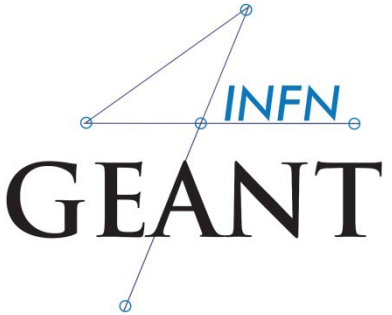


A minimal introduction to OO programming in C++

Luciano Pandola

INFN – Laboratori Nazionali del Sud

Originally based on a presentation by Maria Grazia Pia (INFN-Ge)

C++ and its versions

- C++ is a "live" language, which is evolving in time

  New features, libraries and functionalities are added, to meet new requirements and hardware developments (e.g. multi-threading)

- **Standardization** provided by ISO

  Back-compatibility provided (old codes compile also with new versions)

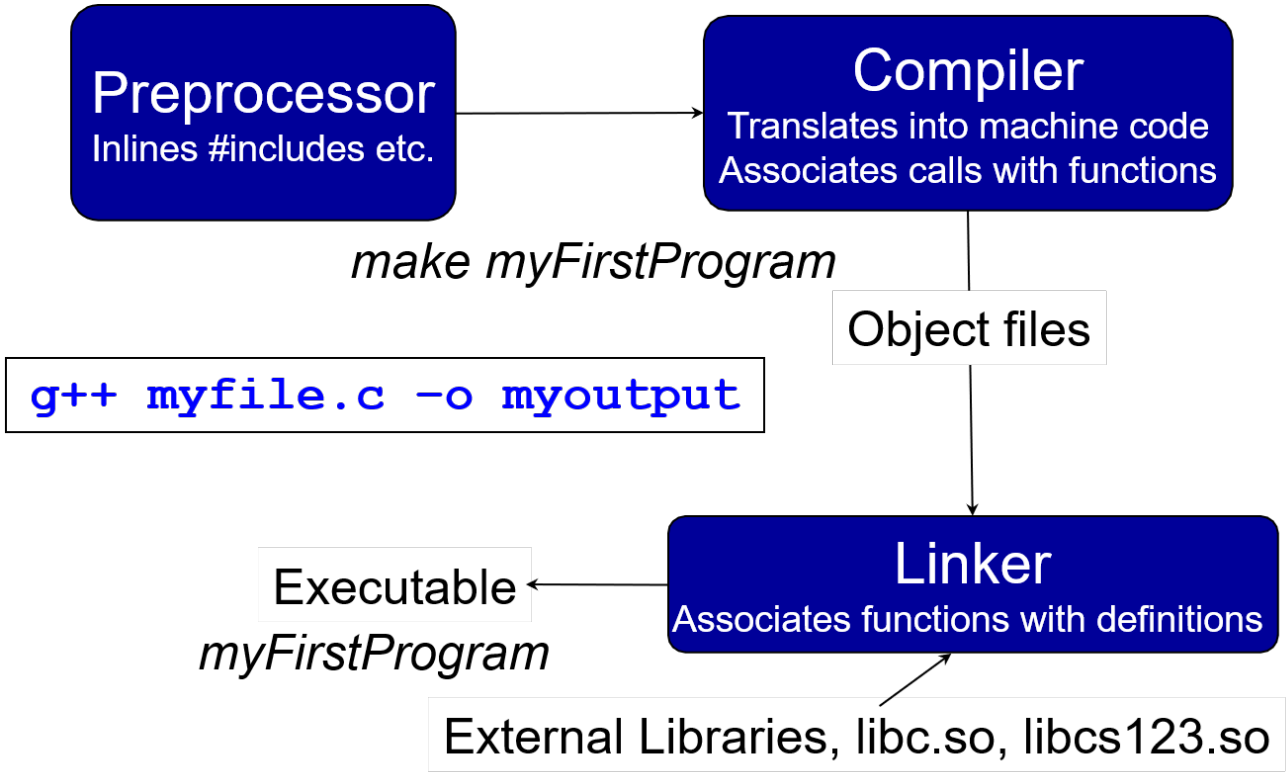
- C++03 used for a long time (ROOT, Geant4). Most scientific projects migrated to C++17 by now
- Getting ready to C++20

C++ standards

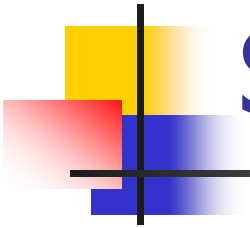
Year	ISO/IEC Standard	Informal name
1998	14882:1998[40]	C++98
2003	14882:2003[41]	C++03
2011	14882:2011[42]	C++11, C++0x
2014	14882:2014[43]	C++14, C++1y
2017	14882:2017[44]	C++17, C++1z
2020	14882:2020[45]	C++20, C++2a
2024	14882:2024[19]	C++23, C++2b
TBA		C++26, C++2c

Compilation

- C++ is a **compiled language**
  - Faster execution and prior syntax control with respect to interpreted languages



Some basic aspects of C++



- All statements must **end with a semi-colon ;**
  - Carriage returns (line breaks) are not meaningful
  - It is useful to use indentation to make the code more clear and readable
- Any C++ application **must** have a **`main()`** function, which returns an int
  - body enclosed in braces {}
- Comments preceded by **`//`**
  - Can start anywhere in the program either at the beginning or right after a statement in the middle of the line
  - Also syntax with `/* ... */`

Variables

Scope of variables

- global variables can be referred from anywhere in the code
- local variables: limited to the block enclosed in braces ({})

Initialization

int a = 0; // assignment operator
int a(0); // constructor

const

the value cannot be modified after definition

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  // declaring variables:
  int a, b; // declaration
  int result = 0;
  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
  // print out the result:
  cout << result << endl;
  const int neverChangeMe = 100;
  // terminate the program:
  return 0;
}
```

All variables MUST be declared

Most common operators

Assignment =

Arithmetic operators +, -, *, /, %

Compound assignment +=, -=, *=, /=, …

```
a+=5; // a=a+5;
```

Increase and decrease ++, --

```
a++; // a=a+1;
```

Relational and equality operators ==, !=, >, <, >=, <=

Logical operators ! (not), && (and), || (or)

Conditional operator (?)

```
a>b ? a : b
// returns whichever is greater, a or b
```

Explicit type casting operator

```
int i; float f = 3.14; i = (int) f;
```

Control structures - loops

for *(initialization; condition; increase) statement;*

```
for (int n=0; n<10; n++)
  {
    cout << n << ", ";
  }
```

Notice: the for loop is executed as long as the "condition" is true. It is the only necessary part of the for structure

The content of the loop must be included in curly brackets {}

- Brackets are optional if there is only one line

```
for (int n=0; n<10; n+2)
    cout << n << ", ";
```

reads until file is over

```
std::ifstream myfile("myfile.dat");
for ( ; !myfile.eof(); )
  {
    int var;
    myfile >> var;
  }
myfile.close()
```

Control structures – if's

```
if (x == 100)
 {
   cout << "x is ";
   cout << x;
 }
```

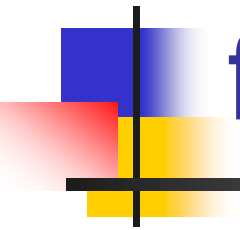
```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

shortcut for (x != 0)

```
if (x)
  cout << "x is not 0";
else
  cout << "x is 0";
```

```
if (x > 50 && x<100)
 {
    cout << "Hello!"
      << endl;
    double y = 2*x;
 }
else
  cout << "Ciao";
```

- The content of the if must be included in curly brackets {}
  - Brackets are optional if there is only one line



Quick intro: pointers and functions (a.k.a. methods)

Reference and pointers - 1

The address that locates a variable within memory is what we call a ***reference*** to that variable

```
x = &y;        // reference operator & "the address of"
```

A variable which stores a reference to another variable is called a **pointer**
Pointers are said to "point to" the variable whose reference they store

```
z = *x;        // z equal to "value pointed by" x
```

pointer

```
x = 0;         // null pointer (not pointing to any valid reference or
                  memory address → initialization)
```

Reference and pointers - 2

```
#include <iostream>
using namespace std;
int main ()
{
  double x = 10.; // declaration
  double* pointer = &x;

  //Let's print
  cout << x << endl;
  cout << pointer << endl;
  cout << *pointer << endl;
  cout << &x << endl;

  // terminate the program:
  return 0;
}
```

variable of type "double" (double-precision real) → value set to 10.

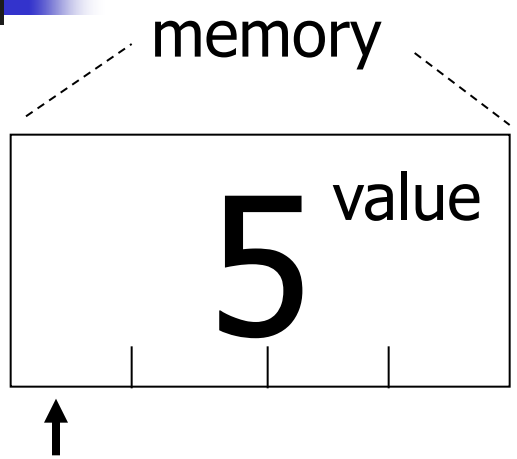
pointer for a "double" variable. Now it contains address of variable x

These lines will print the **content** of variable x (namely, 10)

These lines will print the memory address (=the **reference**) of variable x (something like 0xbf8595d0)

Notice: if we change the value stored in variable x (e.g. x=x+5), the pointer does not change

Pointers



address: 0x23ef14

int value → will take **4 bytes**

```
cout << sizeof(int) << endl;
```

```
int a = 5; //value

int* p = &a; /* address of
  memory where a is stored */

cout << *p << endl; /* content of
       memory address p */

cout << p << endl; /* location of
       memory address p */
```

Retrieving the content using a memory address can be much faster, especially if the content is not a "number" (int/float), but a more complex object/class

Bad and null pointers

- Pointers can point to invalid location in memory
  - Unpredictable behaviour or crash

No problem compiling

```
int main() {

int vect[3] = {1,2,3}; // vector of int

   int* c; // non-initialized pointer
   cout << "c: " << c << ", *c: " << *c <<endl;

   for(int i = 0; i<3; ++i) {  c = &vect[i];
   cout << "c = &vect[" << i << "]: " << c << ", *c: " << *c << endl;
   }

   // bad pointer
   c++; //go to next memory cell
   cout << "c: " << c << ", *c: " << *c <<endl;

   // null pointer causing trouble
   c = 0; //null memory location
   cout << "c: " << c << endl;  cout
   << "*c: " << *c <<endl;

   return 0;
}
```

BOOM → crash at run-time

```
$ g++ -o badptr1 badptr1.cc
$ ./badptr1
c: 0x7c90d592, *c: -1879046974
c = &vect[0]: 0x23eef0, *c: 1
c = &vect[1]: 0x23eef4, *c: 2
c = &vect[2]: 0x23eef8, *c: 3  c:
0x23eefc, *c: 1627945305
c: 0
Segmentation fault (core dumped)
```

Dynamic memory - 1

C++ allows for memory allocation at run-time (amount of memory required is not pre-determined by the compiler)

Operator new

pointer = new type;

```
Student* paul = new Student;
double* x = new double;
```

The operator gives back the **pointer** to the allocated memory area

If the allocation of this block of memory failed,
the failure could be detected by checking if paul took a null pointer value:

```
if (paul == 0) {
  // error assigning memory, take measures
};
```

Dynamic memory - 2

Operator delete

```
delete paul;
```

Dynamic memory should be **freed** once it is no longer needed,
so that the memory becomes available again for other requests of dynamic memory

Rule of thumb: every **new** must be paired by a **delete**

Failure to free memory: **memory leak** (→ system crash)

Dynamic vs. static memory

Two ways for memory allocation:

```
double yy;
double* x;
```

o static ("on the stack")

The amount of memory required for the program is determined at compilation time. Such amount is completely booked during the execution of the program (might be not efficient) → same as FORTRAN

```
double* x = new double;
*x = 10;
delete x;
```

o dynamic ("on the heap")

memory is allocated and released dynamically during the program execution. Possibly more efficient use of memory but requires **care**! You may run out of memory! → crash!

Functions - 1

```
Type name(parameter1, parameter2, ...)
{
  statements…;
  return somethingOfType;
}
```

No return type: void

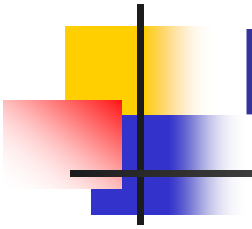
```
void printMe(double x)
{
  std::cout << x << std::endl;
}
```

In C++ *all* function parameters are passed by **copy**.

Namely: if you **modify** them in the function, this will **not affect** the initial value:

```
{
   double x = 10;
   double y = some_function(x);
   ...
}
```

x is still 10 here, even if x is modified inside some_function()



Functions - 2

Arguments can be passed by value and by reference

```
int myFunction (int first, int second);
```

Pass a **copy** of parameters

```
int myFunction (int& first, int& second);
```

Pass a **reference** to parameters
They may be **modified** in the function!

```
int myFunction (const int& first, const int& second);
```

Pass a **const reference** to parameters
They may **not** be **modified** in the function!

More fun on functions - 1

Notice: functions are distinguishable from variables because of () → they are required also for functions without parameters

Default values in parameters

```
double divide (double a, double b=2. )
{
  double r;
  r = a / b;
  return r;
}
```

```
int main ()
{
  cout << divide (12.) << endl;
  cout << divide(12.,3.) << endl;
  return 0;
}
```

More fun on functions - 2

Overloaded functions Same name, different parameter type

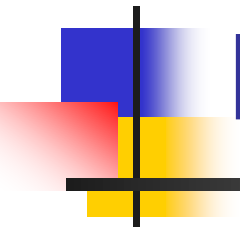
A function cannot be overloaded only by its return type

```
int operate (int a, int b)
{
  return (a*b);
}
```

```
double operate (double a, double b)
{
  return (a/b);
}
```

the compiler will decide which version of the function must be executed

```
{
  cout << operate (1,2) << endl;   //will return 2
  cout << operate (1.0,2.0)<< endl;   //will return 0.5
}
```



Basics of OO

OOP basic concepts

- Object and class
  - A class defines the **abstract characteristics** of a thing (object), including the thing's attributes and the thing's behaviour (e.g. a blueprint of a house)
  - A class can contain variables and functions (methods) → **members** of the class
  - A class is a kind of "user-defined data type", an object is like a "variable" of this type.
- Inheritance
  - "Subclasses" are more specialized versions of a class, which ***inherit*** attributes and behaviours from their parent classes (and can introduce their own)
- Encapsulation
  - Each object exposes to any class a certain ***interface*** (i.e. those members accessible to that class)
  - Members can be **public**, **protected** or **private**

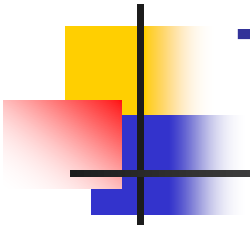
Class and object - 1

Object: is characterized by attributes (which define its state) and operations

A **class** is the blueprint of objects of the same type

```
class Rectangle {
  public:
  Rectangle (double,double); // constructor (takes 2 double variables)
  ~Rectangle() { // empty; } // destructor
  double area () { return (width * height); } // member function
  private:
  double width, height; // data members
};
```

an object is a concrete realization of a class → like house (= object) and blueprint (class). Many objects (= instances) of the same class possible



The class interface

How a class **“interacts”** with the rest of the world. Usually defined in a header (.h or .hh) file:

```
class Rectangle {
  public:
  // Members can be accessed by any object (anywhere else from the world)

  protected:
  // Can only be accessed by Rectangle and its derived objects

  private:
  // Can only be accessed by Rectangle for its own use.
  //No access by derived classes
};
```

Data members

```
class Rectangle {
 private:
    double width, height; // data members
};
```

- Data defined in the scope of a class are called data members of that class
- Data members are defined in the class and can be used by all member functions
  - The data (variables) are shared/visible by all methods
- Contain the actual data that characterizes the content of the class
- Can be public or private
  - public data members **are generally bad** and symptom of bad design

Class and object - 2

```
// the class Rectangle is defined in a way that you need two double
// parameters to create a real object (constructor)

Rectangle  rectangleA (3.,4.); // instantiate an object of type "Rectangle"
Rectangle* rectangleB = new Rectangle(5.,6.);  //pointer of type "Rectangle"

// let's invoke the member function area() of Rectangle
cout << "A area: "  <<  rectangleA.area()   << endl;
cout << "B area: "  <<  rectangleB->area() << endl;

//release dynamically allocated memory
delete rectangleB;                  // invokes the destructor
```

Private members

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

class Rectangle {
  public:
    Rectangle(double val1, double val2)
      {  width = val1;
         height   = val2;
    }
    double width() { return width; }
    double height() { return height; }

    void setWidth(double val)
      { width = val; }
    void setHeight(double val)
      { height = val; }
    double area() { return
    width*height; }
    void print() {
      cout << "Rectangle area: " <<
      area() << endl;
    }

 private:
 double width; // private data member!!!
 double height; // private data member
 };
```

Interface

```
int main() {

  Rectangle r1(1.1223,0.23);
  // setter with no return value
  r1.setWidth( 8.563 );

  // getter to access private data
  double x = r1.width();

  cout << "r1.width():" << r1.width()
       << " r1.height():" <<
       r1.height()
       << endl;
  r1.print();

  return 0;
}
```

```
$ g++ -o class2 class2.cc
$ ./class2
r1.width(): 8.563 r1.height(): 0.23
Rectangle area: 1.9694
```

Private methods

- Can be used only inside other class methods but not from outside

```
class Rectangle {
  public:
      Rectangle(double val1, double val2)
        {  width = val1;
           height   = val2;
      }
      double width() { return width; }
      double height() { return height; }

      void setWidth(double val)
        { width = value; }
      void setHeight(double val)
        { height = val; }
     double area() { return
     width*height; }
      void print() {
        cout << "Rectangle area: " <<
        area() << endl;
      }

  private:
    void reset(){width = 0; height = 0;};
  double width; // private data member!!!
  double height; // private data member
  };
```

Interface

```
int main() {

  Rectangle r1(1.1223,0.23);
  // setter with no return value
  r1.setWidth( 8.563 );

  cout << "r1.width():" << r1.width()
       << " r1.height():" <<
       r1.height()
       << endl;

  r1.reset(); //mistake

   return 0;
}
```

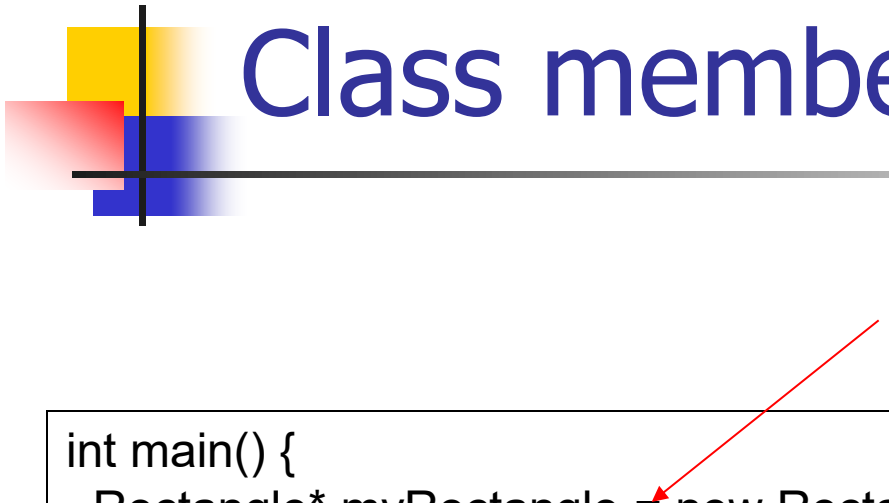
```
$ g++ -o class4 class4.cc  class4.cc: In function
`int main()':
class4.cc:20: error: `void Rectangle::reset()' is
private
class4.cc:35: error: within this context
```

Constructors

```
Rectangle::Rectangle(double val1,double val2)
{
  width = val1; height = val2;
}
```

- Special member functions
  - Required by C++ to create a new object
  - **Must** have the same name of the class
  - Used to initialize data members of an instance of the class
  - Can accept any number of arguments
  - Same rules as any other C++ function applies
- Constructors have no return type
- There can be several (overloaded) constructors
  - Different ways to declare an object of a given type

Class members




```
int main() {
  Rectangle* myRectangle = new Rectangle();  //won't work
  Rectangle* myRectangle = new Rectangle(3.,4.);
  double theArea = myRectangle->area(); //invokes a public member  (function)

  myRectangle->width = 10;  //won't work: tries to access a private member

 delete myRectangle; //invokes the destructor
};
```

Classes: basic design rules

- **Hide** all member variables (use public methods instead)
- **Hide** implementation functions and data (namely those that are not necessary/useful in other parts of the program)
- Minimize the number of public member functions
- Use **const** whenever possible / needed

OK:

A invokes a function of a B object
A creates an object of type B
A has a data member of type B

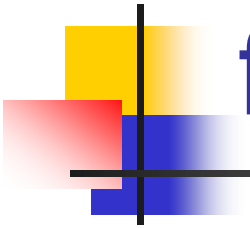
Bad:

A uses data directly from B
(without using B's interface)

Even worse:

A directly manipulates data in B

Separating class interface from implementation



- Clients of your classes only need to know the **interface** of your classes (→ where the doors of the house are!)
- Remember:
  - Users should only rely on public members of your class
  - Internal data structure must be hidden, as they are not needed in applications
- Compiler needs only the declaration of your classes, its functions and their *signature* to compile the application
  - Signature of a function is the exact set of arguments passed to a function and it return type
  - `void myfunction(double, int);` //signature!
- The compiled class code (implementation) is needed only at **link time**
  - Libraries are needed to link not to compile!

Inheritance

- A key feature of C++
- Inheritance allows to create classes derived from other classes
- Public inheritance defines an **"is-a"** relationship
  - *What applies to a base class **applies to its derived classes.*** Derived may add further details

```
class Base {
  public:
    virtual ~Base() {}
    virtual void f() {…}
  private:
    int b; …
};
```

```
class Derived : public Base {
 public:
    virtual ~Derived() {}
    virtual void f() {…}
    …
};
```

Flavours of inheritance

class Derived **: public** Base

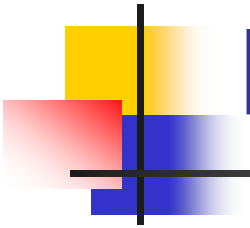
In **Base**	In **Derived**
public	**public**
protected	**protected**
private	**-**

class Derived**: private** Base

public	private
protected	private
private	-

class Derived**: protected** Base

public	protected
protected	protected
private	-



Polymorphism

- Mechanism that allows a derived class to modify the behaviour of a member declared in a base class → namely, the derived class provides an **alternative implementation** of a member of the base class

```
Base* b = new Derived;
b->f();
delete b;
```

Which f() gets called?

Notice: a pointer of the Base class can be used for an object of the Derived class (but **only** members that are present in the base class can be accessed)

Advantage: many derived classes can be treated in the same way using the "base" interface → see next slide

Inheritance and virtual functions - 1

```
class Shape
{
 public:
    Shape();
    virtual void draw();
};
```

Circle and Rectangle are both **derived classes** of Shape.

Notice: Circle has its own version of draw(), Rectangle has not.

```
class Circle : public Shape
{
    public:
        Circle (double r);
        void draw();
        void mynicefunction();
    private:
        double radius;
};
```

```
class Rectangle : public Shape
{
    public:
        Rectangle(double h, double w);
     private:
        double height, width;
};
```

Inheritance and virtual functions - 2

```
Shape* s1 = new Circle(1.);
Shape* s2 = new Rectangle(1.,2.);
```

Use a pointer to the **base class** for derived objects

```
s1->draw(); //function from Circle
s2->draw(); //function from Shape (Rectangle has not its own!)
s1->mynicefunction(); //won't work, function not in Shape!
Circle* c1 = new Circle(1.);
c1->mynicefunction(); //this will work
```

A virtual function defines the interface and provides an implementation; derived classes may provide alternative implementations

Abstract classes and interfaces

```
class Shape
{
    public:
        Shape();
        virtual area() = 0;
};
```

Abstract Interface
a class containing at least one
pure virtual function

A pure virtual function
defines the interface
and delegates the implementation
to derived classes (**no default**!)

Abstract class, cannot be instantiated:

```
Shape* s1 = new Shape(); //won't work
```

Abstract classes and interfaces

```
class Circle : public Shape
{
    public:
        Circle (double r);
        double area();
     private:
        double radius;
};
```

Concrete class

```
class Rectangle : public Shape
{
    public:
        Rectangle(double h, double w);
        double area();
      private:
        double height, width;
};
```

Concrete class

Concrete classes **must** provide their own implementation of the virtual method(s) of the base class

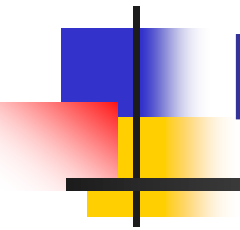
Inheritance and virtual functions

	Inheritance of the **interface**	Inheritance of the **implementation**
Non virtual function	Mandatory	Mandatory (cannot provide alternative versions)
Virtual function	Mandatory	By default Possible to reimplement
Pure virtual function	Mandatory	Implementation is mandatory (must provide an implementation)

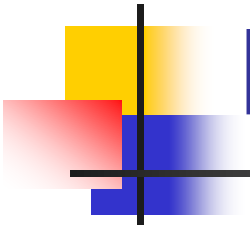
~~Shape* s1 = new Shape~~; *//won't work if Shape is abstract!*

Shape* s2 = new Circle(1.); *//ok (if Circle is not abstract)*

Circle* c1 = new Circle(1.); *//ok, can also use mynicefunction();*



Keeping code tidy



Header and source files

- We can separate the declaration of a class from its implementation
  - Declaration tells the compiler about data members and member functions of a class
  - We know how many and what type of arguments a function has by looking at the **declaration** but we don't know how the function is implemented
  - This is sufficient for compiling (not for linking, though)
- Declaration of a class Segment goes into a file usually called **`Segment.h`** or **`Segment.hh`** suffix
- Implementation of methods goes into the source file usually called **`Segment.cc`**

Organization strategy

Segment.hh

Header file: Class definition

```
double length();
```

Segment.cc

.cc file: Full implementation

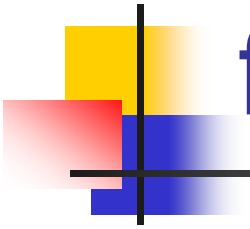
```
double Segment::length() {
  //some kind of implementation
}
```

main.cc

Main function

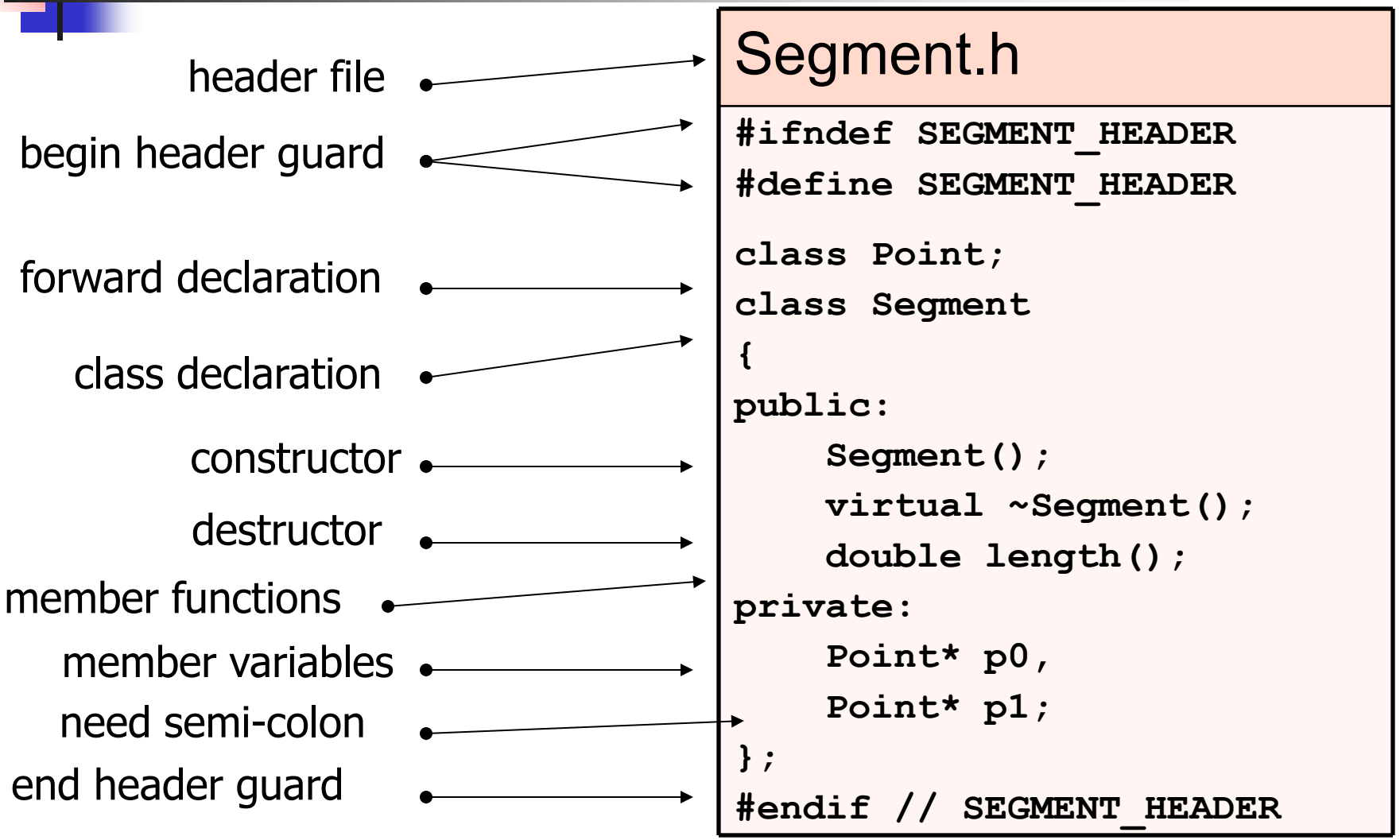
```
Double val = mySegment.length();
```

What should go in the header file (aka .hh)?



- Declaration of the class
  - Interface and data members
- All header files for types and classes used in the header
  - data members, arguments or return types of member functions
- Sometimes when we have very simple methods these are directly implemented in the header file
- Methods implemented in the header file are referred to as inline functions
  - For example, “getter” methods are a good candidate to become inline functions

How a header file looks like



Header and implementation

File Segment.hh

```
#ifndef SEGMENT_HEADER
#define SEGMENT_HEADER

class Point;
class Segment
{
public:
    Segment();
    virtual ~Segment();
    double length();
private:
    Point* p0,
    Point* p1;
};
#endif // SEGMENT_HEADER
```

File Segment.cc

```
#include "Segment.hh"
#include "Point.hh"

Segment::Segment() // constructor
{
    p0 = new Point(0.,0.);
    p1 = new Point(1.,1.);
}
Segment::~Segment() // destructor
{
    delete p0;
    delete p1;
}
double Segment::length()
{
    function implementation …
}
```

A Geant4 application:

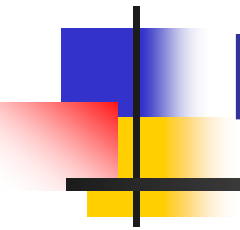
```
[user@vbox B1]$ ls
CMakeLists.txt  exampleB1.out  include       run1.mac  vis.mac
exampleB1.cc    GNUmakefile    init_vis.mac  run2.mac
exampleB1.in    History        README.md     src
[user@vbox B1]$
```

Main program

All headers (.hh) here

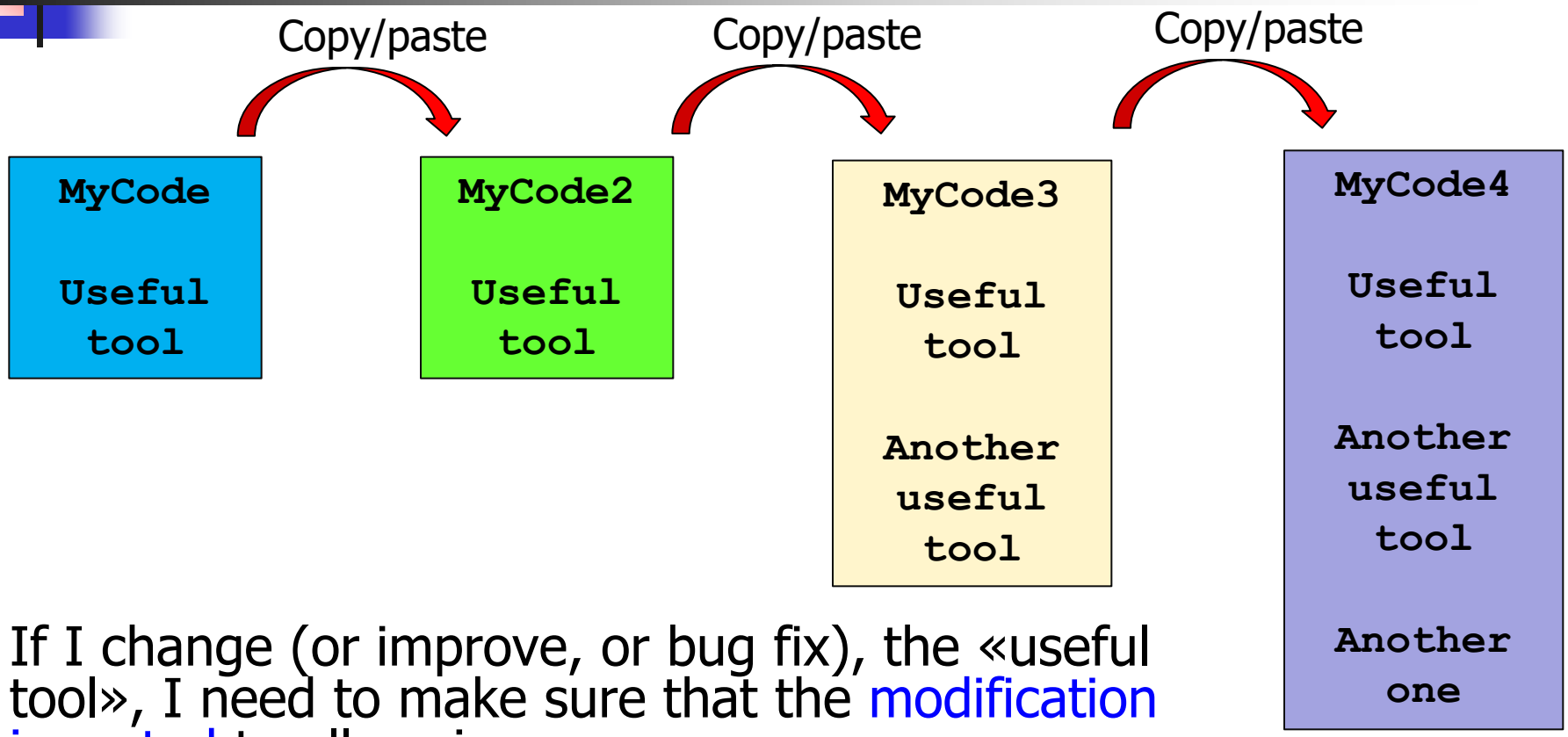
All implementations (.cc) here

.mac and .in are macro files, not compiled



Building bigger projects…

Modular structures are better!



- If I change (or improve, or bug fix), the «useful tool», I need to make sure that the modification is ported to all copies
  - **Error prone!**
  - Idea: modularize the general «useful tools»

How?

Preprocessor
Inlines #includes etc.

Compiler
Translates into machine code
Associates calls with functions

make myFirstProgram

Object files

```
g++ myfile.c -o myoutput
```

Linker
Associates functions with definitions

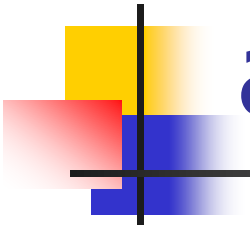
Executable
myFirstProgram

External Libraries, libc.so, libcs123.so

Compile vs. link

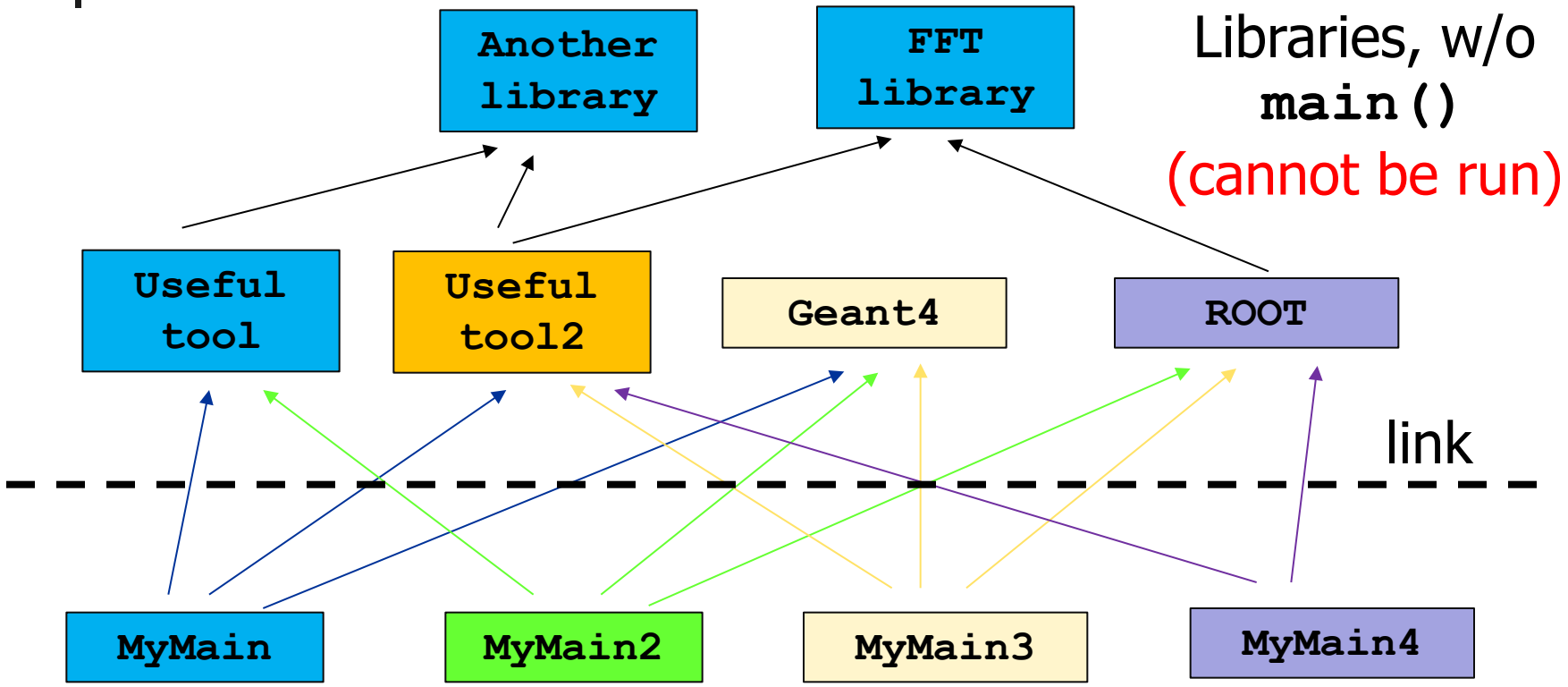
- The main function becomes the **program to run** when the compiler is finished linking the binary application
  - Compiling: translate user code in high level language into binary code that system can use
  - Linking: put binary pieces together, corresponding to methods used in the main function
- Compilation is **not sufficient** for running
  - Functions are declared and defined (the compiler knows which argument each function takes and what's its return type, i.e. the signature)
  - But what the function actually does is unspecified!
  - This generates an object file (not human-readable) which howewer **cannot be run** → it is a **library**
  - Object files are the ingredients for linking

Separating classes and applications

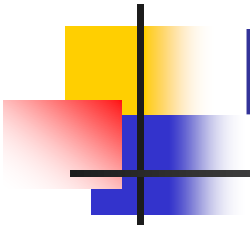


- It's good practice to separate classes from applications
  - Create one file with only your **`main()`** (=application)
  - Use **`#include`** directive to add all classes needed in your application
  - One may need "custom" classes, but also "library" classes (Geant4, ROOT, etc.)
  - Allows multiple usage of the same class
- Include compiled classes (or libraries) when *linking* your application
- If we have a general-purpose piece of code (without a main), we can just compile it
  - The library an be linked together with other libraries and a **`main()`** to produce executables
  - The same library is compiled once and can be used many times

A more efficient layout



Applications, w/ `main()` → executables



Pros and cons

- Only one copy of each library
  - It is re-usable and can be readonly
- Chains of libraries and of dependencies can get complicated
  - The linker must know where libraries and headers are stored in the system
  - If the linker fails to locate any of the required libraries, no executable
    - Or crash at run-time, for dynamic libraries
- Originally done by scripts, **`Makefile`**'s
  - Also allowing to set compiler options
  - More modern tools are now available

cmake

- Cross-platform free and open-source software application for managing the build process of software using a compiler-independent method
  - supports directory hierarchies and multiple libraries
  - can locate executables, headers and libraries
  - **`https://cmake.org/`**
- Gets an input (human-readable) script, **`CMakeList.txt`** and generates **`Makefile`**'s
- Used by Geant4 and by most modern projects
  - You'll get familiar with it
  - Tip: use a version of CMake that came out after your compiler

Now the fun begins ☺