every possible derived object, it must make sense to assign it to a object of the corresponding base class. In doing so, new (additional) components of the derived class are ignored.

- Do not mistake *is-like-a* for *is-a*!

- Use inheritance carefully!

- *Multiple inheritance* `:=` inherit from more than one base class

```
class derived : public base1, public base2 { /* ... */ };
```

- Experts are disagreeing whether multiple inheritance is useful or not
  - It is essential to the natural modeling of real-world problems
  - It is slow, difficult to implement, and no more powerful than single inheritance
- Object-orient Languages:
  - C++, Eiffel, Common LISP Object System (CLOS) offer multiple inheritance
  - Smalltalk, Objective C, Object Pascal do not
  - Java does not, but allows the implementation of multiple interfaces
- Fact: multiple inheritance in C++ opens up a Pandora's box of complexities

➠ **Use multiple inheritance judiciously**

- If a derived class inherits a member name from more than one class, any usage of that name is inherently ambiguous

  ➠ you must explicitly say which member you mean

```
class Lottery {                      class GraphicObj {
public:                              public:
   virtual int draw();                  virtual int draw();
};                                   };

class LotterySimulation: public Lottery, public GraphicObj {...};

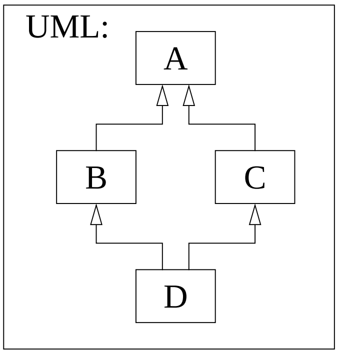
LotterySimulation *pls = new LotterySimulation;
pls->draw();                    // error! ambiguous
pls->Lottery::draw();           // OK
pls->GraphicObj::draw();        // OK
```

- By explicitly qualifying a virtual function, it doesn't act virtual anymore
- It is impossible that `LotterySimulation` can redefine both of the `draw()` functions

Multiple Inheritance — Ambiguity^2: the Diamond

- Sooner or later, you come across the following inheritance relationship:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };
```



- `D` includes two copies of `A`
  - unnecessary duplication
  - no syntactic means of distinguishing them
- Solution: *virtual base classes*

```
class A { /* ... */ };
class B : virtual public A { /* ... */ };
class C : virtual public A { /* ... */ };
class D : public B, public C { /* ... */ };
```

  - only first copy of A is used
  - Problem: specification has to be done of class designer (user/client may not have access)

Multiple Inheritance — Ambiguity^2: the Diamond

- Another problem: an object of class `D` may have *up to five different* addresses:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };

D d;

D* pd1 = &d;    // pd1 is address of D part of D
B* pd2 = &d;    // pd2 is address of B part of D
C* pd3 = &d;    // pd3 is address of C part of D
A* pd4 = (B*) &d;      // pd4 is address of A part of B part of D
A* pd5 = (C*) &d;      // pd5 is address of A part of C part of D
```

- our simple self-test for `operator=()` fails (by checking the address of the object):

```
if (&rhs == this) return;
```

- almost impossible to come up with efficient solution
  (best solution: implement unique object identificator)

- Passing constructor arguments to virtual base classes
  - normally, classes at level `n` of the hierarchy pass arguments to the classes at level `n-1`
  - For virtual base classes, arguments are specified in the classes *most derived* from the base
- Dominance of virtual functions

```
class A { virtual void mf(); /* ... */ };
class B : public A { virtual void mf(); /* ... */ };
class C : public A { /* ... */ };
class D : public B, public C { /* ... */ };

D *pd = new D;
pd->mf();       // A::mf() or B::mf()?
```

  - ➠ normally ambiguous, but `B::mf()` *dominates* if `A` is virtual base class for both `B` and `C`
- Casting restrictions
  - C++ prohibits to cast down from a pointer/reference to a virtual base class to a derived class

➠ **There are cases where multiple inheritance might be useful**

➠ **Rule: try to avoid virtual base classes (diamond-shaped class hierarchy)**

Programming in C++

☆☆☆ More on Arrays ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

More on Arrays — Motivation

❑ What have we got so far with our `array` redesign?

- ❍ size of array need not be constant and known at compile-time
- ❍ `array` assignment and re-assignment
- ❍ optional range checking through derived `safearray`
- ❍ generic usage through templates

❑ What else could be done?

- ❍ adding mathematical operators to get a class `vector`
  - ➠ `vector` can easily implemented by inheriting from `array` or `safearry`
- ❍ what about multi-dimensional arrays like `matrix, array3d, ...?`
  - ➠ implementation?
- ❍ performance of `vector` or `matrix` operations compared to hand-coded loop expressions?

```
for (int i=0; i<size; i++) a[i] = b[i] * c + d[i];
```

- ❍ Functions for selecting parts of an `array` or `vector` (e.g., projections)
- ❍ Sparse arrays, BLAS, LAPACK, `...`

- Derive `vector` from `array` and add mathematical operations

```
#include "array.h"
#include <assert.h>

template<class T> class vector : public array<T> {
public:
   vector(int size = def_array_size) : array<T>(size) {}
   vector(const array<T>& rhs) : array<T>(rhs) {}
   vector(const T* ay, int size) : array<T>(ay, size) {}

   vector<T>& operator+=(const vector<T>& rhs) { // elem. add
      assert (sz == rhs.sz);   // same size?
      for (int i=0; i<sz; ++i) ia[i] += rhs.ia[i];
      return *this;
   }
   vector<T>& operator+=(T rhs) {  // elementwise scalar add
      for (int i=0; i<sz; ++i) ia[i] += rhs;
      return *this;
   }
   // ...
};
```


- Define elementwise addition out of `operator+=` as usual:

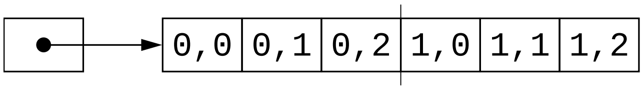
```
template<class T>
vector<T> vector<T>::operator+(const vector<T>& rhs) {
   return vector<T>(*this) += rhs;
}

template<class T>
vector<T> vector<T>::operator+(T rhs) {
   return vector<T>(*this) += rhs;
}
```

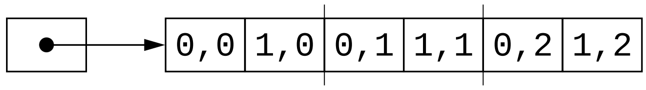
- Global definition of `operator+` to get conversion on both arguments not helpful here, as conversion not defined (for `T=int` even disastrous as `vector` of zeroes of size *n* is added!)

- To complete simple `vector` implementation also implement
  - elementwise substraction, multiplication, division, `...`
  - elementwise trigonometrical (`sin`, `cos`, `...`) and other mathematical functions (`abs`, `...`)
  - dot product, norm, minimum and maximum value or index
  - `...`

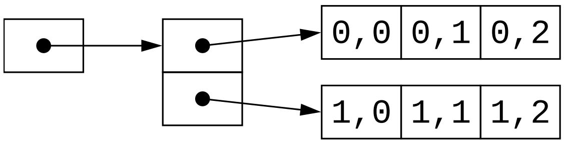
❑ Implementations of `matrix(2,3);` $= \begin{bmatrix} (0,0) & (0,1) & (0,2) \\ (1,0) & (1,1) & (1,2) \end{bmatrix}$



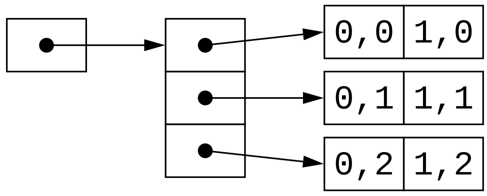
row-major order (C), contiguous



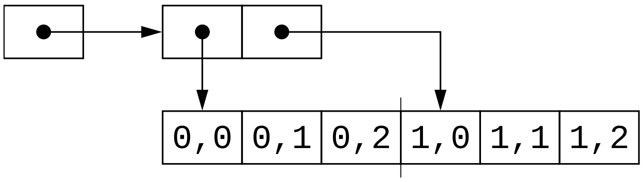
column-major order (Fortran), contiguous



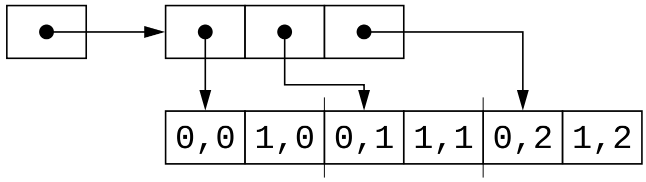
non-contiguous, row vectors



non-contiguous, column vectors



contiguous, row vectors



contiguous, column vectors

➠ even more layouts (sparse, symmetric, packed, `...` representations)

❑ overloaded versions of `operator[]` not possible, e.g., `operator[](int,int)`

❑ but we can use overloaded versions of `operator()` to specify subarray selections

❍ `operator()(int)` is just a replicated version of `operator[]`

```
const T& operator()(int index) const { return ia[index]; }
```

❍ `operator()(int,int)` can be used to select a subarray `[start, end]`

```
array<T> operator()(int start, int end) const {
   return array<T>(ia+start, end-start+1);
}
```

❍ `operator()(int,int,int)` then for strided subarrays `[start, end, stride]`

```
array<T> operator()(int start, int end, int stride) const {
   array<T> r((end-start+stride)/stride);
   for (int i=start, j=0; i<=end; i+=stride) r.ia[j++] = ia[i];
   return r;
}
```

❑ **Not recommended:** could use `operator[]` for indexing with range check, `operator()` without or vice versa (can you remember which is which?)

- Overloaded `operator()` for subarray selection is only sub-optimal solution
  - ➠ e.g., how do you implement it for class `matrix`?
- Better solution:
  - use overloaded `operator()` for indexing in different dimensions
  - use `Range` objects (`Region` for `matrix`, ...) for subarray selection

```
class Range {
public:
   Range (int base);
   Range (int base, int end, int stride = 1);
   ...
};

template <class T>
array<T> array<T>::operator()(const Range&) { /*...*/ }

array<double> a(100), b(100);
a = b(Range(4, 9));
```


- Mathematical relationships between vector, matrix, `...` calls for systematic design
- but multi-dimensional array design can be done in a variety of ways
  - ➠ different uses of arrays require different trade-offs among time, space efficiency, compatibility with existing code (e.g. Fortran libraries), and flexibility
  - ➠ no single array class can cope with the wide range of requirements in practice
- As an example, we have a look on the design used in [Barton and Nackman]
  They distinguish two levels of flexibility:
  - *Rigid arrays*: dimensionality and shape fixed at compile time
  - *Formed arrays*: dimensionality fixed, but shape determined/changeable at runtime

  And they distinguish two kinds based on the way clients can use them:
  - *Interfaced arrays*: shares common *interface base classes* with other array classes; all classes with the same interface can be used interchangeably in client functions
    - ➠ more efficient in code space and programmer time
  - *Concrete arrays*: no common interface base class; no virtual function call overhead
    - ➠ more efficient in function-call time (runtime)

- How can we implement rigid arrays using templates?
  - ➠ Non-type template arguments

```
template<int N, class T>
class array {
private:
   T data[N];
   // ...
};
```

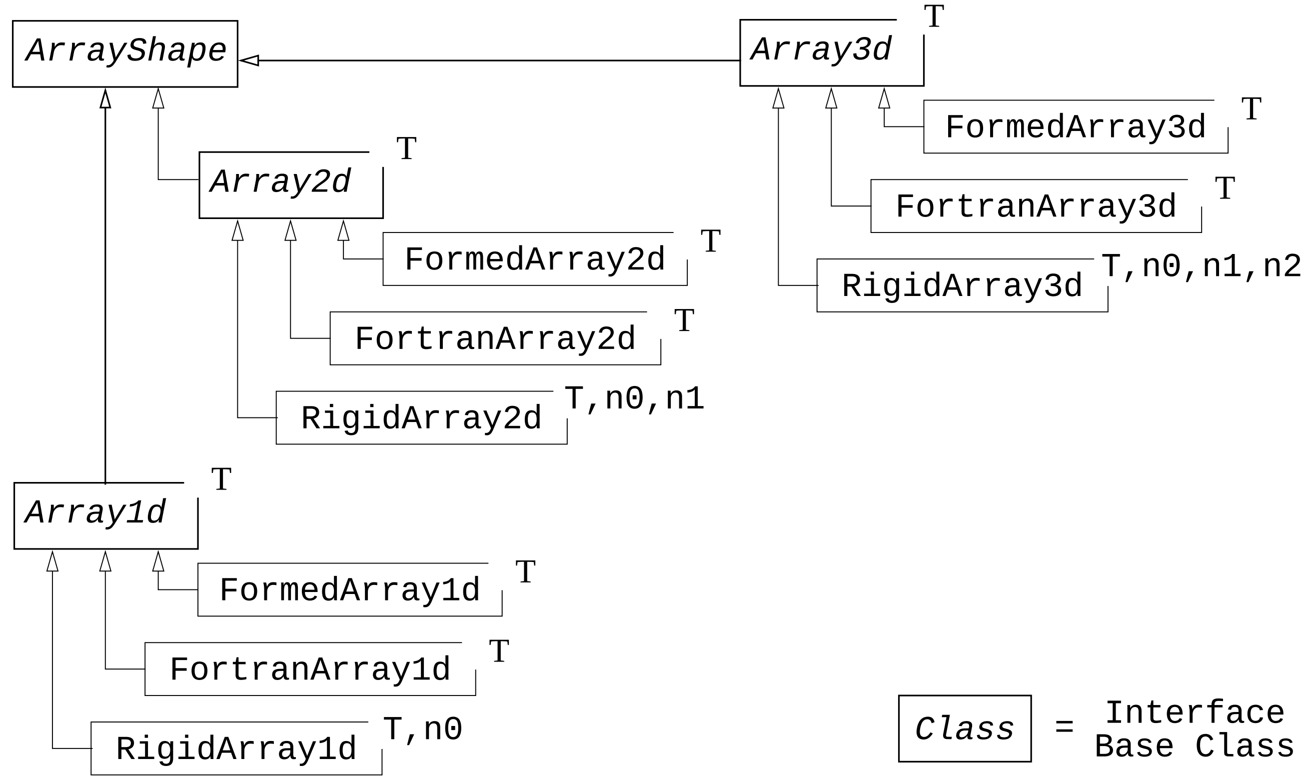
```
Array<100,int>  a;                    // like int a[100];
```

- Recent addition in the ANSI C++ Standard but already implemented in many compilers

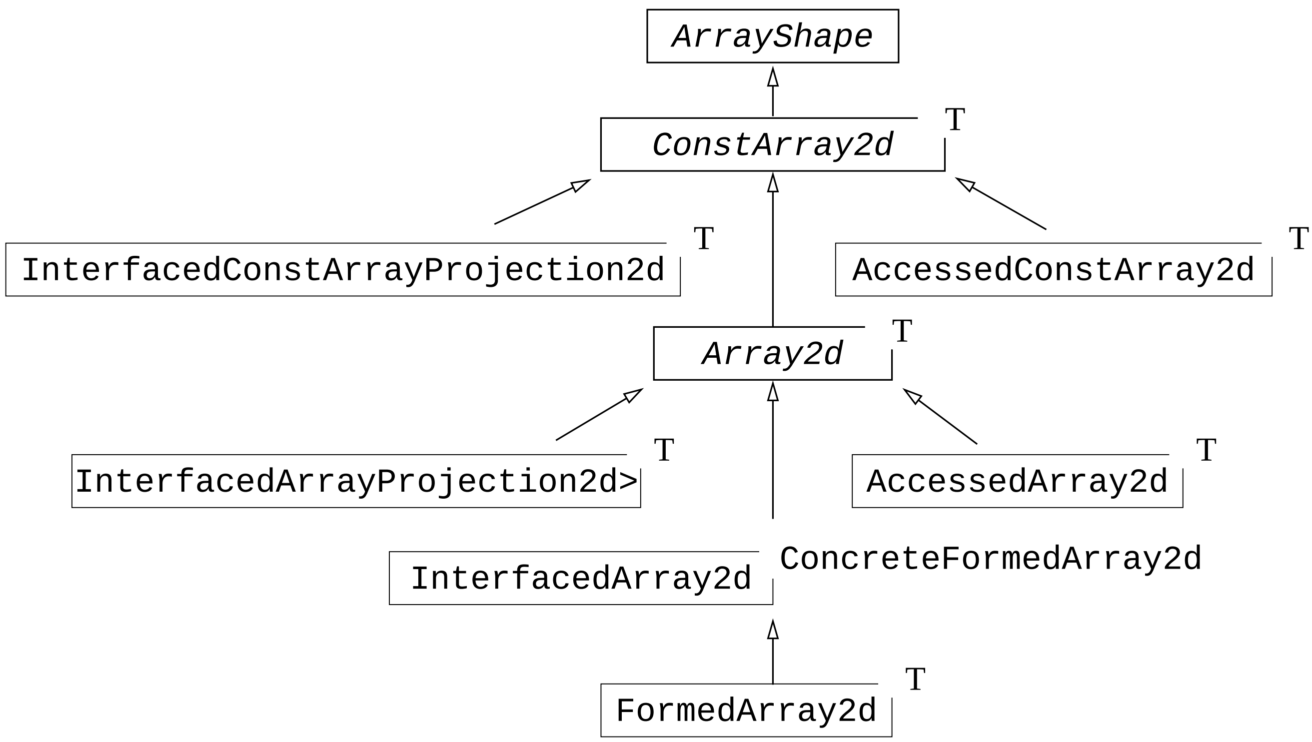
- Example concrete array classes indented for use by client functions

Class name	**Storage layout**
`ConcreteFormedArray1d<T>`	Row major, contiguous
`ConcreteRigidArray1d<T,n0>`	Row major, contiguous
`ConcreteFortranArray1d<T>`	Column major, contiguous
`ConcreteFormedArray2d<T>`	Row major, contiguous
`ConcreteRigidArray2d<T,n0,n1>`	Row major, contiguous
`ConcreteFortranArray2d<T>`	Column major, contiguous
`ConcreteFortranSymPackedArray2d<T>`	Upper triangular, packed
`ConcreteFormedArray3d<T>`	Row major, contiguous
`ConcreteRigidArray3d<T,n0,n1,n2>`	Row major, contiguous
`ConcreteFortranArray3d<T>`	Column major, contiguous

❑ Interfaced array class hierarchy overview



❑ Class hierarchy details for `FormedArray2d<T>` showing 2-dimensional interfaces, projections and accessors



❑ `vector` class definition

```
template <class T, int N>
class vector {
   private:   T *begin, *end;
    public:   vector();
              vector(const vector<T,N>& rhs);
              ~vector();
              T& operator[](int n);
              vector<T,N>& operator=(const vector<T,N>& rhs);
              vector<T,N> operator+(const vector<T,N>& rhs);
...
};
```

❑ Constructor: allocate data space and set end pointer

```
template <class T, int N>
vector<T,N>::vector() : begin(new T[N]), end(begin+N) {}
```

❑ Destructor: free data space

```
template <class T, int N>
vector<T,N>::~vector() { delete [] begin; }
```


❑ Copy Constructor: allocate data space and copy data over

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
   : begin(new T[N]), end(begin+N) {
   T* dest = begin;
   T* src  = rhs.begin;
   while (src != rhs.end) *dest++ = *src++;
}
```

❑ Assignment Operator: after self-test, copy data over

➠ can reuse data space as we know that the vectors have same length

```
template <class T, int N>
vector<T,N>& vector<T,N>::operator=(const vector<T,N>& rhs) {
   if (&rhs == this) return *this;
   T* dest = begin;
   T* src  = rhs.begin;
   while (src != rhs.end) *dest++ = *src++;
   return *this;
}
```

- Addition Operators: elementwise addition, define `operator+()` out of `operator+=()`

```
template <class T, int N>
vector<T,N>& vector<T,N>::operator+=(const vector<T,N>& rhs) {
   T* dest = begin;
   T* src  = rhs.begin;
   while (dest!=end) *(dest++) += *(src++);
   return *this;
}
```

```
template <class T, int N>
vector<T,N> vector<T,N>::operator+(const vector<T,N>& rhs) {
   return vector<T,N>(*this) += rhs;
}
```

- Index Operator: nothing new here

```
template <class T, int N>
T& vector<T,N>::operator[](int n) { return begin[n]; }
```

➠ Rest of class methods left as exercise for the reader!

Problem: Overhead for short `vector` lengths

➠ Template member function specialization using manual loop unrolling and inlining

```
inline vector<double,3>&
vector<double,3>::operator=(const vector<double,3>& rhs) {
   if (&rhs == this) return *this;
   double *dest = begin;              // or:
   double *src  = rhs.begin;          //  begin[0] = rhs.begin[0];
   *dest++ = *src++;                  //  begin[1] = rhs.begin[1];
   *dest++ = *src++;                  //  begin[2] = rhs.begin[2];
   *dest = *src;
   return *this;
}
```

side note: partial specialization would be better (double ➠ T)
but not yet supported by mainstream compilers

```
template<class T> inline vector<T,3>&
vector<T,3>::operator=(const vector<T,3>& rhs) { /*...*/ }
```

❑ Example: elementwise `vector` addition

```
vector<T,N> u, v, w;

// later...                  // typical C (or Fortran-like version)
u = v + w;                   for (int i=0; i<N; ++i) u[i]=v[i]+w[i];
```

❑ Generated code (`GHOST` vector not generated if compiler features "*Return Value Optimization*")

```
v.operator+(w)
➟    vector<T,N> NoName(v);
     ➟    NoName.begin = new T[N]; loop NoName[i] ← v[i];
     NoName.operator+=(w);
     ➟    loop NoName[i] += w[i];
     return NoName;
     ➟    vector<T,N> GHOST(NoName);
          ➟    GHOST.begin = new T[N];
          ➟    loop GHOST[i] ← NoName[i];
          NoName.~vector();
u.operator=(GHOST);
➟    loop u[i] ← GHOST[i];
GHOST.~vector();
```


Problem: Too many temporary objects; e.g. `u = v + w;` ⇒ 2 `new` / `delete` calls!

➟ implement own memory allocation:
`delete` puts object in freelist; `new` uses freelist objects first

❑ Implementation using `Allocate` class modelled after STL:

```
template <class T, int N>
class vector {
private:  T *begin, *end;
          static Allocator<T,N> allocator_data;
public:
  vector() : begin(allocator_data.allocate()), end(begin+N) {}
  ~vector() { allocator_data.deallocate(begin); }
    ...
};

// Definition
template <class T, int N>
Allocator<T,N> vector<T,N>::allocator_data;

// with preallocation: ... ::allocator_data(100);
```

- Class definition

```
template <class T, int N>
class Allocator {
  private:  union {                     // data used either for
               T content[N];            // - vector storage
               Allocator<T,N>* next;    // - pointer in freelist
            };
  public:   Allocator();                // default constructor
            Allocator(int n);           // ctor with preallocation
            T *allocate(void);          // "new"
            void deallocate(T* s);      // "delete"
            ~Allocator();
};
```

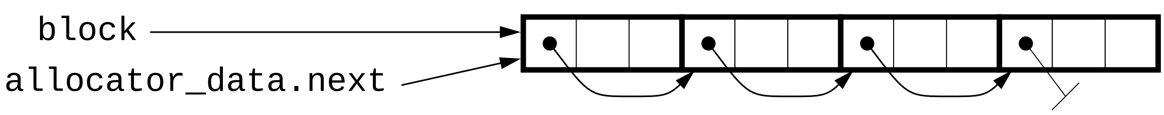
- Default Constructor: setup empty freelist

```
template <class T, int N>
inline Allocator<T,N>::Allocator() { next=0; }
```


- Constructor with Preallocation: allocate block of memory and setup freelist

```
template <class T, int N>
inline Allocator<T,N>::Allocator(int n) {
  Allocator<T,N>* block = new Allocator<T,N>[n];
  for (int i=0; i<n-1; ++i) block[i].next = &(block[i+1]);
  block[n-1].next = 0;
  next = block;
}
```

Example: `Allocator<double,3> vector<double,3>::allocator_data(4);`

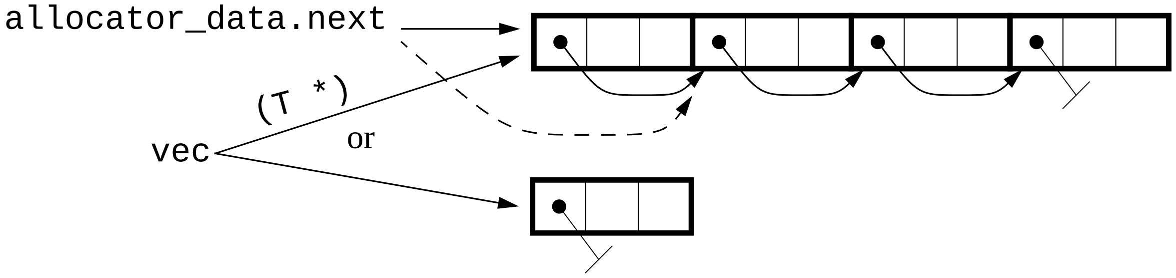


- Destructor

```
template <class T, int N>
inline Allocator<T,N>::~Allocator() {
  if (next) delete next;    // recursive !
}
```

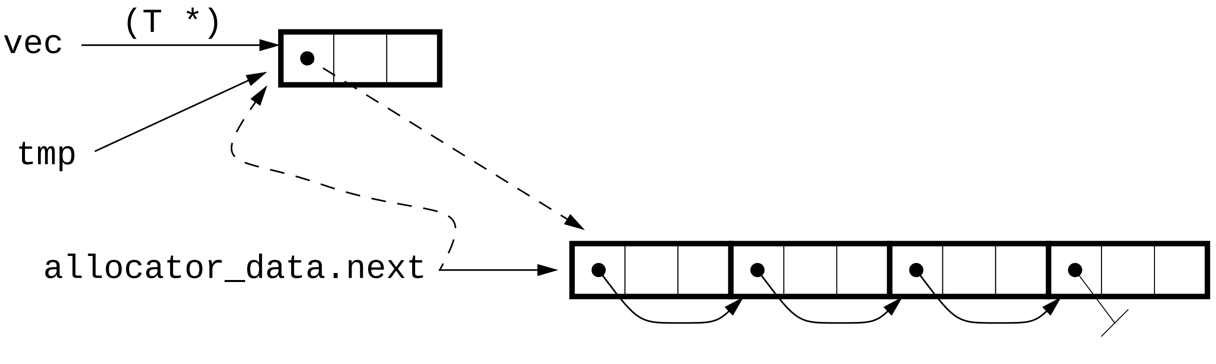
- ❑ Allocate Function ("new")

```
template <class T, int N>
inline T* Allocator<T,N>::allocate(void) {
   if (next) {
      T* vec = next->content;
      next = next->next;
      return vec;
   }  else {
      Allocator<T,N>* tmp = new Allocator<T,N>;
      return tmp->content;
   }
}
```



- ❑ Deallocate Function ("delete")

```
template <class T, int N>
inline void Allocator<T,N>::deallocate(T* vec) {
   Allocator<T,N> *tmp = (Allocator<T,N>*) vec;
   tmp->next = next;
   next = tmp;
}
```



More on Arrays — Standard Optimization: Reference Counting

Problem: copy loops

➠ avoid unnecessary copying using *"copy-on-write"* idiom (implemented by *Reference Counting)*

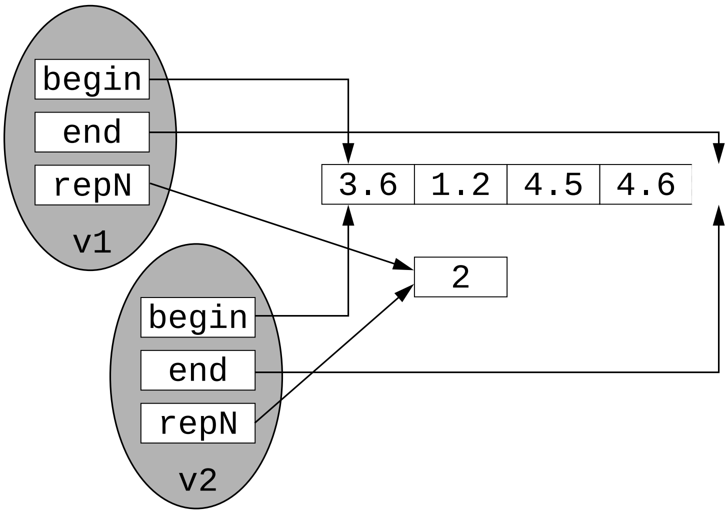
❑ Implementation this time without using *handle/body class* idiom

- ➠ each reference-counted object now has pointer to (shared) reference counter
- ➠ reference counter also managed by optimized memory managment

```
vector<double,4> v1, v2;
v1[0] = 3.6;
v1[1] = 1.2;

...

v2 = v1;
```



More on Arrays — Standard Optimization: Reference Counting

❑ Class `vector` definition

```
template <class T, int N>
class vector {

private:
   T *begin, *end;
   unsigned int *repN;       // reference counter pointer (new)
   static Allocator<T,N>              allocator_data;
   static Allocator<unsigned int, 1> allocator_repN;     // (new)

public:
   vector();
   vector(const vector<T,N>& rhs);
   ~vector();
   T& operator[](int n);

   // ...
};
```

- Constructor now also allocates reference counter and sets it to `1`

```
template <class T, int N>
vector<T, N>::vector(void) : begin(allocator_data.allocate()),
                 end(begin+N), repN(allocator_repN.allocate()) {
  *repN = 1;
}
```

- Copy constructor: shallow copy plus increment of the counter

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
  : begin(rhs.begin), end(rhs.end), repN(rhs.repN) {
  (*repN)++;
}
```

- Destructor only destructs if reference counter goes down to zero

```
template <class T, int N> vector<T, N>::~vector() {
   if (!(--(*repN))) {
      allocator_repN.deallocate(repN);
      allocator_data.deallocate(begin); }
}
```


- Assignment operator: shallow copy plus increment of the counter

```
template <class T, int N>
vector<T, N>& vector<T, N>::operator=(const vector<T, N>& rhs) {
   // self-test
   if (this == &rhs) return *this;

   // destroy lhs if no longer referenced
   if (!(--(*repN))) {
      allocator_repN.deallocate(repN);
      allocator_data.deallocate(begin);
   }

   // increment counter
   (*rhs.repN)++;

   // shallow copy of members
   repN = rhs.repN; begin = rhs.begin; end = rhs.end;
   return *this;
}
```

❑ Index operator: if vector is referenced more than once, make copy

```
template <class T, int N>
T& vector<T, N>::operator[](int n) {
   if ((*repN) == 1)
      return begin[n];
   else {
      (*repN)--;                          // count down old
      repN = allocator_repN.allocate(); // create new ref counter
      *repN = 1;

      // create new data space and copy over
      T* temp = allocator_data.allocate();
      memcpy((void*) temp, (void*) begin, size_t(N*sizeof(T)) );

      begin = temp;
      end   = begin+N;
      return begin[n];
   }
}
```

➠ reference counter especially effective for large arrays

❑ *Temporary Base Class Idiom* = "Reference Counting without counting"

❑ **Idea:** shallow/deep copy flag encoded in types

❑ temporary vectors are represented by class `Tvector`

```
template <class T, int N>
class Tvector {
private:
   T *begin, *end;
   static Allocator<T,N> allocator_data;

public:
```

❑ "Standard" Constructor

```
   Tvector() : begin(allocator_data.allocate()), end(begin+N) {}
```

❑ Copy constructor does "shallow" copy only

```
   Tvector(const Tvector<T,N>& rhs)
      : begin(rhs.begin), end(rhs.end) {}
```

❑ Extra constructor for `vector` copy constructor

```
   Tvector(T* b) : begin(b), end(b+N) {}
```

- Destructor empty as destruction is all handled by class `vector` (`virtual` because `vector` will be derived from `Tvector`)

```
   virtual ~Tvector() {}
```

- "Standard" mathematical operators

```
   Tvector<T, N> operator+(const Tvector<T, N>& rhs);
   // ...
```

- Optimization (see below)

```
   Tvector<T, N> operator+(const vector<T, N>& rhs);
```

- We also need to declare class `vector` a friend:

```
   friend class vector<T, N>;
```

  If your compiler doesn’t yet support friends of templates, we need to use the following "hack"

```
   //protected:  T* start() const { return begin; }
};
```


- `vector` class now subclass of `Tvector`

```
template <class T, int N>
class vector : public Tvector<T,N> {
public:
```

- Default constructor empty as construction is done by `Tvector()`

```
   vector() {}
```

- Deallocate here, as `Tvector` doesn’t delete anything!

```
   ~vector() { allocator_data.deallocate(begin); }
```

- Copy constructor (see below)

```
   vector(const vector<T,N>& rhs);
```

- Assignment and Indexing handled here

```
   T& operator[](int n) { return begin[n]; }
   vector<T,N>& operator=(const vector<T,N>& rhs);
   vector<T,N>& operator=(const Tvector<T,N>& rhs);
   Tvector<T,N> operator+(const vector<T,N>& rhs);
};
```

❑ Copy constructor uses special `Tvector` constructor and does deep copy

```
template <class T, int N>
vector<T,N>::vector(const vector<T,N>& rhs)
   : Tvector<T,N>(allocator_data.allocate()) {
   T* dest = begin; T* src  = rhs.begin;
   while (dest != end) *(dest++) = *(src++);
}
```

❑ temporary variables as well as return type of mathematical operators are now of type `Tvector`

```
template <class T, int N> inline Tvector<T,N>
vector<T,N>::operator+ (const vector<T,N>& rhs) {
   Tvector<T,N> t;
   T* sum1 = begin;
   T* sum2 = rhs.begin;
   T* dest = t.begin;        // can use t.start() if
                             // friend templates don’t work
   while (sum1 != end) *(dest++) = *(sum1++) + *(sum2++);
   return t;
}
```


❑ special assignment operator turn a `Tvector` into a `vector` and handles "`delete`"

```
template <class T, int N>
inline vector<T,N>&
vector<T,N>::operator= (const Tvector<T,N>& rhs) {
   allocator_data.deallocate(begin);
   begin = rhs.begin; // or if necessary: begin = rhs.start();
   end   = rhs.end;   //                  end   = begin+N;
}
```

❑ special optimization for mathematical `Tvector` operations if `this` is already temporary vector (e.g. `(v1+v2)` if `u = v1 + v2 + v3`)

```
template <class T, int N>
inline Tvector<T,N>
Tvector<T,N>::operator+ (const vector<T,N>& rhs) {
   T* sum1 = begin;
   T* sum2 = rhs.begin;
   while (sum1 != end) *(sum1++) += *(sum2++);
   return *this;
}
```

- *Chained Expression Objects*
  - Overloaded operators don't perform the operation but build an expression stack at runtime
  - Assignment operator matches expression stack with library of tuned expression kernels

    `x = v + w;` will execute as `remember expr. "vector + vector"`
    `execute "add(x,v,w)"`

- *Expression templates*
  - Avoid temporary objects in the first place by automatically transform

    ```
    u = v + w;       // vector u, v, w;
    ```

    at compile time (more-or-less) into

    ```
    for (int i=0; i<u.length(); ++i) {
       u[i] = v[i] + w[i];
    }
    ```

    using *Template Meta-Programming* (or *"Compile-Time Programs"*)
  - See Blitz++ (`http://oonumerics.org/blitz/`)
    and PETE (`http://www.acl.lanl.gov/pete/`)

Programming in C++

☆☆☆ The C++ Standard Library and Generic Programming ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

- Namespaces
  - provide a way to partition the global scope
  - a namespace is NOT a module; but it supports techniques related to modularity

```
namespace Chrono {
   class Date {
      /* ... */
   };

   enum Month { Jan, Feb, Mar, Apr, May, Jun,
                Jul, Aug, Sep, Oct, Nov, Dec };

   bool leapyear(int y);
   Date next_weekday(Date d);
}
```

- A `namespace` may only be defined
  - at file scope
  - or nested directly in another `namespace`

- Names declared inside a namespace can be accessed from outside the namespace by using the fully qualified name

  ```
  Chrono::Date d(15, Chrono::May, 1998);
  ```

  ➠ allows mix’n’match between components from different libraries

  ```
  AnotherPackage::Date d2(1998, 5, 15);
  ```

- A namespace can be "unlocked" for the purpose of name lookup with a *using-directive*

  ```
  using namespace Chrono;
  Date d(15, May, 1998);
  ```

  ➠ "if name lookup reaches file scope before the name has been found
    - ✰ search all unlocked namespaces
    - ✰ as well as file scope (which is considered a "special-name" namespace)"

- An individual name can be imported into the current scope with a *using-declaration*

  ```
  using Chrono::Date;
  Date d(15, Chrono::May, 1998);
  ```


The new C++ standard has now an extensive standard library

- Support for the standard C library (but slightly changed to fit C++’s stricter type checking)
- Support for strings (`string`)
- Support for localization and internationalization
- Support for I/O ➠ standardized I/O streams
  - ❍ `string`-based `stringstreams` replaces `char*`-based `strstreams`
- Support for numeric applications
  - ❍ complex numbers: `complex`
  - ❍ special array classes: `valarray`
- Support for general-purpose containers and algorithms
  - ❍ Standard Template Library (STL)

The name *Standard Template Library* is not particularly descriptive because almost everything in the C++ *Standard Library* is a template

- Standard `string` class is actually

```
template < class charT,
           class traits = string_char_traits<charT>,
           class Allocator = allocator>
class basic_string;

typedef basic_string<char> string;
```

- Class `complex` is actually

```
template<class T> class complex;
class complex<float>;
class complex<double>;
class complex<long double>;
```

- Same is true for `valarray`, `ios`, `istream`, `ostream`, **...**

➠ Note: no longer possible to use a forward declaration of these types:

```
class ostream;              // does no longer work!
```


- New C++ headers (note the missing `.h`!):

```
<iostream>     <string>     <vector>
<fstream>      <complex>    <list>       ...
```

  ➠ Do not mix old style and new style headers!
  (e.g., `<iostream.h>` and `<vector>`)

- New C++ headers for C library facilities (not available yet for many compilers)

```
<cassert>   <ciso646>   <csetjmp>   <cstdio>    <ctime>
<cctype>    <climits>   <csignal>   <cstdlib>   <cwchar>
<cerrno>    <clocale>   <cstdarg>   <cstring>   <cwctype>
<cfloat>    <cmath>     <cstddef>
```

- All C++ standard library objects (including the C library interface) are now in namespace `std` !!!

- Compiler vendors often supply migration headers:

  e.g. `<vector.h>`:

```
#include <vector>
using std::vector;
```

- Generic Programming is NOT about object-oriented programming
- Although first research papers appeared 1975, first experimental generic software was not implemented before 1989
  - ➠ not many textbooks on generic programming exist
- First larger example of generic software to become important outside research groups was

  STL (“*Standard Template Library*”)
  - designed by Alexander Stepanov and Meng Lee of HP
  - added to C++ standard in 1994
  - available as public domain software first from HP and now from SGI

- New programming techniques were always based on new *abstractions*
  - often used sequences of instructions ➠ *subroutines*
  - data + interface ➠ *abstract data types*
  - hierarchies of polymorphic ADT’s ➠ *inheritance*
- For *generic programming*:
  - set of requirements on data types ➠ *concept*
- Generic algorithm has
  - generic instructions describing the steps of the algorithm
  - set of requirements specifying the properties the algorithm arguments must satisfiy

➠ only first part can be expressed in C++

➠ templates

- STL is based on four fundamental concepts
  - *containers* hold collections of objects
    - ➠ `bitset`, `vector`, `list`, `deque`, `queue`, `stack`, `set`, `map`, ...
  - *iterators* are pointer-like objects to walk through STL containers
  - *algorithms* are functions that work on STL containers
    - ➠ `find`, `copy`, `count`, `fill`, `remove`, `search`, ...
  - *functors* are function objects used by the algorithms

- To understand these concepts, consider a function to find a value in an `int` array:

```
int find(int array[], int len, int value) {
   int idx;
   for (idx=0; idx<len; ++idx) if (array[idx]==value) return idx;
   return -1;
}
```

  - ➠ can be improved by using pointers to specify begin of search, end of search, and result
  - ➠ also allows for search in subarrays

- To understand the basic principle of STL, look at the rules of C++ (and C) arrays and pointers:
  - ➠ a pointer to an array can legitimately point to any element of the array
  - ➠ or it can point to one element beyond the end of the array (but then it can only be compared to other pointers to the array and it cannot be dereferenced)

  and rewrite `find` to search in the *range* [`begin`, `end`):

```
int* find(int* begin, int* end, int value) {
   while (begin != end && *begin != value) ++begin;
   return begin;   // begin==end if not found
}
```

- You could use `find` function like this:

```
int values[50];
int *firstFive = find(values, values+50, 5);

if (firstFive!=values+50) { //5 found...} else { //not found...}
```

- You can also use `find` to search subranges of the array:

```
int *five = find(values+10, values+20, 5);
```

- Nothing inherent in the `find` function that limits it to array of `int`, so should be `template`:

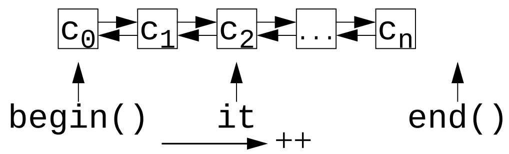
```
template<class T>
T* find(T* begin, T* end, const T& value) {
   while (begin != end && *begin != value) ++begin;
   return begin;
}
```

- Nice, but still to limiting: look at the operations on `begin` and `end`
  - inequality operator, dereferencing, prefix increment, copying (for result!)
  - Why restrict `begin` and `end` to pointers?
  - ➠ allow any object which supports the operations above: *iterators*

```
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& value) {
   while (begin != end && *begin != value) ++begin;
   return begin;
}
```

➠ This version is the actual implementation of `find` in the STL!

- We can apply `find` to every STL container, e.g. `list`




```
list<char> cList;
```

```
// fill cList with values...
```

```
list<char>::iterator it = find(cList.begin(), cList.end(), 'x');
if (it != cList.end()) { // found 'x' ... }
```

➠ `begin()` and `end()` are STL container member functions which return iterators pointing to the beginning and end of the container

- Furthermore, C++ pointers *are* STL iterators, so e.g.,

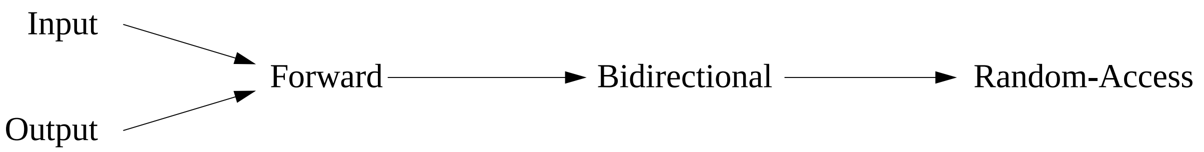
```
int values[50];
```

```
// fill values with actual values ...
```

```
int *firstFive = find(values, values+50, 5);  // calls STL find
if (firstFive != values+50) { // found 5 ... }
```

- A set of requirements (e.g., like for *iterator*) is called a *concept*
- A *concept* can be defined in two (equivalent) ways
  - list of type requirements
  - a set of types
- A *type* `T` is a *model* of a *concept* `C` if
  - `T` satisfies all of `C`’s type requirements
  - `T` belongs to the defining set of types
- *Concept requirements* can**not** fully be described as needed set of member functions (➠ classes)
  - ➠ e.g., `char*` is model of concept iterator but has no member functions
  - ➠ but can be seen as *list of valid expressions*

- Iterator concepts: categorize iterators based on their functionality



Functionality needed:

```
++i  *i  i=j  i==j  i!=j          --i          i[]  i+n  i-n
                                                 i-j  i<j
```

Input iterator `I`: `*I` may only be read once
Output iterator `O`: `*O` may only be written once
Forward iterator `F`: `*F` may be read or written multiple times

- Relation `C1` ⟶ `C2` is called a *refinement* if concept `C2` satisfies all of `C1` requirements and possibly additional requirements
  - ➠ all *models* of `C2` are also models of `C1`
- *Modeling* is relationship between a type and a concept
  Refinement is relationship between two concepts
  (*inheritance* is relationship between two types)

- **STL is just a collection of class and function templates that adhere to a set of conventions**
  - Basic idea behind STL is **simple**
  - STL is **extensible**: you can add your own collections, algorithms, or iterators as long you follow the STL conventions
  - STL is **efficient**: there is no big inheritance hierarchy, no virtual functions, ...
    - STL: C containers + A algorithms
    - Traditional Library: T types $\times C \times A$
  - STL is **portable**

- Disadvantages
  - no error checking (use "safe-STL"!)
  - unusual programming interface

- Using traditional libraries is better than using no library!
  e.g., RogueWave's `tools.h++` (Cray: CCtoollib), `math.h++`, `lapack.h++` (CCmathlib)

- Containers store *collections* of objects (or pointers to objects)

- All STL containers have a standardized interface. Apart from minor differences they provide the same constructors, member functions, and public types

- STL containers are grouped into four categories
  - *sequence* containers
    - linear unsorted collections (`vector`, `deque`, `list`)
    - insert position depends on time / place not value
  - *sorted associative* containers
    - rapid insertion and deletion based on keys (`set`, `map`, `multiset`, `multimap`)
    - insert position depends on value
  - *"almost"* containers (`string`, built-in arrays, `bitset`, `valarray`)
  - container *adapters* (`stack`, `queue`, `priority_queue`)
- There are no *hashed* containers in the C++ Standard STL
  - public domain SGI STL provides them as an extension

- Every STL container declares at least these public types as nested `typedef`s:

Typename	Description
`value_type`	Type of element stored in container (typically `T`)
`size_type`	Signed type for subscripts, element counts, ...
`difference_type`	Unsigned type of distances between iterators
`iterator`	Type of iterator (behaves like `value_type*`)
`const_iterator`	Type of iterator for constant collections (behaves like `const value_type*`)
`reverse_iterator`	For traversing container in reverse order
`const_reverse_iterator`	Same for constant collections
`reference`	Type of reference to element (behaves like `value_type&`)
`const_reference`	Behaves like `const value_type&`

Associative containers provide also:

`key_type`	Type of key
`mapped_type`	Type of mapped value
`key_compare`	Type of comparison criterion


```
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<float> x(5);

  // Get iterators positioned at first and behind "last" element
  vector<float>::iterator first=x.begin(), last=x.end();

  // Get reference to first element and change it
  vector<float>::reference z = x[0];
  z = 8.0;
  cout << "x[0] = " << x[0] << endl;

  // Find the size of the collections
  vector<float>::size_type size=last - first;
  cout << "Size is " << size << endl;
}
```

- Output:

```
x[0] = 8
Size is 5
```

- Every container `Container` provides the following constructors:
  - Default constructor to create empty container

    ```
    Container();
    ```
  - Copy constructor: initialize elements from container of same type

    ```
    Container(const Container&);
    ```
  - Initialize container from container of other type through iterators
    - ➠ element types must be assignment compatible

    ```
    template<class Iterator>
    Container(Iterator first, Iterator last);
    ```
  - Destructor: destroy container and all of its elements

    ```
    ~Container();
    ```
- Examples

```
vector<int> a;                        // calls default ctor
vector<int> b(a);                     // calls copy ctor
list<double> c(a.begin(), a.end());  // calls ctor(iter,iter)
```


- Every STL container provides the following member functions to obtain iterators:

Method	Points to	Associated type
`begin()`	first element	`Container::iterator`
`end()`	one-past-last element	
`rbegin()`	first element of reverse sequence	`Container::reverse_iterator`
`rend()`	one-past-last element of reverse sequence	

- For `const` containers, iterator type is `const_iterator` or `const_reverse_iterator`
- `vector`, `deque`, `string`, `valarray`, and built-in arrays have *random-access* iterators
- `list`, `map`, `multimap`, `set`, and `multiset` have *bidirectional* iterators
- Example: output all values of a vector

not: `it<x.end()`

➠ would require random access!

```
int data[5] = { 23, 42, 666, -89, 5 };
vector<int> x(data, data+5);
for(vector<int>::iterator it=x.begin(); it!=x.end(); ++it)
   cout << *it << endl;
```

note: `++it more efficfient than it++`

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

int main() {
   vector<const char *> mathies(3);
   mathies[0] = "Alan Turing";
   mathies[1] = "Paul Erdos";
   mathies[2] = "Emmy Noether";

   list<const char *> mathies2(mathies.begin(), mathies.end());

   for(list<const char *>::iterator it=mathies2.begin();
                                    it!=mathies2.end(); ++it)
      cout << *it << endl;
}
```

- Output:

```
Alan Turing
Paul Erdos
Emmy Noether
```


- STL containers can be compared using the usual comparison operators
- Equality operator `operator==()`
  - ➠ returns `true` if the containers are the same size
  - ➠ **and** the corresponding elements of each container are equal

  Container elements are compared using `T::operator==()`
- Inequality is defined accordingly to determine if two containers are different
- The other comparison operators
  - ❍ `operator<()`
  - ❍ `operator<=()`
  - ❍ `operator>()`
  - ❍ `operator>=()`
  - ➠ return `true` if first container is *lexicographically* less/less or equal/greater/greater or equal than second container
  - ➠ Containers are compared element by element; first element which differs determines which container is less or greater

STL Containers — Other Member Functions

```
Container a, b;
// fill a and b ...
```

- Assign the contents of one container to the other

  ```
  a = b;
  ```

- Exchange the contents of the two containers

  ```
  a.swap(b);
  ```

- Get the current number of elements stored in container

  ```
  Container::size_type n = a.size();
  ```

- Get the size of the largest possible container

  ```
  Container::size_type maxn = a.max_size();
  ```

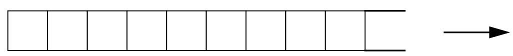
- Check whether container is empty (has no elements)

  ```
  bool isEmpty = a.empty();
  ```

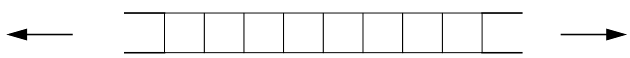

STL Sequence Containers — Overview

- *Sequence* containers provide
  - linear (➠ elements have a predecessor and successor), unsorted collections of elements
  - insert position depends on time / place not value

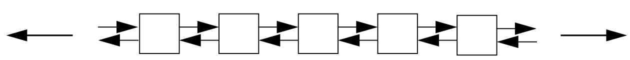
- STL provides three sequence containers:
  - `vector<T>`

    - ➠ dynamic array
    - ➠ fast random access, fast insert and delete *at the end*
  - `deque<T>`

    - ➠ “double-ended queue”
    - ➠ fast random access, fast insert and delete *at both the beginning and end*
  - `list<T>`

    - ➠ doubly-linked list
    - ➠ *slow* random access, fast insert and delete *anywhere*

- In addition to the general constructors for all STL containers
  - `Container();`
  - `Container(const Container&);`
  - ```
    template<class Iterator>
    Container(Iterator first, Iterator last);
    ```

  sequence containers `SeqContainer` also provide
  - Create container with `n` default values

    ```
    SeqContainer(size_type n);
    ```
  - Create container with `n` copies of an element `x`

    ```
    SeqContainer(size_type n, T x);
    ```

- Examples

  ```
  vector<int> x(5);            // => { 0, 0, 0, 0, 0 }
  list<double> y(3, 3.14);     // => { 3.14, 3.14, 3.14 }
  ```



```
SeqContainer a;                // Fill a ...
SeqContainer::iterator it, r; // Point somewhere into a
SeqContainer::iterator f, l;  // Two more iterators
SeqContainer::value_type val; // Some value to insert
```

- Insert element `val` right before the iterator `it`

  ```
  r = a.insert(it, val);        // r points to inserted val
  ```

- Insert `n` copies of element `val` right before the iterator `it`

  ```
  Container::size_type n;
  a.insert(it, n, val);         // void
  ```

- Inserts elements described by iterator range [`f`, `l`) right before the iterator `it`

  ```
  a.insert(it, f, l);           // void
  ```

- Remove element pointed to by iterator `it` or range of elements described by [`f`, `l`)

  ```
  r = a.erase(it);              // r = ++it
  r = a.erase(f, l);            // r = l
  ```

- Remove all elements

  ```
  a.clear();                    // == a.erase(a.begin(), a.end());
  ```
  ```

- *Sequence* containers also provide overloaded forms of a member function `assign`
- `x.assign(...)` has the same effects as `x.clear(); x.insert(x.begin(), ...)` but more efficient
- Assign `n` default values to container

```
assign(size_type n);
```

- Assign `n` copies of an element `x` to container

```
assign(size_type n, T x);
```

- Assign elements from container of other type described by iterators to container
  - ➠ element types must be assignment compatible

```
template<class Iterator>
assign(Iterator first, Iterator last);
```

- Example

```
vector<int> x;
x.assign(3, 42);             // => { 42, 42, 42 }
```


- These operations are provided only for the sequence containers for which they take constant time:

Expression	**Semantics**	**Container**
Getting the first or last element:		
`a.front()`	`*a.begin()`	`vector, list, deque`
`a.back()`	`*--a.end()`	`vector, list, deque`
Adding (`push`) or deleting (`pop`) elements at the beginning or end:		
`a.push_front(x)`	`a.insert(a.begin(),x)`	`list, deque`
`a.push_back(x)`	`a.insert(a.end(),x)`	`vector, list, deque`
`a.pop_front()`	`a.erase(a.begin())`	`list, deque`
`a.pop_back()`	`a.erase(--a.end())`	`vector, list, deque`
Random access to elements:		
`a[n]`	`*(a.begin() + n)`	`vector, deque`
`a.at(n)`	`*(a.begin() + n)`	`vector, deque`

- `at()` provides *bounds-checked* access ➠ throws `out_of_range` if `n >= a.size()`

- Sequence container `vector<T>` provides fast random access to elements
- `vector<T>` can be seen as C style array with a C++ wrapper
  - but will automatically resize itself when necessary
  - adheres to the STL conventions (i.e., defines public types, iterators, ...)
- Usage:

```
#include <vector>
using std::vector;

vector<float> x(10);
```

- `vector<T>` provides *random-access* iterators
  - ➠ **all** STL algorithms may be applied to them

- `vector<T>` also replacement for built-in (especially) dynamic arrays

```
vector<int> x(len);          //instead of:  int *x = new int[len];
                                            ...; delete[] x;

for (int i=0; i<x.size(); ++i) {
   x[i] = 1 + int(6.0*rand()/(RAND_MAX+1.0));  //throw dice
}
```

- Advantages:
  - Size must not be constant
  - `vector<T>` knows its size
  - STL compliant, e.g., STL algorithms and iterator access works too:

```
for (vector<int>::iterator it=x.begin(), it!=x.end(); ++it) {
   cout << " " << *it;
}
```

  - can dynamically increase size (automatically with `insert()` or `push_back()`, not `[]`!!)
  - `vector<T>` objects can be assigned
  - bounds checking possible using `at()`

STL `deque<T>`

- Sequence container `deque<T>`
  - provides fast random access to elements very much like `vector<T>`
  - can be seen as a vector which can grow dynamically on both ends
  - ➠ Unlike `vector<T>`, `deque<T>` supports also constant time insert and erase operations at the beginning **or** end
- As with `vector<T>`, memory allocation and resizing is handled automatically **but** `deque<T>` does **not** provide methods `capacity()` and `reserve()`
- Usage:

```
#include <deque>
using std::deque;

deque<float> x(10);
```

- `deque<T>` provides *random-access* iterators
  - ➠ **all** STL algorithms may be applied to them

STL `list<T>`

- Sequence container `list<T>`
  - provides constant time insert and erase anywhere in the container
    - ➠ unlike `vector<T>` or `deque<T>`, insert or erase operations never invalidate pointers or iterators
  - has automatic storage management
  - does **not** support random access
- `list<T>` is typically implemented as a doubly-linked list
  - ➠ additional storage overhead *per node* (at least two pointers)
- Usage:

```
#include <list>
using std::list;

list<float> x(10);
```

- `list<T>` provides *bidirectional* iterators
  - ➠ **not** all STL algorithms may be applied to them (e.g. `sort()`!)

```
#include <iostream>
#include <list>
using namespace std;

int main() {
   list<const char *> mammals;
   mammals.push_back("cat");
   mammals.push_back("dog");

   list<const char *>::iterator it = mammals.begin();
   ++it;        // 'it' is now at "dog"

   mammals.insert(it, "cow");           // 'it' is still at "dog"

   const char *others[] = { "horse", "pig", "rabbit" };
   mammals.insert(it, others, others+3);// 'it' is still at "dog"

   for(it=mammals.begin(); it!=mammals.end(); ++it)
      cout << " " << *it;
}
```

- Output:

```
cat cow horse pig rabbit dog
```


- STL generic sorting algorithms require *random-access* iterators (which `list<T>` doesn't have)
  - ➠ `list<T>` provides its own efficient `sort()` member function
- Example

```
#include <iostream>
#include <list>
#include <string>
using namespace std;

int main() {
   const char *places[] = { "Paris", "Rom", "London", "Juelich" };

   list<string> placeList(places, places+4);
   placeList.sort();

   for(list<string>::iterator it=placeList.begin();
                                  it!=placeList.end(); ++it)
      cout << " " << *it;
}
```

- Output:

```
Juelich London Paris Rom
```

- What sequence container should be used?
  - Use `vector<T>` when
    - fast, random-access is needed to the contents
    - insertions and deletions usually occur at the end of the sequence
    - size of the collection is reasonably constant
      or occasional resizing can be tolerated
      or size is known in advance
  - If frequent insertions and deletions at the beginning of the sequence are also required, consider using `deque<T>`
  - If frequent insertions and deletions are necessary everywhere in the sequence, use `list<T>`

- STL provides *adaptors* which turn *sequence* containers into containers with *restricted interface*
  - In particular, no iterators provided; can only be used through the specialized interface
  - No “real” STL containers (STL algorithms cannot be used with the adapted containers)

  There are `stack<T>`, `queue<T>`, and `priority_queue<T>`


```
template <class T, class Container = deque<T> >  class stack {
public: typedef typename Container::value_type value_type;
        typedef typename Container::size_type size_type;

        explicit stack(const Container& = Container());
        bool empty() const;
        size_type size() const;
        value_type& top();               // get top element
        const value_type& top() const;
        void push(const value_type& x);  // add element on top
        void pop();                      // remove top element
};
```

- `Container` can be any STL container with `empty()`, `size()`, `back()`, `pop_back()`, and `push_back()`
- Example:

```
#include <stack>
using std::stack;

stack<char> s1;                // uses deque<char> for storage
stack<int, vector<int> > s2;   // uses vector<int>
```

```
template <class T, class Container = deque<T> >  class queue {
public: // the same typedefs as stack
        explicit queue(const Container& = Container());
        bool empty() const;
        size_type size() const;
        value_type& front();             // get front element
        const value_type& front() const;
        value_type& back();              // get back element
        const value_type& back() const;
        void push(const value_type& x);  // add element to back
        void pop();                      // remove front element
};
```

- `Container` can be any STL container with `empty()`, `size()`, `front()`, `back()`, `push_back()`, and `pop_front()` ➠ `vector` cannot be used!
- Example:

```
#include <queue>
using std::queue;

queue<char> q1;
```



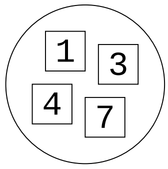
```
template <class T, class Ctr = vector<T>, class Cmp =
         less<typename Ctr::value_type> >  class priority_queue {
public:
   // the same typedefs as stack
   explicit priority_queue(const Cmp&=Cmp(), const Ctr&=Ctr());
   bool empty() const;
   size_type size() const;
   const value_type& top() const;    // get "top" element
   void push(const value_type& x);   // add element
   void pop();                       // remove "top" element
};
```

- *"top"* element `:=` element with *highest priority* ➠ *largest* element based on `Cmp`
- Container `Ctr` must provide *random-access* iterators ➠ `list` cannot be used!
- Example:

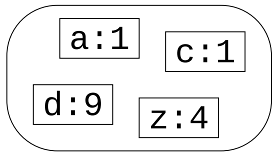
```
#include <queue>
using std::priority_queue;

priority_queue<char> pq1;
```

- *Associative* containers
  - store the elements based on *keys*
  - are sorted (based on keys)

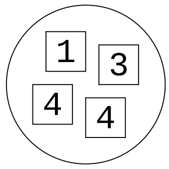


`set<int>`

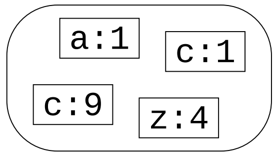


`map<char,int>`

- STL provides four associative containers:
  - `map<Key, T>`
    - ➠ collection of (key, value) pairs
    - ➠ fast retrieval based on keys
    - ➠ each key in map is *unique* ➠ one value per key
  - `set<T>`
    - ➠ `map` where key is value itself
  - `multimap<Key,T>` and `multiset<T>`
    - ➠ versions of `map` and `set` where keys must not be unique
    - ➠ more than one value per key



`multiset<int>`



`multimap<char,int>`

- maps called *associative array* or *table*, multimap *dictionary*, and multiset *bag* in other languages

- Associative containers are *sorted*
  - based on `key_type::operator<()` or
  - a user-specified predicate `Cmp`
    - ➠ `Cmp` must define *strict weak ordering* (`[1]+[2]` *is partial ordering*)

```
[1]  Cmp(x,x) is false                              // irreflective
[2]  If Cmp(x,y) and Cmp(y,z), then Cmp(x,z)        // transitive
[3]  If equiv(x,y) and equiv(y,z), then equiv(x,z)
```

- Associative containers have an optional extra template parameter specifying the sorting criterion ➠ default is `key_type::operator<()`
- Equality of keys is tested using `key_type::operator==()`
  - ➠ with user-specified sorting criterion `Cmp`

    `equiv(x,y)` is defined as `!(Cmp(x,y) || Cmp(y,x))`

- **Note**: `operator<()` for C style strings (`char *`) compares pointers not the contents!
  - ➠ use:

```
bool strLess(const char *p1, const char *p2)
                   { return strcmp(p1, p2) < 0; }
```

STL Associative Containers — Finding Elements

- The member functions return iterator `end()` to indicate “*not found*”
- There are five member functions for locating a key `k` in an associative container `ac`
  - Return iterator to element with key `k`

    ```
    iterator it = ac.find(k);
    ```
  - Find number of elements with key `k`

    ```
    size_type c = ac.count(k);
    ```
  - Find *first* element with key `k`

    ```
    iterator it = ac.lower_bound(k);
    ```
  - Find *first* element with key *greater* than `k`

    ```
    iterator it = ac.upper_bound(k);
    ```
  - Get subsequence [`lower_bound`, `upper_bound`) at once

    ```
    pair<iterator,iterator> p = ac.equal_range(k);
    ```
- Of course, the last three member functions are more useful for `multiset` and `multimap`

STL `set<T>` and `multiset<T>` — Basics

- Associative container `set<T>` keeps a *sorted* collection of *unique* values
- Associative container `multiset<T>` keeps a *sorted* collection of values
- ➠ for `set` and `multiset`, key is value itself!
- ➠ `T` == `value_type` == `key_type`
- Usage:

  ```
  #include <set>
  using std::set;

  set<float> x;                        // sorted by <
  set<int, greater<int> > y;           // sorted by >

  std::multiset<double> z;
  ```
- `set<T>` and `multiset<T>` provide *bidirectional* iterators
  - ➠ **not** all STL algorithms may be applied to them

- The position of elements in an *associative* container are determined by the sorting criterion
  - ➠ insertion member functions do **not** have iterator parameter specifying position for insert
- Two more forms of `insert` are provided for a `set` or `multiset` container `SetContainer`
  - Insert value `val`

```
SetContainer c;
SetContainer::iterator it = c.insert(val);
```

  - Add elements from other container described by iterator range [`f`, `l`)

```
c.insert(f, l);                        // void
```

- For `set`, keys are only inserted if they do not already occur in the container.
  Also, instead of returning an iterator, `insert(val)` returns a `pair<iterator,bool>` where the second part is `false` when the value was already present and was not inserted.
- In addition to the usual ways of deleting elements (`erase(it)`, `erase(f,l)`, and `clear()`), associative containers also allow deletion based on keys (== values for sets):

```
c.erase(key);
```



```
#include <iostream>
#include <set>
using namespace std;

int main() {
   set<int> x;

   x.insert(42);
   x.insert(496);
   x.insert(6);
   x.insert(42);
   x.insert(-5);

   for(set<int>::iterator it=x.begin(); it!=x.end(); ++it)
      cout << " " << *it;
   cout << endl;
}
```

- Output:

```
 -5 6 42 496
```

```
#include <iostream>
#include <set>
using namespace std;

int main() {
  multiset<int> x;

  x.insert(42);
  x.insert(496);
  x.insert(6);
  x.insert(42);
  x.insert(-5);

  for(multiset<int>::iterator it=x.begin(); it!=x.end(); ++it)
    cout << " " << *it;
  cout << endl;
}
```

- Output:

```
 -5 6 42 42 496
```


STL `map<Key,T>` and `multimap<key,T>` Basics

- Associative container `map<Key,T>` keeps a *sorted* collection of *unique* (key, value) *pairs*
- Associative container `multimap<Key,T>` keeps a *sorted* collection of (key, value) *pairs*
- `map` and `multimap` have the same forms of `insert` (and `erase`) like `set`s, **but** their `value_type` is defined as `pair<const Key,T>`
  - ➠ `insert` member functions take parameter of type `pair<const Key,T>`
  - ➠ iterators point to `pair<const Key,T>`

  (`const Key` because changing keys could destroy sorting)
- Usage:

```
#include <map>
using std::map;

map<int,int> m;                              // int -> int
std::multimap<char*,int,strLess> mm;    // char* -> int
```

- `map<Key,T>` and `multimap<Key,T>` provide *bidirectional* iterators
  - ➠ **not** all STL algorithms may be applied to them

- STL provides small utility class `pair<T1,T2>` which essentially looks like this:

```
template<class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair(const T1& f, const T2& s) : first(f), second(s) {}
};
```

- Example of `map insert`

```
map<string,float> numbers;
numbers.insert(pair<string,float>("Pi", 3.141592653589));
numbers.insert(pair<string,float>("e",  2.718281828459));
```

- To make `pair`s slightly less ugly, STL provides `make_pair()` function:

```
numbers.insert(make_pair("Pi", 3.141592653589));
```

- In addition, `map` (but not `multimap`) provides an overloaded `operator[]`:

```
numbers["Pi"] = 3.141592653589;
```

  - ➠ lookup key and return reference to value; if not found, *insert* `pair(key,T())`!
  - ➠ For lookup without modifying map use `find()` or `count()`!


```
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main() {
  map<string,double> prices;

  prices["sugar"] = 2.99;
  prices["salt"]  = 1.87;
  prices["flour"] = 2.49;

  cout << "Sugar costs " << prices["sugar"] << endl;

  map<string,double>::iterator h = prices.find("salt");
  cout << "Salt costs " << h->second << endl;
}
```

- Output:

```
Sugar costs 2.99
Salt costs 1.87
```

STL `multimap<Key,T>` Example

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
typedef pair<string,string> Pair;
typedef multimap<string,string> strMap;
typedef pair<strMap::iterator, strMap::iterator> iterPair;

int main() {
  strMap phonebook;
  phonebook.insert(Pair("Hans", "0123 / 454545"));
  phonebook.insert(Pair("Lisa", "0999 / 12345"));
  phonebook.insert(Pair("Moni", "0180 / 999999"));
  phonebook.insert(Pair("Hans", "0771 / 16528"));

  iterPair ip = phonebook.equal_range("Hans");
  for(strMap::iterator it=ip.first; it!=ip.second; ++it)
    cout << "Hans's phone number is " << it->second << endl;
}
```

- Output:

```
Hans's phone number is 0123 / 454545
Hans's phone number is 0771 / 16528
```


STL Algorithms

- STL provides a large assortment of algorithms which can be applied to *any* data structure which adheres to the STL conventions (especially provides iterators)
- Defined in the header file `<algorithm>`
- STL algorithms can be grouped into four categories:
  - *Nonmutating sequence algorithms* operate on containers *without* making changes to their contents (e.g., `find`)
    - ➠ can also be applied to `const` container
  - *Mutating sequence algorithms* operate on containers *changing* their contents (e.g., `fill`). Often, mutating algorithms come in two versions `foo()` and `foo_copy()`:
    - ✰ *In-place version*: replaces original contents of container with results
    - ✰ *Copying version*: places the result in a new container
  - *Sorting-related algorithms* sort containers or operate only on *sorted* containers
    - ➠ `set<T>`, `map<T>`, sorted `vector<T>`s, ...
  - *Generalized numeric algorithms* defined in header `<numeric>`

- Some STL algorithms have a version which takes an extra function parameter.
- For example, one version of `sort()` has the following prototype:

```
template<class RandomIter, class Compare>
void sort(RandomIter first, RandomIter last, Compare cmp);
```

`cmp` is a function parameter specifying the sorting criterion for sorting the range [`first`, `last`)

- For example, suppose we have a very simple class `Person`:

```
struct Person {
   Person(const string& first, const string& last) :
                              firstName(first), lastName(last) {}
   string firstName, lastName;
};

ostream& operator<<(ostream& ostr, const vector<Person>& folks) {
   vector<Person>::const_iterator it;
   for (it=folks.begin(); it!=folks.end(); ++it)
      ostr << it->firstName << " " << it->lastName << endl;
   return ostr;
}
```


- We now want to be able to sort a vector of `Person` according to either the first or last name
- Function parameters can be expressed in two ways:
  - as *pointer to functions*:

```
bool compareFirstName(const Person& p1, const Person& p2) {
   return p1.firstName < p2.firstName;
}
```

  - as *function objects* or *functors* `:=` objects which behave like functions
    - object with function call operator `operator()` defined

```
struct compareLastName {
   bool operator()(const Person& p1, const Person& p2) {
      return p1.lastName < p2.lastName;
   }
};
```

    - biggest advantage: function objects can have local *state*
    - also: more efficient because can be inlined (no indirect function calls!)
- Non-modifying *Functions* or *functors* which return `bool` are also called *predicates*

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() {
  vector<Person> folks;

  folks.push_back(Person("Hans", "Lustig"));
  folks.push_back(Person("Lisa", "Sommer"));
  folks.push_back(Person("Moni", "Hauser"));
  folks.push_back(Person("Herbert", "Hauser"));

  // -- sort with function pointer
  sort(folks.begin(), folks.end(), compareFirstName);
  cout << "Sorted by first name:" << endl << folks << endl;

  // -- sort with functor; note the () to call default ctor to get an object!
  sort(folks.begin(), folks.end(), compareLastName());
  cout << "Sorted by last name:" << endl << folks << endl;
}
```


- To keep the following description of the STL algorithms compact
  - the `template<...>` specification was omitted
  - secondary forms of the same algorithm are not described in full detail (e.g., the `_copy` version or versions with an additional optional parameter)
  - the template parameter names are significant:

Name	**Meaning**
`In`	input iterator
`Out`	output iterator
`For`	forward iterator
`Bi`	bidirectional iterator
`Ran`	random-access iterator
`Op, BinOp`	unary and binary operation (function or functor)
`Pred, BinPred`	unary and binary predicate
`Cmp`	sorting criterion with strict weak ordering

Linear Search

- Find first element equal to `value`; return `last` when not found

```
In find(In first, In last, const T& value);
```

- Find first element for which unary predicate `pred` is true

```
In find_if(In first, In last, Pred pred);
```

- Example: use `find_if` to find all persons whose last name matches a string (using `string::find()`!)
  - ➠ define predicate functor

```
struct lastNameMatch {
   lastNameMatch(const string& pattern) : patt(pattern) {}

   bool operator()(const Person& p) {
      return p.lastName.find(patt) != string::npos;
   }

private:
   string patt;
};
```



```
int main() {
   vector<Person> folks;

   folks.push_back(Person("Hans", "Lustig"));
   folks.push_back(Person("Lisa", "Sommer"));
   folks.push_back(Person("Moni", "Hauser"));
   folks.push_back(Person("Herbert", "Hauser"));

   vector<Person>::iterator it = folks.begin();

   while (true) {
      it = find_if(it, folks.end(), lastNameMatch("use"));
      if (it == folks.end()) break;
      cout << it->firstName << " " << it->lastName << endl;
      ++it;
   }
}
```

- Output:

```
Moni Hauser
Herbert Hauser
```

Linear Search (cont.)

- Find first element out of any values in [`first2`, `last2`) in the range [`first1`, `last1`)

```
For find_first_of(For first1, For last1, For first2, For last2);
```

- Find first consecutive duplicate element

```
For adjacent_find(For first1, For last1);
```

- Find *first* occurrence of subsequence [`first2`, `last2`) in [`first1`, `last1`)

```
For search(For first1, For last1, For first2, For last2);
```

- Find *last* occurrence of subsequence [`first2`, `last2`) in [`first1`, `last1`)

```
For find_end(For first1, For last1, For first2, For last2);
```

- Find first occurrence of subsequence of `count` consecutive copies of `value`

```
For search_n(For first1, For last1, Size count, const T& value);
```

➠ All return `last1` for indicating not found

➠ These five algorithms are also available with an additional parameter `pred` specifying a binary predicate to be used instead of `operator==()` to check for equality

Counting Elements

- Count number of elements equal to `value`

```
difference_type count(In first, In last, const T& value);
```

- Count number of elements that satisfy predicate `pred`

```
difference_type count_if(In first, In last, Pred pred);
```

Minimum and Maximum

- Return the minimum or maximum of two elements

```
const T& min(const T& a, const T& b);
const T& max(const T& a, const T& b);
```

- Find smallest and largest element

```
For min_element(For first, For last);
For max_element(For first, For last);
```

➠ These four functions are also available with an additional parameter `cmp` specifying a sorting criterion to be used instead of `operator<()` for comparing elements

Comparing Two Ranges

- ❑ Is range1 equal to range2 (i.e., same elements in same order)?

```
bool equal(In first1, In last1, In first2);
```

- ❑ Find first position in each range where the ranges differ

```
pair<InIt1, InIt2> mismatch(In first1, In last1, In first2);
```

- ➠ Second range starting at `first2` must be at least as long as range1 [`first1`, `last1`)
- ➠ Both also available with additional parameter specifying binary predicate `pred`
- ❑ Is range1 lexicographically less than range2?

```
bool lexicographical_compare(In first1, In last1,
                             In first2, In last2);
```

- ➠ Also available with additional parameter specifying sorting criterion `cmp`

for_each

- ❑ Apply unary function or function object `f` to each element in the sequence
  Return `f` (useful for functors which keep / calculate state information)

```
Op for_each(In first, In last, Op f);
```


Copying Ranges

- ❑ Copy range1 [`first`, `last`) to range2 starting at `result`

```
Out copy(In first, In last, Out result);
```

- ❑ Copy range1 [`first`, `last`) to range2 ending at `result` backwards
  - ➠ use this version if range1 and range2 *overlap* (start of range2 is between `first` and `last`)

```
Bi copy_backward(Bi first, Bi last, Bi result);
```

- ➠ The result range must have enough elements to store result!
  (the elements are *assigned* the result values, they are not *inserted*)

- ❑ Example:

```
block<int,6> x = { 1, 2, 3 4, 5, 6 };
vector<int> not_ok, ok(x.size());

copy(x.begin(), x.end(), not_ok.begin());     // undefined!
copy(x.begin(), x.end(), ok.begin());         // OK!
```

Transforming Elements

❑ Apply unary operation `op()` to elements in range [`first`, `last`) and write to `res`

```
Out transform(In first, In last, Out res, Op op);
```

❑ Apply binary operation `op()` to corresponding elements in range1 [`first1`, `last1`) and range2 starting at `first2` and write to range starting at `res`

```
Out transform(In first1, In last1, In first2, Out res, BinOp op);
```

❑ Example:

```
inline int Square(int z) { return z*z; }

block<int,6> x = { 1, 2, 3 4, 5, 6 };
vector<int> res(x.size());

transform(x.begin(), x.end(), res.begin(), Square);

for (vector<int>::iterator it=res.begin(); it!=res.end(); ++it)
  cout << *it << " ";
cout << endl;
```

Output:

```
1 4 9 16 25 36
```


Replacing Elements

❑ Replace all elements with value `old` in-place with value `new`

```
void replace(For first, For last, const T& old, const T& new);
```

❑ Replace all elements satisfying predicate `pred` in-place with value `new`

```
void replace_if(For first, For last, Pred pred, const T& new);
```

➠ Also available as `replace_copy` and `replace_copy_if` with additional third parameter describing result range starting at output iterator `out`

Filling Ranges

❑ Assign `value` to all elements in range [`first`, `last`) or [`first`, `first+n`)

```
void fill(For first, For last, const T& value);
void fill_n(Out first, Size n, const T& value);
```

❑ Assign result of calling `gen()` to all elements in range [`first`, `last`) or [`first`, `first+n`)

```
void generate(For first, For last, Generator gen);
void generate_n(Out first, Size n, Generator gen);
```

Removing Elements

- ❑ "Remove" all elements equal to `value`

```
For remove(For first, For last, const T& value);
```

- ❑ "Remove" all elements which satisfy predicate `pred`

```
For remove_if(For first, For last, Pred pred);
```

- ❑ "Remove" all consecutive duplicate elements (using `operator==()` or predicate `pred`)

```
For unique(For first, For last);
For unique(For first, For last, BinPred pred);
```

- ➠ Also available as `remove_copy`, `remove_copy_if`, and `unique_copy` with additional third parameter describing result range starting at output iterator `out`
- ➠ **Note**: These functions do **not** really remove the elements from the sequence but move them to the end returning the position where the "removed" elements start
- ➠ To really delete the elements from a Container `C` use `erase()`, e.g., for `remove`:

```
C.erase(remove(C.begin(), C.end(), x), C.end());
```


Permuting Algorithms

- ❑ Reverse range in-place or copy reverse range to range starting at `res`

```
void reverse(Bi first, Bi last);
Out reverse_copy(Bi first, Bi last, Out res);
```

- ❑ Exchange [`first`, `middle`) and [`middle`, `last`) in-place or copy to `res`

```
void rotate(For first, For middle, For last);
Out rotate_copy(For first, For middle, For last, Out res);
```

- ❑ Transform range into next or previous lexicographical permutation

```
bool next_permutation(Bi first, Bi last);
bool prev_permutation(Bi first, Bi last);
```

  - ➠ also available with additional parameter specifying sorting criterion `cmp`

- ❑ Randomly re-arrange elements (using the internal or an user-specified random-number generator)

```
void random_shuffle(Ran first, Ran last);
void random_shuffle(Ran first, Ran last, RandomNumberGen& rand);
```

Partitions

- ❑ Reorder the elements so that all elements satisfying `pred` precede the elements failing it

  `Bi partition(Bi first, Bi last, Pred pred);`

- ❑ Same as partition but preserves relative order of elements

  `Bi stable_partition(Bi first, Bi last, Pred pred);`

Swapping Elements

- ❑ Exchange values of elements `a` and `b`

  `void swap(T& a, T& b);`

- ❑ Exchange values of elements pointed to by `it1` and `it2` (`swap(*it1, *it2)`)

  `void iter_swap(For it1, For it2);`

- ❑ Swap the corresponding elements of range1 [`first1`, `last1`) and range2 starting at `first2`

  `For swap_ranges(For first1, For last1, For first2);`

Sorting Ranges

- ❑ Sort elements in ascending order (smallest element first)

  `void sort(Ran first, Ran last);`

- ❑ Like `sort()` but preserves the relative order between equivalent elements

  `void stable_sort(Ran first, Ran last);`

- ❑ Put smallest `middle-first` elements into [`first`, `middle`) in sorted order

  `void partial_sort(Ran first, Ran middle, Ran last);`

- ❑ Put smallest `rlast-rfirst` elements of [`first`, `last`) into [`rfirst`, `rfirst+N`) in sorted order; `N` is minimum of `last-first` and `rlast-rfirst`

  `Ran partial_sort_copy(In first, In last, Ran rfirst, Ran rlast);`

- ❑ “Sort” sequence so that element pointed to by `nth` is at correct place
  - ➠ elements in [`first`, `nth`) are smaller than the elements in [`nth`, `last`)
  - ➠ good for calculating medians or other quantiles

  `void nth_element(Ran first, Ran nth, Ran last);`

➠ All functions also available with additional parameter specifying sorting criterion `cmp`

Binary Search

- Determine whether `value` is in sequence [`first`, `last`) using binary search

```
bool binary_search(For first, For last, const T& value);
```

- Returns first / last position where element equal to `value` could be inserted without destroying ordering

```
For lower_bound(For first, For last, const T& value);
For upper_bound(For first, For last, const T& value);
```

- Return `pair<lower_bound, upper_bound>`

```
pair<For, For> equal_range(For first, For last, const T& value);
```

Merging Two Sorted Ranges

- Combine sorted ranges [`first1`, `last1`) and [`first2`, `last2`) into `res`

```
Out merge(In first1, In last1, In first2, In last2, Out res);
```

- Combine the two consecutive sorted ranges [`first`, `middle`) and [`middle`, `last`) in-place

```
void inplace_merge(Bi first, Bi middle, Bi last);
```

➠ All functions on page also available with additional parameter specifying sorting criterion `cmp`

Set Operations

- Is every element in [`first1`, `last1`) included in range [`first2`, `last2`)?

```
bool includes(In first1, In last1, In first2, In last2);
```

- Determine set-like union, intersection, difference, and symmetric difference of the ranges [`first1`, `last1`) and [`first2`, `last2`) and write to `res`

```
Out set_union(In first1, In last1,
              In first2, In last2, Out res);

Out set_intersection(In first1, In last1,
                     In first2, In last2, Out res);

Out set_difference(In first1, In last1, In first2,
                   In last2, Out res);

Out set_symmetric_difference(In first1, In last1,
                             In first2, In last2, Out res);
```

➠ Input ranges need only be *sorted*, they must be no real *sets* (duplicate elements allowed)

➠ All functions also available with additional parameter specifying sorting criterion `cmp`

Heap Operations

➠ *heap* (here) is a tree represented as a sequential range where each node is less than or equal to its parent node ➠ `*first` is largest element

- ❑ Turn range [`first`, `last`) into heap order

```
void make_heap(Ran first, Ran last);
```

- ❑ Add element at `last-1` to heap [`first`, `last-1`)

```
void push_heap(Ran first, Ran last);
```

- ❑ "Remove" (move to the end) largest element

```
void pop_heap(Ran first, Ran last);
```

- ❑ Turn heap into sorted range

```
void sort_heap(Ran first, Ran last);
```

➠ All functions also available with additional parameter specifying sorting criterion `cmp`

- ❑ Calculate the sum of `initval` plus all the elements in [`first`, `last`) using `T::operator+()` or binary function `op`

```
T accumulate(In first, In last, T initval);
T accumulate(In first, In last, T initval, BinOp op);
```

- ❑ Calculate the sum of `initval` plus the results of `first1[i]*first2[i]` for the range [`first1`, `last1`) using `operator+()`, `operator*()` or binary functions `op1`, `op2`

```
T inner_product(Input1 first1, Input1 last1,
                Input2 first2, T initval);
T inner_product(Input1 first1, Input1 last1,
                Input2 first2, T initval, BinOp op1, BinOp op2);
```

- ❑ Calculate running sum of all elements (or running result of `op()`)

```
Out partial_sum(In first, In last, Out result);
Out partial_sum(In first, In last, Out result, BinOp op);
```

- ❑ Calculate differences $x_i - x_{i-1}$ for elements in [`first+1`, `last`) (or use `op()` instead of `-`)

```
Out adjacent_difference(In first, In last, Out result);
Out adjacent_difference(In first, In last, Out result, BinOp op);
```

➠ These functions are defined in header `<numeric>`

- `block`: minimal container which adheres to STL container convention (like built-in array)

```
#include <stddef.h>

template <class T, size_t N>
struct block {
   T data[N];                          // public data

   // -- public types --
   typedef T value_type;
   typedef T& reference;
   typedef const T& const_reference;
   typedef ptrdiff_t difference_type;
   typedef size_t size_type;

   // -- iterators --
   typedef T* iterator;
   typedef const T* const_iterator;

   iterator begin() { return data; }
   const_iterator begin() const { return data; }

   iterator end() { return data+N; }
   const_iterator end() const { return data+N; }
```



```
   // -- member access --
   reference operator[](int i) { return data[i]; }
   const_reference operator[](int i) const { return data[i]; }

   // -- other member functions --
   size_type size() const { return N; }
   size_type max_size() const { return N; }
   bool empty() const { return N==0; }
   void swap(block& x) {
      for (size_t n = 0; n < N; ++n)
         std::swap(data[n], x.data[n]);
   }

   operator T*() { return data; }   // provided for compatibility
                                    // with built-in arrays
};
```

- Because `block` is defined as a POD (Plain Old Data) type, initialization syntax still works!

```
block<int, 6> x = { 1, 4, 7, 3, 8, 4 };
```

- We still need to define global comparison operators `operator==()` and `operator<()`

```
template <class T, size_t N>
bool operator==(const block<T,N>& lhs, const block<T,N>& rhs) {
   for (size_t n = 0; n < N; ++n)
      if (lhs.data[n] != rhs.data[n])
         return false;
   return true;
}

template <class T, size_t N>
bool operator<(const block<T,N>& lhs, const block<T,N>& rhs) {
   for (size_t n = 0; n < N; ++n)
      if (lhs.data[n] < rhs.data[n])
         return true;
      else if (rhs.data[n] < lhs.data[n])
         return false;
   return false;
}
```

- STL contains *algorithm adaptors* which define `operator!=()`, `operator<=()`, `operator>()`, and `operator>=()` out of the two operators above

- If we want `block` to be a *reversible* container, we need to add also

```
#include <iterator>

template <class T, size_t N>
struct block {
   // ...
   typedef std::reverse_iterator<iterator>        reverse_iterator;
   typedef std::reverse_iterator<const_iterator>
                                                const_reverse_iterator;

   reverse_iterator rbegin() { return reverse_iterator(end()); }
   const_reverse_iterator rbegin() const {
      return const_reverse_iterator(end());
   }

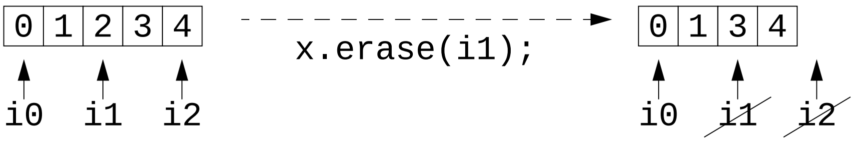
   reverse_iterator rend() { return reverse_iterator(begin()); }
   const_reverse_iterator rend() const {
      return const_reverse_iterator(begin());
   }

   // ...
};
```

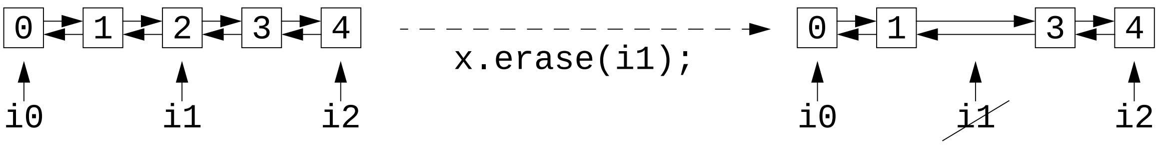
Advanced STL: Sequence Containers   Invalidating Iterators/Pointers

- `vector<T>` and `deque<T>` member functions which *modify* the container (`insert`, `erase`) ***can*** invalidate iterators and pointers into the container
- *invalidated* iterator `:=` the iterator points to another element or becomes invalid
- This is not true for `list<T>` with the exception of a iterator/pointer to a deleted element
- Example:

```
vector<int> x(5);
vector<int>::iterator i0=x.begin(), i1=x.begin()+2,
                      i2=x.begin()+4;
```




```
list<int> x(5);
list<int>::iterator i0=x.begin(), i1=i0, i2;
++i1; ++i1; i2=i1; ++i2; ++i2;
```



Advanced STL: `vector<T>`   Resizing Behavior

- When adding elements to a full `vector<T>`, automatic *resizing* occurs
- How this resizing occurs is implementation dependent
- ➠ resizing vector one by one element would be horribly inefficient
- ➠ usual methods are to double the size of the vector or to add fixed-sized blocks and copy over old elements
- ➠ `vector<T>` has two counts associated with it
  - its *size* `:=` the number of elements currently stored in the vector
  - its *capacity* `:=` the number of elements which could be stored without resizing
  - ➠ always: size <= capacity
- The following member functions related to resizing are available:
  - Get the size of a vector

    ```
    vector<T>::size_type size();
    ```
  - Get the capacity of a vector

    ```
    vector<T>::size_type capacity();
    ```

Advanced STL: vector<T> — Resizing Behavior

- Resize the vector to be of length n

  ```
  resize(size_type n, T initValue = T());
  ```

  - ➠ if n < size() elements at the end are removed
  - ➠ if n > size() additional elements initialized with initValue are added to the end

- Ensure that the *capacity* is at least n

  ```
  reserve(size_type n);
  ```

  - ➠ can be used to pre-allocate vector memory if approximate size if known in advance
  - ➠ saves memory allocations and copies!

- Example

```
const int n = 9000000;
vector<float> numbers;
numbers.reserve(n);
for (int i = 0; i < n; ++i) number.push_back(random());
```

➠ without reserve, several dozens of resizes would occur. With the last resize, vector would probably allocate a 64Mbyte buffer, copy over the 32Mbyte old values, and 29Mbyte would be wasted.

Advanced STL: list<T> — List Reordering: Splicing

- A list<T> has the advantage that it can be reordered by changing links

➠ list reorder functions don’t *copy* elements like insert() rather they modify the list data structures that refer to the elements

- The following reorder member functions are provided:

  - Move contents of list2 into list1 just before iter, leave list2 empty

    ```
    list<T> list1, list2;
    list<T>::iterator iter;            // points into list1
    list1.splice(iter, list2);
    ```

  - Move element pointed to by iter2 from list2 into list1 just before iter.
    Element is removed from list2. list1 and list2 may be the same list

    ```
    list<T>::iterator iter2;           // points into list2
    list1.splice(iter, list2, iter2);
    ```

  - Move the range [i2, j2) from list2 into list1 just before iter.
    The range is removed from list2

    ```
    list<T>::iterator i2, j2;          // point into list2
    list1.splice(iter, list2, i2, j2);
    ```

- Combine two *sorted* lists by moving elements from `list2` into `list1` while preserving order.

  ```
  list1.merge(list2);
  ```

  ➠ if one of the list is **not** sorted, `merge` still produces one list (however unsorted)

- There are also `sort` and `merge` with a second argument specifying the ordering criterion *cmp*

  ```
  list1.sort(cmp);
  list1.merge(list2, cmp);
  ```

- (Really) remove duplicates that appear consecutively or
  elements that appear consecutively and both satisfy the predicate *`pred`*

  ```
  list1.unique();
  list1.unique(pred);
  ```

- (Really) remove all elements with the value *`val`* or
  that satisfy predicate *`pred`*

  ```
  list1.remove(val);
  list1.remove_if(pred);
  ```

- Reverse all elements in the list

  ```
  list1.reverse();
  ```


- C-style built-in arrays, `string`s, `valarray`s, and `bitset`s are also containers
  - ➠ however, each lacks some aspect or the other of the STL standard containers
  - ➠ these “almost” containers are not completely interchangeable with STL containers

- Built-in arrays
  - supplies subscripting and random-access iterators in the form of ordinary pointers
  - provides no public types and doesn’t know its size (like ➠ `block<T>`)
- `std::string` defined in `<string>`
  - provides subscripting, random-access iterators, and most of STL conventions
  - but implementation is optimized for use as a string of characters
- `std::bitset<N>` defined in `<bitset>`
  - like a `vector<bool>` but provides operations to manipulate bits, is of fixed size `N`, and is optimized for space
- `std::valarray` defined in `<valarray>`
  - is a (badly defined) vector for optimized numeric computation

- STL library also provides some predefined predicates and functors in header `<functional>`
  - ➠ unnecessary to invent tiny functions just to implement trivial function objects
- The following basic predicates and arithmetic functors are defined:

Predicate	#Args	Op
`equal_to`	2	`==`
`not_equal_to`	2	`!=`
`greater`	2	`>`
`less`	2	`<`
`greater_equal`	2	`>=`
`less_equal`	2	`<=`
`logical_and`	2	`&&`
`logical_or`	2	`||`
`logical_not`	1	`!`

Arith.Func.	#Args	Op
`plus`	2	`+`
`minus`	2	`-`
`multiplies`	2	`*`
`divides`	2	`/`
`modulus`	2	`%`
`negate`	1	`-`

- These basic predicates and functors are all templates with one parameter specifying the base type
  - ➠ `greater<foo>` is a functor behaving like `foo::operator>()`

- more complex functors can be composed out of trivial ones with the help of *functor adaptors*
- The following adaptors are provided by the STL:

Function	#Args	Action of Generated Functor
`bind2nd(f,y)`	1	Call binary STL functor `f` with `y` as 2nd argument
`bind1st(f,x)`	1	Call binary STL functor `f` with `x` as 1st argument
`not1(f)`	1	Negate unary predicate `f`
`not2(f)`	2	Negate binary predicate `f`
`mem_fun(f)`	0 or 1	Transform 0- or 1-argument member function `f` into functor (call through pointer)
`mem_fun_ref(f)`	0 or 1	Transform 0- or 1-argument member function `f` into functor (call through reference)
`ptr_fun(f)`	1 or 2	Transform pointer to unary or binary function `f` into functor

➠ adapters are simple forms of a *higher-order function* `:=` takes a function argument and produces a new function from it

- Example: write function that deletes all numbers smaller and including 1000 in a sorted vector
- First, we could write our usual *predicate* function:

```
bool bigger1000(int n) { return n > 1000; }
```

- Then, we use STL algorithms, to *find* and *erase* elements in the vector:

```
void g1(vector<int>& numbers) {
   vector<int>::iterator vi =
         find_if(numbers.begin(), numbers.end(), bigger1000);
   numbers.erase(numbers.begin(), vi);
}
```

- Can even do away with the local variable `vi`

```
void g2(vector<int>& numbers) {
   numbers.erase(numbers.begin(),
         find_if(numbers.begin(), numbers.end(), bigger1000));
}
```


Advanced STL — Example: Predefined Functors

- Use binary functor "*greater*" `greater<T>`

```
greater<int> gt;              // gt(3,4) => false
```

- Need adapter functor "*apply with 2nd argument bound to value*" `bind2nd()` to fix value 1000

```
(bind2nd(gt,1000)) (999)      // => false
(bind2nd(gt,1000)) (1001)     // => true
```

- Now we can replace `bigger1000` by predicate function:

```
void g3(vector<int>& numbers) {
   greater<int> gt;
   numbers.erase(numbers.begin(),
      find_if(numbers.begin(), numbers.end(), bind2nd(gt,1000)));
}
```

- Of course, default constructor call `greater<int>()` can replace local variable `gt`:

```
void g4(vector<int>& numbers) {
   numbers.erase(numbers.begin(),
         find_if(numbers.begin(), numbers.end(),
               bind2nd(greater<int>(), 1000)));
}
```

- Read persons from standard input and store them sorted by first name in a list
  - First, define appropriate comparison function

```
bool cmpPtrFirstName(const Person* p1, const Person* p2) {
   return p1->firstName < p2->firstName;
}
```

  - Next, use it to find the right place to insert the new person in the list

```
list<Person*> folks;
string name, fname;

while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   list<Person*>::iterator it = folks.begin();
   while ( it != folks.end() ) {
      if ( cmpPtrFirstName(p, *it) ) break;
      ++it;
   }
   folks.insert(it, p);
}
```

- How about using STL algorithms (and predefined functors)?

- Use STL `find_if` algorithm and `bind1st` and `ptr_fun` functor adaptors to find the right place in the list

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   list<Person*>::iterator it
      = find_if(folks.begin(), folks.end(),
                bind1st(ptr_fun(cmpPtrFirstName), p));
   folks.insert(it, p);
}
```

- Can even do without local variable `it`

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   folks.insert(find_if(folks.begin(), folks.end(),
                bind1st(ptr_fun(cmpPtrFirstName), p)), p);
}
```

- Is it possible to avoid the need for functor adaptor `ptr_fun`?

- Write STL compatible functor by deriving from `binary_function<arg1_type, arg2_type, ret_type>` helper class:

```
struct cmpPtrFirstNameFtor :
   public binary_function<const Person*, const Person*, bool> {

   bool operator()(const Person* p1, const Person* p2) const {
      return p1->firstName < p2->firstName;
   }
};
```

- Then:

```
while ( cin >> fname >> name ) {
   Person* p = new Person(fname, name);
   folks.insert(find_if(folks.begin(), folks.end(),
                bind1st(cmpPtrFirstNameFtor(), p)), p);
}
```

- Can we do without functor adaptor `bind1st`?

- Implement unary functor with state (remembering the person to compare to):

```
struct cmpPtrWithFirstName :
   public unary_function<const Person*, bool> {

   cmpPtrWithFirstName(const Person *pers) : ref(pers) {}

   bool operator()(const Person *other) const {
      return ref->firstName < other->firstName;
   }

   const Person* ref;
};
```

- Usage:

```
folks.insert(find_if(folks.begin(), folks.end(),
             cmpPtrWithFirstName(p)), p);
```

- Note! Probably more efficient to simply insert persons into the list and sort them once:

```
while ( cin >> fname >> name )
   folks.push_back(new Person(fname, name));
folks.sort(cmpPtrFirstName);
```

The STL header `<iterator>` provides

- iterator *primitives*: utility classes and functions to simplify the task of defining new iterators
- iterator *operations*:
  - Increments (or decrements for negative `n`) iterator `i` by `n`

    ```
    void advance(InputIter& i, Distance n);
    ```
  - Returns the number of increments or decrements needed to get from `first` to `last`

    ```
    difference_type distance(InputIter first, InputIter last);
    ```
- *predefined* iterators:
  - Insert iterators ➠ assignment to iterator inserts value in container

    ```
    back_inserter(Container& x);
    front_inserter(Container& x);
    inserter(Container& x, Iterator i);
    ```
  - Stream iterators ➠ stepping iterator means reading/writing values from/to stream

    ```
    istream_iterator(istream& s);
    ostream_iterator(ostream& s, const char* delim);
    ```
  - Reverse iterators (see `block<T>` example)

- `ostream_iterator`s useful for printing container (`cont<T> C`) if `operator<<` for type of container element `T` exists. Instead of

  ```
  for (cont<T>::iterator it=C.begin(); it!=C.end(); ++it)
     cout << *it << "\n";
  ```

  you can use

  ```
  copy(C.begin(), C.end(), ostream_iterator<T>(cout, "\n"));
  ```

- Also useful for directly printing results of mutating sequence algorithms:

  ```
  inline int Square(int z) { return z*z; }

  block<int,6> x = { 1, 2, 3, 4, 5, 6 };
  ostream_iterator<int> ot(cout, " ");

  transform(x.rbegin(), x.rend(), ot, Square);
  cout << "--- ";
  transform(x.begin(), x.end(), x.rbegin(), ot, multiplies<int>());
  cout << endl;
  ```

  Output:

  ```
  36 25 16 9 4 1 --- 6 10 12 12 10 6
  ```

- Result range of mutating sequence algorithms (e.g., `transform`) must have enough elements to store result (because the elements are ***assigned*** the result values, they are not ***inserted***)

➠ Make sure result container is large enough:

```
block<int,6> x1 = { 1, 2, 3, 4, 5, 6 };
vector<int> res1(x1.size());
```

```
copy(x1.begin(), x1.end(), res1.begin());     // OK!
```

➠ res1: {1, 2, 3, 4, 5, 6}

```
copy(x.begin(), x.end(), res1.rbegin());      // reversed!
```

➠ res1: {6, 5, 4, 3, 2, 1}

➠ Or use iterator adaptors:

```
block<int,6> x2 = { 1, 2, 3, 4, 5, 6 };
list<int> res2;
```

```
copy(x2.begin(), x2.end(), back_inserter(res2));  // OK!
```

➠ res2: {1, 2, 3, 4, 5, 6}

```
copy(x2.begin(), x2.end(), front_inserter(res2)); // reversed!
```

➠ res2: {6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6}


```
#include <fstream.h>         // get the I/O facilities
#include <vector.h>          // get the vector facilities
#include <algorithm.h>       // get the operations on containers
#include <iterator.h>        // get the iterator facilities

int main (int argc, char *argv[]) {
   ifstream ii(argv[1]);  // setup input
   ofstream oo(argv[2]);  // setup output
   vector<int> buf;       // vector used for buffering

   // initialize buffer from input
   copy(istream_iterator<int>(ii),
        istream_iterator<int>(),     // def ctor => EOF iterator
        back_inserter(buf));

   // sort the buffer
   sort(buf.begin(), buf.end());

   // copy to output
   copy(buf.begin(), buf.end(), ostream_iterator<int>(oo,"\n"));

   return !(ii.eof() && oo);
}
```

- STL Containers always contain ***copies*** of objects
  - ➠ adding objects to or getting objects from a container means copying
  - ➠ moving objects in a sequence container (because of `insert` or `erase`) means copying
  - ➠ changing order in a sequence container (because of `sort`, `reverse`, ...) means copying

➠ make sure copying objects works correctly
- ❍ pointer members ➠ deep copy?!
- ❍ derived objects ➠ slicing!

- for “heavy” objects or in the presence of inheritance:

➠ use containers of pointers to objects

➠ better: use *smart pointers* for automatic memory management


```
class P {
public:
   P(const string& _n) : n(_n), i(1) { print(cout, "Construct"); }

   P(const P& p) : n(p.n), i(p.i+1) { print(cout, "Copy"); }

   P& operator=(const P& p) {
      print(cout, "Delete");
      n = p.n;  i = p.i+1;
      print(cout, "Copy Assign");
      return *this;
   }

   ~P() { print(cout, "Delete"); }

   ostream& print(ostream& ostr, const char* action) const {
      return ostr << action << "(" << n << i << ")\n";
   }

private:
   string n;     // My name is "n"
   int i;        // I am the i-th copy
};
```

```
ostream& operator<<(
  ostream& ostr,
  const P& p) {
  return p.print(ostr, "Out");
}

vector<P> vec1, vec2;

vec1.push_back( P("Bob") );
vec1.push_back( P("Joe") );
vec1.push_back( P("Sue") );

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<P>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<P>(cout));
```

Possible output (depends on resize behavior):

```
Construct(Bob1)
Copy(Bob2)        //passing by value
Delete(Bob1)      //-
Construct(Joe1)
Copy(Joe2)        //passing by value
Delete(Joe1)      //-
Construct(Sue1)
Copy(Bob3)        //vector resize
Copy(Joe3)        //vector resize
Copy(Sue2)        //passing by value
Delete(Bob2)      //-
Delete(Joe2)      //-
Delete(Sue1)      //-
Copy(Bob4)
Delete(Bob4)
Copy Assign(Sue3)
Out(Bob3)
Out(Joe3)
Out(Sue2)
Out(Sue3)
Delete(Sue3)
Delete(Bob3)
Delete(Joe3)
Delete(Sue2)
```



```
ostream& operator<<(
  ostream& ostr,
  const P* p) {
  return p->print(ostr, "Out");
}

vector<P*> vec1, vec2;

vec1.push_back( new P("Bob") );
vec1.push_back( new P("Joe") );
vec1.push_back( new P("Sue") );

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<P*>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<P*>(cout));
```

- Output:

```
Construct(Bob1)
Construct(Joe1)
Construct(Sue1)
Out(Bob1)
Out(Joe1)
Out(Sue1)
Out(Sue1)

//no Delete?!
```

➠ Can do manual `delete` but might get complicated in larger program!

➠ Use smart pointer!

```
typedef shared_ptr<P> PPtr;

ostream& operator<<(
  ostream& ostr,
  const PPtr& p) {
  return p->print(ostr, "Out");
}

vector<PPtr> vec1, vec2;

vec1.push_back(PPtr(new P("Bob")));
vec1.push_back(PPtr(new P("Joe")));
vec1.push_back(PPtr(new P("Sue")));

vec2.push_back( vec1[0] );
vec2[0] = vec1[2];


copy(vec1.begin(), vec1.end(),
  ostream_iterator<PPtr>(cout));

copy(vec2.begin(), vec2.end(),
  ostream_iterator<PPtr>(cout));
```

- Output:

```
Construct(Bob1)
Construct(Joe1)
Construct(Sue1)
Out(Bob1)
Out(Joe1)
Out(Sue1)
Out(Sue1)
Delete(Bob1)
Delete(Joe1)
Delete(Sue1)
```

- Note! `Sue1` gets (correctly) deleted just once!

- Idea: Implement pointer-like class which automatically deletes content if necessary
- Example: Shared ownership based on reference counting

```
template<typename T>
class shared_ptr {

private:
   T*     px;     // contained pointer
   long*  pn;     // pointer to reference counter

public:
   explicit shared_ptr(T* p = 0) : px(p), pn(new long(1)) {}

   ~shared_ptr() { if (--*pn == 0) { delete px; delete pn; } }

   shared_ptr(const shared_ptr& r) : px(r.px) { ++*(pn = r.pn); }

   shared_ptr& operator=(const shared_ptr& r);

   T& operator*() const  { return *px; }
   T* operator->() const { return px; }
   T* get() const        { return px; }
   bool unique() const   { return *pn == 1; }
};
```

```
template<typename T>
shared_ptr<T>& shared_ptr<T>::operator=(const shared_ptr<T>& r) {
   if (pn != r.pn) {
      // increment new reference counter
      ++*(r.pn);
      // decrement old reference counter, delete if last
      if (--*pn == 0) { delete px; delete pn; }
      // copy pointer and counter
      px = r.px;
      pn = r.pn;
   }
   return *this;
}
```

- Example smart pointer too simple for professional use
  - other ownership models (no copy allowed, copy transfers ownership, shared, ...)
  - const and inheritance issues
  - multi-threading?

➠ better: see smart pointer collection at `boost.org`

- To eliminate all objects in a container `C` that have a particular value `val`:
  - `vector`, `deque`, `string`: use `erase/remove` idiom

    `C.erase(remove(C.begin(), C.end(), val), C.end());`
  - `list`: use `list` method `remove(val)`
  - associative container: use container method `erase(key)`

- To eliminate all objects in a container that satisfy a particular predicate `pred`:
  - `vector`, `deque`, `string`: use `erase/remove_if` idiom
  - `list`: use `list` method `remove_if(pred)`
  - associative container: use `remove_copy_if` and then `swap` elements

```
AssocCtr C, OK;

remove_copy_if(C.begin(), C.end(),
               inserter(OK, OK.end()), pred);
C.swap(OK);
```

- ❑ Which method should be used to find objects in STL containers?
- ➠ the faster and simpler, the better!
  - ❍ unsorted containers: can only use linear-time algorithms `find`, `find_if`, `count`, and `count_if`
  - ❍ sorted sequence containers: can use faster `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` ***algorithms***
  - ❍ associative containers: can use faster `binary_search`, `lower_bound`, `upper_bound`, and `equal_range` ***methods***
- ➠ Note! the linear-time algorithms use equality to test whether two objects are the same, the others equivalence!

  equality ⇔ `x1 == x2`

  equivalence ⇔ `! ( x1 < x2 ) && ! ( x2 < x1 )`

- ❑ Summary:

What You Want to Know	Algorithm to Use		Member Function to Use	
	Unsorted Range	Sorted Range	`set` or `map`	`multiset` or `multimap`
Does desired value exist?	`find`	`binary_search`	`count`	`find`
Where is first object of desired value?	`find`	`equal_range`	`find`	`find` or `lower_bound`
Where is first object with a value not preceding desired value?	`find_if`	`lower_bound`	`lower_bound`	`lower_bound`
Where is first object with a value succeeding desired value?	`find_if`	`upper_bound`	`upper_bound`	`upper_bound`
How many objects have desired value?	`count`	`equal_range`	`count`	`count`
Where are all objects with desired value?	`find` (iteratively)	`equal_range`	`equal_range`	`equal_range`

- Use `vector<T>` instead of built-in arrays and `string` instead of `char*`!
- What about calling old code?

- Consider, we need to call an old C function

```
void doSomething(const int *pInts, size_t numInts);
```

but we have

```
vector<int> v;
```

➠ pass address of first element and size (But don't forget empty vectors!)

```
if ( !v.empty() ) { doSomething(&v[0], v.size()); }
```

- What about

```
void doSomething(const char *pString);
string s;
```

➠ Use string method `c_str()`

```
doSomething(s.c_str());
```


Books on STL

- Austern, *Generic Programming and the STL*,
  Addison-Wesley, 1998, ISBN 0-201-30956-4.
  - ➠ Most up-to-date and complete book on STL, good tutorial and reference
- Josuttis, *The C++ Standard Library – A Tutorial and Reference*
  Addison-Wesley, 1999, ISBN 0-201-37926-0.
  - ➠ Most up-to-date and complete book on **whole** C++ standard library
    (including iostream, string, complex, ...)
- Meyers, *Effective STL*,
  Addison-Wesley, 2001, 0-201-74962-9.
  - ➠ 50 specific ways to improve your use of the standard template library

General C++ Books (but cover STL very well)

- Stroustrup, *The C++ Programming Language*, **Third Edition**
- Lippman and Lajoie, *C++ Primer*, **Third Edition**

STL WWW

WWW STL Information

- SGI STL (public-domain implementation plus well-organized on-line documentation)
  http://www.sgi.com/tech/stl/
- Adapted SGI STL (value-added and more portable version of SGI STL)
  http://www.stlport.org
- BOOST: free, peer-reviewed, portable C++ libraries
  http://www.boost.org

STL Portability

- Watch out for differences between original public domain STL from HP and the C++ standard:

	HP (SGI) STL	**C++ Standard STL**
container adaptors stack, queue, ...	Adaptor<Container<T> >	Adaptor<T> Adaptor<T, Container<T> >
iterator access	(*iter).field	iter->field or (*iter).field
scope of items	global scope	in namespace std
headers	<algo.h> <function.h> <stack.h> <map.h>, <multimap.h> <set.h>, <multiset.h>	<algorithm>, <numeric> <functional> <stack>, <queue> <map> <set>
* functor	times<T>	multiplies<T>
algorithms: count, count_if, distance	int n = 0; func(..., n);	int n; n = func(...);
I/O iterators (ostream analog)	istream_iterator <T,Distance>	istream_iterator <T,charT,traits,Distance>

Programming in C++

☆☆☆ Advanced C++ ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de
Forschungszentrum Jülich
Germany

- Private inheritance allows to implement a class in terms of another one:
  - ➠ e.g. try to implement `Set` as `List` without `head`/`tail` and new `insert` through *derivation with (compile-time) cancellation* (**but**: not allowed in ARM C++!!!)

```
class List {
public:
   elem* head();
   elem* tail();
   int size();
   bool has(elem&);
   void insert(elem&);
};
```

```
class Set: public List {
public:
   void insert(elem&);
private:
   List::head;  // error! in ARM C++
   List::tail;  // error! in ARM C++
};
```

- However, it works the other way:

```
class Set: private List {
public:
   void insert(elem&);
   List::size;
   List::has;
};
```

- In new C++ Standard:
  - derivation with cancellation now allowed
  - "*using declaration*" should be used instead of access specifiers:

```
using List::size;
using List::has;
```

- But reuse through private inheritance has a severe limitation:
  - ➠ the following is illegal because no class may appear as a direct base class twice

```
class Arm { ... };
class Leg { ... };

class Robot: Arm, Arm, Leg, Leg {    // illegal!
private:
   // Robot specific members
public:
   Robot();
};
```

- Solution: use *class members* or *layering* (nested classes)

```
class Robot {
private:
   Arm leftarm, rightarm;
   Leg leftleg, rightleg;
   ...
```

➠ **avoid private inheritance if possible**

- Sometimes, the design of a class requires an auxiliary class
  - ➠ Solution: nested classes

```
class Stack {
private:
   class StackNode {
      T data;
      StackNode *next;
      StackNode( const T& d,
                 StackNode *n)
         : data(d), next(n) {}
   };
   StackNode *top;

public:
   Stack();
   ~Stack();
   void push(const T& data);
   T pop();
};
```

- ➠ if compiler doesn't support nested classes use private global class with friend

```
class StackNode {
private:
   T data;
   StackNode *next;
   StackNode( const T& d,
              StackNode *n)
        : data(d), next(n) {}

   friend class Stack;
};

class Stack {
public:
   Stack();
   ~Stack();
   // ...
};
```

❑ Nested classes can now be forward declared just like other classes:

```
class Outer {
  class Inner;
public:
  Inner* getCookie();
private:
  class Inner { /* ... */ };
};
```

❑ The definition can now also be outside the class:

```
class Outer {
  class Inner;
public:
  Inner* getCookie();
};

class Outer::Inner { /* ... */ };
```


❑ C programmers are already familiar with the C style cast syntax:

```
(Type)Expression              // double d = (double) 5;
```

❑ C++ introduced the functional style cast syntax

```
Type(Expression)              // double d = double(5);
```

❑ In specific circumstances, casts are useful (and necessary)

➠ But "old" style casts convert everything in everything (compiler believes you)

❑ 4 new casts added to C++ to allow finer control over casting

➠ new syntax `xxx_cast<type>(expr)` is easier to locate for tools/programmer

➠ it is harder to type (many believe: a good thing)

➠ makes clear what kinds of casts are "meaningful" (useful)

- `comst` / `volatile` conversion
  - `const_cast`
  - allows only to change the "constness" (of pointer and references) in a conversion:

```
void foo(double&);
const double PI = 3.1415;
foo(const_cast<double&>(PI));        // OK
int d = const_cast<int&>(PI);        // error: more than const!
```

- compile–time checked conversion
  - `static_cast`
  - allows to perform any conversion where there is an implicit conversion in the opposite direction (most frequent use of casts in C):

```
int total = 500, days = 9;
double rate = static_cast<double>(total) / days;

enum color {red=0, green, blue}; int i = 2;
color c = static_cast<color>(i);   // c = blue;

foo(static_cast<int>(PI));        // error: changes constness!
```


- run-time checked conversion
  - `dynamic_cast`
  - allows to perform *safe casts* down or across an inheritance hierarchy
  - Failed casts return null pointer (when casting pointers) or exception (with references)

```
base b, *pb;
derived d, *pd;

pb = &d;                                // fine

// ... later
pd = dynamic_cast<derived *>(pb);       // non-null only if pb
                                        // really points to
                                        // derived object

d = dynamic_cast<derived>(b);           // error: no pointer!
```

- unchecked conversion
  - `reinterpret_cast`
  - allows to perform any conversion
  - result is nearly always implementation-defined
  - non-portable!

```
typedef void(*FuncPtr)();
FuncPtr funcPtrArray[10];

int doSomething();

funcPtrArray[0] = reinterpret_cast<FuncPtr>(&doSomething);
```

- If your compiler doesn’t support the new style casts yet, you can use the following approximation

```
#define static_cast(TYPE,EXPR)       (TYPE)(EXPR)
#define reinterpret_cast(TYPE,EXPR)  (TYPE)(EXPR)
#define const_cast(TYPE,EXPR)        (TYPE)(EXPR)

// doesn’t tell you when the casts fails; use with care
#define dynamic_cast(TYPE,EXPR)      (TYPE)(EXPR)
```


- Current C++ class libraries often use their "home-brewed" dynamic type framework:

```
if (shape_ptr->myType() == RectangleType) {
  Rectangle *rec_ptr = (Rectangle *)shape_ptr;
  // ...
}
```

- Every major library came up with there own facility; therefore, a standardisation took place:

```
if (Rectangle *rec_ptr = dynamic_cast<Rectangle *>(shape_ptr)) {
  // rec_ptr in scope and not null
}
// rec_ptr no longer in scope
```

- There is also a type enquiry operator `typeid` returning a reference to a type description object of type `type_info` (it can be compared and has a name)

```
if (typeid(*shape_ptr) == typeid(Circle)) {
  // it’s a Circle pointer
}

const type_info& rt = typeid(*shape_ptr);
cout << "shape_ptr is a pointer to " << rt.name() << endl;
```

- Recall `operator[]` from `safeArray`:

```
#include <assert.h>
#include "safearray.h"

T& safeArray<T>::operator[](int index) {
   assert(index >= 0 && index < size());       // exit on failure
   return Array<T>::operator[](index);
}
```

- Printing error message and exit is sometimes too drastic

- error codes / error return values not intuitive and error-prone
  - ➠ sometimes error code cannot returned or error return value does not exist (see above)
  - ➠ can be ignored by caller
  - ➠ must be passed up calling hierarchy (in deeply nested function calls)
  - ➠ could use `setjmp` / `longjmp`; fine for C, but don't call destructors of local objects

➠ **Solution: use exception handling!**

- Exception Handling consists of three parts:
  - ❍ `try`ing to execute possibly failing code
  - ❍ `throw`ing an exception in case of failure
  - ❍ `catch`ing the exception and handling it (exception handler)

`try` Block:

```
try {
   // code from which exception may be thrown
}
```

- Region of program from which control can be transferred directly to an exception handler elsewhere in program
- Transferring control from try block to exception handler is called "*throwing an exception*"

`catch` Block:

- Syntax:

```
try {
   // code from which exception may be thrown

} catch ( exception-type-1 &e1 ) {
   // exception handling code for type 1 exceptions

} catch ( exception-type-2 &e2 ) {
   // exception handling code for type 2 exceptions

}
```

- Region of program that contains exception-handling code
- Each thrown exception has a type
- When exception is thrown, control transfers to first catch block in an (dynamically) enclosing block with matching type
- For efficiency, catch exceptions *by reference*
  - avoids call to copy constructor of exception object
  - avoids *slicing* (i.e. automatic conversion to base object) of exception object

`throw` Statement:

- Syntax:

```
throw exception-type;
```

- Causes immediate transfer of control to exception handler (`catch` block) of matching type
- *Copy* of exception object is thrown (automatically)
- Throwing an exception causes abnormal exit from a function
- Normal exit calls destructor for all local objects. For consistency, throwing an exception invokes destructors for all objects instantiated since entering `try` block
- Order of destructor invocation is reverse of order of object instantiation

```
void foo (int i) {
   String s1 = "a string";
   Complex c1(i,i);
   if (some_error) {
      throw SomeException;    // destructor called for c1, then s1
   }
   String s2 = "another string";
}
```

Default `catch` Blocks:

- ❑ Syntax:

```
catch (...) {
   // exception handling code for default case
}
```

- ❑ Matches *all* exception types
- ❑ Use at most one default `catch` block for each `try` block
- ❑ Default `catch` block (if any) should be the *last* `catch` block accompanying a given `try` block
- ❑ Good programming practice dictates that every `try` block should have a default `catch` block

Re-throwing Exceptions:

- ❑ allows some exceptions to be handled locally, others to be handled by high-level general purpose exception handlers
- ❑ Re-throw current exception by placing throw statement with no type specifier: `throw;`

Exception Hierarchies

- ❑ Exception types can be user-defined (class) types
- ❑ Exception hierarchy = hierarchy (build using C++ inheritance mechanism) of user-defined exception types
- ❑ Exception base class = "category" of exceptions
- ❑ Derived exception class = individual exception type
- ❑ catch block for base type catches all exceptions in category
- ❑ Example from Standard C++ Library:

```
class runtime_error : public exception {};
class underflow_error : public runtime_error {};
class range_error : public runtime_error {};
```

Standard Exceptions

❑ Standard exceptions thrown by the language:

Name	Thrown by
`bad_alloc`	`new`
`bad_cast`	`dynamic_cast`
`bad_typeid`	`typeid`
`bad_exception`	exception specification
`out_of_range`	`at`
	`bitset<>::operator[]`
`invalid_argument`	`bitset constructor`
`overflow_error`	`bitset<>::to_ulong`
`ios_base::failure`	`ios_base::clear`

➠ Standard exceptions do **NOT** include asynchronous events like UNIX signals (e.g., segmentation violation) or math library errors (e.g., overflow, div by zero)!

❑ Standard exceptions hierarchy:

- `exception`
  - `runtime_error`
    - `range_error`
    - `overflow_error`
    - `underflow_error`
  - `logic_error`
    - `length_error`
    - `domain_error`
    - `out_of_range`
    - `invalid_argument`
  - `bad_alloc`
  - `bad_cast`
  - `bad_exception`
  - `bad_typeid`
  - `ios_base::failure`

Advanced Exception Handling

Exception Specifications:

❑ allow to specify list of exceptions, functions might throw

```
void f1() throw(E1, E2);     // f1 can throw E1 or E2 exceptions
void f2() throw();           // f2 doesn’t throw exceptions
void f3();                   // exception spec for f3 unknown!
```

❍ is checked at runtime

❍ `unexpected()` is called if exception specification doesn’t hold

❍ which by default calls `terminate()` (can be changed with `set_unexpected()`)

Function try blocks:

❑ allow to catch exceptions in a method body and its member initialization list

```
derived::derived(int i) try : base(i) {
   // body ...
}
catch (...) {
   // exception handler for initializer list + body ...
}
```

- Improved version of class `safeArray`:
  - Exception defined in `safearray.h`:

```
class InvalidIndex {
public:
   InvalidIndex(int i) : idx(i) {}
   int invalid() { return idx; }
private:
   int idx;
};
```

    - ➠ should actually be a subclass of `exception`, `logic_error`, or `out_of_range`
    - ➠ should also be nested class inside `safeArray`

  - New version of `operator[]:`

```
T& safeArray<T>::operator[](int index) {
   if (index < 0 || index >= size()) throw InvalidIndex(index);
   return Array<T>::operator[](index);
}
```


- Simple example for exception handling:

```
#include <iostream.h>
#include "safearray.h"

void initialize (safeArray<int>& a, int val) {
   for (int i=0; i<=a.size(); ++i) a[i] = val;
}

int main(int, char**) {
   safeArray<int> a(10);

   try {
      initialize(a, 42);
   } catch (InvalidIndex& idx) {
      cerr << "Access to invalid index a["
           << idx.invalid() << "]" << endl;
   }
}
```

➠ **making programs fool-proof with exceptions is still hard; not much experience for now**

➠ For more details and guidelines see Stroustrup, *3rd. Edition* or Meyers, *More Effective C++*

- Compilation-unit local (file local) declarations possible through *unnamed namespaces*:

```
namespace {
   int internal_version;
   void check_filebuf();
}
```

is equivalent to

```
namespace SpecialUniqueName {
   int internal_version;
   void check_filebuf();
}
using namespace SpecialUniqueName;
```

➠ **use unnamed namespaces instead of** `static` **declarations**:

```
   // -- deprecated!
   static int internal_version;
   static void check_filebuf();
```


- A shorter alias for a long namespace name can also be defined, e.g.,

```
namespace Chrono = Chronological_Utilities;
```

This allows you to mix’n’match libraries more easily:

```
namespace lib = Modena;
//namespace lib = RogueWave;
//namespace lib = std;

// never need to change this:
lib::string my_name;
```

- Namespaces are not yet supported by many current compilers. You can partition the global namespace now by using `struct` and using fully qualified names to access them

```
struct MyNamespace {
   static const float version = 1.6;
};

cout << "Version " << MyNamespace::version << endl;
```

For convenience, you can provide typedefs, references, ... as short-cuts in a separate header file, so people without global namespace problems can use them in a "normal" way

Templates have been extended by the standard committee in many ways.
(beyond how they are described in most books)

- Template friend declarations and definitions are permitted in class definitions and class template definitions.
  - Example: class definition that declares a template class `Peer<T>` as a `friend`:

```
class Person {
private:
   float my_money;
   template<class T> friend class Peer;
};
```

  - Example: class template that declares a template class `foo<T>` as a `friend`:

```
template <class T>
class bar {
   T var;
   friend class foo<T>;
};
```


Advanced C++ Advanced Templates

- The new keyword `typename` can be used instead of `class` in template parameter lists

```
template<typename T> class List { /*...*/ };
```

It is also used for resolving ambiguities:

```
template<class T> class Problem {
public:
   void ack() {
      typename T::A * B;      // without typename, could be
   }                          // interpreted as multiplication
};                            // T::A * B;
```

- Templates and `static` members:

```
template <class T> class X {
   static T s;
};
```

```
template <class T> T X<T>::s = 0;
```

If needed, specialization:

```
template<> double X<double>::s = 1.0;
```

❑ Template as template arguments

```
template<class T> class List;
template<class T> class Vector;

template<class T, template<class U> class C = List>
class Group {
  C<T> container;
  //...
};

Group<int>          group_int_list;
Group<int, Vector>  group_int_vector;
```

❑ Explicit instantiation (currently highly unportable)

```
template<typename T> class List {
  bool has(T&) { /* ... */ }
};

template class List<int>;   // request inst. of List for T=int
template bool List<elem>::has(elem&);  // inst. of member 'has'
```


❑ Partial Specialization

```
template<class U, class V> class relation;  #1 primary template
template<> class relation<int, int>;        #2 specialization
template<class T> class relation<T*, T*>;   #3 partial special.

relation<char*, char*> r1;   # uses 3
relation<int*, char*> r2;    # uses 1 (different pointer types!)
```

Note: function templates cannot be partial specialized, however similar results can sometimes be achieved with template function overloading

❑ Use of `template` to disambiguate template names (can only be necessary for calling member template functions inside other templates)

```
class X {
public:  template<int Nt> X* alloc();
};

template<class T> void f(T* p) {
  T* p1 = p->alloc<200>();          // ERROR: < means less than
  T* p2 = p->template alloc<200>(); // OK
}
```

- Current compilers typically only support the *inclusion model* for template usage
  - ➠ Bodies of
    - ✰ member functions of class templates
    - ✰ function templates

    need to be "*included*" before they can be used
  - ➠ Cannot be "*compiled away*" in separate template implementation file (e.g., `.cpp`)

- New keyword `export` to support the so-called *separation model*:

```
// main.C:
template<typename T> T twice(T); // template declaration only

int main() {
   int i = 2;
   int j = twice(i);
}

// template_implementation.cpp:
export template<typename T> T twice(T t) { return t*t; }
```


- We know already static class member:
  - ❍ *Declaration*: in class definition (in `.h`)

    ```
    class Complex { .... static int num_numbers; ... };
    ```
  - ❍ *Definition*: needed elsewhere (in `.C`, ***not*** in `.h`)

    ```
    int Complex::num_numbers = 0;
    ```

- As the initialization occurs outside the class, this wouldn't allow class related constants
  - ➠ Initialization is now allowed for integral `static const` data members
  - ❍ *Definition* + *Declaration*s now in class definition (in `.h`)

    ```
    class Data {
       static const int max_size = 256;
       // enum {max_size = 256};  // use this if above doesn't work
       char buffer[max_size];
    };
    ```
  - ❍ *Definition* elsewhere (in `.C`) no longer needed

- *Abstract* constness of data members (`const` to the user/client), also called: conceptual constness
- *Concrete* constness of a data member (`const` to the implementation), also: bitwise constness

- Example: non-normalized `Rational` class

```
class Rational {
   void reduce() const;      // allowed for constant numbers
   ...                       // because it doesn't change
                             // "abstract" value
private:
   mutable long num;
   mutable long den;
};

void Rational::reduce() const {
   long g = gcd(num, den);
   num /= g;                          // ERROR without mutable
   den /= g;                          // ERROR without mutable
}
```


- Problem:

```
class Vector {
public:
   Vector(int len);   // constructs a Vector len elements long
};

void print(const vector& v);
print(3);    // creates temp Vector(3) and passes it to print!
```

➠ Obviously the constructors are needed but they are not always suitable as conversions as well

- Solution: mark constructor "non-converting" with `explicit` (disallows implicit conversions)

```
class Vector {
public:
   explicit Vector(int len);
};

print(3);    // now error!

Vector v1(3), v2 = Vector(3);         // OK
Vector v3 = 3;                        // error!
```

- Variables can be declared inside conditions (and for statements)

```
if (int d = prim(345)) {          while (Word& w = readnext()) {
   i /= d;                           process(w);
   break;                         }
}
```

- You can now overload functions on enumeration types

```
enum Status { OK, Busy, Stopped };
void process(Status s);
void process(int n);         // was error, now OK
```

- There are now separate versions of `new` and `delete` for array allocation for overloading

```
class Foo {
   void* operator new(size_t);        // already available
   void  operator delete(void *);
   void* operator new[](size_t);      // recently added to C++
   void  operator delete[](void *);
};
```

- pointer to `extern "C"` functions is now a different type than pointer to global C++ function

- Scope of declaration inside `for` loop changed
  - ➠ now only valid inside loop body
- Template name lookup and template instantiation procedures changed
- `new` now throws `bad_alloc` exception when out of memory

```
OLD:                  NEW1:                    NEW2:
                      #include <new>           #include <new>
                                               try {
X *p=new X;           X *p=new(nothrow)X;        X *p=new X;
if (p==NULL) {        if (p==NULL) {           } catch(bad_alloc &) {
   do_something();      do_something();          do_something();
}                     }                        }
```

- . . .

- Good summary:

  Jack W. Reeves
  (B)leading Edge: Moving to Standard C++
  C++ Report, Jul/Aug 1997

- Incompatibilities between Cfront's `iostream` and Standard `iostream`. Biggest changes are:
  - The Standard C++ library puts most library identifiers in the namespace `std`, e.g.,
    `ostream` is now `std::ostream`
    - ➠ import identifiers using `namespace` declarations
  - stream classes are now templates taking the character type as parameter
    `typedef basic_istream<char,char_traits<char> > istream;`
  - Base class ios is split into character type dependent and independent portions
    - ➠ `ios::`*`flags`* now are `ios_base::`*`flags`*
  - Can now throw exceptions in addition to setting error flag bits
  - Internationalization using locale
  - Assignment and copying of streams is prohibited
  - File descriptors (through member function `fd()`) are not supported any longer
  - `string` based `stringstream` class replaces `char*` based `strstream`

- `char *p = "a string";`
  - ➠ String literals are now `const char*`
- Postfix `operator++` on `bool` operand
  - ➠ Don’t use it
- `static` keyword to declare objects local to file scope
  - ➠ Use unnamed `namespace`
- access declaration
  - ➠ using declarations
- `strstream` class
  - ➠ Use `stringstream` class
- Standard C Library headers
  - ➠ use new C++ C Library headers

Programming in C++

☆☆☆ Object-Oriented Design ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Object-Oriented Design Motivation

❑ Ranking of software lifecycle activities

1.) Design

2.) Design

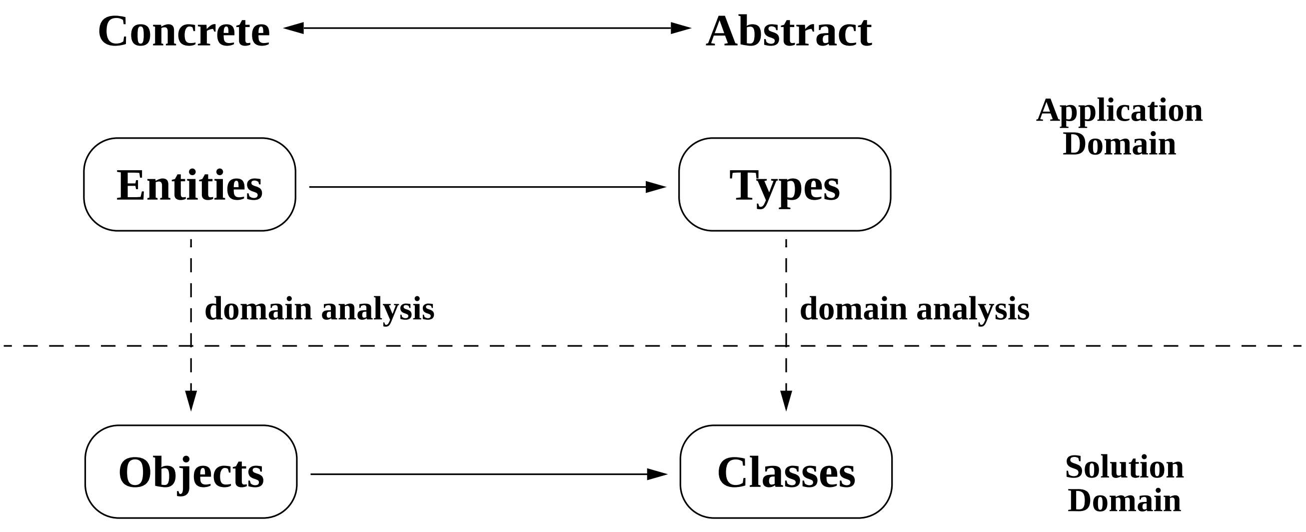
3.) Design

4.) Implementation

❑ "*The time-consuming thing to learn about C++ is not syntax, but design concepts.*" [Stroustroup]

❑ "*If the design is right, the implementation is trivial.*
If you get stuck how to implement something, go back to design."

- The most important single aspect of software development is to be clear about what you are trying to build
- Successful software development is a long-term activity
- The systems we construct tend to be at the limit of complexity that we and our tools can handle
- There are no "cookbook" methods that can replace intelligence, experience, and good taste in design and programming
- Experimentation is essential for all nontrivial software development
- Design and programming are iterative activities
- The different phases of a software project, such as design, programming, and testing, cannot be strictly separated
- Programming and design cannot be considered without also considering the management of these activities



- In addition,
  - type relationships can be modelled with inheritance (public or private)
  - entity relationships can be modelled with object hierarchies

- Identify the entities in your application domain
  - resources, events, (hardware) parts, software interfaces, **concepts**, ...
- Identify the behaviours of the entities
  - services, tasks, functionality, responsibilities, ...
- Identify relationships/dependencies between entities
  - *is-a, is-kind-of, is-like, has-a, is-part-of, uses-a, creates-a,* ...
- Broaden Design (**Guideline**: should be implementable in at least 2 different ways)
  - to be able to reuse later (design + source code!)
  - helps the system evolve / maintain
  - to be sure the implementation can fulfil its requirements
- Create a C++ design structure from the entities
  - define classes (and their interfaces!), objects, inheritance and object hierarchy, ...
- Implement, Fine-Tune, Test, Maintain

➠ **Note**: this is not a sequential process rather than a back-and-forth or cyclic one

Summary:

➠ Say what you mean

➠ understand what you are saying

- A common base class means common characteristics

- Public inheritance means *is-a* or *is-kind-of*

```
class derived : public base { /* ... */ };
```

  - every object of type `derived` is also an object of type `base`, but not vice-versa!
  - *Liskov substitution principle*:
    can every usage of an object of type `base` be replaced by an object of class `derived`?
  - *additional* functionality and/or data makes good subclass!

  Don't mistake *is-like-a* for *is-a* !

  - find higher abstraction, make parent: e.g. `set` is-like-a `list`, derive from `collection`

- Private inheritance means *is-implemented-in-terms-of*

```
class derived : private base { /* ... */ };
```

  - ➠ implementation issue; no design-level conceptional relationship
  - ➠ use only if access to protected members needed or to redefine virtual functions, otherwise use layering

- Layering (nested classes) means *has-a*, *is-part-of*, or *is-implemented-in-terms-of* between classes

```
class Inner { /* ... */ };
class Outer  { Inner I; /* ... */ };
```

- Class member(s) means *has-a*, *is-part-of*, or *is-implemented-in-terms-of* between objects

```
class foo { /* ... */ };
class bar1 { foo  I; /* ... */ };                    // 1-to-1 mapping
class barN { foo *I; /* ... */ };                    // 1-to-n mapping
```

- Member function(s) that take parameter of another class means *uses-a*

```
foo::func(const bar& b) { /* ... */ }
```


- *The Open/Closed Principle* (Bertrand Meyer): Software entities (classes, modules, etc) should be open for extension, but closed for modification.
  - ➠ New features can be added by *adding new code* rather than by *changing working code*
  - ➠ Thus, the working code is not exposed to breakage

- *The Liskov Substitution Principle*: Derived classes must be usable through the base class interface without the need for the user to know the difference.

- *Principle of Dependency Inversion*: Details should depend upon abstractions. Abstractions should not depend upon details.
  - ➠ All high level functions and data structures should be utterly independent of low level functions and data structures.

- Dependencies in the design must run in the direction of stability. The dependee must be more stable than the depender. (stability `:=` probable change rate)
  - ➠ The more stable a class hierarchy is, the more it must consist of abstract classes. A completely stable hierarchy should consist of nothing but abstract classes.
  - ➠ Executable code changes more often than the interfaces

- Use classes to represent concepts
- Keep things as private as possible
  - Once you publicize an aspect of your library (method, class, field), you can never take it out
- Watch out for the "giant object syndrome"
  - Objects represent concepts in your application, not the application itself!
  - Don’t add features "just in case"
- If you must do something ugly, at least localize the ugliness inside a class
- Don’t try technological fixes for sociological problems

- How should objects be created and destroyed?
  (constructors, destructor, class specific `new`/`delete`)
- How does object initialization differ from assignment? (constructors, `operator=()`)
- What does it mean to pass objects of new type by value? (copy constructor)
- What are the constraints on legal values for the new type?
  (error checking in constructors and `operator=()`)
- Does the new type fit into an inheritance graph? (virtuality of functions, ...)
- What kind of conversions are allowed? (constructors, conversion operators)
- What operators and functions make sense for the new type? (public interface)
  - Strive for class interfaces that are complete and minimal
- What standard operators and functions should be explicitly disallowed? (declare private)
- Who should have access to the members of the new type? (access modes, friends)
- How general is the new type? (class template?)

Goal: user-defined classes should be indistinguishable from built-in types!

Object-Oriented Design (both books use C++ for examples)

- ❑ Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing, 1994, ISBN 0-8053-5340-2.
- ❑ Budd, *Introduction to Object Oriented Programming*, Second Edition, Addison-Wesley, 1996, ISBN 0-201-82419-1.

Aspects of Object-Oriented Design in C++

- ❑ Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995, ISBN 0-201-63361-2.
  - ➠ Provides an overview of the ideas behind patterns and a catalogue of 23 fundamental patterns
- ❑ Carroll and Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995, ISBN 0-201-51284-X.
  - ➠ Discusses many practical aspects of library design and implementation

➠ **But remember:**
Design and programming, like bicycle or swiming, cannot be learned by reading books!

Programming in C++

☆☆☆ class `std::string` ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

`std::string` Overview

- *string* `:=` sequence of *characters* `:=` objects of type `charT` (including e.g., `'\0'`)
- Standard `string` class is actually

```
namespace std {
   template < class charT,
              class traits = string_char_traits<charT>,
              class Allocator = allocator>
   class basic_string {
   public:
      static const unsigned npos = -1;
      /* ... */
   };

   typedef basic_string<char> string;
   typedef basic_string<wchar_t> wstring;

}
```

➠ To keep description of `std::string` simple in the following pages
- `string` is used instead of `std::basic_string<char>`
- `traits` and `Allocator` objects are ignored

- *Positions* in a string of *n* characters are specified
  - as an object of type `size_t` (typically `unsigned int`)
  - in the range *0* (first character) to *n-1* (last character)
- *Substrings* are specified by the start position `pos` and the number of characters `len`
  - length of substring is determined by the minimum of `len` and `size()-pos`
  - there is **no** separate `substring` class in the standard (but it is easy to define one)
- The `string` local special constant `npos`
  - can be used to specify the length *"all the remaining characters"*
  - is used by the search algorithms to indicate the result *"not found"*
- Member functions of `string` throw an exception
  - `out_of_range` if a specified `pos` is not in the range *0* to *n-1*
  - `length_error` if a specified `len` would construct a string larger than the maximal possible length

- Object definitions for all examples in the `std::string` chapter:

```
#include <string>
using namespace std;


const string str0 = "***";
const string str1 = "0123456789";

string str2 = str1;
string str3 = "345";

const char *cstr0 = "***";
const char *cstr1 = "abcdefghij";

char cstr2[16];

const char *cstr3 = "345";
```

- Arguments to `string` member functions fall into the following categories:

1.) an object of class `string` or a substring of it

```
(... const string& str, size_t pos=0, size_t len=npos ...)
```

Examples:

```
str1                                  // => "0123456789"
str1, 6                               // => "6789"
str1, 3, 4                            // => "3456"
str1, 3, 20                           // => "3456789"
```

2.) a C/C++ built-in string (zero-terminated array of `char`) or the first `len` characters of it

```
(... const char* s, size_t len=npos ...)
```

Examples:

```
cstr1                                 // => "abcdefghij"
cstr1, 5                              // => "abcde"
"*****"                               // => "*****"
"*****", 3                            // => "***"
```


3.) a repetition of `len` characters `c`

```
(.... size_t len, char c ...)
```

Examples:

```
5, '*'                                // => "*****"
2, 'a'                                // => "aa"
```

4.) a (sub)`string` object specified by an *iterator* range

```
(.... InputIterator first, InputIterator last ...)
```

Examples:

```
str1.begin(), str1.end()              // => "0123456789"
str1.begin()+2, str1.end()-2          // => "234567"
str1.rbegin(), str1.rend()            // => "9876543210"
```

- Constructors

```
explicit string();
string(const string& str, size_t pos = 0, size_t len = npos);
string(const char* s, size_t len);
string(const char* s);
string(size_t len, char c);

template<class InputIterator>
string(InputIterator begin, InputIterator end);
```

- Destructor

```
~string();
```

- Examples

```
string s1;                    // => ""
string s2(str1, 3, 3);        // => "345"
string s3(cstr1, 3);          // => "abc"
string s4(3, '*');            // => "***"
string s5(str1.begin(), str1.end());            // => "0123456789"
string s6(str1.rbegin(), str1.rend());          // => "9876543210"
```


- Assignment operator with *value semantics* (i.e., conceptional deep copy)

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
```

- Member function `assign`: modifies and returns `this`

```
string& assign(const string&);
string& assign(const string& str, size_t pos, size_t len);
string& assign(const char* s, size_t len);
string& assign(const char* s);
string& assign(size_t len, char c);

template<class InputIterator>
string& assign(InputIterator first, InputIterator last);
```

- Examples

```
str2.assign(str1, 3, 3)              // => "345"
str2.assign(cstr1, 3)                // => "abc"
str2.assign(3, '*')                  // => "***"
str2.assign(str1.begin(), str1.end())          // => "0123456789"
```

- `string` provides member types and operations similar to those provided by STL containers
  - ➠ all STL algorithms can be applied to `string`
  - ➠ but only few are useful as `string` often provides more optimized versions directly
  - ➠ examples of useful ones are comparison or searches based on user-defined predicates
- Most useful are the usual iterators pointing to the first and one-after-the-last position
  - ➠ `string` iterators are *random access iterators*

```
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```


- Like STL's `vector<T>`, `string`s have a
  - ❍ *length* `:=` number of characters stored in string
  - ❍ *capacity* `:=` number of characters which could be stored in string without new memory allocation
- Length related member functions

```
size_t size() const;                    // returns length
size_t length() const;                  // same as size()

size_t max_size() const;                // maximal possible length

bool empty() const;                     // size() == 0?
```

- Capacity related member functions

```
size_t capacity() const;                // returns capacity

void resize(size_t len, char c = '\0'); // shrink or enlarge capacity to len
                                        // new characters are initialized with c

void reserve(size_t len = 0);           // inform string that at least len
                                        // characters are needed, change
                                        // capacity as needed
```

- Individual characters of a `string` can be accessed through subscripting.
  It comes in two forms: with and without range check
  - The result of using `operator[](pos)` is *undefined* if `pos >= size()`

    ```
    const_reference operator[](size_t pos) const;
    reference operator[](size_t pos);
    ```
  - Member function `at(pos)` throws `out_of_range` if `pos >= size()`

    ```
    const_reference at(size_t pos) const;
    reference at(size_t pos);
    ```

  Otherwise, both return the character at position `pos`

- To access last character of a string `str` use `str[str.size()-1]`

- Examples:

  ```
  str1[5]                    // => '5'
  str1.at(5)                 // => '5'

  str1[42]                   // => undefined
  str1.at(42)                // => throws out_of_range
  ```


- Strings can be compared to (sub)strings or (sub)arrays of characters
- If `pos1` and `len1` are specified only this part of the string is compared
- Returns
  - `0` if the (sub)strings are equal
  - a negative number if the string is lexicographically before the argument character object
  - a positive number otherwise

  ```
  int compare(const string& str) const;
  int compare(size_t pos1, size_t len1, const string& str) const;
  int compare(size_t pos1, size_t len1, const string& str,
          size_t pos2, size_t len2) const;
  int compare(const char* s) const;
  int compare(size_t pos1, size_t len1, const char* s,
          size_t len2 = npos) const;
  ```

- Examples

  ```
  str1.compare(str1)         // => 0
  str1.compare(3, 3, str1)   // => >0
  ```

- Member function `append` allows adding characters described by the arguments to the end (short-cut for `insert` at the end of the string)
- `append` modifies the string itself (`*this`) and returns the modified string

```
string& append(const string& str);
string& append(const string& str, size_t pos, size_t len);
string& append(const char* s, size_t len);
string& append(const char* s);
string& append(size_t len, char c);

template<class InputIterator>
string& append(InputIterator first, InputIterator last);
```

- `operator+=` and `push_back` are provided as a conventional notation for the most common forms of `append`

```
string& operator+=(const string& str);
string& operator+=(const char* s);
string& operator+=(char c);

void push_back(const char);
```


- Examples

```
str2.append(str1)           // => "01234567890123456789"
str2.append(str1, 3, 3)     // => "0123456789345"
str2.append(cstr1, 3)       // => "0123456789abc"
str2.append(cstr1)          // => "0123456789abcdefghij"
str2.append(3, '*')         // => "0123456789***"
str2.append(str1.rbegin(), str1.rend())
                            // => "01234567899876543210"

str2 += str1                // => "01234567890123456789"
str2 += cstr1               // => "0123456789abcdefghij"
str2 += '*'                 // => "0123456789*"

str2.push_back('*')         // => "0123456789*"
```

- Member function `insert` allows adding characters described by the arguments to the string
- characters can either be inserted
  - *before* the index position `pos`
    - ➠ modifies the string itself (`*this`) and returns the modified string

```
string& insert(size_t pos, const string& str);
string& insert(size_t pos, const string& str,
        size_t pos2, size_t len);
string& insert(size_t pos, const char* s, size_t len);
string& insert(size_t pos, const char* s);
string& insert(size_t pos, size_t len, char c);
```

  - *before* the position described by iterator into the same string `p`

```
iterator insert(iterator p, char c);

void insert(iterator p, size_t len, char c);

template<class InputIterator>
void insert(iterator p,
        InputIterator first, InputIterator last);
```


- Examples

```
str2.insert(3, str0)          // => "012***3456789"
str2.insert(3, str0, 1, 1)    // => "012*3456789"
str2.insert(3, cstr0, 1)      // => "012*3456789"
str2.insert(3, cstr0)         // => "012***3456789"
str2.insert(3, 3, '*')        // => "012***3456789"

str2.insert(str2.begin(), '*')        // => "*0123456789"
str2.insert(str2.begin(), 3, '*')     // => "***0123456789"
str2.insert(str2.begin(), str1.rbegin(), str1.rend())
                                      // => "98765432100123456789"
```

- Member function `erase` deletes a range of characters described by the arguments from the string

- Delete a range of characters described by a start position `pos` and a length `len`

```
string& erase(size_t pos = 0, size_t len = npos);
```

- Delete the single character specified by iterator `position` or
  delete a range of characters described by iterator pair `first` and `last`
  - ➠ Returns an iterator pointing to the element immediately following the element(s) being erased

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
void clear();                          // == erase(begin(), end())
```

- Examples

```
str2.erase(3, 3)             // => "0126789"
str2.erase(3)                // => "012"
str2.erase(str2.begin())     // => "123456789"

str2.erase(str2.begin(), str2.end())     // => ""
str2.clear()                             // => ""
str2.erase()                             // => ""
```


- Member function `replace` allows changing a range of characters to other characters described by the arguments
  - ➠ replace `:=` assignment to a substring

- `replace` modifies the string itself (`*this`) and returns the modified string

- The range of characters to `replace` can be described by
  - ❍ a start position `pos1` and the length `len1`

```
string& replace(size_t pos1, size_t len1, const string& str);
string& replace(size_t pos1, size_t len1, const string& str,
        size_t pos2, size_t len2);
string& replace(size_t pos, size_t len1, const char* s,
        size_t len2);
string& replace(size_t pos, size_t len1, const char* s);
string& replace(size_t pos, size_t len1, size_t len2, char c);
```

or

- a pair of iterators `i1` and `i2`
  - ➠ `i1` and `i2` must be valid iterators into the string (`*this`)

```
string& replace(iterator i1, iterator i2, const string& str);

string& replace(iterator i1, iterator i2, const char* s,
         size_t len);

string& replace(iterator i1, iterator i2, const char* s);

string& replace(iterator i1, iterator i2, size_t len, char c);

template<class InputIterator>
string& replace(iterator i1, iterator i2,
        InputIterator j1, InputIterator j2);
```


- Examples

```
str2.replace(3, 3, str0)              // => "012***6789"
str2.replace(3, 3, str0, 1, 1)        // => "012*6789"
str2.replace(3, 3, cstr0, 1)          // => "012*6789"
str2.replace(3, 3, cstr0)             // => "012***6789"
str2.replace(3, 3, 3, '*')            // => "012***6789"
str2.replace(str2.begin(), str2.end(), str0)         // => "***"
str2.replace(str2.begin(), str2.end(), cstr0, 1)     // => "*"
str2.replace(str2.begin(), str2.end(), cstr0)        // => "***"
str2.replace(str2.begin(), str2.end(), 3, '*')       // => "***"
str2.replace(str2.begin(), str2.end(), str0.begin(), str0.end())
                                                     // => "***"
```

- Conversion to C/C++ style string
  - returns pointer to `string` owned character array containing copy of string

    ```
    const char* data() const;
    ```
  - the same thing but `'\0'` terminated

    ```
    const char* c_str() const;
    ```
  - copy (parts of) string into user-supplied buffer (**note**: not `'\0'`-terminated!)

    ```
    size_t copy(char* s, size_t len, size_t pos = 0) const;
    ```
- Exchange the contents of two strings

  ```
  void swap(string&);
  ```
- Select substring (`len` characters starting from position `pos`)

  ```
  string substr(size_t pos = 0, size_t len = npos) const;
  ```
- Examples

  ```
  int i = str1.copy(cstr2, 3, 3); cstr2[i] = '\0'    // cstr2="345"
  str2.swap(str3)              // => str2="345" str3="0123456789"
  str1.substr(3, 3)            // => "345"
  ```


- Find *substring* described by arguments in string starting at position `pos`

  ```
  size_t find(const string& str, size_t pos = 0) const;
  size_t find(const char* s, size_t pos, size_t len) const;   /*X*/
  size_t find(const char* s, size_t pos = 0) const;
  size_t find(char c, size_t pos = 0) const;
  ```
- Find *substring* described by arguments in string searching *backwards* from `pos`

  ```
  size_t rfind(const string& str, size_t pos = npos) const;
  size_t rfind(const char* s, size_t pos, size_t len) const;  /*X*/
  size_t rfind(const char* s, size_t pos = npos) const;
  size_t rfind(char c, size_t pos = npos) const;
  ```
- **Note**: in forms marked `/*X*/` `len` characters of `s` are searched from position `pos` in `*this`
- Both return start position of substring in string, or `string::npos` if not found
- Examples

  ```
  str1.find(str3, 0)                 // => 3
  str1.find(cstr1, 0)                // => string::npos
  str1.rfind(str3)                   // => 3
  str1.rfind('3', string::npos)      // => 3
  ```

- Find first/last *character* out of character set described by arguments in string forward/backward from position `pos`

```
size_t find_first_of(const string& str, size_t pos = 0) const;
size_t find_first_of(const char* s, size_t pos, size_t len)
          const;
size_t find_first_of(const char* s, size_t pos = 0) const;
size_t find_first_of(char c, size_t pos = 0) const;

size_t find_last_of(const string& str, size_t pos = npos) const;
size_t find_last_of(const char* s, size_t pos, size_t len) const;
size_t find_last_of(const char* s, size_t pos = npos) const;
size_t find_last_of(char c, size_t pos = npos) const;
```

- Both return start position of character in string, or `string::npos` if not found

- Examples

```
str1.find_first_of(str3, 0)          // => 3
str1.find_first_of(cstr3, 0, 3)      // => 3
str1.find_last_of(cstr3, string::npos)  // => 5
str1.find_last_of('3', string::npos)    // => 3
```


- Find first/last *character* **not** in character set described by arguments in string forward/backward from position `pos`

```
size_t find_first_not_of(const string& str, size_t pos = 0)
         const;
size_t find_first_not_of(const char* s, size_t pos, size_t len)
         const;
size_t find_first_not_of(const char* s, size_t pos = 0) const;
size_t find_first_not_of(char c, size_t pos = 0) const;

size_t find_last_not_of(const string& str, size_t pos = npos)
         const;
size_t find_last_not_of(const char* s, size_t pos, size_t len)
         const;
size_t find_last_not_of(const char* s, size_t pos = npos) const;
size_t find_last_not_of(char c, size_t pos = npos) const;
```

- Both return start position of character in string, or `string::npos` if not found

- Examples

```
str1.find_first_not_of(cstr3, 0)            // => 0
str1.find_last_not_of(str3, string::npos)    // => 9
```

- Concatenation is implemented as global function `operator+`
  - to allow non-`string` arguments (character arrays and single `char`'s) on left side of operator
  - to avoid creation of temporary string objects resulting from automatic conversion

```
string operator+(const string& lhs, const string& rhs);
string operator+(const char* lhs,   const string& rhs);
string operator+(char lhs,          const string& rhs);
string operator+(const string& lhs, const char* rhs);
string operator+(const string& lhs, char rhs);
```

- Examples

```
str1 + str1                  // => "01234567890123456789"
cstr1 + str1                 // => "abcdefghij0123456789"
'*' + str1                   // => "*0123456789"
str1 + cstr1                 // => "0123456789abcdefghij"
str1 + '*'                   // => "0123456789*"
```


- The equality operators `operator==` and `operator!=` are also implemented as global functions for the same reasons

```
bool operator==(const string& lhs, const string& rhs);
bool operator==(const char* lhs,   const string& rhs);
bool operator==(const string& lhs, const char* rhs);

bool operator!=(const string& lhs, const string& rhs);
bool operator!=(const char* lhs,   const string& rhs);
bool operator!=(const string& lhs, const char* rhs);
```

- Examples

```
str1 == str1                 // => true
cstr1 == str1                // => false
str1 == cstr1                // => false

str1 != str1                 // => false
cstr1 != str1                // => true
str1 != cstr1                // => true
```

- So are the rest of the comparison operators

```
bool operator< (const string& lhs, const string& rhs);
bool operator< (const string& lhs, const char* rhs);
bool operator< (const char* lhs, const string& rhs);

bool operator> (const string& lhs, const string& rhs);
bool operator> (const string& lhs, const char* rhs);
bool operator> (const char* lhs, const string& rhs);

bool operator<=(const string& lhs, const string& rhs);
bool operator<=(const string& lhs, const char* rhs);
bool operator<=(const char* lhs, const string& rhs);

bool operator>=(const string& lhs, const string& rhs);
bool operator>=(const string& lhs, const char* rhs);
bool operator>=(const char* lhs, const string& rhs);
```

- Examples

```
str1 >= str1                 // => true
cstr1 > str1                 // => true
cstr1 < str1                 // => false
```


- `operator>>` reads a *whitespace-terminated word*
  - string is expanded as needed to hold the word
  - initial and terminating whitespace is not entered into the string

```
istream& operator>>(istream& is, string& str);
```

- `operator<<` writes string contents to an output stream

```
ostream& operator<<(ostream& os, const string& str);
```

- `getline` reads a line terminated by `delim` into `str`
  - string `str` is expanded as needed to hold the line
  - delimiter `delim` is not entered into the string

```
istream& getline(istream& is, string& str, char delim = '\n');
```

- Exchange the contents of the strings `lhs` and `rhs`

```
void swap(string& lhs, string& rhs);
```

Programming in C++

☆☆☆ English – German Dictionary ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

abstract base class	abstrakte Basisklasse
abstract/concrete constness	abstrakte(logische)/tatsächliche Unveränderbarkeit
access specifier	Zugriffsspezifikation
aligment	(Speicher)Ausrichtung
arithmetic types	arithmetische Typen
array	Feld
automatic object	automatisches Objekt
assignment	Zuweisung
associativity	Assoziativität
basic types	elementare Typen, grundlegende Typen
base class	Basisklasse
catch block	Ausnahmebehandlungsblock
cast	cast, explizite Typ(en)konversion/Typ(en)umwandlung
character	Textzeichen
class	Klasse
comment	Kommentar
compound statement	(Anweisungs)Block, Anweisungsfolge, zusammengesetzte Anweisung
conditional compilation	bedingte Übersetzung

constructor	Konstruktor
control flow	Steuerfluß
conversion (operator)	(Typ)Umwandlungsoperator
diasy-chaining	aneinanderreihen, verketten, kaskadieren
data hiding	Geheimnisprinzip
data member	Datenelement
declaration	Deklaration
definition	Definition
default parameter	Standardparameter(wert), vorbelegte Parameter Parameterwertvorgabe
derived class	abgeleitete Klasse
derived types	abgeleitete Typen
destructor	Destruktor
do until loop	Durchlaufschleife
dynamic binding	dynamische Bindung
encapsulation	Kapselung
enumeration	Aufzählung
exception handling	Ausnahme(fall)behandlung
expression	Ausdruck

floating-point	Fliesskomma
for loop	Zählschleife
function body	Funktionsrumpf
function call	Funktionsaufruf
function head	Funktionskopf
global	global
header file	Schnittstellendatei
heap	dynamischer Speicher, Freispeicher
hide	überdecken, verdecken
identifier	Bezeichner
include directive	Dateieinfügeanweisung
indirection	Dereferenzierung
inheritance	Vererbung
inline	inline
integer	ganzzahlig
integral promotion	integrale Promotion
integral types	integrale Typen
iteration	Schleife

jump statement	Sprunganweisung
keyword	Schlüsselwort, reservierter Bezeichner
label	Sprungmarke
labeled statement	benannte Anweisung
literal	Literal (auch "Konstante")
member function	Elementfunktion, Methode
member template	Elementschablone
memory allocation	Speicheranforderung
memory deallocation	Speicherfreigabe
memory management	Speicherverwaltung
multiple Inheritance	Mehrfachvererbung, mehrfache Vererbung
namespace	Namensraum, Namensbereich
nested classes	(ein)geschachtelte Klassen
operator	Operator
operator delete	Löschoperator, Speicherfreigabe
operator new	Erzeugungsoperator, Speicherbelegung
overloading	Überladen
runtime type identification (RTTI)	Typermittlung zur Laufzeit, Laufzeit-Typinformation

placement	Plazierung
pointer	Zeiger
polymorphism	Polymorphie
precedence	Priorität
preprocessor directives	Präprozessordirektiven
private member	privates Element
protected member	geschütztes Element
public member	öffentliches Element
pure virtual	rein virtuell, abstrakt
qualified name	qualifizierter Bezeichner / Name
recursive	rekursiv
reference	Verweis / Referenz
relational operator	Vergleichsoperator
return type	Funktionswerttyp
scope	Gültigkeitsbereich, Bezugsrahmen
signed	vorzeichenbehaftet
stack	Stapel
statement	Anweisung
static binding	statische Bindung

static member	Klassenmethode, Klassenvariable, objektloses Element
static object	statisches Objekt
storage class	Speicherklasse
struct	Struktur, Verbund, Rekord
template	Schablone, generische / parametrisierte Klasse, Musterklasse
throw exception	Ausnahmebehandlungs(code) anstossen, Ausnahme auswerfen
unsigned	vorzeichenlos
union	varianter Rekord
virtual function	virtuelle Funktion
while loop	Abweisungsschleife

Programming in C++

☆☆☆ Index ☆☆☆

Dr. Bernd Mohr
b.mohr@fz-juelich.de

Forschungszentrum Jülich
Germany

Symbols

A

B

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W