Introduction to C++

- Only seven lectures + exercises
- Examination through programming project
- Assume little experience of programming
- C++ language
- Object orientation, generic programming
- The standard library

Meeting Dates

Proposed: Thursdays at 10-12.
Room NB at Theoretical Physics is available most weeks.

Thu 20/3	Introduction. (pp 1–41)
Thu 27/3	Basic language features. (pp 42–76)
Thu 3/4	Object orientation. (pp 77–101)
Thu 10/4	Templates and namespaces. (pp 102–127)
Thu 24/4	The standard library. (pp 128–151)
Tue 29/4	Containers and iterators. (pp 152–188)
Thu 8/5	(K327) Tricks, pitfalls, general advice. (pp 189–212)
September	Deadline for projects.

The Examination Project

A three week course is not enough to learn C++. You need to get your hands dirty programming.

The project should be a program or a part of a program which is useful for your research and which shows you have understood C++ to some extent. Choose a project consulting with your supervisor and Carl.

(Note also that these are Leif's notes, lightly revised by Carl.

There is no real time limit on the project, but it should typically be finished after the summer / early fall.

We will be available for questions and advice until it is ready.
(and afterwards)

This course will give you an introduction to the C++ language. You will not be able to write good C++ programs after this course, but you will be able to learn on your own.

I will try to explain how C++ works *behind the scenes* to help you understand the why C++ was designed the way it was.

Object oriented analysis/design is a science of of its own. I will only give you the basic philosophy of object orientation.

There will be no course book, only these slides.

Use "The C++ Programming Language" fourth edition by Bjarne Stroustrup as a reference for the exercises and project.

There is a free electronic book: "Thinking in C++" 2nd edition by Bruce Eckel. It's got good reviews on Amazon but may be a bit outdated.

`http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html`

Other useful resources:

- General C++ info `http://www.accu.org/`
- C++ Guru of the week `http://www.gotw.ca`
- newsgroups: `comp.lang.c++.moderated`, `comp.std.c++`
- IRC: ##C++ and ##C++-beginners on Freenode, `irc.freenode.net`

You will need a C++ compiler

If you are on Linux, use g++ (which is part of GCC, the GNU Compiler Collection): `http://www.gnu.org/software/gcc/gcc.html`

If you have a Windows machine, you may want to install CygWin from `http://www.cygwin.com`
(don't forget to select gcc/g++ and an editor, e.g. Emacs)

What is C++

A computer language based on C but *more* (The meaning of '++' will become clear later.)

C++ is not simple, but it is versatile

Keywords are type safety, object orientation and generic programming.

Easy access to fundamental system calls but also easy to make high-level abstractions.

Try to detect as many bugs as possible at compile-time

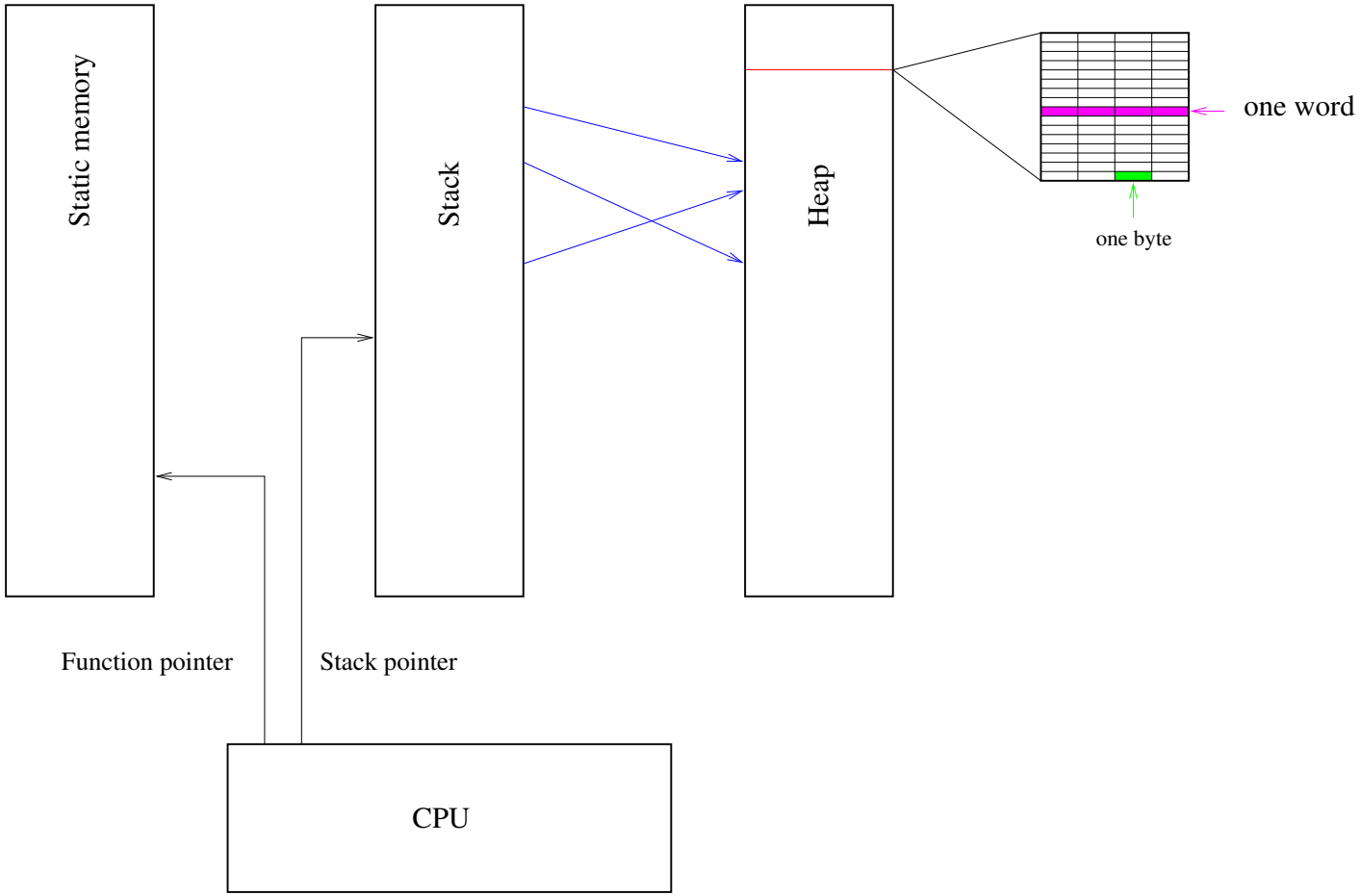
C++ has been standardized by ISO (in 1998, with a minor update in 2003). The next major update, C++11, is gradually becoming more widely implemented.

What is a computer

In Swenglish: Data machine – a machine for manipulating data.

A program tells the computer how to read data from a device (disk, keyboard, mouse, scanner, network, . . .) into the memory, manipulate it and write it to another device (disk, screen, printer, network, . . .).

The computers memory is logically divided into parts: Static memory with the instructions sequenced into functions (the program). The stack is used for locally created data and communication between functions. The heap is used for dynamically created global data.


Static memory
Stack
Heap
one word
one byte
Function pointer
Stack pointer
CPU

Data are made of (binary) bits. bits are grouped into bytes (typically 8 bits per byte). bytes are grouped into words (typically 4 or 8 bytes per word).

These words could mean anything, but there are some meanings of a word which are intrinsic to the computer, typically *an integer number*, *a floating point number*, *the address of another word*, or *an instruction*.

The basic instructions in a computer are very simple and can do stuff like *read a word from a device and store it at a specific address in memory*, *take two words at specified addresses and, treating them as floating point number add them and put the resulting word at some other address*, or *copy a number of words starting at some address to another address*.

When a program is run, instructions are read sequetially from the static memory and executed. To avoid repetition of instuctions, the program is divided into functions which can be called several times during the execution of the program.

A function is like a mathematical function, it takes arguments and it returns a value.

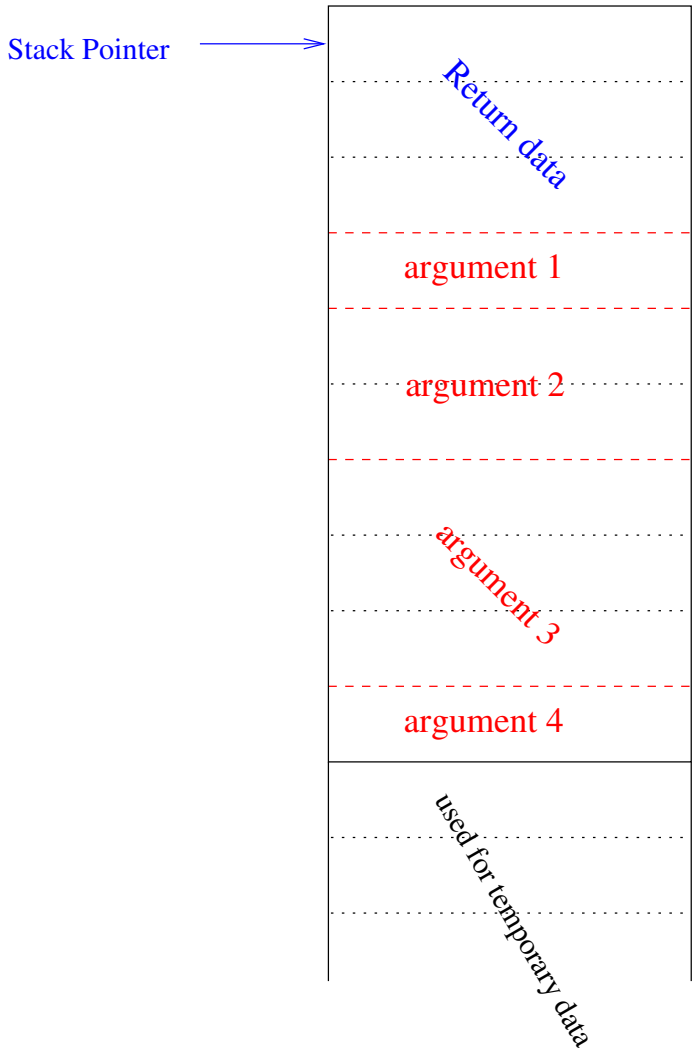
But it may also have *side effects*, such as writing on the screen or allocating data on the heap.

When executing a program the operating system calls the main function, the mother of all functions in a program.

Inside one function, the same instructions may be executed over and over in a loop.

When one function calls another function, the following has to be done

- *push* the value of the current stack pointer to the stack
- push the address of the next instruction to be executed after the function call
- push the values of the arguments to the stack
- make room on the stack for the called function to place its return value
- set the function pointer to the address containing the first instruction of the function to be called.



The called function only knows the address on the stack where the arguments are stored. The calling function only knows the address on the stack where the return value is placed.

The called function has no way of knowing if the calling function has put the right number and right type of arguments on the stack. The calling function has no way of knowing that the called function has put the right type of return value on the stack. It is the responsibility of the programmer to ensure that this works. Some languages helps the programmer. C++ has *strong type checking*.

A programming language is meant to help the programmer to formulate his problem in a way the computer can handle it.

The early languages had a structure very close to the computer it self and only gave a little help to the programmer.

E.g. Fortran defines a few new data types (complex numbers, character string, vectors/arrays of data types) and operations on them. Just as the computer as a whole. Fortran (and C) can be defined in term of a set of functions (or *subroutines*) acting on a common set of data (*common blocks*). Fortran does not enforce type checking.

If all you have is a hammer, any problem will look like a nail.

Most problems are not most efficiently formultated in terms of *functions acting on a global pool of data*.

C++ allows you to formulate problems in terms of user-defined abstract data types and user-defined operations on them and relations between them – the data structures have functions associated with them.

In any software design process, you begin by stating the problem.

In old languages you then look at the verbs in the statement and identify them with functions.

With object oriented programming you start by looking at the nouns and then associate them with user-defined types. Then you look at the relationship between different *objects* and different *classes* of objects.

In the end what the computer deals with is functions acting on a global pool of unspecified data.

The structure of a C++ program

A C++ program usually consists of several header files and source files.

You declare data types and functions in a header file (`filename.h`)

You define (implement) functions in a source file (`filename.cc`). To ensure that all arguments and return values are correct, you `#include` the relevant header file in the source file.

You compile each source file into an object file (`filename.o`). The source file and the headers it `#include`s make up one translation unit (TU).

You link all object files you need for a program into an executable file (`a.out`).

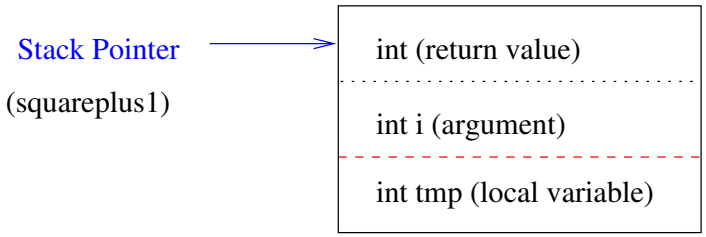
A stupid example:

```
//                                increment.h

int increment(int i);
// declares a function 'increment' taking
// an integer as argument and returning
// another integer
```

```
//                                squareplus1.h

int squareplus1(int i);
// declare a function 'squareplus1' taking
// an integer argument and returning
// another integer.
```

```
//                                increment.cc
#include "increment.h"

int increment(int i) {
  return i+1;
}
// implements the function 'increment'
// simply returning the value of the
// argument plus one.
```

```
//                                squareplus1.cc
#include "squareplus1.h"
#include "increment.h"

int squareplus1(int i) {
  int tmp = increment(i);
  return tmp*tmp;
}
// implement the function 'squareplus1'.
// First create a temporary integer giving
// it the value of the argument plus one,
// then return the square of the temporary
// integer
```



Compiling and running a main program:

```
//                                          main1.cc
#include "squareplus1.h"

int main() {
  return squareplus1(2);
}
// A main program calculating the square of
// two plus one. 'main' is always
// implicitly declared for you.
```

```
% g++ -c increment.cc
% g++ -c squareplus1.cc
% g++ -c main1.cc
% g++ main1.o squareplus1.o increment.o
% ./a.out
% echo $?
9
%
```

A (slightly less stupid) C++ program:

```
//                                        hello.cc
#include <iostream>

int main() {

  std::cout << "Hello world!" << std::endl;
  return 0;

}
```

```
% g++ -c hello.cc
% g++ hello.o -o hello
% ./hello
Hello world!
%
```

Everything after `//` on a line is treated as a comment and is disregarded by the compiler.

There are also so-called block comments (`/* a comment */`)
Use them with care

Every line starting with `#` (such as `#include`) is a pre-processor directive. The preprocessor is run before the proper compilation, and does things like including header files and removing comments.

The C++ compiler comes with a standard library including a standardized set of types and functions. The corresponding object files are normally linked automatically, but you must include the relevant header files: e.g. `#include <iostream>` or `#include <cmath>` if you want to use them.

E.g. the `cmath` header file contains function declarations such as

```
double std::sqrt(double);
double std::sin(double);
```

etc.

Some code snippets

```
std::vector<long> data(10);           // declare a vector of 10 long integers
long sum = 0;                         // declare a long integer set to 0
for ( int i = 0; i < 10; ++i ) {      // loop over the elements of the
  data[i] = i*(i-1);                  // vector set them to something
  sum += data[i];                     // and add them together in sum
}
std::cout << sum << std::endl;        // Write out the sum

std::string name = "John";            // declare a character string and set it
name += " Smith";                     // concatenate the charcater string
std::cout << "Length of " << name     // write out the full name
          << " is " << name.length()
          << std::endl;
```

C++ code is divided up in *statements*. A statement can run over several lines and is terminated with a semicolon `;`

A *compound statement* is a sequence of statements enclosed in curly brackets {}

All variables, types and functions must be declared before they are used. A *declaration* tells the compiler what something is, whereas a *definition* in some sense actually creates it. It is common with statements that both declare and define a variable.

An *identifier* (name of a variable, function or type) is a letter (a-z,A-Z) followed by any number of letters, digit and underscores. (`::` is the scope resolution operator and will be decribed later)

Some names are *keywords* and may not be used as identifiers, e.g. `long`, `this`, `operator`, `template`, `public`, `for`, ...

The following fundamental data types are pre-defined:

`int`	`char`
`long (int)`	`(un/signed) char`
`short (int)`	`float`
`unsigned (int)`	`double`
`unsigned long (int)`	`long double`
`unsigned short (int)`	`bool`
`(unsigned) long long (int)`	`void`

`void` is not really a data type – its merits will become clear later

The `sizeof` operator will give the size of a type or an object in bytes. E.g. `sizeof(char)` is 1 (by definition), `sizeof(int)` is typically 4 (but machine dependent). But look out! If `foo` is a pointer, `sizeof(foo)` is the size of the pointer, not the size of the object it points to.

Things you can do with fundamental types

```
int i = 1;
long j = (i+2)*3;
double d = 4.0;
float f = float(sqrt(d));
j += i++;                   // add i to j and then increment i
j *= --i;                   // decrement i and then multiply j with i
d = f+3.0*i;                // equivalent to f+(3*i) as expected
if ( d == 5.0 ) i = j%2;    // % is the modulus operator
```

==, +, ++, +, %, etc. are operators. Operators have different precedence and bind in different directions e.g.

`a+b*c` means `a+(b*c)` not `(a+b)*c` and

`a+b+c` means `(a+b)+c`

On unix/linux, the operator man page (`man 7 operator`) is a convenient list of all the operators.

If you need integers of a certain size, C99 provides useful new types, found in header <cstdint>

int8_t	uint8_t
int16_t	uint16_t
int32_t	uint32_t
int64_t	uint64_t

`if ( condition-expression ) statement;`

If `condition-expression` is true (the expression gives a non-zero value), execute `statement`.

`if ( condition-expression ) statement1; else statement2;`

If `condition-expression` is true, execute `statement1` otherwise execute `statement2`.

```
double d;
std::cin >> d;
if ( d > 0.0 ) d = sqrt(d);
else d = d*d;

if ( d < 10 )
  if ( d == 1.0 ) std::cout << "one";
  else {
    std::cout << d;
  }
```

`else` always refers to the last `if`. (So indent your code well and be generous with {}!)

Looping in C++

`for ( statement1; condition; statement2 ) statement3;`

First execute `statement1`, then if `condition` is true, execute `statement3` and `statement2`. If `condition` is still true execute `statement3` and `statement2` again, etc. `statement3` may be a single statement or several encosed in {}.

```
int v[20];                          // read in integers from terminal
                                    // into vector 'v'.
for ( int i = 0; i < 20; ++i ) { // At most 20 objects should be read
  v[i] = 0;
  if ( i == 13 ) continue;          // Don't assign to v[13]
                                    // (could mean bad luck)
  std::cin >> v[i];
  if ( v[i] == 0 ) break;           // Stop reading if we read a 0
}
if ( i < 10 ) v[10] = 0;            // ERROR 'i' is not defined
                                     // outside of the loop.
```

`continue` skips the rest of `statement3`. `break` leaves the loop.

`while (condition) statement;`

Check `condition`. If it is true, execute `statement`. Then check `condition` again and execute `statement` if it is true. Continue until `condition` is false.

```
int v[20]
int i = 0;
while ( i < 20 ) {
  v[i] = 0;
  if ( i == 13 ) continue;    // OOOPS! loop is continued but 'i'
                              // is not incremented
  std::cin >> v[i];
  if ( v[i] == 0 ) break;
  ++i;
};
if ( i < 10 ) v[10] = 0;      // OK, 'i' is declared outside of loop
```

`do statement while (condition);`

Execute `statement`. Then if `condition` is true execute `statement` again, etc.

```
int v[20]
int i = 0;
do {
  v[i] = 0;
  if ( i == 13 ) continue;   // OK, loop is continued and 'i'
                             // is incremented in the while condition
  std::cin >> v[i];
  if ( v[i] == 0 ) break;
} while ( ++i < 20 );
if ( i < 10 ) v[10] = 0;     // OK,
```

`switch ( expression ) statement;`

Evaluate `expression` jump to the corresponding `case` label (or the optional `default` label if no match is found) in `statement` and continue execution from there

```
int i;
std::cin >> i;
switch (i) {
  case 0:
    std::cout << "none";
    break;
  case 1:
    std::cout << "one";
    break;
  case 2:
  case 3:
  case 4:
    std::cout << "a hand-full";
    break;
  default:
    std::cout << "a lot";
}
```

```
int i;
std::cin >> i;
if ( i == 0 ) {
  std::cout << "none";
}
else if ( i == 1 )
  std::cout << "one";
else if ( i == 2 || i == 3 || i == 4 )
  std::cout << "a hand-full";
else
  std::cout << "a lot";
```

The case label must be a constant expression because it is evaluated at compile time. Note also the importance of `break` between cases.

Functions:

```
double sqr(double d) {
  return d*d;
}

unsigned long fact(unsigned long n) {
  if ( n < 2 ) return 1;
  else return n*fact(n-1);
}

int sqr(int i) {
  return i*i;
}
```

```
void print(double d) {    // returns 'void'
  std::cout << d;         // i.e. nothing
  return;                 // Not needed at
}                         // end of function.

void printendl(void) {    // Takes no args.
  std::cout << std::endl; // (void) the same
}                         // as just ()

int i = 3;
double d = 5;

long j = fact(i);
double d2 = sqr(d); // calls sqr(double)
print(d2);
printendl();
sqr(i);             // calls sqr(int) return
                    // value disregarded
```

Functions may be *recursive*, as `fact` above. `sqr(int)` and `sqr(double)` are two different functions. When you call `sqr(something)`, the compiler will choose the relevant one by examining the type(s) of the argument(s).

Another fundamental data type is a pointer. A pointer is an address of a word/byte in the memory, with information of the type of the data at that address.

```
int j = 1;
int *jp;            // Declare a pointer 'jp' to an int
jp = &j;            // jp now points to the memory location of 'j'
int i = 1 + *jp;    // '*jp' gives the value of the int 'jp' is
                    // pointing to (dereferencing)
double *dp = jp;    // ERROR pointer to double poining to an int.
void *vp = &i;      // 'void*' can point to anything - it is just an address
j = *vp + 1;        // ERROR we cannot dereference 'void*' since we
                    // do not know the type.
dp = (double*) vp; // OOPS dp now point to 'j. Allowed, but result undefined
```

`&variable` means *the address of* `variable`

`*pointer` means *the value of the data* `pointer` *points to*.

Pointers and arrays

```
double darr[10];                // declare an array of ten doubles
                                // consecutive in the memory
double * dv = darr;             // 'darr' is in fact a pointer to the first
                                // element in the array.
darr[0] = 1;                    // set the first (zero'th) element
darr[1] = *darr + 1.0;          // '*darr' is the same as 'darr[0]'
dv += 2;                        // 'dv' now points to the third element
*dv = darr[0] + *(darr+1);      // 'darr[i]' is the same as '*(darr+i)'
std::cout << darr << " "
          << dv << std::endl; // prints out the actual addresses in hex
                                // form.
```

Looking at the output from the last line, the difference between the two pointers will be two times the number of bytes a double occupies on the current machine.

```
void swap(double * p, double * q) {  // A function taking two pointers
  double tmp = *p;                   // to double as arguments and
  *p = *q;                           // returns nothing.
  *q = tmp;
}

void noswap(double p, double q) {    // This function would not do
  double tmp = p;                    // anything
  p = q;
  q = tmp;
}

double d1 = 1.0;
double d2 = 2.0;
swap(&d1, &d2);
std::cout << d1 << " "
          << d2 << std::endl;        // The output will be '2 1'
noswap(d1, d2);                      // Only swaps local copies of
                                     // d1 and d2
```

C++ has *call by value* semantics of functions.

References:

```
int i = 1;
int * ip = &i;          // 'ip' is a pointer to 'i'
int & ir = i;           // 'ir' is now an alias or reference for 'i'
*ip = 2;                // 'ir', '*ip' and 'i' now gives '2'
ir = 3;                 // 'ir', '*ip' and 'i' now gives '3'
int j = ir;             // Also 'j' is 3
j = 4;                  // 'i', '*ip' and 'ir' still gives 3
```

```
void swap(double & p, double & q) {  // A function taking two references
  double tmp = p;                    // to double as arguments and
  p = q;                             // returns nothing.
  q = tmp;
}

double d1 = 1.0;
double d2 = 2.0;
swap(d1, d2);                        // swaps the values of 'd1' and 'd2'
```

A reference is like a pointer which is 'permanently dereferenced'. In compiled form, the function `swap` here expects two arguments containing the addresses of the data in `d1` and `d2`. The compiled code of `swap(double *, double *)` would typically be identical to `swap(double &, double &)`

A reference can be used as a return value:

```
int & element(int * iarr, int i) {  // Returns a reference to the i'th
 return iarr[i];                    // element of 'iarr'. Remember
}                                   // iarr[i] is the same as *(iarr+i)

int v[10];
element(v, 3) = 4;                  // 'v[3]' is now 4
```

You can have function calls on either side of an = sign.

Anything which can be on the left side of an = sign is called an lvalue. Any other expression is called an rvalue.

The `const` keyword

```
const int ci = 5;          // Declare a variable which may not be changed
ci = 3;                    // ERROR try to change the value of a const variable
int & ir = ci;             // ERROR try to bind a normal reference to a const
const int & cir = ci;      // OK 'cir' is a 'const' reference
const int * cip = &ci;     // OK 'cip' is a pointer to const
cir = 3 ;                  // ERROR try to change a const reference.

double x = 1.0;
const double & cxr = x;    // OK a const reference may refer to a non const
x = 3.0;                   // Both 'x' and 'cxr' gives 3.0
cxr = 1.0;                 // ERROR cannot change the value through a
                           // const reference

double & d = 1;            // ERROR '1' is const. normal reference must be
                           // bound to an lvalue
const double & cdr = 1;    // OK cdr is a const reference
```

In the last line, what happens behind the scenes is equivalent to

```
const double tmp = double(1);
const double & cdr = tmp;
```

First Exercise

1. Compile and run the `hello.cc` program.

2. Write, compile and run a small program using the `sizeof` operator to find out the sizes of the fundamental types including pointers on your computer and print them out. Also examine `sizeof("Hello")` and `sizeof(char[5])`.

3. Write a program that prints the numbers between 1 and 100 (one per line). Modify it so that multiples of 3 are replaced with "Fizz", multiples of 5 with "Buzz" and multiples of both with "FizzBuzz".

User defined types:

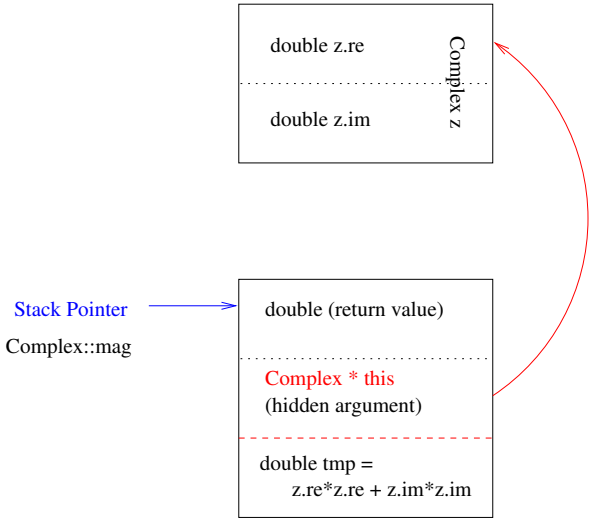
```
class Complex {     // Define a new type called 'Complex'
public:
  double re;        // with two double members re and im.
  double im;
};

Complex z;          // z is a variable of type Complex
z.re = 1.0;         // access the individual members of
z.im = 0.2;         // the Complex variable
Complex w;
w = z;              // Memberwise copy.
Complex * zp = &z;  // 'zp' is a pointer to a Complex
zp->im = zp->re;    // Access the members from a pointer.
                    // equivalent to (*zp).im = (*zp).re;
Complex u = w + z;  // ERROR '+' operator not defined for
                    // Complex.
```

A user defined type is called a `class`. An instance of a class is called an **object**. Instances of fundamental types may also be referred to as objects.

C has `struct`s but not classes. In C++ we may speak of POD (plain old data) types – objects that are just an aggregate of data (like this example).

User defined types can also have function members:

```
class Complex {
public:
  double re;
  double im;

  double mag() {                  // A member function
    return sqrt(re*re + im*im); // returning the absolute
  }                               // value of the complex.
};

Complex z;
z.re = 3.0;
z.im = 5.0;
Complex * zp = &z;                 // zp points to z
zp->im = 4.0;                      // z.im is now 4.0
double d = z.mag();                // d should now be 5.0
double d2 = zp->mag();              // d2 is also 5.0
```



The compiled function corresponding to `mag` typically looks like `double mag(Complex * this)` and is called with a pointer to the Complex varaible. In fact, in any member function, you have a pointer to the object it is called for implicitly declared with the name `this`.

A few special member functions are implicitly defined also for User defined types

```
class Complex {
public:
  double re;
  double im;

  Complex() :
    re(0.0), im(0.0) {}
  Complex(const Complex & z) :
    re(z.re), im(z.im) {}

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

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

```
Complex z;
// uses Complex(),
// setting all members to 0

Complex v(1.0, 3.1);
// uses Complex(double, double)
// 'v.re' is 1.0, 'v.im' is 3.0

Complex w(z);
Complex u = z;
// both uses
// Complex(const Complex &)

z = u;
// uses operator=(const Complex &)
```

A member function with the same name as the name of the class is called a constructor (a.k.a. *ctor*).

`Complex()` is the default constructor. By default you get an empty one, which does not initialize the member variables. Defining any constructor kills the empty default constructor.

`Complex(const Complex &)` is the copy constructor. If it is not defined, the values of the members will be copied memberwise.

`operator=(const Complex &)` is the assignment operator. If it is not defined, data will be assigned memberwise.

You have the right to remain silent. You have the right to a copy constructor and an assignment operator. If you cannot afford to write these, they will be written for you by the compiler.

```
Complex operator+(Complex a, Complex b) {   // Define the behavior
  Complex r;                                // of the + operator for
  r.re = a.re + b.re;                       // Complex
  r.im = a.im + b.im;                       // (as a free function,
  return r;                                 // not a class member)
}

Complex z, w, u;
u = w + z;                                  // This is now OK
u = operator+(w, z);                        // This is equivalent
```

Operator overloading: We can define operators for Complex.

Operators can be defined to do anything a function can do, but you cannot change the number of arguments or the precedence of operators: `a+b*c` will always be interpreted as `operator+(a,operator*(b,c))` and not `operator*(operator+(a,b),c)`.

Operator overloading is useful, but only in certain situations:

Mathematical operators for mathematical objects (e.g. arbitrary precision numbers).

`operator<<` and `operator>>` when working with input/output streams.

`operator[]` for accessing an element in an array-like class.

`operator()` in function objects (functors) – we will get to these in later lectures.

Declarations and implementations in different files

```
//                                    Complex.h
class Complex {
public:
  Complex();
  // Default constructor
  Complex(const Complex &);
  // Copy constructor

  Complex(double r, double i);
  // Constructor from doubles

  Complex & operator=(const Complex &);
  // Assignment operator

  Complex operator+(const Complex &) const;
  // Addition operator (member version)

  double mag() const;
  // Normal member function

  double re;
  double im;
};
```

```
//                                    Complex.cc
#include "Complex.h"

Complex::Complex() : re(0.0), im(0.0) {}

Complex::Complex(const Complex & z)
  : re(z.re), im(z.im) {}

Complex & Complex::
operator=(const Complex & z) {
  re = z.re;
  im = z.im;
  return *this;
}

Complex Complex::
operator+(const Complex & z) const {
  Complex sum;
  sum.re = re + z.re;
  sum.im = im + z.im;
  return sum;
}

double Complex::mag() const {
  return sqrt(re*re+im*im);
}
```

Outside of the class declaration the member functions are accessed using the *scope resolution* operator: `Complex::mag()`.

The `const` qualifier in `double mag() const;` means that the function does not change the object in any way. The compiled function will behave like `double mag(const Complex * this);`

Inside a member function, the data members of an object can be referred to by name. `this->re = 2.0;` is allowed, but is equivalent to `re = 2.0;`. (In a const member function `re = 2.0;` is not allowed, because the implicit `this` pointer points to a const object.)

The member `Complex::operator+(const Complex &)` is preferred before the *global* `operator+(const Complex &, const Complex &)`. `a + b` is translated to `a.operator+(b)`

Function calls are somewhat expensive. If a function is declared `inline` and the compiler can see the function body, it may choose to expand the function within the calling function to avoid expensive copying of argument and return values.

```
class Complex {
  public:
  double mag() const {
    return sqrt(re*re + im*im);
  }
  inline double phase() const;

  double re;
  double im;
};

inline double Complex::phase() const {
  return atan2(re,im);
}
```

Member functions which are defined (implemented) within a class declaration are implicitly declared `inline`. Note that the function body must be visible everywhere the function is called and can therefore not be placed in the `.cc` file.

Private members:

```
class Complex {
public:
  inline double real() const;   // return the real part
  inline double imag() const;   // return the imaginary part
  inline double mag() const;    // return the magnitude
  inline double phase() const;  // return the phase

  inline void setReal(double);  // set the real part
  inline void setImag(double);  // set the imaginary part
  inline void setMag(double);   // set the magnitude
  inline void setPhase(double); // set the phase

private:
  double re;
  double im;
};

Complex z;
z.re = 1.0;                     // ERROR 're' is private
z.setReal(1.0);
double d1 = z.im;               // ERROR 'im' is private
double d2 = z.imag();
```

WHY?

WHY?

This is encapsulation and the essence of data abstraction. Isolate the interface to a data type from its representation.

A complex number is defined by how it behaves under mathematical operations, not by the fact that it can be represented by two real numbers on the form $(x + iy)$. In fact, a complex number may just as well be represented by two other real numbers representing the magnitude and the phase $r \cdot e^{i\phi}$.

If the data members are private, we ensure that the outside functions using compex number are not depending on the underlying representation. So if we wish to change the underlying representation no other code using `Complex` need to be changed (only re-compiled).

```
class Complex {
public:
// same as above ...
private:
  double re;
  double im;
};

double Complex::real() const {
  return re;
}

double Complex::phase() const {
  return atan2(re, im);
}

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

void Complex::setMag(double r) {
  double phi = phase(); // leave phase
  re = r*cos(phi);      // intact
  im = r*sin(phi);
}
```

```
class Complex {
public:
// same as above ...
private:
  double r;
  double phi;
};

double Complex::real() const {
  return r*cos(phi);
}

double Complex::phase() const {
  return phi;
}

void Complex::setImag(double im) {
  double re = real();      // leave real
  r = sqrt(re*re + im*im); // part intact
  phi = atan2(re, im);
}

void Complex::setMag(double m) {
  r = m;
}
```

So far we have only created data on the stack - so called automatic variables. Such variables are automatically destroyed in the end of the enclosing *scope*.

```
double & somefunction(long i) {
  int j = 1;                          // j is created here
  if ( i < 0 ) {
    double d = double(i+3*j);         // d is created here
  }                                   // d is destroyed here
                                      // and the memory can be
                                      // used for other things
  double f = 0.0;                     // f is created here (could use
                                      // the same memory as d).
  if ( i < 0 ) {
    f = double(i+3*j);
  }

  return f;                           // PROBLEM: f is still around
                                      // before the return statement
}                                     // but is destroyed after the
                                      // function returns. The calling
                                      // function is given a reference
                                      // to a non existent variable.
```

The *scope* is typically limited by {}

Dynamic memory – free store

It is possible to allocate data on the *free store* (a.k.a. *heap*). This will live until it is explicitly destroyed.

```
Complex * cp =
   new Complex(1.0, 1.0);  // Create a Complex on the heap, construct it
                           // as 1+i and return a pointer to it.
cp->setReal(3.0);          // 'cp' is used like any other pointer.
delete cp;                 // Destroy the complex pointed to by 'cp'
                           // and allow the memory to be used for
                           // something else.

cp = new Complex[10];      // Create ten Complexes on the heap and
                           // return a pointer to the first element
cp[4] = Complex(1.0, 1.0); // used like any other array
delete [] cp;              // Delete all ten Complexes (note [])

Complex & tricky() {
  Complex * cp =
    new Complex(1.0, 0.0);
  return *cp;              // This is fine. The pointer is destroyed
}                          // but the Complex it pointed to is still
                           // around. WARNING: who shall delete the
                           // Complex?
```

Case study: Character Strings

C++ has inherited rudimentary support for character strings from C, based on *zero-terminated arrays of characters*

```
const char * h = "Hello";     // 'h' now points to an array of six characters,
                              // where the last 'h[5]' is the null char '\0'
const char * w = "World";     // 'w' points to another array.

char * hw = h + w;            // ERROR you cannot add two pointers.
char hw[11];                  // 'hw' points to an array of size 11.

strcpy(hw, h);                // 'strcpy' is a standard C function. Copies the
                              // characters (including final '\0') from the array
                              // pointed to by 'h' to the one pointed to 'hw'
strcat(hw, w);                // Copies the characters from the array pointed
                              // to by 'w' to the one pointed to by 'hw' but
                              // starting at the first occurence of '\0' in 'hw'
                              // 'hw' now points to an array with 'HelloWorld\0'
                              // Note that we have to allocate the size of 'hw'
                              // to include the '\0' and we have to know the
                              // length in advance.
```

Very cumbersome and non-intuitive.

Let's try to write a character string class encapsulating the C character array.

```
class String {
public:

  String();
  // Default constructor

  String(const String &);
  // Copy constructor

  String(const char *);
  // Constructor from C-string

  ~String();
  // Destructor

private:

  char * cstring;
  // The internal representation.

  int len;
  // Number of chars in the string.

};
```

```
String::String() :
  cstring(new char[1]), len(0) {
  cstring[0] = '\0';
}
// Initialize with 'empty' string. len is 0

String::String(const String & s) :
  cstring(new char[s.len+1]), len(s.len) {
    strcpy(cstring, s.cstring);
}
// Allocate enough room, and copy characters.

String::String(const char * cs) :
  len(strlen(cs)) {
  cstring = new char[len+1];
  strcpy(cstring, cs);
}
// Allocate enough room, and copy characters.

String::~String() {
  delete [] cstring;
}
// Will be called each time a 'String' object
// is destroyed.
```

```
class String {
public:

// ...

String & operator=(const String &);
// Assignment from other string.

String & operator=(const char *);
// Assignment from C-string

String & operator+=(const String &);
String & operator+=(const char *);
// Concatenate this string with
// another.

String operator+(const String &) const;
String operator+(const char *) const;
// Concatenate two strings.

char operator[](int) const;
// Access a character.

int length() const;
//return the length of the string.

};
```

```
String & String::operator=(const String & s) {
  if ( this == &s ) return *this;
  // Avoid self copy.
  delete [] cstring;
  len = s.len;
  cstring = new char[len+1];
  strcpy(cstring, s.cstring);
  return *this;
}

String & String::operator=(const char * cs) {
  delete [] cstring;
  len = strlen(cs);
  cstring = new char[len+1];
  strcpy(cstring, cs);
  return *this;
}

String & String::operator+=(const String & s) {
  char * newstr = new char [len + s.len + 1];
  strcpy(newstr, cstring);
  strcat(newstr, s.cstring);
  delete [] cstring;
  cstring = newstr;
  len += s.len;
  return *this;
}
```

```
String String::operator+(const String & s) const {
  String r = *this;
  r += s;
  return r; // Note: return by value here.
}

char String::operator[](int i) const {
  return cstring[i];
}

int String::length() const {
  return len;
}
```

```
String h = "Hello";
String w = "World";
String hw;
hw = h;
hw += " ";
hw += w + "!";      // hw.cstring is now 'Hello World!\0'
char a = hw[0];
hw[5] = 'w';        // ERROR operator[] returns an rvalue
```

Some lessons learned:

- If the creation of an object involved allocation of resources, you must provide a destructor (a.k.a *dtor*) to do the de-allocation when the object is destroyed.

- Care must be taken in assignment operators to avoid problems with self-assignment.

- The privacy of members is on *per class* basis. An object of a class may access the private members of another object of the same class in its member functions.

- Never use `C`-style character strings with zero-terminated arrays. `C++` has a standard string class, `std::string`, which you access by `#include <string>`

Case study 2: Vector3

```
//                                        Vector3.h
#ifndef Vector3_H
#define Vector3_H

class Vector3 {

public:

  Vector3(double xx = 0.0, double yy = 0.0,
          double zz = 0.0);
  Vector3(const Vector3 &);
  // The constructors.

  ~Vector3();
  // The destructor.

  double x() const;
  double y() const;
  double z() const;
  // The components in cartesian
  // coordinate system.

  void x(double);
  void y(double);
  void z(double);
  // Set the components in cartesian
  // coordinate system.

  double phi() const;
  // The azimuth angle.

  double theta() const;
  // The polar angle.

  double mag() const;
  double mag2() const;
  // The magnitude (squared)

  double perp() const;
  double perp2() const;
  // The transverse component (squared)
```

```
void phi(double);
// Set phi keeping mag and theta constant

void theta(double);
// Set theta keeping mag and phi constant

void mag(double);
// Set mag keeping theta and phi constant

Vector3 & operator=(const Vector3 &);
// Assignment.

Vector3 & operator+=(const Vector3 &);
Vector3 operator+(const Vector3 &) const;
// Addition.

Vector3 & operator-=(const Vector3 &);
Vector3 operator-(const Vector3 &) const;
// Subtraction.

Vector3 operator-() const;
// Unary minus.
```

```
Vector3 & operator*=(double);
Vector3 operator*(double) const;
// Scaling with real numbers.

double dot(const Vector3 &) const;
// Scalar product.

Vector3 cross(const Vector3 &) const;
// Cross product.

double angle(const Vector3 &) const;
// The angle w.r.t. another 3-vector.
```

```
  void rotateX(double);
  void rotateY(double);
  void rotateZ(double);
  // Rotates the Vector3 around the
  //  x-, y- and z-axis.

  void rotate(double, const Vector3 &);
  // Rotates around the axis specified
  // by another Vector3.

private:

  double dx;
  double dy;
  double dz;
 // The components.
};


Vector3 operator*(double a, const Vector3 &);
// Scaling of 3-vectors with a real number

#endif /* HEP_THREEVECTOR_H */
```

```
//                                  Vector3.cc
#include "Vector3.h"

#include <cmath>

Vector3::Vector3(double xx, double yy,
                 double zz)
: dx(xx), dy(yy), dz(zz) {}

Vector3::Vector3(const Vector3 & p)
: dx(p.x()), dy(p.y()), dz(p.z()) {}

Vector3::~Vector3() {}

double Vector3::x() const {
  return dx;
}

double Vector3::y() const {
  return dy;
}

double Vector3::z() const {
  return dz;
}
```

```
void Vector3::x(double xx) {
  dx = xx;
}

void Vector3::y(double yy) {
  dy = yy;
}

void Vector3::z(double zz) {
  dz = zz;
}

double Vector3::phi() const {
  return x() == 0.0 && y() == 0.0 ?
         0.0 : std::atan2(y(),x());
}

double Vector3::theta() const {
  return
    x()==0.0 && y()==0.0 && z()==0.0 ?
    0.0 : std::atan2(perp(),z());
}

double Vector3::mag2() const {
  return x()*x() + y()*y() + z()*z();
}
```

```
double Vector3::mag() const {
  return std::sqrt(mag2());
}

double Vector3::perp2() const {
  return x()*x() + y()*y();
}

double Vector3::perp() const {
  return std::sqrt(perp2());
}

void Vector3::theta(double th) {
  double ma   = mag();
  double ph   = phi();
  x(ma*std::sin(th)*std::cos(ph));
  y(ma*std::sin(th)*std::sin(ph));
  z(ma*std::cos(th));
}

void Vector3::phi(double ph) {
  double ma   = mag();
  double th   = theta();
  x(ma*std::sin(th)*std::cos(ph));
  y(ma*std::sin(th)*std::sin(ph));
  z(ma*std::cos(th));
}
```

```
void Vector3::mag(double ma) {
  double th = theta();
  double ph = phi();
  x(ma*std::sin(th)*std::cos(ph));
  y(ma*std::sin(th)*std::sin(ph));
  z(ma*std::cos(th));
}

Vector3 & Vector3::operator=(const Vector3 & p) {
  x(p.x());
  y(p.y());
  z(p.z());
  return *this;
}

Vector3 & Vector3::operator+=(const Vector3 & p) {
  dx += p.x();
  dy += p.y();
  dz += p.z();
  return *this;
}

Vector3 Vector3::operator+(const Vector3 & p) const {
  Vector3 ret = *this;
  ret += p;
  return ret;
}
```

```
Vector3& Vector3::
operator-=(const Vector3 & p) {
  dx -= p.x();
  dy -= p.y();
  dz -= p.z();
  return *this;
}

Vector3 Vector3::
operator-(const Vector3 & p) const {
  Vector3 ret = *this;
  ret -= p;
  return ret;
}

Vector3 Vector3::operator-() const {
  return Vector3(-dx, -dy, -dz);
}

Vector3& Vector3::operator*=(double a) {
  dx *= a;
  dy *= a;
  dz *= a;
  return *this;
}
```

```
Vector3 Vector3::
operator*(double a) const {
  Vector3 ret = *this;
  ret *= a;
  return ret;
}

double Vector3::
dot(const Vector3 & p) const {
  return x()*p.x() + y()*p.y() + z()*p.z();
}

Vector3 Vector3::
cross(const Vector3 & p) const {
  return Vector3(y()*p.z()-p.y()*z(),
                 z()*p.x()-p.z()*x(),
                 x()*p.y()-p.x()*y());
}

double Vector3::
angle(const Vector3 & q) const {
  double ptot2 = mag2()*q.mag2();
  return ptot2 <= 0.0 ? 0.0 :
         std::acos(dot(q)/std::sqrt(ptot2));
}
```

```
void Vector3::rotateX(double angle) {
  double s = std::sin(angle);
  double c = std::cos(angle);
  double yy = y();
  y(c*yy - s*z());
  z(s*yy + c*z());
}

void Vector3::rotateY(double angle) {
  double s = std::sin(angle);
  double c = std::cos(angle);
  double zz = z();
  z(c*zz - s*x());
  x(s*zz + c*x());
}

void Vector3::rotateZ(double angle) {
  double s = std::sin(angle);
  double c = std::cos(angle);
  double xx = x();
  x(c*xx - s*y());
  y(s*xx + c*y());
}
```

```
void Vector3::rotate(double angle,
                     const Vector3 & axis) {
  rotateZ(-axis.phi());
  rotateY(-axis.theta());
  rotateZ(angle);
  rotateY(axis.theta());
  rotateZ(axis.phi());
}

Vector3 operator * (double a,
                    const Vector3 & p) {
  return Vector3(a*p.x(), a*p.y(), a*p.z());
}

Vector3 p1(1, 1);    // p1.dz is 0 by default
p1.rotateX(3.14);
Vector3 p2 = 3.0*p1; // Uses global
                     // operator*(double,Vector3)
Vector3 p3(1.0, 1.0, 1.0);
p3.rotate(-3.14, p1);
```

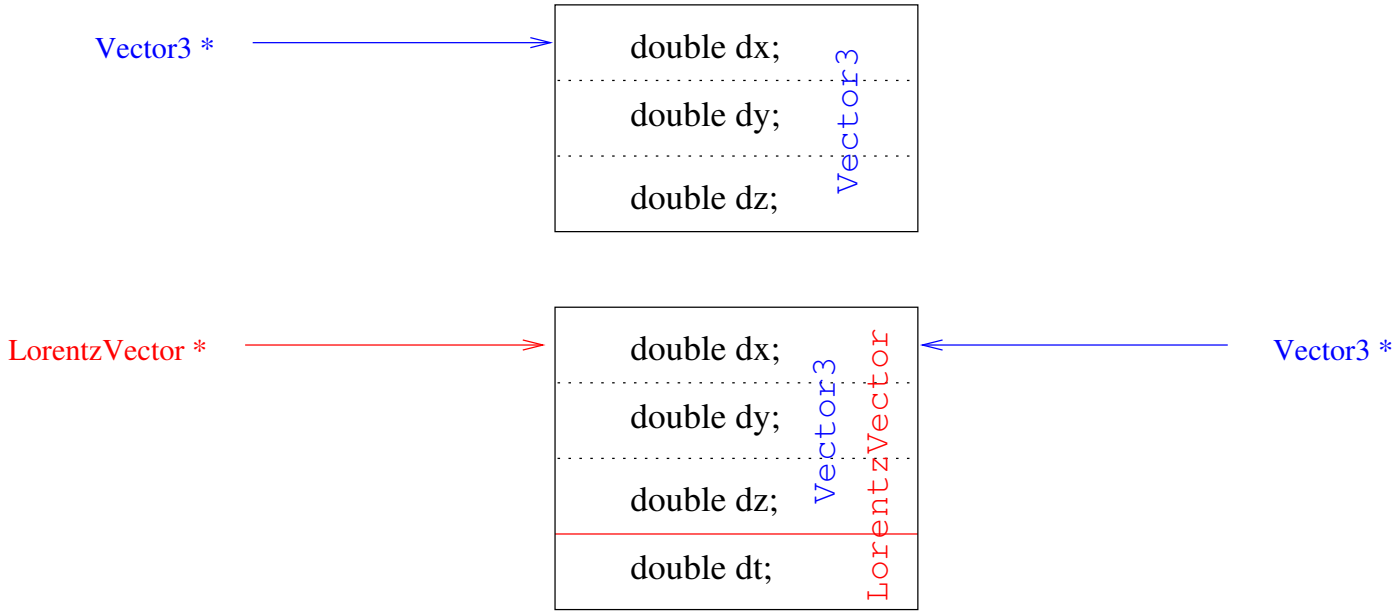
Case 2 continued: `LorentzVector`

A Lorentz vector can be thought of as a 3-vector with an extra time component. Anything you can do with a 3-vector you can also do with the space-like components of a Lorentz vector. A Lorentz vector **Is a** 3-vector.

```
class Vector3 {
  // ...
};

class LorentzVector: public Vector3 {

private:

  double dt;

};
```

`LorentzVector` **inherits** the properties of `Vector3`. `Vector3` is a **base class** of the **derived class** `LorentzVector`.

A pointer to an object contains the address of the first (data-) member of that object*. A pointer to a `Vector3` can be thought of as the address of `Vector3.dx`.



A pointer to a `Vector3` can point to an object of class `LorentzVector`.

*Strictly speaking this is only guaranteed for `POD types`, classes that only contain *plain old data* and lack (non-static) member functions.

```
//                              LorentzVector.h
#ifndef LorentzVector_H
#define LorentzVector_H

#include "Vector3.h"

class LorentzVector: public Vector3 {

public:

  LorentzVector(double=0, double=0,
                double=0, double=0);
  // Constructor giving the
  // components x, y, z, t.

  LorentzVector(const Vector3 &, double);
  // Constructor giving a 3-Vector and
  // a time component.

  LorentzVector(const LorentzVector &);
  // Copy constructor.

  ~LorentzVector();
  // The destructor.
```

```
  double t() const;
  // Get time component.

  void t(double);
  // Set time component.

  Vector3 vec3() const;
  // Get copy of spatial component.

  void vec3(const Vector3 &);
  // Set spatial component.

  LorentzVector &
  operator=(const LorentzVector &);
  // Assignment.

  LorentzVector &
  operator+=(const LorentzVector &);
  LorentzVector
  operator+(const LorentzVector &) const;
  // Additions.

  LorentzVector &
  operator-=(const LorentzVector &);
  LorentzVector
  operator-(const LorentzVector &) const;
  // Subtractions.
```

```
  LorentzVector operator - () const;
  // Unary minus.

  double s2() const;
  // Invariant length squared.

  double s() const;
  // Invariant length.

  double dot(const LorentzVector &) const;
  double
  operator*(const LorentzVector &) const;
  // Scalar product.

  void boost(double, double, double);
  void boost(const Vector3 &);
  // Lorentz boost.

private:

  double  dt;

};

#endif /* LorentzVector_H */
```

```
//                          LorentzVector.cc
#include "LorentzVector.h"
#include <cmath>

LorentzVector::
LorentzVector(double xx, double yy,
              double zz, double tt)
  : Vector3(xx, yy, zz), dt(tt) {}

LorentzVector::
LorentzVector(const Vector3 & p, double tt)
  : Vector3(p), dt(tt) {}

LorentzVector::
LorentzVector(const LorentzVector & p)
  : Vector3(p.x(), p.y(), p.z()),
    dt(p.t()) {}

LorentzVector::~LorentzVector() {}

double LorentzVector::t() const {
  return dt;
}

void LorentzVector::t(double tt) {
  dt = tt;
}
```

```
Vector3 LorentzVector::vec3() const {
  return *this;
}

void LorentzVector::
vec3(const Vector3 & p) {
  Vector3::operator=(p);
}

LorentzVector & LorentzVector::
operator=(const LorentzVector & q) {
  Vector3::operator=(q);
  t(q.t());
  return *this;
}

LorentzVector LorentzVector::
operator+ const LorentzVector & q) const {
  return
    LorentzVector(x()+q.x(), y()+q.y(),
                  z()+q.z(), t()+q.t());
}

LorentzVector & LorentzVector::
operator+=(const LorentzVector & q) {
  Vector3::operator+=(q.vec3());
  dt += q.t();
  return *this;
}
```

```
LorentzVector LorentzVector::
operator-(const LorentzVector & q) const {
  return
    LorentzVector(x()-q.x(), y()-q.y(),
                  z()-q.z(), t()-q.t());
}

LorentzVector & LorentzVector::
operator-=(const LorentzVector & q) {
  Vector3::operator-=(q.vec3());
  dt -= q.t();
}

LorentzVector LorentzVector::
operator-() const {
  return
     LorentzVector(-x(), -y(), -z(), -t());
}

double LorentzVector::s2() const {
  return t()*t() - mag2();
}

double LorentzVector::s() const {
  double ss = s2();
  return ss < 0.0 ? -std::sqrt(-ss)
                  : std::sqrt(ss);
}
```

```
double LorentzVector::dot(const LorentzVector & q) const {
  return t()*q.t() - z()*q.z() - y()*q.y() - x()*q.x();
}

double LorentzVector::operator * (const LorentzVector & q) const {
  return dot(q);
}

void LorentzVector::boost(double betax, double betay, double betaz) {
  boost(Vector3(betax, betay, betaz));
}

void LorentzVector::boost(const Vector3 & beta) {
  double b2 = beta.mag2();
  double gamma = 1.0 / std::sqrt(1.0 - b2);
  double bp = beta.dot(*this);
  double gamma2 = b2 > 0 ? (gamma - 1.0)/b2 : 0.0;

  x(x() + gamma2*bp*beta.x() + gamma*beta.x()*t());
  y(y() + gamma2*bp*beta.y() + gamma*beta.y()*t());
  z(z() + gamma2*bp*beta.z() + gamma*beta.z()*t());
  t(gamma*(t() + bp));
}
```

Alternatives to inheritance

Containment by inheritance.
`LorentzVector` IsA `Vector3`.

Containment by value.
`LorentzVector` HasA `Vector3`.

Containment by reference.
`LorentzVector` Points to a `Vector3`.

```
class LorentzVector: public Vector3 {
  // ...
};

class LorentzVector {
private:
  Vector3 vec3;
public:
// Reimplement all Vector3 member
// functions 'forwarding' the call
// to vec3
};

class LorentzVector {
private:
  Vector3 * vec3;
public:
// Reimplement all Vector3 member
// functions 'forwarding' the call
// to vec3. Also allocate vec3 in
// each constructor. deallocate it
// in the destructor.
};
```

Inheritance seems like a good alternative in this case

Friends and protected members

A `friend` function (or class) can acces the private parts of a class. A *derived* class can access `protected` members of a base class

```
Vector3 operator*(double a,
                  const Vector3 & p) {
  return Vector3(a*p.x(),a*p.y(),a*p.z());
}

class Vector3 {
public:
  friend Vector3
    operator*(double, const Vector3 &);

// ... as before

private:
  double dx, dy, dz;

};

Vector3 operator*(double a,
                  const Vector3 & p) {
  return Vector3(a*p.dx, a*p.dy, a*p.dz);
}
```

```
class Vector3 {
public:
  // ... as before
protected:
  double dx, dy, dz;

};

class LorentzVector: public Vector3 {
public:
  double s2() const {
    return dt*dt - dz*dz - dy*dy - dx*dx;
  }

private:
  double dt;

};

LorentzVector q;  // ERROR dx is still
q.dx = 3.0;       // private to the rest
                  // of the world.
```

Second Exercise

Take the code for `Vector3` and `LorentzVector` from `http://cbbp.thep.lu.se/~carl/cpp/` Compile it and write small programs and functions using them. Try to check that all rotations and boosts are correctly implemented.

Types and typedefs

C++ is a strongly typed language. The language makes it hard to do stupid things (including some that don't look very stupid).

```
class X {
public:
  int a;
};
class Y {
public:
  int a;
};
X x1;
X x2 = x1;      // OK. implicit copy-ctor
int a1 = x1.a;
int a2 = x2;    // ERROR type mismatch
Y y1 = x1;      // ERROR type mismatch

typedef X Z;    // Z is now an alias for X

Z z1 = x1;      // OK, Z and X are the same

int * ip;            // pointer to int
const int * cip;     // pointer to const int
int & ir = *ip;      // reference to int
```

```
const int & cir = 2;
    // reference to const int
typedef int * IP;
    // IP is an alias for 'pointer to int'
typedef int & IR;
    // IR is an alias for reference to int
IP ip2 = ip
    // OK ip2 points to the same int as ip
IR ir2 = ir;
    // OK ir2 refers to the same int as ir
const IP cip2 = cip;
    // ERROR cip2 is a const pointer to a
    // (non-const) int, not a pointer to a
    // const int
int * ip3 = cip;
    // ERROR ip3 is a pointer to a (non-
    // const) int, not to a const int
const int * cip3 const = cip;
    // OK cip3 cannot be changed later
```

Which function will be called depends on the type of the arguments, but not on the type of the return value.

```
double abs(double d);
float abs(float f);
int abs(int i);
double abs(Vector3 p);
float abs(Vector3 p);// ERROR same argument
                     // type exists before

double d = abs(-14); // abs(int)
d = abs(-14.0);      // abs(double)
short c = -3;
abs(c);              // abs(int)
                     // short is 'promoted'
long l = -2;
l = abs(l);          // ERROR ambiguous

LorentzVector q;
abs(q);              // abs(Vector3)
                     // LorentzVector IsA
                     // Vector3
```

```
class MyLong {
public:
  MyLong(long l): a(l) {}
  long a;
};
MyLong sqr(MyLong);
sqr(2);       // sqr(MyLong);
              // User-defined conversion

class MyDouble {
public:
  MyDouble(): a(0) {}
  MyDouble(double d): a(d) {}
  operator double() const { return a;}
  double a;
};
MyDouble md;
md = abs(md); // abs(double)
              // User-defined conversion
```

Promotion between built-in types:
`char` → `short` → `int` → `long` and `float` → `double`
but also
`int` → `float`, `int` → `double`, `long` → `float`, etc.

'`float abs(float)`', '`double abs(double)`', '`double abs(Vector3)`' and '`double Vector3::abs()' const` have the same name in the code, but they are different functions and will internally be given a unique name. This is called *name mangling*. My compiler names them `abs__Ff`, `abs__Fd`, `abs__FG7Vector3` and `abs__C7Vector3`.

Different compilers have different name mangling, you can therefore not link together object files from different compilers.

When finding out which function to call, the compiler tries different thing in the following order:

1. Try to find an exact match
2. Try to find a trivial conversion . (If the argument is `const int &` any variable of type `int`, `const int` or `int &` is trivially converted.)
3. Try to find a promotion. (eg. `int`→`double`)
4. Try to find a standard conversion. Eg. `LorentzVector*` can be converted to a `Vector3*`
5. Try to find a user-defined conversion.

This is a simplification. The actual *lookup-rules* are **very** complicated, but luckily the options are usually not many.

Function pointers

```
double sqrt(double);
double log(double);
int round(double);
double sqr(int);

typedef double (*DDFunc)(double);
// DDFunc is an alias for 'pointer to a function taking
// one double argument and returning a double'

double sumfn(double * v, int n, DDFunc fn) {
  double sum = 0;
  for ( int i = 0; i < n; ++i )
    sum += (*fn)(v[i]);
  return sum;
}

double dvec[10];
double s = sumfn(dvec, 10, &sqrt);
s = sumfn(dvec, 10, &log);
s = sumfn(dvec, 10, &round); // ERROR bad return type
s = sumfn(dvec, 10, &sqr);   // ERROR bad argument type
```

The syntax for function pointers can be rather confusing so it is generally a good idea to make a typedef for the function pointer type, as in the example on the previous slide.

C++ has inherited the notion of function pointers from C, where they are very common. In C++ you will likely only use them if your program uses software libraries written in C (for example GSL for numerical stuff or SDL for graphics).

The C++ replacement (of sorts) for function pointers are *functors* (function objects), which will be covered in lecture 6. In C++ there are also special pointers to class member variables and functions, but we will not cover those here.

Base class pointer to derived class

```
class Employee
{
public:
  Employee(std::string s, int sal) :
    theName(s), theSalary(sal) {}

  int salary() const {
    return theSalary;
  }

  int payCheck() const {
    return salary();
  }

private:
  std::string theName;
  int theSalary;
};
```

```
class Boss: public Employee
{
public:
  Boss(std::string s, int sal, int b) :
    Employee(s, sal), theBonus(b) {}

  int bonus() const {
    return theBonus;
  }

  int payCheck() const {
    return salary() + bonus();
  }

private:
  int theBonus;
};
```

```
Boss * ceo = new Boss("Gill Bates", 10000, 200000);
std::cout << ceo->payCheck() << "\n";  // 210000
Employee * emp = ceo;
std::cout << emp->payCheck() << "\n";  // 10000
```

Virtual member functions

```
class Employee
{
public:

  Employee(std::string s, int sal) :
    theName(s), theSalary(sal) {}

  virtual ~Employee() {}

  int salary() const {
    return theSalary;
  }

  virtual int payCheck() const {
    return salary();
  }

private:
  std::string theName;

  int theSalary;
};
```

```
class Boss: public Employee
{
public:

  Boss(std::string s, int sal, int b) :
    Employee(s, sal), theBonus(b) {}

  virtual ~Boss() {}

  int bonus() const {
    return theBonus;
  }

  virtual int payCheck() const {
    return salary() + bonus();
  }

private:
  int theBonus;
};
```

```
Employee * staff[10];      // An array of ten pointers to Employee
staff[0] = new Employee("John Smith", 10000);
staff[1] = new Boss("Gill Bates", 100000, 200000);
staff[2] = new Employee("Calvin Hobbes", 12000);
// ...

int sumpay = 0;
for ( int i = 0; i < 10; ++i ) {
  sumpay += staff[i]->payCheck();
}
```

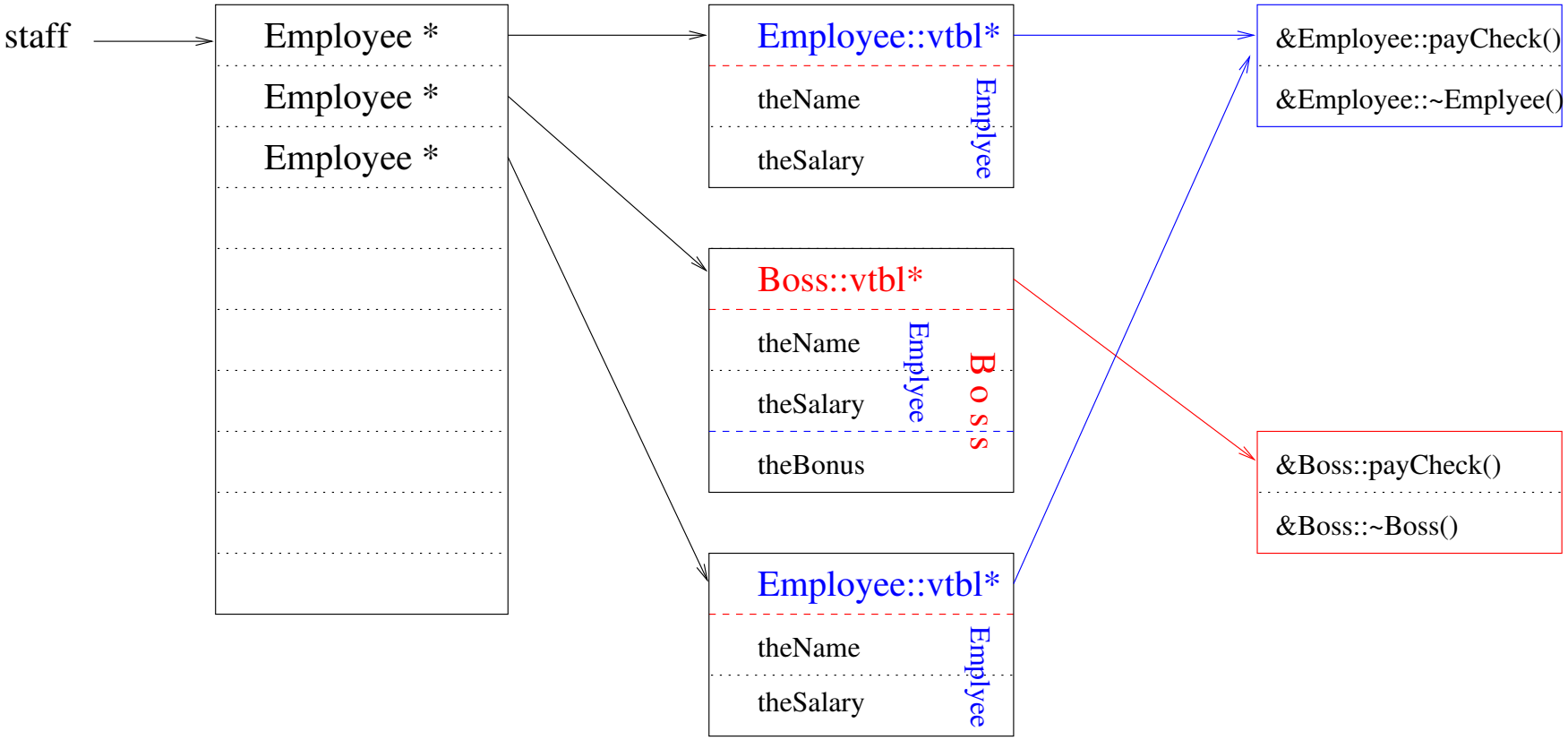
`staff[0]->payCheck()` will call `Employee::payCheck()` but `staff[1]->payCheck()` will call `Boss::payCheck()`

```
Employee * e1 = staff[1];
e1->payCheck();              // Call Boss::payCheck()
Employee & e2 = *staff[1];
e2.payCheck();               // Call Boss::payCheck()

Employee e3 = *staff[1];     // Copy only Employee part
e3.payCheck();               // Call Emplayee::payCheck();
```

`Boss` IsA `Employee`. The one summing up the pay checks need not know how the check is calculated, just that every `Employee` has a way of doing it.

This is Polymorfism!

It is one of the most important ingredients of Object Orientation. How does it work?



When inheriting from a *base class*, the *derived* class inherits all members. The member *functions* may however be overridden.

If a member function is declared `virtual` in the base class, it will be overidden if a member function with the same name and same signature is declared in the derived class. The member of the derived class will then be called even if the object is accessed though a pointer or a reference to the base class.

```
for ( int i = 0; i < 10; ++i ) delete staff[i];
```

`delete staff[0];` will call `~Employee()`, but `delete staff[1];` will call `~Boss()` first and then `~Employee()`

Polymorphic classes **must** declare the destructor in the base class `virtual`.

Abstract base clases

A triangle is a shape. A rectangle is a shape, A square is a rectangle which in turn is a shape. So it seems natural to have `Shape` as a base class for objects which are shapes.

```
class Shape {
public:

  Shape() {}
  Shape(const Shape &) {}

  virtual ~Shape() {}

  Shape & operator=(const Shape &) {
    return *this;
  }

  virtual double area() const {
   return 0;
  }

};
```

```
class Triangle: public Shape {
public:
  Triangle();
  Triangle(const Triangle &);
  Triangle(double b, double h);
  virtual ~Triangle() {}
  Triangle & operator=(const Triangle &);

  double base() const;
  double height() const;
  void base(double);
  void height(double);

  virtual double area() const {
    return base()*height()/2.0;
  }

private:
  double theBase;
  double theHeight;
};
```

```
class Rectangle: public Shape {
public:

  Rectangle();
  Rectangle(const Rectangle &);
  Rectangle(double b, double h);
  virtual ~Rectangle() {}
  Rectangle & operator=(const Rectangle &);

  double base() const;
  double height() const;
  void base(double);
  void height(double);

  virtual double area() const {
    return base()*height();
  }

private:
  double theBase;
  double theHeight;
};
```

```
class Square: public Rectangle {
public:

  Square();
  Square(const Square &);
  explicit Square(double side);
  virtual ~Square() {}

  void base(double);
  void height(double);
  // Both sets both the base and height

  Square & operator=(const Square &);
};

Triangle tr(1.0, 3.0);
Square sq(4.0);
Rectangle re(1.0, 1.0);
Shape * s1 = &tr;
Shape * s2 = &sq;
Shape * s3 = &re;
Shape s;              // OOOPS! what is
                      // a 'Shape' object?
```

Shape, Triangle, Rectangle and Square build up a *class hierarchy*.

We can define a *pure* virtual function

```
class Shape {
public:

  virtual ~Shape() {}

  Shape & operator=(const Shape &) {
    return *this;
  }

  virtual double area() const = 0;

protected:

  Shape() {}
  Shape(const Shape &) {}

};
```

```
Shape s;              // ERROR Shape is abstract
                      // (and Shape() is protected)
Square sq(4.0);
Rectangle & re = sq; // OK A Suare IsA Rectangle
re.height(2.0);       // OOOPS! sq is no longer a
                      // square

double somefn(const Square &);
                      // Some function taking a
                      // Square as argument

double d = somefn(sq);
d += somefn(3.0);     // Error 3.0 is not
                      // convertible to Square
                      // since Square(double) was
                      // declared 'explicit'
d += somefn(Square(3.0));
                      // OK.
```

Any class with a pure virtual function is an *abstract* base class and an object of this class cannot be instatiated.

A square is a rectangle in strict geometrical sense. But in C++ Square should not inherit from Rectangle.

Since a derived class can be accessed as its base class, it is seldom a good idea to have a derived class which *restricts* the behavior of the base class. (In Java, inheritance is declared as "`class Square extends Shape`".)

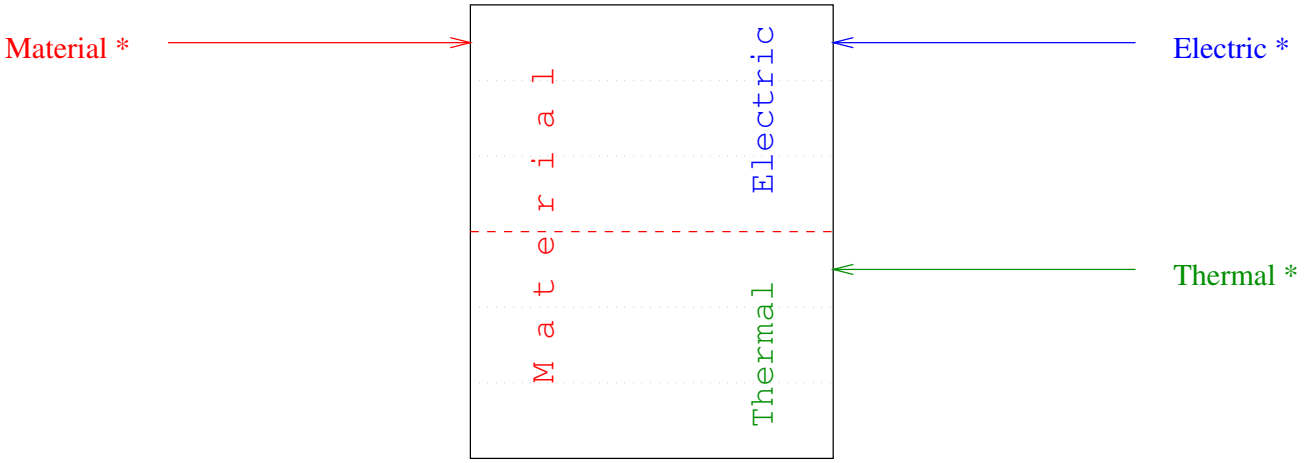
Constructors taking one argument of a different type causes automatic type conversion. This may look pretty cool, but may lead to bugs which are difficult to detect. Use `explicit` keyword unless you are absolutely sure you want automatic type conversion.

Multiple inheritance

A derived class may have several base classes

```
class Electric {
  // abstraction for anything
  // with electric proprties
  // probably abstract base
public:

  virtual ~Electric();
  double conductivity() const;
  // ...
};

class Thermal {
  // abstraction for anything
  // with thermal properties
  // probably abstract base.
public:

  virtual ~Thermal();
  double meltingpoint() const;
};
```

```
class Material: public Electric,
                public Thermal {
public:
  // ...
};

Material * m = new Material;

Electric * e = m;

Thermal * t = m;

t = e;                              // ERROR
```



Multiple inheritance is not trivial. Can cause strange name clashes e.g. if we have both `Electric::conductivity()` and `Thermal::conductivity()`. Problems may also occur if `Electric` and `Thermal` has a common base class (in which case we may use virtual inheritance).

See CPL3 15.2

A pointer to a derived class can be automatically converted (*cast*) to a pointer to its base class, but not the other way around. You can use `dynamic_cast`.

```
Square sq(1.0);
Shape * sp = &sq;
Rectangle * rp = sp;                                  // ERROR
rp = dynamic_cast<Rectangle*>(sp);                    // This is allowed
Triangle * tp = dynamic_cast<Triangle*>(sp); // This is also allowed
                                                      // but sp does not point
                                                      // to a Triangle!
if ( tp ) std::cout << tp->base();
else std::cout << "Error" << std::endl;
```

`dybamic_cast` only work on (pointers and references to) classes with virtual functions. If the cast fails, `dynamic_cast` returns `0`.

Any pointer can be assigned the value `0`. Also any pointer can be converted into an integer and can be used in a conditional expression. If the pointer is `0` the condition is `false` otherwise it is `true`.

Forward declarations

In bubble chamber experiments we can measure *tracks* which are connected with *vertices*.

Each track has a creation vertex and a decay vertex. Each vertex has an incoming, and a number of outgoing tracks.

To define a class `Track` we need to know about `Vertex` and vice versa. Which should we declare first? – a typical chicken–egg situation.

```
// Track.h
#ifndef Track_H
#define Track_H

class Vertex; // Forward declaration

class Track {

  // ...

public:
  double distance() const;

private:
  Vertex creationVertex;
  // ERROR we cannot use actual objects
  // of Vertex since it hasn't been
  // fully declared yet

  Vertex * decayVertex;
  // OK. We can handle pointers to an
  // undeclared type as long as we do not
  // try to access the object pointed to.

};
#endif
```

```
// Vertex.h
#ifndef Vertex_H
#define Vertex_H

#include <vector>
#include "Vector3.h"

class Track; // Forward declaration

class Vertex {

  // ...

public:
  const Vector3 & position() const;

private:
  Track * incomingTrack;

  std::vector<Track*> outgoingTracks

  Vector3 thePosition;

};

#endif
```

```
// Track.cc
#include "Track.h"

double Track::distance() const {
  return (creationVertex->position() - decayVertex->position()).mag();
}
// ERROR Vertex is still not fully declared

#include "Vertex.h"

double Track::distance() const {
  return (creationVertex->position() - decayVertex->position()).mag();
}
// OK now.
```

The `static` keyword

```
const Vector3 xhat(1.0, 0.0, 0.0);
const Vector3 yhat(0.0, 1.0, 0.0);
const Vector3 zhat(0.0, 0.0, 1.0);
// unit vectors in the x,y,z directions
// 'But they pollute the namespace'

class Vector3 {
public:

  static const Vector3 & xHat();
  static const Vector3 & yHat();
  static const Vector3 & zHat();

private:

  static const Vector3 xhat;
  static const Vector3 yhat;
  static const Vector3 zhat;

};
```

```
const Vector3 & Vector3::xHat() {
  return xhat;
}
const Vector3 Vector3::xhat(1.0,0.0,0.0);
// Defined exactly once in the program.

Vector3 p;
p.rotate(3.14, p.xHat());
p.rotate(3.14, Vector3::xHat());
p.rotate(3.14, p.xhat);  // ERROR private

void somefn(double d) {
  static int count = 0;
  ++count;
  // Count calls to this function.
  // 'count' is set to 0 at first call.
}

static inline int sqr(int x) { ...
  // Function is not exported to object
  // file; may be useful in header files.
}
```

A static member variable gives one object which is common to all objects of the class. It is accessed like an ordinary member variable.

A static member function is like an ordinary one, but does not have the hidden `this` argument and can therefore not access non-static members.

All static class members are created before `main()` is called and destroyed after `main()` has returned.

A static local variable in a function is created when the statement is first executed, and is destroyed after `main()` has returned.

Third Exercise

Write a hierarchy of classes that can be used to represent a mathematical expression of one varible, x. Start from an abstract base class Expression with this function:

```
// Returns the value of this expression given a value for x
virtual double evaluate(double x) const = 0;
```

The idea is that an expression such as $x + 5$ can be handled as an Addition that points to its two operands, a Variable (ie x) and a Constant (here: 5). Write derived classes with suitable constructors, destructors, member variables and implementations of `evaluate`. Note that Constant::evaluate will ignore the value of x. Write the following classes, as described by the declarations of their constructors:

```
// Evaluates to a constant value
Constant(double c);
// Evaluates to x
Variable();
// Evaluates to the sum of the two expressions
Addition(const Expression *e1, const Expression *e2);
// Evaluates to the product of the two expressions
Multiplication(const Expression *e1, const Expression *e2);
```

With these classes it should be possible to describe an expression such as $x(x+7)+2$ as follows:

```
Expression *e = new Addition(
   new Multiplication(new Variable,
   new Addition(new Variable, new Constant(7))),
   new Constant(2));
std::cout << e->evaluate(0.5) << "\n";   // should print 5.75
delete e;   // should delete all sub-expressions
```

When this works, add to Expression a function that returns the derivative of the expression, and implement it in the derived classes.

```
virtual Expression * derivative() const = 0;
```

Usage example, with the same expression as above:

```
Expression *de = e->derivative();
std::cout << de->evaluate(0.5) << "\n";   // should print 8.0
```

Please submit your solution to carl@thep.lu.se no later than 12:00 on Wednesday 9/4.

Third Exercise, solutions

```
class Expression {
public:
  virtual ~Expression();
  virtual double eval(double x) const = 0;
  virtual Expression * deriv() const = 0;
};

class Constant: public Expression {
public:
  Constant(double c): c(c) {}
  double eval(double x) const override {
    return c;
  }
  Expression * deriv() const override {
    return new Constant(0);
  }
private:
  double c;
};
```

```
class Multiplication: public Expression {
public:
  Multiplication(Expression *o1,
    Expression *o2): op1(o1), op2(o2) {}
  ~Multiplication() {
    delete op1;  delete op2;
  }
  double eval(double x) const {
    return op1->eval(x) * op1->eval(x);
  }
  Expression * deriv() const {
    return new Addition(
      new Multiplication(op1, op2->deriv()),
      new Multiplication(op2, op1->deriv()));
  }
private:
  Expression *op1, *op2;
};
```

Destructors delete the sub-expressions but a) it's easy to miss some and b) we get two pointers to the same object in `Multiplication::deriv()`. A crash is coming...

Solution 1: Clone the sub-expressions in `Multiplication::deriv()` by adding `virtual Expression * clone() const;` to all classes. Doesn't solve problem of managing pointers.

Solution 2: Use a smart pointer like `unique_ptr` or `shared_ptr` to solve one or both problems. We'll see these soon...

Take-home message: Pointers, like arrays, should be used with great care.

C++11 adds `override` and `final` for checking that a function overrides a virtual function in a base class. See example in `Constant`. A `final` function cannot be overridden in a derived class (this allows devirtualization, a useful optimization).

Templates

```
double sqr(double x) {
  return x*x;
}

int sqr(int x) {
  return x*x;
}

float sqr(float x) {
  return x*x;
}

double d = 3;
d = sqr(d);
// Uses double sqr(double)
```

```
template <typename T>  // or <class T>
T sqr(T x) {
  return x*x;
}

double d = 3;
d = sqr(d); // 'Instantiate'
            // double sqr<double>(double)
            // and call it.
Complex z(1.0,2.0);
z = sqr(z); // call Complex sqr<Complex>(Complex)
```

This is an example of *Generic Programming*.

A class can be templated.

```
template <typename T>
class vector {
public:

  typedef T value_type;

  explicit vector(int);
  // create a vector of given size.

  vector(const vector<T> &);

  vector<T> & operator=(const vector<T> &);

  T & operator[](int i);
  const T & operator[](int i) const;
  // subscript operator

  int size() const;

private:

  T * vec;
  int theSize;
  // The internal representation.

};
```

```
template <typename T>
vector<T>::vector(int sz)
  : vec(new T[sz]), theSize(sz) {}

template <typename T>
vector<T>::vector(const vector<T> & v)
  : vec(new T[v.theSize]),
    theSize(v.theSize) {
  for ( int i = 0; i < theSize; ++i )
    vec[i] = v[i];
}

template <typename T>
vector<T> & vector<T>::
operator=(const vector<T> & v) {
  if ( this == &v ) return *this;
  delete [] vec;
  theSize = v.theSize;
  vec = new T[theSize];
  for ( int i = 0; i < theSize; ++i )
    vec[i] = v[i];
  return *this;
}

template <typename T>
vector<T>::~vector() {
  delete [] vec;
}
```

```
template <typename T>
T & vector<T>::operator[](int i) {
  return vec[i];
}

template <typename T>
const T & vector<T>::
operator[](int i) const {
  return vec[i];
}

template <typename T>
int vector<T>::size() const {
  return theSize;
}
```

```
vector<int> iv(20);  // instantiate the class
                     // vector<int>
iv[5] = 3;           // uses non-const operator[]

vector<Complex> cv(30);  // a vector of 30 Complex
cv[13] = Complex(1.0,3.0);

vector<long> lv = cv;    // ERROR type mismatch

const vector<int> & civ = iv;
civ[2] = 42;             // ERROR uses const
                         // operator[] which
                         // returns const int&
template <typename T>
T sumvector(const vector<T> & v) {
  T sum = T();
  for ( int i = 0; i < v.size(); ++i ) sum += v[i];
  return sum;
}

template <typename Cont>
typename Cont::value_type sum(const Cont & c) {
  typedef typename Cont::value_type T;
  T sum = T();
  for ( int i=0; i < c.size(); ++i ) sum += c[i];
  return sum;
}

Complex sumz = sum(cv);
```

Member functions can be templated:

```
template <typename T>
class vector {
public:
// as before ...

  template <typename U>
  vector<T> & operator=(const vector<U> &);
  // Assignment from vector of other type

};

template <typename T>
template <typename U>
vector<T> & vector<T>::
operator=(const vector<U> & v) {
  if ( this == &v ) return *this;
  delete [] vec;
  theSize = v.theSize;
  vec = new T[theSize];
  for ( int i = 0; i < theSize; ++i )
    vec[i] = v[i];
  return *this;
}
```

```
vector<int> iv(10);
vector<long> lv(30);
vector<Complex> cv(30);

lv = iv;  // OK uses templated operator=

iv = cv   // ERROR assignment from Complex
          // to int in templated operator=
```

Template specialization

```
template <>
class vector<char> {
public:
  vector(const vector<char> & v) {
    : vec(new char[v.theSize]), theSize(v.theSize) {
    memcpy(v.vec, vec, theSize); // memcpy is a standard
  }                              // C function for
                                 // copying raw memory

  vector<char> & operator=(const vector<char> & v) {
    if ( this == &v ) return *this;
    delete [] vec;
    theSize = v.theSize;
    vec = new char[theSize];
    memcpy(v.vec, vec, theSize);
  }

  // Reimplement all other members again.
};

vector<char> cv1(1000000);
vector<char> cv2 = cv1; // Uses memcpy;
vector<int> iv1(1000000);
vector<int> iv2 = iv1;  // Still uses memberwise copy
```

Traits

```
template <typename T>
class CopyTraits {
public:
  static void copy(T * tout,
                   const T *tin, int n) {
    for ( int i = 0; i < n; ++i )
      tout[i] = tin[i];
  }
};

template <typename T>
class vector {
public:
  vector(const vector & v)
    : vec(new T[v.theSize]),
      theSize(v.theSize) {
    CopyTraits<T>::copy(vec, v.vec, theSize);
  }

// Similarly for operator=
// The rest as before.

};
```

```
template <>
class CopyTraits<char> {
public:
  static void copy(char * tout,
                   const char *tin,
                   int n) {
    memcpy(tout, tin, n);
  }
};

template <>
class CopyTraits<double> {
public:
  static void copy(double * tout,
                   const double *tin,
                   int n) {
    memcpy(tout, tin, n*sizeof(double));
  }
};
```

Partial template specialization

We can try to be even more general:

```
template <typename T>
class TypeTraits {
public:
  static const bool trivial = false;
};

template <typename T,
          bool Trivial = TypeTraits<T>::trivial>
class CopyTraits {
public:
  static void copy(T * tout, const T *tin, int n){
    for ( int i=0; i < n; ++i ) tout[i] = tin[i];
  }
};

template <typename T>
class CopyTraits<T, true> {
public:
  static void copy(T *tout, const T *tin, int n) {
    memcpy(tout, tin, n*sizeof(T));
  }
};
```

```
template <>
class TypeTraits<char>
 public:
  static const bool trivial = true;
};

template <>
class TypeTraits<int>
 public:
  static const bool trivial = true;
};
```

A templated function will not be instantiated for a given template argument type/value unless it is called.

Compilers typically need to see the implementation of a templated function when it is called – do not put templated function definitions in `.cc` files.

Excessive templating increases the compile time, but there is in general no run-time overhead.

Templates are difficult but very versatile.

```
template <unsigned int i>
class Factorial {
public:
  static const unsigned long value = i*Factorial<i-1>::value;
};

template <>
class Factorial<1> {
public:
  static const unsigned long value = 1;
};

template <>
class Factorial<0> {
public:
  static const unsigned long value = 1;
};

int main() {
  std::cout << Factorial<13>::value << std::endl;
}   // Calculated at compile-time
```

```
template <typename T>
class MagVector {

  // exactly like vector<T> above but:

public:

  double mag(int i) const {
    return (*this)[i].mag();
  }

};
```

```
MagVector<Vector3> v3v(10);
// This is fine

MagVector<Complex> cv(20)
// Also fine

MagVector<int> iv(12);
// Fine, although int does not have a
// mag() member function

double d = iv.mag(3);
// ERROR: int does not have a
// mag() member function
```

Name lookup with templated functions is approximately as follows: First the ordinary lookup of non-templated functions is tried looking for an exact match or a trivial conversion. Then templated functions are tried, where template argument matching must be exact or via a trivial conversion. No other templated functions are tried, but search for non-templated functions continues with promotions, standard conversions and user-defined conversions.

A templated function called with explicit template argument (as in `sqr<double>(2);`) is treated like any other non-templated function call.

```
double max(double d1, double d2) {
  return d1 > d2? d1: d2;
}

template <typename T>
T max(T t1, T t2) {
  return t1 > t2? t1: t2;
}
```

```
int i = 1;
long l = 2;
float f = 3;
double d = 4;

max(l, l); // max<long>
max(i, l); // max(double,double) note that no
           // conversions are tried to find
           // suitable template function
max(f, d); // max(double,double)
max(f, f); // max<float>
```

Namespaces and resolution

Different variables, functions and types may have the same name in C++. The scope resolution operator :: is sometimes needed to access the right variable etc.

```
int pi = 0;

class MathConstants {
public:

  static const double pi = 3.14159265358979323846;
  static const double e = 2.7182818284590452354;
  static const double sqrt2 = 1.41421356237309504880;

};

void somefn(int & j) {
  int pi = 0;
  ++pi;                              // Increment local pi
  ::pi = 4;                          // set global pi
  double d = 2*MathConstants::pi;  // Use static member
}
```

We can use a `namespace` to logically separate out identifiers

```
namespace MathConstants {
  const double pi = 3.14159265358979323846;
  const double e = 2.7182818284590452354;
  const double sqrt2 = 1.41421356237309504880;

  // all these identifiers can be accessed though MathConstants::

}

double e = 3.0;

using MathConstants::pi;           // Make 'pi' available in this scope.
double d = std::sin(pi/e);         // Uses MathConstants::pi;
d *= MathConstants::sqrt2;
d *= sqrt2;                        // ERROR no 'sqrt2' declared
using namespace MathConstants;     // Make all identifiers from
                                   // MathConstants available.
d *= sqrt2;                        // OK now
d += std::sin(pi/e);               // ERROR ambigous which 'e'?
```

A `using` directive adds identifiers into the current scope.

All types, functions and objects in the C++ standard library are in `namespace std`. Recall standard I/O: `std::cout`, or the functions in `<cmath>` such as `std::sqrt(double)`.

Namespaces are very important when designing code to be used by others. The need for them in small projects is not as obvious.

We want to have the code as readable as possible - this calls for simple and descriptive names on types and functions. But simple names often give name clashes. E.g. the standard library defines a character string type called `string`. In High Energy Physics we can treat the field between quarks as a *string*-like field – we may like to call this class of objects `string`.

We could also have different implementations of the same concept. We previously defined a `vector` class. The same is defined in `<vector>` doing (approximately) the same thing.

```
#include <vector>

namespace MyClasses {

  template <typename T>
  class vector {
    // same as before ...
  };

}
```

```
int somefn() {

  using MyClasses::vector;
  // Change this to 'using std::vector'
  // to use the standard implementation
  // Nothing else need to be changed

  using std::string;

  vector<double> dv(10);

  dv[9] = 2.8;

  string s = "Hello";

}
```

A namespace can be aliased:

```
namespace MyStd = std;
// namespace MyStd = SomeOtherProvidersStd;

// ...

MyStd::vector<int> iv;
```

Namespaces are open

```
namespace MyClasses {
  class X;
}

namespace MyClasses {
  class Y;            // Both classes X and Y are declared
}                     // in MyClasses
```

Namespaces can be nested:

```
namespace MyProject {

  namespace MathClasses {

    class Complex;

  }
}
```

Namespaces are transitive

```
 namespace MyClasses {
   using std::vector;
   // ...
 }

 MyClasses::vector<int> iv;
```

Avoid `using` directives in header files.

In particular, **never** import a whole namespace in header files neither in the global namespace or in a named namespace.

```
// MyClasses.h

using namespace std;

// Whatever...
```

```
// MyClasses.h

namespace MyClasses {

  using namespace std;

}
```

Possible users of your code may want to be able to use other implementations of e.g. the standard library.

The standard C++ library

The C++ standard specifies a number of standard classes and functions to be supplied with a standard-conforming compiler.

All identifiers declared by the standard library are defined inside `namespace std {}`.

Some functions have been inherited from C, these are found in the header files such as

- `<cstring>` functions associated with C-style character strings
- `<cmath>` the standard mathematical functions. Also definition of some constants such as `M_PI`.
- `<ctime>`
- `<cstdio>`

The C++-specific library can be divided into

- Character string stuff in `<string>`, ...
- I/O stuff in `<iostream>`, `<fstream>`, `<sstream>`, ...
- Containers in `<vector>`, `<list>`, `<set>`, `<map>`, `<iterator>`, ...
- General utilities in `<utility>`, `<algorithm>`, `<memory>`, `<functional>`...
- Numeric stuff in `<complex>`, `<valarray>` and `<numerics>`.
- ...

Utilities

You only need to define `bool T::operator==(const T &) const;` and `bool T::operator<(const T &) const;` for a given class `T`, the other comparison operators are defined as templates in `<utility>`.

Templated `min` and `max` functions are declared in `<algorithm>`

The useful class `pair<T1, T2>` is defined in `<utility>`

Smart pointers are in the header file `<memory>`. See CPL4 34.3.

A `unique_ptr<T>` is a wrapper around a pointer to a T such that the object is deleted when the pointer is destructed. As the name implies, the pointer cannot be copied (but moved).

A `shared_ptr<T>` is more powerful but more expensive. It does reference counting, and deletes the object when the last pointer to it is destructed.

For both pointer types, the objects must be created with `new`. A good way to create a `shared_ptr` is with `make_shared` :

```
shared_ptr<Expression> expr;
expr = make_shared<Multiplication>(op1, op2->deriv());
// as opposed to:
expr = shared_ptr<Expression>(new Multiplication(op1, op2->deriv()));
```

Fourth Exercise

- Rewrite your `Expression` class hierarchy from the Third Exercise using `shared_ptr<Expression>` instead of bare pointers. A typedef for this pointer type may come in handy.

- Define operators for multiplication and addition of expressions (via `shared_ptr<Expression>`) so that you don't have to call the constructors explicitly everywhere else. Pay attention to const-correctness: use `shared_ptr<const Expression>` where appropriate. Use these operators where possible.

- Place your classes and operators in their own namespace. Don't import the entire namespace in your main; instead verify that the operators are found even though they are in a namespace.

- The function `std::pow(a, b)` computes a^b for floating point types. Implement `pow(expr, double)` in terms of a new Expression subclass called Power.

- Define `template<class T> T sqr(const &T)` which squares its argument. It could be costly to evaluate the same expression twice as `Expression::evaluate` would do, so specialize `sqr` for your expressions using `pow(expr, 2)`

- `dynamic_pointer_cast<T>(shared_ptr<U>)` is the "smart" equivalent of `dynamic_cast<T*>(U*)`. Use this in your addition operator to detect addition with the constant 0 and optimize this case appropriately.

Fourth Exercise, solution

A commented solution to the exercise can be found at

`http://cbbp.thep.lu.se/~carl/cpp/expressions_ex4.cc`

std::string

Recall the problems with C-style strings. They were implemented as pointers to arrays of characters – adding and copying pointers is not the same as adding and copying the character strings themselves.

There are two standard character string types defined in <string>: string and wstring (for 16-bit characters). They are typically declared as follows:

```
template <class charT,
          class traits = string_char_traits<charT>,
          class Allocator = alloc >
class basic_string;

typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

`std::string` works as expected.

```
using std::string;
string s1, s2, s3;

s3 = s1 + s2;          // Concatenation
                       // and assignment

s1 = "Hello";          // Automatic conversion
                       // from c-strings

const char * cp = s2; // ERROR: no conversion
                       // to c-string

cp = s2.c_str();       // OK, and is 0-terminated

s1 += '!';             // Concatenation with character.

char c = s2[3];        // Indexing.

if ( s1 == s2 );       // Check for equivalence

if ( s1 < s2 );        // Lexicographical comparison

std::cout << s3;       // Stream I/O
```

In addition there are member functions such as `insert`, `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, `find_last_not_of`, `replace`, `substr` all with different arguments.

The position of a character in the string (as returned by the *find* functions or as arguments to the *insert/append/replace* functions) is described by an integer of a type which is typedefed to `std::string::size_type`

`std::string` has a static const member `const std::string::size_type std::string::npos` representing *no-position* when returned by the *find* functions

```
string s1 = "Hello";
int h = s1.find('H');
if ( h != string::npos )
  s1[h] = 'h';
```

Never use `C`-style strings

Always use `std::string`

Exceptions

```
template <typename T>
T & vector<T>::operator[](int i) {
  if ( i >= size() ) // Panic! What to do here?
  return vec[i];
}
```

There are basically five ways of dealing with run-time errors

1. Abort the program and dump core. Fine, but let's try to do better.
2. Call a user-supplied error function. IEEE standard specifies a `matherr` function.
3. Return an impossible value I.e `LorentzVector::mag()` returns `-sqrt(-mag2())` above. Not always possible.
4. Set a global error variable. Fine, but what if no one checks?
5. Throw an exception!

```
class OutOfRange {
public:
  // Whatever
};

template <typename T>
T & vector<T>::operator[](int i) {
  if ( i < 0 || i >= size() )
    throw OutOfRange();
    // create an OutOfRange object
    // and throw it
  return vec[i];
}
```

```
vector<int> iv(10);
int i, j;
while (1) {
  std::cin >> i >> j;
  try {
    iv[i] = j;
    break;
  }
  catch (OutOfRange) {
    std::cout << "Out of range! try again!";
  }
}
```

An exception can be an object of any type.

To be able to *catch* an exception thrown from a function, you must enclose the function call in a `try` block followed by `catch` statements.

The `catch` statment looks like a function definition, and normal argument matching is done. If the thrown object cannot be converted to the any of the types in the `catch` statements, the exception is not caught but is passed to the next enclosing try block.

A `try` block can be followed by several `catch` statements

```
class MathError{};
class Overflow: public MathError{};
class ZeroDivide: public MathError{};

// ...

try {
  // ...
}
catch (ZeroDivide) {
  // Handle divide by zero here.
}
catch (MathError) {
  // Handle any other MatheError here
}
catch (...) {
  // Handle any other exception here
}
```

The `catch` statements are tried in order. If one with matching argument is found this one is executed and the rest are ignored, otherwise the next is tried.

Functions can declare what exceptions they throw, but this is deprecated. C++11 instead adds `noexcept` which allows certain optimizations. (For more info, read the book.)

```
class OutOfRange{};

template <typename T>
class vector {
public:

  int size() const throw();
  // Does not throw any exception

  // C++11 equivalent:
  // int size() const noexcept;

  T & operator[](int) throw(OutOfRange);
  // may throw an OutOfRange

  vector<T> & operator=(const vector<T> &);
  // may throw any exception

  // ...

};
```

```
template <typename T>
int vector<T>::size() const {
  return theSize;
}
// ERROR! must have the same 'throw' specs

int somefn(vector<int> & iv, int i)
  throw() {
  return iv[i];
}
// OOPS! not true. operator[] may throw.
// The compiler is not required to give
// an error
```

If no matching `catch` statement is found, `std::terminate()` is called to terminate the program.

Exceptions are ordinary objects and may carry information. They can be caught by value or by reference.

`std::exception` is the base class of some standards exception classes. It has a fairly minimal interface, with a message string.

```
#include <stdexcept> // standard exceptions

template <typename T>
T & vector<T>::operator[](int i) {
  if ( i < 0 || i >= size() )
    throw std::range_error("out of range in vector");
  return vec[i];
}

try {
  // ...
}
catch (const std::range_error & rerr) {
  std::cerr << rerr.what() << std::endl;
}
```

If you call a function which may throw an exception you do not know how to handle, but you may e.g. need to do some cleanup you can catch the exception and re-throw it.

```
double f(int);

double g(int i) {
  try {
    f(i);
  }
  catch (std::range_error) {
    std::cerr << "Something went wrong. "
              << "Hopefully someone else can fix it."
    throw;
  }
}
```

An exception is not necessarily an error. It simply tells you that the called function could not do what it was supposed to.

You could use exceptions for other things. E.g. inside nested loops, where a `break` statement only exits the inner-most loop, throwing an exception can be used to exit the whole loop. Exceptions should be saved for exceptional events, not least because they can be very expensive to handle.

```
vector< vector<int> > matrix;
// ...
bool done = false;
for ( int i = 0; i < 10; ++i ) {
  for (int j = 0; j < 10; ++j ) {
    if ( matrix[i][j] == 0 ) {
      done = true;
      break;
    }
  }
  if ( done ) break;
}
```

```
vector< vector<int> > matrix;
// ...
try {
  for ( int i = 0; i < 10; ++i ) {
    for (int j = 0; j < 10; ++j ) {
      if ( matrix[i][j] == 0 ) throw true;
    }
  }
}
catch (bool) {}
```

Exceptions are new to C++.

Exceptions are difficult.

Enumerations

```
 namespace MySpace {

   const int hearts = 0;
   const int spades = 1;
   const int diamonds = 2;
   const int clubs = 3;

}

using namespace MySpace;
int suit = hearts;

suit = 1;
suit = 14;
```

```
namespace MySpace {

  enum CardSuit {
    hearts, spades,
    diamonds, clubs };

}

using namespace MySpace;
CardSuit suit = hearts;

int i = suit;         // All enum types are
                      // convertible to int
suit = 1;             // ERROR type mismatch
suit = CardSuit(14); // OK
```

`enum` introduces a new integer type, and a number of constant values of that type. The integer type is guaranteed to be big enough to hold all the constants.

```
namespace QM {

  enum Colour {
    triplet = 3, anti_triplet = -3,
    singlet = 0, octet = 8,
    undefined = -99999 };

  enum Parity {
    positive, negative,
    undefined }; // ERROR undefined already
                 // defined

}
```

Input and Output in C++

The C++ standard requires a number of classes for providing input and output of streams of characters.

The standard <iostream> contains the declaration of the std::istream and std::ostream classes representing general input and output streams respectively. It also contains definitions of the objects std::cin, std::cout and std::cerr for the standard input, output and error streams connected to the calling process.

The file <fstream> contains definitions of the std::ifstream and std::ofstream classes for input and output connected to named files.

The file <sstream> contains definitions of the std::istringstream and std::ostringstream classes for input and output from/to std::string objects in memory.

I/O of objects are performed with the *left*- and *right-shift* operators. These were inherited from C and are used to shift the bit pattern of integers to the left or right:

```
int i = 16;
int j = i >> 2;      // Shift all bits two steps
                     // to the right eg. divide by 4
int k = j << 2 << 3; // Shift the value of j 2 steps to the left.
                     // Then shift the resulting bit pattern 3
                     // more steps to the left
```

In <iostream> these operators are defined as member functions of the istream and ostream approximately as follows:

```
class ostream {
public:

  ostream & operator<<(int);
  ostream & operator<<(double);
  // etc...

};

void somefn(ostream & os) {
  int i = 42;
  double d = 3.14;
  os << i << d; // is equivalent to
  (os.operator<<(i)).operator<<(d);
}
```

```
class istream {
public:

  istream & operator>>(int &);
  istream & operator>>(double &);
  // etc...

};

void somefn(istream & is) {
  int i;
  double d;
  is >> i >> d; // equivalent to
  (is.operator>>(i)).operator>>(d);
}
```

`istream` defines `operator>>` for all fundamental types. Similarly `ostream` defines `operator<<` for all fundamental types.

To implement I/O for user-defined types we use non-member (global) left- and right-shift operators:

```
ostream & operator<<(ostream & os,
                     const Complex & z) {
  os << "(" << z.real() << ","
             << z.imag() << ")";
  return os;
}
```

```
istream & operator>>(istream & is,
                     Complex & z) {
  // Handle all cases 'float', '(float)'
  // and '(float,float)'
  double re = 0;
  double im = 0;
  char c = '\0';

  is >> c;
  if ( c == '(' ) {
    is >> re >> c;
    if ( c == ',' ) is >> im >> c;
    if ( c != ')' ) throw ios::failure(is);
  } else {
    is.putback(c);
    is >> re;
  }
  if ( is ) z = Complex(re, im);
  return is;
}
```

Error handling in I/O streams are not done with exceptions. Instead a stream will be put in a *bad state* after an error. This must be checked.

There is an implicit conversion from `istream` and `ostream` to an integer value to be used in conditional statements:

```
char c;
while ( cin >> c ) cout << c;
```

Also the unary not operator is defined

```
if ( !cin ) {
  cerr << "Something went wrong. ";
  cerr << "Maybe read beyond EOF.";
}
```

Also the following member functions are defined:
`bool good() const;` (the next operation might succeed),
`bool eof() const;` (end of input reached),
`bool fail() const;` (the next operation will fail) and
`bool bad() const;` (the stream is corrupted).

Output formatting

There are a number of ways to tell an `ostream` how to format the output. Here are some examples:

```
cout.setf(ios::oct, ios::basefield);      // Set the basefield flag for
                                          // octal output of integers
cout << 42 << endl;                       // output: '52'
cout.width(5);                            // next format had width 5
cout << 42 << endl;                       // output: '   52'
cout << 42 << endl;                       // output: '52'
cout.width(5);
cout.fill('#');
cout << 42 << endl;                       // output: '###52'
cout.width(5);
cout.setf(ios::left, ios::adjustfield);
cout << 42 << endl;                       // output: '52###'
cout.setf(ios::dec, ios::basefield);      // Set the base to decimal
cout.width(5);
cout << 9999999 << endl;                  // output: '9999999'
cout << 1.234567890 << endl;              // output: '1.23457'
cout.precision(3);
cout << 1.234567890 << endl;              // output: '1.23'
cout.setf(ios::scientific,                // use 'scientific' format
          ios::floatfield);               // of floats
cout << 1.234567890 << endl;              // output: '1.235e+00'
```

Stream Manipulators

`iostream` has the following operator member defined:

```
ostream & ostream::operator<<(ostream & (*omanip)(ostream &)) {
  return (*omanip)(*this);
}
```

This enables you to do things like:

```
ostream & setoct(ostream & os) {
  os.setf(ios::oct, ios::basefield);
  return os;
}

cout << setoct << 34 << endl; // ouput: '42'
```

Standard manipulators are defined in `<iomanip>`.

<iomanip> also defines manipulators taking arguments. It works approximately like this.

```
class IntManip {
public:

  IntManip(void (*fn)(ostream &, int), int in)
    : func(fn), i(in) {}

  void (*func)(ostream &, int);
  int i;

};

ostream & operator<<(ostream & os, IntManip & m) {
  m.func(os, m.i);
  return os;
}

void set_precision(ostream & os, int i) {
  os.precision(i);
}

IntManip setprecision(int i) {
  return IntManip(set_precision, i);
}

cout << setprecision(3) << 1.23456789 << endl; // output: '1.23'
```

file streams

The `ofstream` and `ifstream` clases are defined in `<fstream>`

```
ofstream os("filename.dat");   // Open a file for output

os << 42 << endl;              // used as an ostream. In fact
                               // ofstream IsA ostream

os.close();                    // close the file
os.open("anotherfile.dat");    // open another file associated
                               // with the same ofstream object.

int readnumber(string file) {
  ifstream is(file);   // Open a file for input. Before C++11 you
                       // would have to say file.c_str()
  int i;
  is >> i;
  return i;
}                      // The file is safely closed when the
                       // ifstream object is destructed

int j = readnumber("filename.dat");
```

String Streams

You can use `ostringstream` and `istringstream` as defined in `<sstream>` to e.g. convert between character strings and numbers

```
ostringstream os;
os << 42;
string s = os.str();        // s is now "42"

istringstream is(s);
int i;
is >> i;                    // i is now 42

static inline int toint(const std::string &s)
{
   std::istringstream ss(s);
   int i;
   ss >> i;
   if(!ss || !ss.eof())  // Was the whole string used?
      throw "error converting '" + s + "' to int";
   return i;
}
```

Fifth Exercise

Wouldn't it be nice if we could read those expressions from a file instead of having to put them into the program in that strange notation?

Using strings and streams, write a parser for equations.

To keep things simple, you can require whitespace between *tokens* in the input stream, so you can read one token at a time from the stream using >>. Example:

```
x + x * 2 + 15
```

In order to implement operator precedence correctly (* before +), I recommend writing a recursive function that keeps track of the precedence outside of the current subexpression, e.g.

```
shared_ptr<Expression> parse(std::istream &in, int precedence);
```

Containers and Iterators

In almost all code you will need to use containers of different kinds, and you need to be able to loop over the objects inside a container.

The simplest container is an array:

```
double darr[100];
for ( int i = 0; i < 100; ++i ) {
  somefn(darr[i]);
}
```

Looping over the elements in a container is called *iterating*. Not all containers have objects which can be accessed through the subscript operator, `operator[]`.

Here is an alternative way of iterating through the objects of a container using pointer arithmetics:

```
double darr[100];
for ( double * it = darr; it != darr+100; ++it) {
  somefn(*it);
}
```

An *iterator* is a generalization of a pointer. All container classes in the standard library are associated with different classes of iterators, which can be used in similar ways as pointers can be used for arrays.

```
#include <vector>

vector<double> dv(100);
for ( vector<double>::iterator it = dv.begin(); it != dv.end(); ++it )
  somefn(*it);

// C++11 range-based for loop hides the iterator stuff
for ( double v : dv )
  somefn(v);
```

All standard-compliant containers must implement the member functions `begin()` and `end()`.

`begin()` must return an iterator pointing to the *first* object in the container.

`end()` must return an iterator pointing to *one past the last object* in the container. (For `double darr[100]`, `darr+100` points to the next memory location after the array.) In general if the iterator `it` points to the last object in a continer `cont` then `++it == cont.end()` is true.

All iterator classes must have `operator++(int)` and `operator++()` (pre- and post-increment) defined so that `++it` and `it++` make the iterator point to the *next* object in the container.

All iterators must have `operator*()` and `operator->()` defined to dereference and selecting a member of the object pointed to in the container

All iterators must have `operator==` defined so that if and only if two iterators, `it1` and `it2`, point to the same object, `it1 == it2` is true.

There are different categories of iterators

Output iterators are only guaranteed to have the increment and the dereference operators.

Input iterators also have the member selection and equality operators. (but the dereferenced object is not necessarily assignable).

A forward iterator is both an output *and* input operator

A bidirectional iterator is a forward iterator but also have decrement operators (`operator--(int)` and `operator--()`).

A random-access iterator is a bidirectional iterator but also have the operators `[]`, `+`, `-`, `+=`, `-=` and `<` defined.

The C++ standard specifies a number of standard container classes. But the standard does not specify how they are implemented, only how they should behave.

In particular, the CPU time behavior with increasing container size of different operations is specified

container type	subscript	middle insert	front insert	back insert	find	iterator category
`vector`	const	$\mathcal{O}(n)^+$		const$^+$	$\mathcal{O}(n)$	random
`list`		const	const	const	$\mathcal{O}(n)$	bidirectional
`deque`	const	$\mathcal{O}(n)$	const	const	$\mathcal{O}(n)$	random
`set`		$\mathcal{O}(\log(n))$			$\mathcal{O}(\log(n))$	bidirectional
`map`	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$			$\mathcal{O}(\log(n))$	bidirectional
`multimap`		$\mathcal{O}(\log(n))$			$\mathcal{O}(\log(n))$	bidirectional

The 'const+' for e.g. back insertions means that normally the CPU time needed for such operations is independent of the size of the vector, but sometimes the vector may need to re-allocate all objects which is a $\mathcal{O}(n)$ operation.

In practice this means that a `std::vector` implementation nearly always will allocate more objects than needed for the current size.

Which container to choose depends on the operations you want to do and how many object you want to store. Note that although $\log(100) \approx 5$ you do not know the constant factor in front.

std::vector

A `vector` is the container which is much like an ordinary array (C++11 also adds `std::array`). The declaration is as follows:

```
template <typename T, typename Alloc = alloc>
class vector {
public:
  typedef T value_type;
  typedef value_type * pointer;
  typedef const value_type * const_pointer;
  typedef value_type & reference;
  typedef const value_type & const_reference;
  typedef /* implementation defined */ size_type;
  typedef /* implementation defined */ difference_type;

  // ...

};
```

`size_type` is the integer type used to give the size of the vector.
`difference_type` is the integer type used to describe the difference between two iterators.
`Alloc` is a class which used to allocate new objects in the vector.

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:
  typedef /* implementation defined */ iterator;
  typedef /* implementation defined */ const_iterator;
  typedef /* implementation defined */ reverse_iterator;
  typedef /* implementation defined */ const_reverse_iterator;

  // ...

};
```

`vector<T>::iterator` is of random-access type.

`vector<T>::const_iterator` is the same but returns a const reference when it is dereferenced.

`vector<T>::reverse_iterator` is an iterator where `++` steps one step backwards in the vector (used with `rbegin()` / `rend()`)

`vector<T>::const_reverse_iterator` is the same but returns a const reference when it is dereferenced.

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:

  vector();
  // An empty vector
  explicit vector(size_type n, const value_type & t = T());
  // A vector of size n with all objects initialized to t
  vector(const vector<T, Alloc> &);
  // Copy constructor

  template <typename InputIterator>
  vector(InputIterator first, InputIterator last);
  // Constructor initialized from a range of iterators
  // [first,last)

  ~vector();
  // Destructor

  // ...

};
```

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:

  iterator insert(iterator position, const value_type & t = T());
  // Insert a copy of t before position. return position
  iterator insert(iterator position, size_type n,
                  const value_type & t = T());
  // Insert n copies of t before position. return position of the
  // last inserted object

  template <typename InputIterator>
  void insert(iterator position, InputIterator first, InputIterator last);
  // Insert copies of the objects in the range [first,last)
  // before position.

  void push_back(const value_type & t);
  // Add a copy of t to the end

  vector<T, Alloc> & operator=(const vector<T, Alloc> &);
  // assignment

  // ...

};
```

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:
  iterator erase(iterator position);
  // Remove object at position. return pointer to the next object.

  iterator erase(iterator first, iterator last);
  // Remove objects in the range [first,last), return pointer to
  // the object pointed to by last.

  void pop_back();
  // Remove the last element.

  void clear();
  // Remove all elements.

  reference front();
  const_reference front() const;
  reference back();
  const_reference back() const;

  reference operator[](size_type i);
  const_reference operator[](size_type i) const;

  // ...

};
```

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:

  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;

  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;


  // ...

};

// looping backwards over the vector:
vector<double> dv(100);
for (vector<double>::reverse_iterator it = dv.rbegin();
     it != dv.rend(); it++ ) {
  somefn(*it);
}
```

```
template <typename T, typename Alloc = alloc>
class vector {

  // ...
public:

  size_type size() const;
  // The size
  bool empty() const;
  // size() == 0
  size_type capacity() const;
  // The allocated size capacity() >= size()

  void resize(size_type n, const value_type & t = T());
  // Change the size. If new objects need to be created
  // copy from t

  void reserve(size_type n);
  // Change the capacity whithout changing the size
  // If n < capacity() nothing happens.

  // ...

};
```

```
#include <vector>
using std::vector;

vector<double> dv;

for (vector<double>::iterator it = dv.begin();
     it != dv.end(); ++it ) {
  if ( *it == 0 ) dv.push_back(2.0);
  // What happens to 'it' here?
}
```

If the size of a vector is increased so that the objects need to be reallocated, all iterators pointing into the vector are invalidated, and the effects of using them is undefined.

This can be predicted. If a vector is increased with `n` objects and `n <= capacity() - size()` then no re-allocation is done. Note that you can use `reserve()` to change the capacity.

std::deque

A deque (double-ended queue) is like a vector but also has member functions `void push_front(const T &)`, `void pop_front()` and `T & front()`.

The front operations are as efficient as the back operations. Insertions in the middle are still expensive.

std::stack

A stack is a *container adapter* restricting a suitable container.

```
template <typename T, typename C = std::deque<T> >
class std::stack {
public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit stack(const C & cc = C()) : c(cc) {}

  bool empty() const { return c.empty(); }
  size_type size() { return c.size(); }

  value_type & top() { return c.back(); }
  const value_type & top() const { return c.back(); }

  void push(const value_type & t) { c.push_back(t); }
  void pop() { c.pop_back(); }

protected:
  C c;
};
```

std::queue

```
template <typename T, typename C = std::deque<T> >
class std::queue {
public:
  typedef typename C::value_type value_type;
  typedef typename C::size_type size_type;
  typedef C container_type;

  explicit queue(const C & cc = C()) : c(cc) {}

  bool empty() const { return c.empty(); }
  size_type size() { return c.size(); }

  value_type & back() { return c.back(); }
  const value_type & back() const { return c.back(); }
  value_type & front() { return c.front(); }
  const value_type & front() const { return c.front(); }
  void push(const value_type & t) { c.push_back(t); }
  void pop() { c.pop_front(); }

protected:
  C c;
};
```

Neither `stack` or `queue` have iterators defined. (If you need them you probably didn't need a stack or a queue.)

std::list

`list` has efficient insertions anywhere. On the other hand it does not have a subscript operator and only bi-directional iterators.
A list is typically built up from list items

```
template <typename T, typename Alloc = alloc>
class std::list {

  class list_item {
  public:
    T t;
    list_item * prev;
    list_item * next;
  };

  list_item * head;
  list_item * last; // always point to an empty object.
  // ...
};
```

Iterators into a `std::list` is never invalidated except if the object it points to is erased.

Associative containers

An `std::map` associates a given *key* with a *value*.

```
#include <map>
using std::map;

map<string,int> ages;

ages["Jill"] = 4;
ages["Jack"] = 5;

std::cout << ages["Jack"] << std::endl;

for ( map<string,int>::iterator it = ages.begin();
      it != ages.end(); ++it )                          // Should write:
  std::cout << it->first << " is " << it->second        // Jack is 5 years old
            << " years old" << std::endl;               // Jill is 4 years old
```

The elements of a map are automatically ordered according to their key.

```
template <typename Key, typename T, typename Cmp = less<Key>,
          typename Alloc = alloc>
class std::map;

template <typename T>
class std::less {
public:
  bool operator()(const T & t1, const T & t2) const {
    return t1 < t2;
  }
};
```

The requirement on the `Key` type is that class `less<Key>` exists and makes sense. Or that `map` is given a template argument of class `Cmp` with
`bool Cmp::operator()(const Key &, const Key &) const;` defined.

```
template <typename T1, typename T2>
class std::pair {
public:
  T1 first;
  T2 second;
};
```

`map<Key,T>::iterator` points to a `pair<const Key,T>` and is bi-directional. We cannot change the key of an object, but we can change the object for a given key.

```
map<int,double> intmap;

intmap[1] = 2.0;  // Works like a sparse vector

for ( map<int,double>::iterator it = intmap.begin();
      it != intmap.end(); ++it )
  cout << it->first << '\t' << it->second << endl;
```

A map is typically implemented as a binary tree, to be able to achieve $\mathcal{O}(\log(n))$ efficiency for subscript and find operations.

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Alloc = alloc>
class map {
public:

  typedef Key key_type;
  typedef T data_type;
  typedef T mapped_type;
  typedef pair<const Key, T> value_type;
  typedef Compare key_compare;
  typedef /* implementation defined */ value_compare;
  // Comparisons between pair<const Key, T>

  typedef /* implementation defined */ pointer;
  typedef /* implementation defined */ const_pointer;
  typedef /* implementation defined */ reference;
  typedef /* implementation defined */ const_reference;
  typedef /* implementation defined */ iterator;
  typedef /* implementation defined */ const_iterator;
  typedef /* implementation defined */ reverse_iterator;
  typedef /* implementation defined */ const_reverse_iterator;
  // Bidirectional iterators

  typedef /* implementation defined */ size_type;

  // ...

};
```

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Alloc = alloc>
class map {
public:

  // ...

  map();
  explicit map(const Compare& comp);
  // We can have different Compare object for different
  // objects of the same map class (why?)

  template <typename InputIterator>
  map(InputIterator first, InputIterator last);
  template <typename InputIterator>
  map(InputIterator first, InputIterator last, const Compare& comp);
  // Iterators must point to pair<const Key, T>

  map(const map<Key, T, Compare, Alloc>& x);

  map<Key, T, Compare, Alloc>&
  operator=(const map<Key, T, Compare, Alloc>& x);

  key_compare key_comp() const;
  value_compare value_comp() const;

  // ...

};
```

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Alloc = alloc>
class map {
public:

  // ...

  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
  bool empty() const;
  size_type size() const;
  size_type max_size() const;

  T& operator[](const key_type& k);
  // If no Key k is present create a new entry and return a reference
  // the T object.
  // Note: no const operator[]

  // ...

};
```

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Alloc = alloc>
class map {
public:

  // ...

  pair<iterator,bool> insert(const value_type& x);
  // pair<iterator,bool>::second is false if key existed before
  iterator insert(iterator position, const value_type& x);
  template <typename InputIterator>
  void insert(InputIterator first, InputIterator last);

  void erase(iterator position);
  size_type erase(const key_type& x);
  void erase(iterator first, iterator last);
  void clear();

  // ...

};
```

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Alloc = alloc>
class map {
public:

  // ...

  iterator find(const key_type& k);
  const_iterator find(const key_type& k) const;
  // Returns end() if key k is not present.

  size_type count(const key_type& k) const;
  // Returns 1 or 0

  iterator lower_bound(const key_type& k);
  const_iterator lower_bound(const key_type& k);
  // First entry with key >= k or end()

  iterator upper_bound(const key_type& k);
  const_iterator upper_bound(const key_type& k) const;
  // First entry with key > k or end()

  pair<iterator,iterator> equal_range(const key_type& k);
  pair<const_iterator,const_iterator> equal_range(const key_type& k) const;
  // return pair<iterator,iterator>(lower_bound(k), upper_bound(k));

};
```

std::multimap

`std::multimap` is like `std::map` but many objects can have the same key.

For obvious resaons `multimap` does not have `operator[]` or `find()`. On the other hand, the `count()`, `lower_bound()`, `upper_bound()` and `equal_range()` makes more sense.

std::unordered_map

`std::unordered_map` is new in `C++11`. Works like an ordinary `map`, but also takes a hash function as template argument. If you think you need a hash table, use `std::map`, and if you find that you *really* need a performance increase, try `unordered_map`.

std::set

Is like a `map` where the key of an object is the object itself. Can be used as a self-sorting list.

`set<T>::iterator` is bidirectional and points to the object itself.

`operator[]` makes no sense for `set`.

Note that all containers are containers of objects. If you want to store polymorphic objects such as `Shapes`, you may be in for a surprise.

```
#include <list>
#include "Shape.h"
#include "LorentzVector.h"

std::list<Vector3> vectorList;
LorentzVector q;
vectorList.push_back(q);      // Only Vector3 part is
                              // copied into the list.
std::list<Shape> shapeList;   // ERROR: Shape is an abstract base class.
std::list<Shape &> shapeList; // ERROR:
std::list<Shape *> shapeList; // OK

Rectangle r;
shapeList.push_back(&r);      // OK
```

Containers are great for managing object lifetime. Combine with `unique_ptr` or `shared_ptr` if you need managed pointers to objects.

Function objects

A function object is any object of a class with `operator()` defined.

Note that `operator()` is the only operator which does not have a fixed number of arguments.

We have already seen `less<T>` which defines `bool operator()(const T &, const T &) const;`. This is an example of a *predicate* function object. Also other such function classes are defined: `equal_to<>`, `greater_equal<>`, etc.

Other standard function classes are `plus<>`, `minus<>`, `multiplies<>` etc.

Algorithms

The standard library contains a number of algorithms for containers, based on knowing the iterators to the first and (one past) the last objects. They also rely on function objects.

```
template <typename InputIterator, typename Function>
Function std::for_each(InputIterator first, InputIterator last, Function f) {
  for ( ; first != last; ++first) f(*first);
  return f;
}

template <typename InputIterator, typename T>
InputIterator std::find(InputIterator first, InputIterator last,
                        const T& value) {
  while (first != last && *first != value) ++first;
  return first;
}

template <typename InputIterator, typename Predicate>
InputIterator std::find_if(InputIterator first, InputIterator last,
                       Predicate pred) {
  while (first != last && !pred(*first)) ++first;
  return first;
}
```

```
#include <algorithm>

class sumit {
public:
  sumit() : sum(0) {}
  sumit(const sumit & s): sum(s.sum);

  void operator()(double x) {
    sum += x;
  }

  operator double() const {
    return sum;
  }

private:
  double sum;
};

double dv[100];

double sum = std::for_each(dv, dv+100, sumit());
```

```
template <typename InputIterator, typename OutputIterator>
OutputIterator std::copy(InputIterator first, InputIterator last,
                         OutputIterator result) {
  while ( first != last ) *result++ = *first++;
  return result;
}

template <typename InputIterator, typename OutputIterator, typename Predicate>
OutputIterator dont_copy_if(InputIterator first, InputIterator last,
                            OutputIterator result, Predicate pred) {
  for ( ; first != last; ++first)
    if ( !pred(*first) ) *result++ = *first;
  return result;
}

class even {
public:
  bool operator()(int i) {
    return !(i%2);
  }
};

vector<int> iv1;
// fill iv1
vector<int> iv2;
dont_copy_if(iv1.begin(), iv1.end(), iv2.begin(), even());
// OOOPS iv2.begin() points to nonsense since iv2 has size 0.

dont_copy_if(iv1.begin(), iv1.end(), back_inserter(iv2), even());
```

Exercise Six A: Molly Malone

Dublin fishmonger Molly Malone writes down her sales in a file:

```
cod halibut halibut
cod herring halibut
```

Your task is to process this information. Write a program that takes the name of a file (using `argc` and `argv`), and makes a summary of the sales sorted by the number of fish:

```
3 halibut
2 cod
1 herring
```

Hints: You may want to first try without sorting the output. A map is good for counting how many there are of each species. Each output line is a pair of a number and a string. Sorting backwards or printing backwards will give the same result.

Exercise Six B: ODE solver

Starting from a solution to Exercise 5, e.g. `http://cbbp.thep.lu.se/~carl/cpp/expressions_ex5.cc`, extend the expressions to multiple variables.

`parseexpressions` will have to figure out what's variable names in the expressions. Put the function into a class to help it hold the required mapping between variable names and indexes.

Hint: If m is a `map<string,size_t>` then `m.insert(make_pair(str, m.size()))` only alters m if `str` isn't already a key. Do this for n different keys and they will get values $0, \ldots, n$.

A reasonable design choice is to let the `Variable` class know the index of its variable, and let `evaluate()` take a vector of variable values (`vector<const double> &`). Printing the `Variable` objects with names becomes more difficult unless you also let them know the variable name.

If you get this to work, write an ODE solver using e.g. the Euler method. That is, interpret n equations of n variables as the time derivatives of the variables. Start from some state, compute the derivatives, take small steps in the variable values and keep stepping until time τ.

$$x_i(t+h) = x_i(t) + hf(x_1(t), \ldots, x_n(t))$$

Exercise six A, solution

```
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>

int main(int argc, const char **argv)
{
  // Check argument count
  if(argc != 2) {
    std::cerr << "usage: " <<
      argv[0] << " fishfile\n";
    return 1;
  }

  // Open file
  std::ifstream in(argv[1]);
  if(!in) {
    std::cerr << "failed to open " <<
      argv[1] << "\n";
    return 1;
  }
```

```
  // Read and count fish, summarized by name
  std::map<std::string, int> fishcount;
  std::string s;
  while(in >> s)
    fishcount[s]++;

  // Flip the order in the pairs and
  // store them all in a vector
  std::vector<std::pair<int,
    std::string>> countfish;
  for(auto p : fishcount) {
    countfish.push_back(
      std::make_pair(p.second, p.first));
  }

  // Sort by count, descending, then print
  sort(countfish.rbegin(), countfish.rend());
  for(auto p : countfish) {
    std::cout << p.first << " " <<
      p.second << "\n";
  }
}
```

Object Oriented Software Design

Leif says:

1. Formulate your problem in a spoken language.
2. Identify nouns with classes/objects.
3. Identify verbs with (member) functions.

Before starting coding, think very carefully about which classes are needed and how they interact.

In addition, adjectives may indicate inheritance or template class specialization. Adverbs may be template function specializations.

I think this approach may obscure the main issues.

Choosing the right tools

Describe your project in terms of parts that have small interfaces to other parts.

Can you find a hierarchy where parts don't need knowledge of the context they are used in?

Think in terms of objects and containers. If you have a bunch of things, you can probably use a vector. Should the bunch then be its own class with additional variables and functions?

Can you avoid `new` and `delete` by using containers and automatic variables.

Do you need to mix different classes of objects?

Can it be solved by variables in the objects (different data or pointers to different objects of some other class) or do you need completely different objects? If so, you need polymorphism (virtual functions and all that).

Don't overdo it – polymorphism can be expensive and may be unsuitable for some scientific computation.

A guiding principle: Aim to never replicate information unless it's needed for optimization.

Rules and recommendations

Leif's Rule 1: All rules have exceptions

Carl's Rule 2: Aim to never replicate information unless it's needed for optimization.

Comments

C++ is a language. Written code should be readable – but only to people who speak C++. Do not write unnecessary comments.

```
b = a; // b is equal to a
```

Put comments at

- the beginning of each file,
- the definition of each class,
- the declaration of each (member) function.
- In long function definitions (which should be avoided), comment distinct parts of the code.
- In any function definition, comment difficult to read code e.g. containing tricks

Put declarations in header files. The reader should be shown as little of the implementation and data representation as possible. (PIMPL is useful if you really want to hide things.)

```
// This is a dummy class
class dummy {
// ...
public:
  double x() const {
    return theX;
  }
};
```

```
#ifndef dummy_h
#define dummy_h
// This is a dummy class
class dummy {
// ...
public:
  inline double x() const;
  // The x component;
};
#endif
```

Put inline and template function definitions in separate files and `#include` them in the header file.

The header file should work like a UNIX *man-page*, where the usage of the class is documented.

Only one class per header file. The name of the header file should be the same as the class name.

Always define a default constructor, copy constructor and assignment operator for all classes.

If any of these does not make sense, make them private (or delete them in C++11).

Even if the definition is empty or does exactly what the compiler would do in their absence, define them anyway.

Always define the destructor, even if the definition is empty or would do exactly what the compiler would do in their absence.

Remember to make destructors virtual if the class is a part of a hierarchy with anything virtual.

Only introduce overloaded operators if it is absolutely clear what they mean.

Always make member functions `const` if they do not change the logical state of the object.

If a member function changes internal member variables irrelevant for the object's logical state (e.g. for optimization), make these variables `mutable`.

If a const member function accesses objects outside of the class, these should not be changed (i.e. if they are owned by the object).

Use a consistent naming scheme in addition to consistent indentation etc.

- Identifiers constructed from several words could be `capitalisedLikeThis` or `written_like_this`.
- Class names may (or may not) begin with a upper-case letter.
- Member variables could have a `_leading` or `trailing_` underscore to help the reader and avoid clashes with other identifiers. (Note that a leading underscore should not be followed by underscore or a capital letter.)
- There are many styles – the important thing is to be consistent to reduce the potential for confusion.

Always start by making member variables private.

Only introduce friends or make members protected if it improves data hiding, i.e. when the alternative is to making things public. Never do it to simplify notation.

Never use inheritance to represent a has-a relationship. Instead use a member variable, possibly a pointer.

Pass object arguments as copies or as references unless you want to transfer the ownership of dynamically allocated objects, in which case they should be passed as pointers.

But: if you use smart pointers you should pass pointers so that the pointers work as expected – you don't want to be stranded with a reference to a deleted object.

When allocating objects with `new`, think hard about who is supposed to `delete` them and make this very clear. Likewise if allocating using legacy (read: C) libraries.

When you `delete`, set the pointer to `0` (`C++11 nullptr`).

When passing a pointer or a reference as an argument, make them `const` if the function should not change the logical state of the corresponding objects.

If you find that you need to use `dynamic_cast`, think through your design again.

If you find that you need to use `const_cast`, think through your design again.

Always inline simple accessor member functions.

Never inline functions which are more than one or two lines long.

Optimize your code only if you **know** you have a performance problem.

BUT: Don't de-optimize your code for no reason. Passing a `const vector<int> &` instead of a `vector<int>` is not an optimization, it's good practice and does not complicate the code beyond adding a few characters.

Use the standard library, especially the containers and algorithms.

Never use built-in arrays or character strings.

If possible, write a working program before trying to replace `std::` with something special purpose.

If the standard library is not enough, try GPL'ed packages first before commercial ones. And only as a last resort roll your own.

Try to make your classes and functions general and reusable.

Separate the reusable from the problem-specific. (Example: In many of my projects I reuse a small function for seeding a random number generator from `/dev/random`)

Copying code within a project is generally bad. Refactor and generalize instead.

Try to make each class (hierarchy) as independent as possible from the rest of the classes in your project.

Last rule: All rules have exceptions.

Addendum: Case study, detector simulation

Simulate particles propagating through a detector

A particle propagates through the detector until it interacts with the material in the detector, decays or leaves the detector.

If a particle decays, propagate its decay products.

If a particle interacts and interaction products are produced, propagate these.

If a particle interacts with an active part of the detector, a hit is registered.

After all particles have been propagated, take all hits and reconstruct the tracks of all particles.

Simulate particles propagating through a detector

```
class Particle;
class Detector;
```

Choose `simulate(Particle &, Detector &);` rather than `Particle::simulate(Detector &);` or `Detector::simulate(Particle &);`

Choose `Detector::propagate(Particle &);` rather than `Particle::propagate(Detector &);` Or maybe propagate should really have been propagator and we should have `class Propagator` and `Propagator::propagate(Particle &, Detector &);`

through implies that the detector occupies space. We may need a `class Volume;`

`Detector` would typically be an abstract base class. "Silicon Vertex Detector", "Electromagnetic Calorimeter" or "Bubble chamber" are `Detectors`, but there is no plain detector.

`Particle` could be polymorphic. `class Lepton;`, `class Hadron;`, `class Nucleus;`, `class Electron: public Lepton;`

But maybe `Particle` is a concrete class which *HasA* set of quantum numbers.

Particle decays

`Particle::decay();`? Or maybe `class Decayer` and `Decayer::decay(Particle &);` or both?

A particle may have a pointer to a `Decayer` which knows how to decay the particle (`bool Decayer::canDecay(Particle &);`).

Particle interacts with material in detector

`class Material` may be polymorphic. A `Detector` has/is `Volume` and has/is `Material`.

Who knows how to do what? `Particle::interact(Material &);` or `Material::interact(Particle &);` or
`class Interactor; Interactor::interact(Particle &, Material &);`

A particle leaves the detector

Again `Detector` occupies a `Volume`.

A particle interacts and interaction products are formed

Trivial: interactions products (just as decay products) are also `particles` and should be propagated.

A particle interacts with an active part of the detector.

```
class DetectorPart: public Volume {
public:
  virtual void interact(Particle &);

  // ...

};

class ActiveDetectorPart: public DetectorPart {
public:
  virtual void interact(Particle &);

  // ...

};

class Detector {

  // ...

  SomeContainer<DetectorPart *> theParts;

};
```

```
class CompoundDetectorPart: public DetectorPart {
  // ...
  SomeContainer<DetectorPart *> theParts;
};

class AtomicDetectorPart: public DetectorPart, public Material {};

class Detector: public CompoundDetectorPart {};
```

A hit is registered

reconstruct tracks

```
class DetectorPart {
  // ...
  CompoundDetectorPart * parentPart;
  // ...
public:
  virtual void record(Hit * hit) {
    parent->record(hit);
  }
  // ...
};

class Detector {
  // ...
  SomeContainer<Hit *> hits;
  // ...
public:
  virtual void record(Hit * hit) {
    hits.insert(hit);
  }
  SomeOtherContainer<Track> reconstruct();
  // ...
};
```

A particle leaves a track in the detector.

Does a `Particle` have a `Track`? Or maybe a `Track` IsA `Particle`?

But some particles will leave no track (e.g. neutrinos). A single reconstructed track may correspond to two collinear particles or no particle at all.

`Particle` knows nothing about `Track`. Track may be associated with zero or more `Particles`.

```
class Track {
  // ..
  SomeContainer<const Particle *> associatedParticles;
  // ..
};
```