FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Ausbildung von
Mathematisch-Technischen Software-Entwicklern

Programming in C++
Part I

Bernd Mohr

FZJ-JSC-BHB-0154

1. Auflage

(letzte Änderung: 19.09.2003)

Publikationen des JSC stehen in druckbaren Formaten (PDF auf dem WWW-Server des Forschungszentrums unter der URL: <http://www.fz-juelich.de/jsc/files/docs/> zur Verfügung. Eine Übersicht über alle Publikationen des JSC erhalten Sie unter der URL: **<http://www.fz-juelich.de/jsc/docs>** .

Beratung

Tel: +49 2461 61 -nnnn

Auskunft, Nutzer-Management (Dispatch)

Das Dispatch befindet sich am Haupteingang des JSC, Gebäude 16.4, und ist telefonisch erreichbar von

Montag bis Donnerstag	8.00 - 17.00 Uhr
Freitag	8.00 - 16.00 Uhr

Tel. 5642 oder 6400, Fax 2810, E-Mail: **dispatch.jsc@fz-juelich.de**

Supercomputer-Beratung

Tel. 2828, E-Mail: **sc@fz-juelich.de**

Netzwerk-Beratung, IT-Sicherheit

Tel. 6440, E-Mail: **junet-helpdesk@fz-juelich.de**

Rufbereitschaft

Außerhalb der Arbeitszeiten (montags bis donnerstags: 17.00 - 24.00 Uhr, freitags: 16.00 - 24.00 Uhr, samstags: 8.00 - 17.00 Uhr) können Sie dringende Probleme der Rufbereitschaft melden:

Rufbereitschaft Rechnerbetrieb:	Tel. 6400
Rufbereitschaft Netzwerke:	Tel. 6440

An Sonn- und Feiertagen gibt es keine Rufbereitschaft.

Fachberater

Tel. +49 2461 61 -nnnn

Fachgebiet	**Berater**	**Telefon**	***E-Mail***
Auskunft, Nutzer-Management, Dispatch	E. Bielitza	5642	***dispatch.jsc@fz-juelich.de***
Supercomputer	W. Frings	2828	***sc@fz-juelich.de***
JuNet/Internet, Netzwerke, IT-Sicherheit	T. Schmühl	6440	***junet-helpdesk@fz-juelich.de***
Web-Server	Dr. S. Höfler-Thierfeldt	6765	***webmaster@fz-juelich.de***
Backup, Archivierung	U. Schmidt	1817	***backup.jsc@fz-juelich.de***
Fortran	D. Koschmieder	3439	***fortran.jsc@fz-juelich.de***
Mathematische Methoden	Dr. B. Steffen	6431	***mathe-admin@fz-juelich.de***
Statistik	Dr. W. Meyer	6414	***mathe-admin@fz-juelich.de***
Mathematische Software	Dr. B. Körfgen, R. Zimmermann	6761, 4136	***mathe-admin@fz-juelich.de***
Graphik	Ma. Busch, M. Boltes	4100, 6557	***graphik.jsc@fz-juelich.de***
Videokonferenzen	M. Sczimarowsky, R. Grallert	6411, 6421	***vc.jsc@fz-juelich.de***
Oracle-Datenbank	M. Wegmann, J. Kreutz	1463, 1464	***oracle.jsc@fz-juelich.de***

Die jeweils aktuelle Version dieser Tabelle finden Sie unter der URL:
<http://www.fz-juelich.de/jsc/allgemeines/beratung>

Programming in C++

Part I

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

Forschungszentrum Jülich
Germany

Version 19 September 2003

Programming in C++

☆☆☆ Introduction ☆☆☆

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

Forschungszentrum Jülich
Germany

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++*")

- ➠ Many changes in compilers in recent years; expect more
- ➠ 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

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/`

- 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/`

Some parts of 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)
- icpc (Intel)
- 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

Basics — General Remarks

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;
}
```

	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
```


	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

```
const double PI=3.1415;
const char SP=' ';
const double NEG_PI=-PI;
const double TWO_PI=2*PI;

typedef int Length;
enum State {
     error, warn, ok
};

int a, b;
double x;
enum State s;
int i = 396;
```

Pascal

```
const
  PI = 3.1415;
  SP = ' ';
  NEG_PI = -PI;
  ❖
type
  Length = integer;
  State =
   (error, warn, ok);
var
  a, b : integer;
  x : real;
  s : State;
  ❖
```

Fortran

```
real,parameter::PI=3.1415
character,parameter::SP=' '
real,parameter::NEG_PI=-PI
real,parameter::TWO_PI=2*PI

❖
❖

integer a, b
real x
❖
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!

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]`!

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

- `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;
```


ANSI C	Pascal	Fortran
`int i;`		`integer,target :: i`

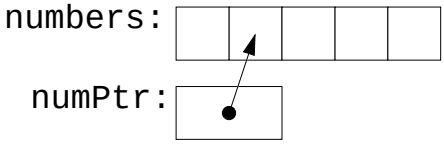
`int *a;`
`char *b, *c;`
`Engine *mp;`
	`var`
	`a : ^integer;`
	`b, c : ^char;`
	`mp : ^Engine;`
		`integer,pointer :: a`
		`character,pointer :: b, c`
		`type(Engine),pointer :: mp`

`a = &i;`
`*a = 3;`
`b = 0;`
	❖
	`a^ := 3;`
	`b := nil;`
		`a => i`
		`a = 3`
		`nullify(b)`

`mp = malloc(` `  sizeof(Engine));`	`new(mp);`	`allocate(mp)`
`(*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

```
#include "file"

global constant, type,
variable, function and
procedure declarations

int main()
   /* C comment */

{
   local declarations
   statements
}
```

Pascal

```
❖

(* Pascal comment1 *)

program AnyName;
   { Pascal comment2 }
   global constant, type,
   variable, function and
   procedure declarations
begin

   statements
end.
```

Fortran

```
include 'file'

module Global
   global constant, variable decls
   contains
   function, procedure decls
end module

program AnyName
! Fortran comment

use Global

local declarations
statements
end [program]
```

- ANSI C and Fortran: declarations in any order

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`

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')`

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

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

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();
```


- 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

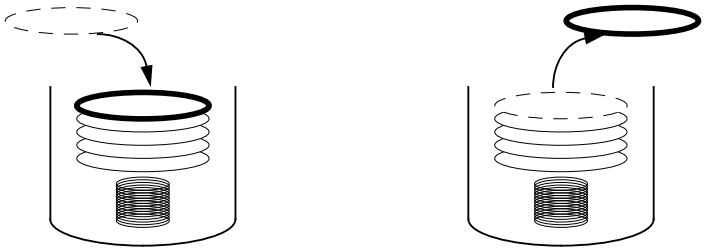
Forschungszentrum Jülich
Germany

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++?**

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...**

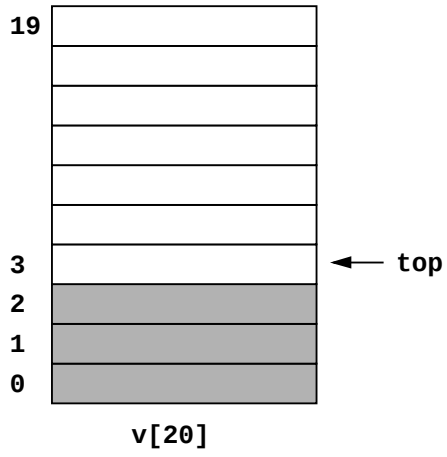

```
typedef struct {
   float v[20];
   int top;
} Stack;

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 `empty()` 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

From C to C++ New Keywords

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

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

C99

`[iso646.h]`

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

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

C99

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

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

C99

```
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
```

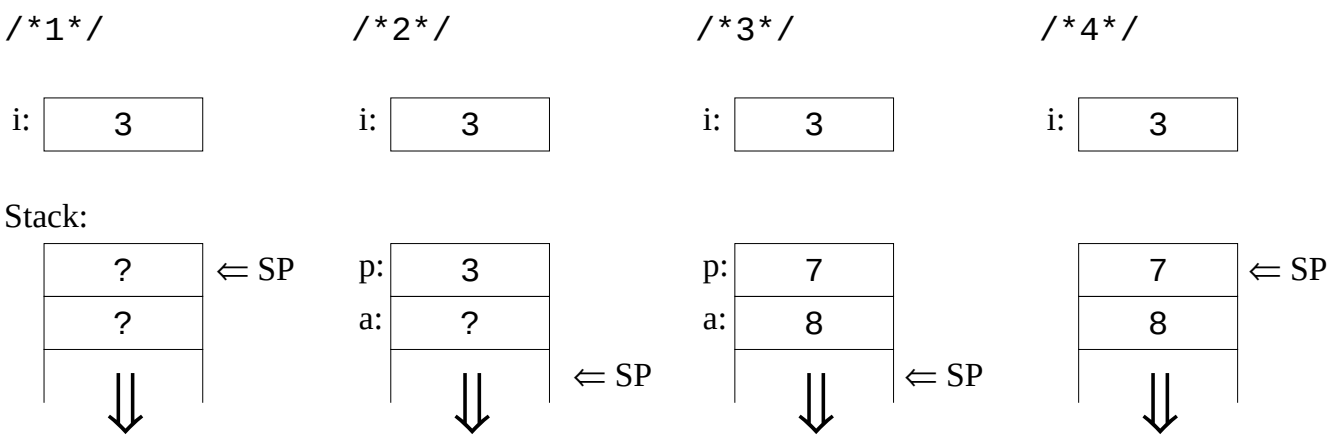
```
/* pointers */

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

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

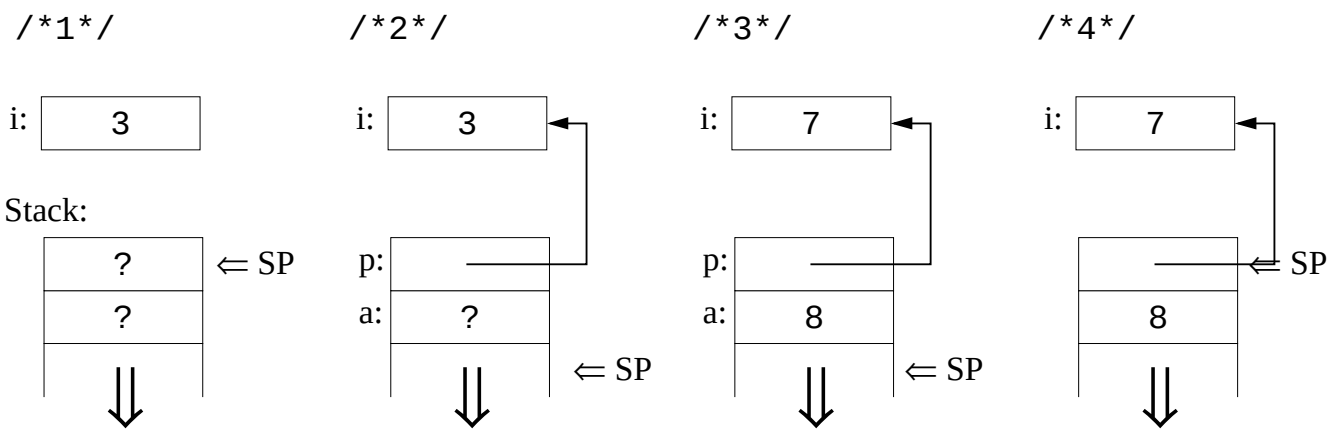
- The primary use of a reference type is as an *parameter type* or *return type* of a function. They are rarely used directly.

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




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

```
void func(int& p) {
   int a = p + 5;
   p = 7;
}
int i = 3;
func(i);
...
```



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

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.

C99

```
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 f(int);
f();      // calls f()
f(10);    // calls f(int);
```

```
void g(int x=0);

g();         // calls  g(0);
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>` instead of `<stdio.h>`
- IOstreams are part of the standard C++ library

```
using namespace std;
```

- 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>
using namespace std;

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? ";

   if ( ! cin >> birth ) birth = -1;
}
```

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)"
```


- 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++

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

C99

- 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);
}
```


- New keywords
- Alternative representations
- `//` comment
- `struct XXX` introduces new type (➠ `typedef struct XXX XXX;`)
- Declarations and statements in any order
- Default arguments for functions
- [Reference type]
- Passing values in and out of functions per reference
  - ➠ output parameter ➠ `Type& t`
  - ➠ efficiency ➠ `const Type& t`
- [Inline functions]
- Function overloading
- Operator overloading (for user-defined types)
- Boolean type: `bool`, `true`, `false`
- Input/output:

```
#include <iostream>
using namespace std;
cout << var
cin  >> var
if ( handle ) ...
```

- `new`/`delete` instead of `malloc`/`free`
- `extern "C"`
- Include guards
- Declare constants with `const` instead of `#define`

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;
   i = 5;
   /*...*/
   j = 5 * i;
   /*...*/
}
```

① allocation of object *i* of type `int`

② initialization

③ usage

④ [destruction not needed for built-in types]

⑤ leaves scope: deallocation of memory

For classes: special member functions

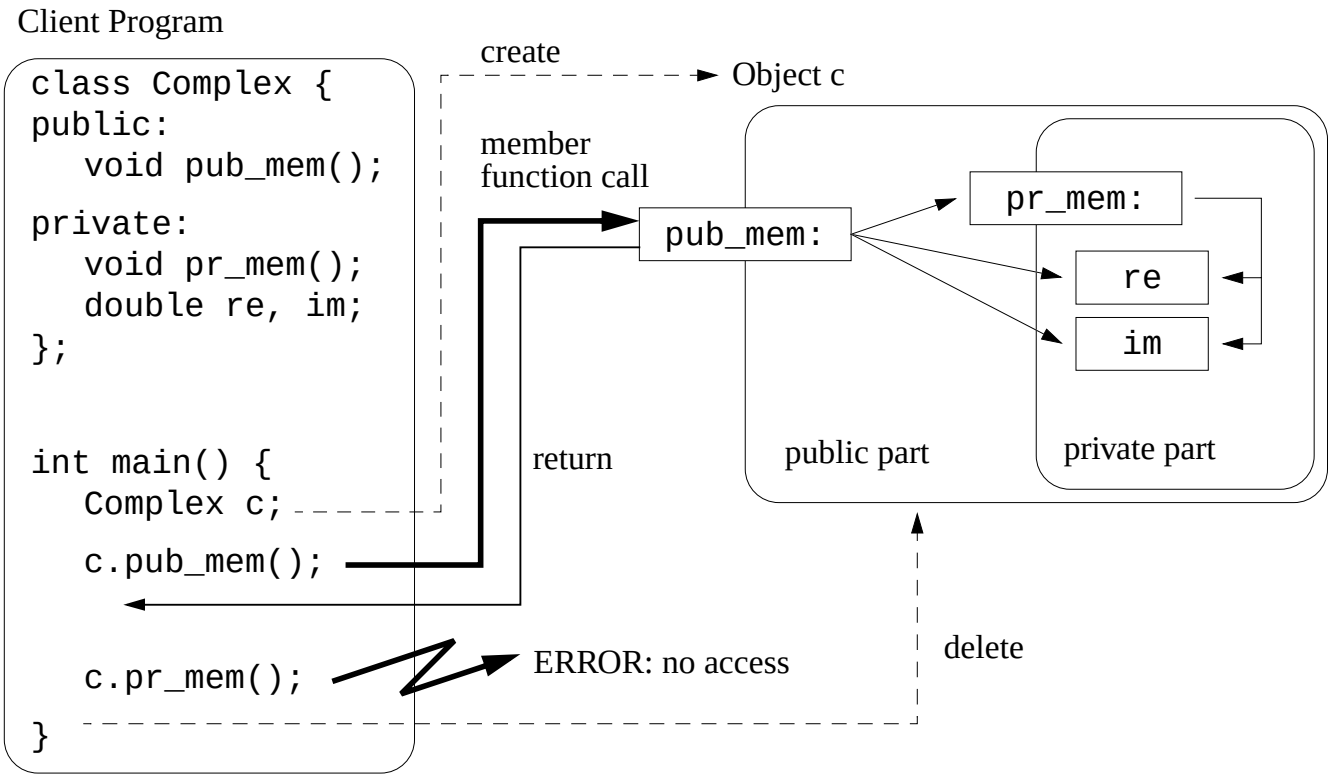
- *constructor*: automatically called after allocation for class specific initialization ②
- *destructor*: automatically called prior to deallocation for class specific clean-up ④


```
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*
- access specifiers (`public`, `private`, `protected`) can appear in any order and repeatedly



- 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?

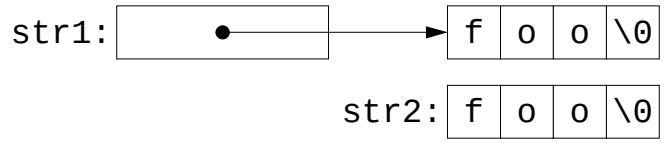
- Free store allocation (and deallocation)
  - Pointers usually indicate this
  - Who should do this and keep track? (class or user?)
  - How about the constructors and destructor?
- Pointer value vs. what it points to
  - For object assignment, do we want memberwise assignment of pointer fields?
    - ➠ 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

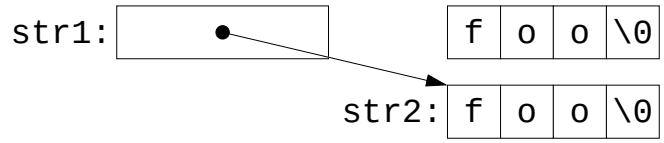
- C++ character strings are really *NUL terminated arrays of char* (as in C)
  - Beware of difference between arrays and pointers!

```
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!
```



```
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!**

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)**


```
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`)

Pointer Data Members — `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....

Pointer Data Members — Constructors/Destructor

❑ 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

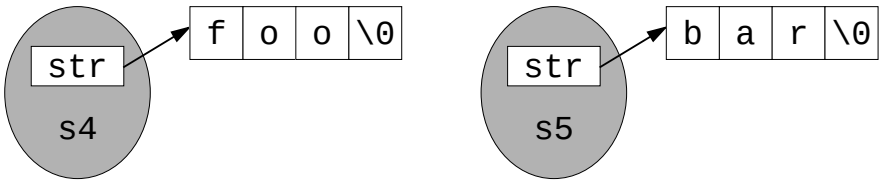
```
class String {
public:
   const char *c_str(void) {
      return (const char *) str;
   }
};
```

- 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:

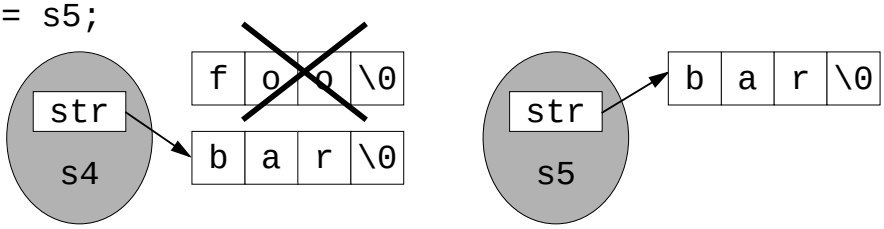
```
String s1, s2(s1), s3 = "I am s3";
char *cs1 = s2.c_str();              // error!
const char *cs2 = s1.c_str();        // OK
cout << s3.c_str() << endl;          // OK
```



```
String& operator=(const String& rhs) {      //poor implementation
   delete [] str;
   set_str (rhs.str);
   return *this;
}
String s4("foo"), s5("bar");
```




```
s4 = s5;
```



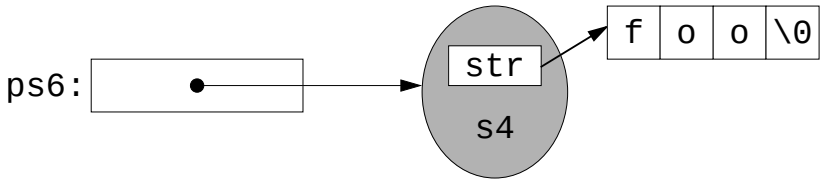
- Recall: reference returned for daisy-chaining

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

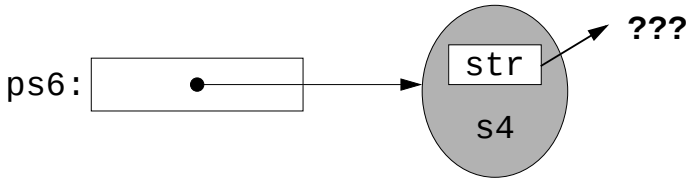
```
String& operator=(const String& rhs) {     //poor implementation
   delete [] str;
   set_str (rhs.str);
   return *this;
}
```

- 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
  - ➠ **∃ pointer data fields ⇒ define copy constructor and** `operator=()` **!!!!**

- 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**

- `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>
using namespace std;
#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

```
#include <iostream>
#include <string>
using namespace std;

string s1 = "hello";
string s2 = s1 + ", world!";

cout << s2 << endl;
```

  - 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
  - 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,                    // ctor(double)
             Complex(3.4, 4.5),      // ctor(double, double)
             c,                      // copy ctor
             ca1[2]                  // copy ctor
           };
}
```

  - ❍ 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` $\not\Longrightarrow$ `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

- 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!**

- 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

- 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)

- 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`

- 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;
}
```


- 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;
```

  - Application specific functions members

- 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

More Class Examples — `Rectangle` (C to C++)

```
#include <stdio.h>

struct RE {
   int h_, w_;
};

typedef struct RE Rectangle;

void re_set(Rectangle* r,
            int h, int w)
   { r->h_ = h; r->w_ = w; }

int re_height(Rectangle r)
   { return r.h_; }

int re_width(Rectangle r)
   { return r.w_; }

int main() {
   Rectangle r;
   re_set(&r, 5, 8);
   printf("%d\n", re_height(r));
}
```

```
#include <iostream>
using namespace std;

struct Rectangle {
   int h_, w_;
};


void re_set(Rectangle& r,
            int h, int w)
   { r.h_ = h; r.w_ = w; }

int re_height(const Rectangle& r)
   { return r.h_; }

int re_width(const Rectangle& r)
   { return r.w_; }

int main() {
   Rectangle r;
   re_set(r, 5, 8);
   cout << re_height(r) << endl;
}
```

More Class Examples — `Rectangle` (Simple "Class")

```
#include <iostream>
using namespace std;

struct Rectangle {
   int h_, w_;
};

void re_set(Rectangle& r,
            int h, int w)
   { r.h_ = h; r.w_ = w; }

int re_height(const Rectangle& r)
   { return r.h_; }

int re_width(const Rectangle& r)
   { return r.w_; }


int main() {
   Rectangle r;
   re_set(r, 5, 8);
   cout << re_height(r) << endl;
}
```

```
#include <iostream>
using namespace std;

struct Rectangle {
   int h_, w_;


   void set(int h, int w)
      { this->h_ = h;
        this->w_ = w; }

   int height() const
      { return this->h_; }

   int width() const
      { return this->w_; }

};

int main() {
   Rectangle r;
   r.set(5, 8);
   cout << r.height() << endl;
}
```


More Class Examples — `Rectangle` (`this` Not Necessary)

```
#include <iostream>
using namespace std;

struct Rectangle {
   int h_, w_;


   void set(int h, int w)
      { this->h_ = h;
        this->w_ = w; }

   int height() const
      { return this->h_; }

   int width() const
      { return this->w_; }

};

int main() {
   Rectangle r;
   r.set(5, 8);
   cout << r.height() << endl;
}
```

```
#include <iostream>
using namespace std;

struct Rectangle {
   int h_, w_;


   void set(int h, int w)
      { h_ = h;
        w_ = w; }

   int height() const
      { return h_; }

   int width() const
      { return w_; }

};

int main() {
   Rectangle r;
   r.set(5, 8);
   cout << r.height() << endl;
}
```

More Class Examples — `Rectangle` ("real" Class)

```
#include <iostream>
using namesapce std;

struct Rectangle {

   int h_, w_;


   void set(int h, int w)
      { h_ = h; w_ = w; }

   int height() const
      { return h_; }

   int width() const
      { return w_; }

};

int main() {
   Rectangle r;
   r.set(5, 8);
   cout << r.height() << endl;
}
```

```
#include <iostream>
using namespace std;

class Rectangle {
private:
   int h_, w_;

public:
   Rectangle(int h, int w)
      { h_ = h; w_ = w; }

   int height() const
      { return h_; }

   int width() const
      { return w_; }

};

int main() {
   Rectangle r(5, 8);

   cout << r.height() << endl;
}
```


More Class Examples — `Rectangle` (Outlined Methods)

```
#include <iostream>
using namespace std;

class Rectangle {
private:
   int h_, w_;

public:
   Rectangle(int h, int w)
      { h_ = h; w_ = w; }

   int height() const
      { return h_; }

   int width() const
      { return w_; }

};


int main() {
   Rectangle r(5, 8);
   cout << r.height() << endl;
}
```

```
#include <iostream>
using namespace std;

class Rectangle {
private:
   int h_, w_;

public:
   Rectangle(int h, int w)
      : h_(h), w_(w) {}
   int height() const;
   int width() const;
};

int Rectangle::height() const
   { return h_; }

int Rectangle::width() const
   { return w_; }

int main() {
   Rectangle r(5, 8);
   cout << r.height() << endl;
}
```

More Class Examples — Complex (Simple C)

```
#include <stdio.h>

struct Complex {
   double re, im;
};

void add(struct Complex *res,
         struct Complex c1,
         struct Complex c2) {
   res->re = c1.re+c2.re;
   res->im = c1.im+c2.im;
}

void mult(struct Complex *res,
          struct Complex c1,
          struct Complex c2) {
   res->re = c1.re*c2.re-c1.im*c2.im;
   res->im = c1.re*c2.im+c1.im*c2.re;
}

void cplx_print(struct Complex c) {
   printf("(%g,%gi)", c.re, c.im);
}

int main() {
   /* calculate r=(c1+c2)*c3 */
   struct Complex c1, c2, c3, r, t;
   c1.re = 1; c1.im = 1;
   c2.re = 2; c2.im = 2;
   c3.re = 3; c3.im = 3;
   add(&t, c1, c2);
   mult(&r, t, c3);
   printf("r = ");
   cplx_print(r);
   printf("\n");
}
```


More Class Examples — Complex (Better C)

```
#include <stdio.h>

typedef struct {
   double re, im;
} Complex;

void cplx_init(Complex *c,
               double r, double i) {
   c->re = r; c->im = i;
}

void cplx_add(Complex *res,
              Complex *c1,
              Complex *c2) {
   cplx_init(res, c1->re+c2->re,
                  c1->im+c2->im);
}

void cplx_mult(Complex *res,
               Complex *c1,
               Complex *c2) {
   cplx_init(res,
      c1->re*c2->re - c1->im*c2->im,
      c1->re*c2->im + c1->im*c2->re);
}

double cplx_real(Complex *c) {
   return c->re;
}

double cplx_imag(Complex *c) {
   return c->im;
}

void cplx_print(Complex *c) {
   printf("(%g,%gi)", c->re, c->im);
}

int main() {
   /* calculate r=(c1+c2)*c3 */
   Complex c1, c2, c3, r, t;
   cplx_init(&c1, 1, 1);
   cplx_init(&c2, 2, 2);
   cplx_init(&c3, 3, 3);
   cplx_add(&t, &c1, &c2);
   cplx_mult(&r, &t, &c3);
   printf("r = ");
   cplx_print(&r);
   printf("\n");
}
```

```
#include <stdio.h>

struct Complex {
   double re, im;
};

void cplx_init(Complex& c,
               double r, double i) {
   c.re = r; c.im = i;
}


void cplx_add(Complex& res,
              const Complex& c1,
              const Complex& c2) {
   cplx_init(res, c1.re+c2.re,
                  c1.im+c2.im);
}

void cplx_mult(Complex& res,
               const Complex& c1,
               const Complex& c2) {
   cplx_init(res,
         c1.re*c2.re - c1.im*c2.im,
         c1.re*c2.im + c1.im*c2.re);
}
```

```
double cplx_real(const Complex& c) {
   return c.re;
}

double cplx_imag(const Complex& c) {
   return c.im;
}

void cplx_print(const Complex& c) {
   printf("(%g,%gi)", c.re, c.im);
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1, c2, c3, r, t;
   cplx_init(c1, 1, 1);
   cplx_init(c2, 2, 2);
   cplx_init(c3, 3, 3);
   cplx_add(t, c1, c2);
   cplx_mult(r, t, c3);
   printf("r = ");
   cplx_print(r);
   printf("\n");
}
```



```
#include <stdio.h>

struct Complex {
   double re, im;
};

void cplx_init(Complex& c,
               double r, double i) {
   c.re = r; c.im = i;
}

Complex operator+(const Complex& c1,
                  const Complex& c2){
   Complex res;
   cplx_init(res, c1.re+c2.re,
                  c1.im+c2.im);
   return res;
}

Complex operator*(const Complex& c1,
                  const Complex& c2){
   Complex res; cplx_init(res,
         c1.re*c2.re - c1.im*c2.im,
         c1.re*c2.im + c1.im*c2.re);
   return res;
}
```

```
double cplx_real(const Complex& c) {
   return c.re;
}

double cplx_imag(const Complex& c) {
   return c.im;
}

void cplx_print(const Complex& c) {
   printf("(%g,%gi)", c.re, c.im);
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1, c2, c3, r;
   cplx_init(c1, 1, 1);
   cplx_init(c2, 2, 2);
   cplx_init(c3, 3, 3);
   r = (c1 + c2) * c3;
   printf("r = ");
   cplx_print(r);
   printf("\n");
}
```

```
#include <iostream>
using namespace std;

struct Complex {
   double re, im;  };

void cplx_init(Complex& c,
               double r, double i) {
   c.re = r; c.im = i;
}

Complex operator+(const Complex& c1,
                  const Complex& c2){
   Complex res;
   cplx_init(res, c1.re+c2.re,
                  c1.im+c2.im);
   return res;
}

Complex operator*(const Complex& c1,
                  const Complex& c2){
   Complex res; cplx_init(res,
         c1.re*c2.re - c1.im*c2.im,
         c1.re*c2.im + c1.im*c2.re);
   return res;
}
```

```
double cplx_real(const Complex& c) {
   return c.re;
}

double cplx_imag(const Complex& c) {
   return c.im;
}

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.re
        << "," << c.im << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1; cplx_init(c1, 1, 1);
   Complex c2; cplx_init(c2, 2, 2);
   Complex c3; cplx_init(c3, 3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```



```
#include <iostream>
using namespace std;

struct Complex {

   double re, im;

   void init(double r, double i) {
      this->re = r; this->im = i;
   }

   Complex
   operator+(const Complex& c2) const {
      Complex res;
      res.init(this->re+c2.re,
               this->im+c2.im);
      return res;
   }

   Complex
   operator*(const Complex& c2) const {
      Complex res;
      res.init(
      this->re*c2.re-this->im*c2.im,
      this->re*c2.im+this->im*c2.re);
      return res;
   }
```

```
   double real() const {
      return this->re;
   }

   double imag() const {
      return this->im;
   }
};

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.re
        << "," << c.im << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1; c1.init(1, 1);
   Complex c2; c2.init(2, 2);
   Complex c3; c3.init(3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```

More Class Examples — `Complex` (`this` Not Necessary)

```
#include <iostream>
using namespace std;

struct Complex {

   double re, im;

   void init(double r, double i) {
      re = r; im = i;
   }

   Complex
   operator+(const Complex& c2) const {
      Complex res;
      res.init(re+c2.re, im+c2.im);
      return res;
   }

   Complex
   operator*(const Complex& c2) const {
      Complex res;
      res.init(re*c2.re - im*c2.im,
               re*c2.im + im*c2.re);
      return res;
   }

   double real() const {
      return re;
   }

   double imag() const {
      return im;
   }

};

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.re
        << "," << c.im << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1; c1.init(1, 1);
   Complex c2; c2.init(2, 2);
   Complex c3; c3.init(3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```


More Class Examples — `Complex` (Access Control)

```
#include <iostream>
using namespace std;

class Complex {

private:
   double re, im;

public:
   void init(double r, double i) {
      re = r; im = i;
   }

   Complex
   operator+(const Complex& c2) const {
      Complex res;
      res.init(re+c2.re, im+c2.im);
      return res;
   }

   Complex
   operator*(const Complex& c2) const {
      Complex res;
      res.init(re*c2.re - im*c2.im,
               re*c2.im + im*c2.re);
      return res;
   }

   double real() const {
      return re;
   }

   double imag() const {
      return im;
   }

};

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.real()
        << "," << c.imag() << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1; c1.init(1, 1);
   Complex c2; c2.init(2, 2);
   Complex c3; c3.init(3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```

```
#include <iostream>
using namespace std;

class Complex {
private:
   double re, im;

public:
   Complex(double r, double i) {
      re = r; im = i;
   }

   Complex
   operator+(const Complex& c2) const {
      Complex res(re+c2.re,
                  im+c2.im);
      return res;
   }

   Complex
   operator*(const Complex& c2) const {
      Complex res(re*c2.re-im*c2.im,
                  re*c2.im+im*c2.re);
      return res;
   }

   double real() const {
      return re;
   }

   double imag() const {
      return im;
   }

};

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.real()
        << "," << c.imag() << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1(1, 1);
   Complex c2(2, 2);
   Complex c3(3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```



```
#include <iostream>
using namespace std;

class Complex {

private:
   double re, im;

public:
   Complex(double r = 0.0,
           double i = 0.0) {
      re = r; im = i;
   }

   const Complex
   operator+(const Complex& c2) const {
      return Complex(re+c2.re,
                     im+c2.im);
   }

   const Complex
   operator*(const Complex& c2) const {
      return
      Complex(re*c2.re - im*c2.im,
              re*c2.im+im*c2.re);
   }

   double real() const {
      return re;
   }

   double imag() const {
      return im;
   }
};

ostream&
operator<<(ostream& ostr,
           const Complex& c) {
   return
   ostr << "(" << c.real()
        << "," << c.imag() << "i)";
}

int main() {
   // calculate r=(c1+c2)*c3
   Complex c1(1, 1);
   Complex c2(2, 2);
   Complex c3(3, 3);
   Complex r = (c1 + c2) * c3;
   cout << "r = " << r << endl;
}
```

```
#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); }
```

More Class Examples — `FStack` Class Implementation (.cpp)

- `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;
```


More Class Examples — `Employee` Class Interface (.h)

```
#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
```

More Class Examples — `Employee` Class Implementation (.cpp)

- 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); }
```


More Class Examples — `Rational` Class Definition (.h)

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

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream>
using namespace std;

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);
   // 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;
      }
   }
}
```

More Class Examples — `Rational` Constructor (.cpp)

- 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;
   }
}
```


More Class Examples — `Rational` Addition (.cpp)

- 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;
}
```

More Class Examples — `Rational` Multiplication (.cpp)

- 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;
}
```


More Class Examples — `Rational` Division (.cpp)

- 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;
}
```

More Class Examples — `Rational` Boolean Operators (.cpp)

- 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());
}
```


More Class Examples — `Rational` Boolean Operators (.cpp)

- 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?

More Class Examples — `toRational` Conversion (.cpp)

- 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();
   }
}
```


More Class Examples — `toRational` Conversion (.cpp)

- 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

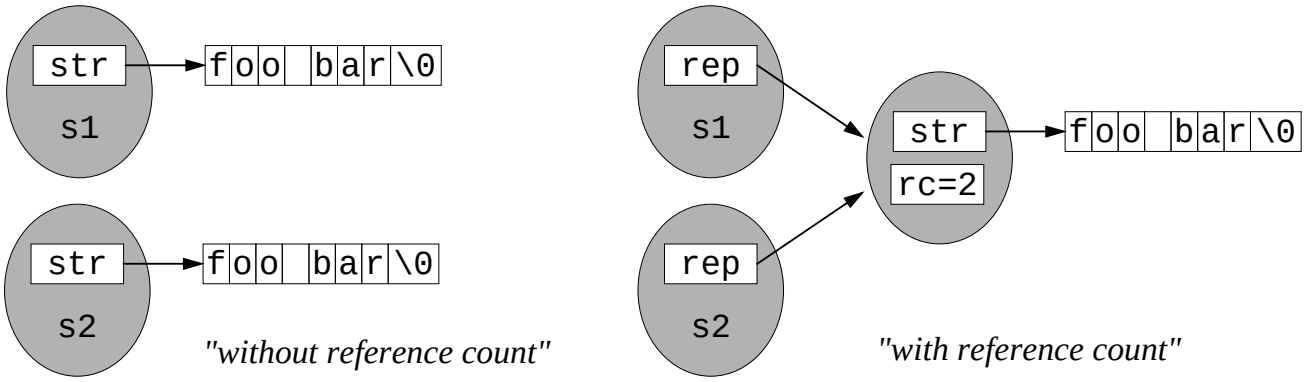
```
Rational binomial(int n, int k) {
   if (n < 0) { cerr << "1st out of range" << endl; exit(1); }
   if (k < 0 || k > n) {
       cerr << "2nd out of range" << endl; exit(1); }
   if (k > n-k) k = n-k;
   if (k == 0) return 1;
   return Rational(n-k+1, k) * binomial(n, k-1);
}
```

- 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>` instead of `<stdio.h>`
- IOstreams are part of the standard C++ library

```
using namespace std;
```

- 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`

```
int get();
```

- `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)

```
int c2;
c2 = cin.get();
```

- 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_base::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_base::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>`"
- 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 open mode* by using optional second argument
  - `ios_base::in` open for reading
  - `ios_base::out` open for writing
  - `ios_base::ate` start position at end of file
  - `ios_base::app` append mode: all additions at end of file
  - `ios_base::trunc` delete old file contents at open
  - `ios_base::binary` open file in binary mode
- The mode is constructed by or-ing the predefined values

```
// append to file foo
ofstream foofile("foo", ios_base::out|ios_base::app);
```

- Comparison `open` mode flags with `stdio` equivalents (ignoring `ios_base::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 open 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_base::in);
   ...
   infile.close();
   infile.clear();
}
```

- Complicated way to calculate harmonic number H_{100}

```
#include <fstream>
using namespace std;
#include <stdlib.h>

int main(int, char**) {
   ofstream out("iotest.out");
   if (!out) {
      cerr << "error opening '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 << "error opening '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!**)

➠ All constants are part of class `ios_base`, e.g., `ios_base::showpos`

- Stream class member functions
  - to get format flags

    `fmtflags flags()`
  - to set format flags (flags combined by using "`|`" operator): `flags = f`

    `fmtflags flags(long f)`
  - to set or unset additional flags: `flags |= f` or `flags &= ~f`

    `fmtflags setf(fmtflags f)` `fmtflags unsetf(fmtflags f)`
    `fmtflags setf(fmtflags f, fmtflags 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)`

  } return old values
- or alternatively: Stream class *manipulators* (inserted in stream instead of values)


```
inline void ifmt(ostream& s, int w, ios_base::fmtflags b)
                {s.width(w); s.setf(b,ios_base::basefield);}
inline void ffmt(ostream& s, ios_base::fmtflags f)
                {s.setf(f,ios_base::floatfield);}

int i = 12345; double d = 3.1415;

cout.fill('.');
cout.setf(ios_base::showbase);
ifmt(cout,10,ios_base::dec); cout << i;                 // "%#10d"
ifmt(cout,10,ios_base::oct); cout << i;                 // "%#10o"
ifmt(cout,10,ios_base::hex); cout << i;                 // "%#10x"
cout << endl;
cout.precision(3);
ffmt(cout,ios_base::fmtflags(0)); cout << d << "  ";    // "%.3g"
ffmt(cout,ios_base::scientific);  cout << d << "  ";    // "%.3e"
ffmt(cout,ios_base::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

- Also: `[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>`"

```
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>
#include <iomanip>
using namespace std;


ostream& general(ostream& ostr) {
   ostr.setf(ios_base::fmtflags(0), ios_base::floatfield);
   return ostr;
}


int main() {
   double pi = 3.1415;
   cout << scientific << pi << endl;
   cout << general << pi << endl;
}
```

```
#include <iostream>
#include <iomanip>
using namespace std;

// define manipulator general 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 C++ standard library strings (`string`)
- C++ equivalent to `sscanf` and `sprintf`

- Use `#include <sstream>`
- Has classes `ostringstream`, `istringstream`, and `stringstream`
- `sstream` member function `str()` returns constructed string

- Example

```
#include <sstream>
#include <string>
using namespace std;

ostringstream os;
int i = 15, j, k;

os << i << " is a number, in hex: 0x" << hex << i;
string s = os.str();

istringstream is("15 18 798");
is >> i >> j >> k;
```


- Problem: `operator<<()` for compound types does not obey `width` stream attribute
  - ➠ use `ostringstream` for temporary buffering!

- Example: `Complex::operator<<()`

```
#include <sstream>
using namespace std;

ostream& operator<<(ostream& ostr, const Complex& rhs) {
  ostringstream buf;
  buf.flags(ostr.flags());          // copy stream flags
  buf.precision(ostr.precision());  // copy precision

  buf << "(" << rhs.real() << "," << rhs.imag() << "i)";
  ostr << buf.str();
  return ostr;
}
```

- Another example: `Rational::operator<<()`

```
#include <sstream>
using namespace std;

ostream& operator<< (ostream& ostr, const Rational& r) {
   if (r.denominator() == 1L)
      ostr << r.numerator();

   else {
      ostringstream buf;
      buf.flags(ostr.flags());        // copy stream flags
      buf.fill(ostr.fill());          // copy fill character

      buf << r.numerator() << "/" << r.denominator();

      ostr << buf.str();
   }
   return ostr;
}
```


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(streamoff nbytes, seekdir 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(streamoff nbytes, seekdir base);
```

- Current value is returned by the `tellp()` function: `streampos tellp();`

Base Position Specification

- `base` can be `ios_base::beg` (beginning of file), `cur` (current position), `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...

Array Redesign — `FArray` Con/Destructors

- 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);
```


Array Redesign — `FArray` Other Member functions

- 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) {
   // check vality
   assert(size >= 0);

   // 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

Array Redesign — `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
```


Array Redesign — `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>
using namespace std;
#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);
};
```


Class Templates Member Definition outside of Class Definition

- 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)

Class Templates — stack.h

- 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;
```


Class Templates — General Remarks

- 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)