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

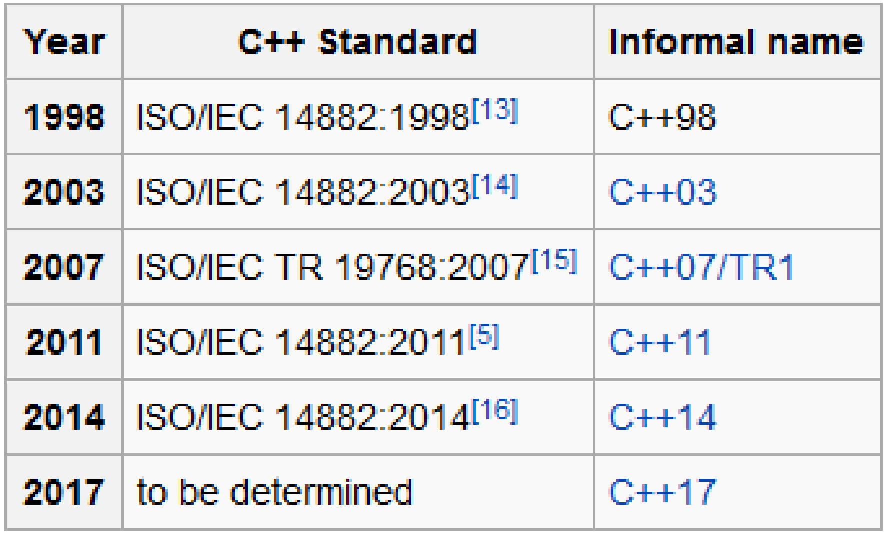
- C++ is a "live" language, which is evolving in time

  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 are migrating to C++11
  - ROOT 6, Geant4 10.2 (next)



Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998[13]	C++98
2003	ISO/IEC 14882:2003[14]	C++03
2007	ISO/IEC TR 19768:2007[15]	C++07/TR1
2011	ISO/IEC 14882:2011[5]	C++11
2014	ISO/IEC 14882:2014[16]	C++14
2017	to be determined	C++17

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

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

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
```

The class interface in C++

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
};
```

Class member functions

```
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
};

Rectangle::Rectangle(double v1,double v2)
{
 width = v1; height=v2;
}
```

Short functions can be defined “inline”. More complex functions are usually defined separately

type class::function()

(but costructor has **no type**)

Class members

constructor needs two parameters

```
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

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

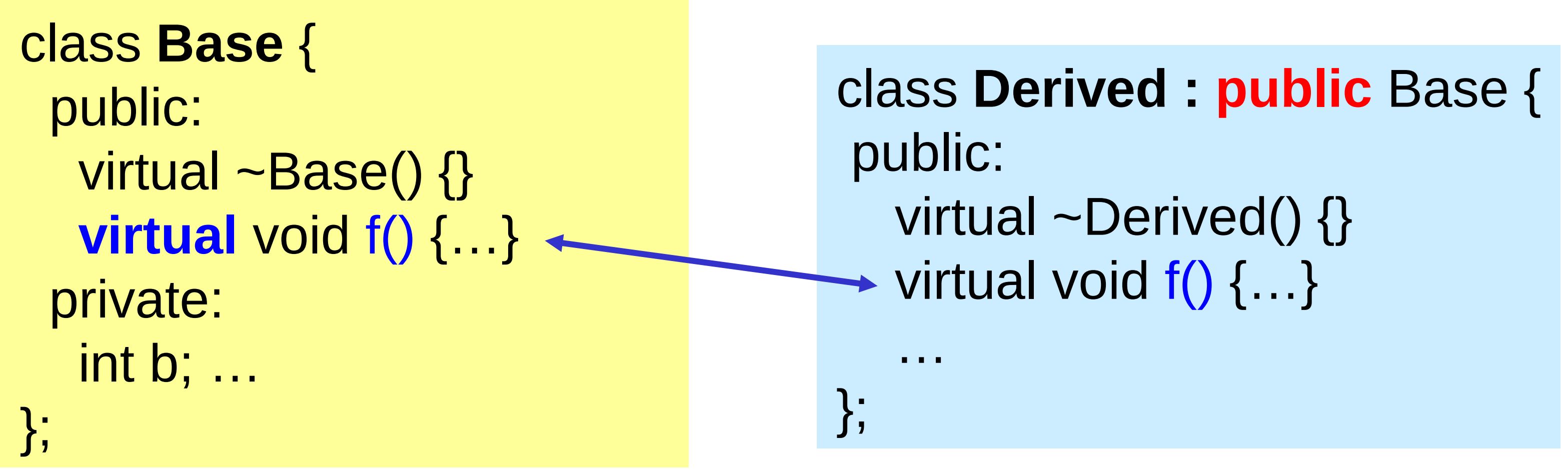
public	private
protected	private
private	-

class Derived**: protected** Base

public	protected
protected	protected
private	-

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() {…}
    …
};
```

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.);
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
```

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

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();*

Utilities from C++11

NEW

```
class Shape
{
 public:
    Shape();
    virtual void draw() const;
    virtual void Ciao(int i);
};
```

```
class Circle : public Shape
{
    public:
          Circle (double r);
          void draw()  override;
          void Ciao (size_t i) override;
};
```

Possible error: **signature mismatch**! The **draw()** and **Ciao()** in the derived class are seen as new functions, **not as overloads → OK for the compiler**

May cause that the version of the functions which is called is **different** from what it is intended, e.g. you want the draw() from Circle and you get the draw() from Shape.

Keyword (optional) **override:** say that the function is meant to override something from the base class → will cause a **compiler error** in case of signature mismatch

Utilities from C++11

NEW

In some cases you want to prevent a function to be overridden or even a **class to be derived** → **final** keywork

```
class Shape final
{
};
```

```
class Circle : public Shape
{
};
```

Compiler error

In some cases, the variable type can be deduced by the compiler → **why to explicitly write it**? Error-prone**.** Keyword **auto**

```
std::map<int,double> mymap;
for (map<int,double>::iterator itr = mymap.begin() ; itr<mymap.end(); itr++)
{}
```

```
std::map<int,double> mymap;
for (auto itr = mymap.begin() ; itr<mymap.end(); itr++)
{}
```

A few practical issues and miscellanea

Organization strategy

image.hh

Header file: Class definition

```
void SetAllPixels(const Vec3& color);
```

image.cc

.cc file: Full implementation

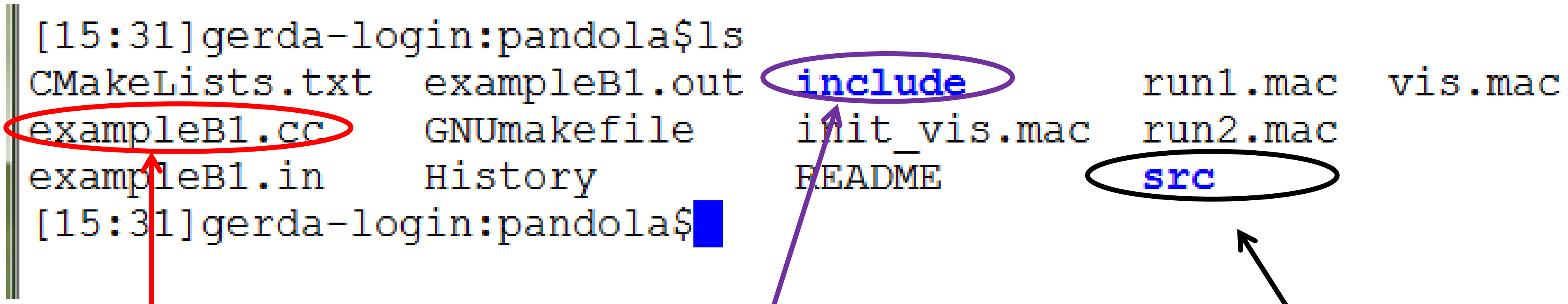
```
void Image::SetAllPixels(const Vec3& color) {
    for (int i = 0; i < width*height; i++)
      data[i] = color;
}
```

main.cc

Main function

```
myImage.SetAllPixels(clearColor);
```

A Geant4 application:



```
[15:31]gerda-login:pandola$ls
CMakeLists.txt  exampleB1.out  include       run1.mac  vis.mac
exampleB1.cc    GNUmakefile    init_vis.mac  run2.mac
exampleB1.in    History        README        src
[15:31]gerda-login:pandola$
```

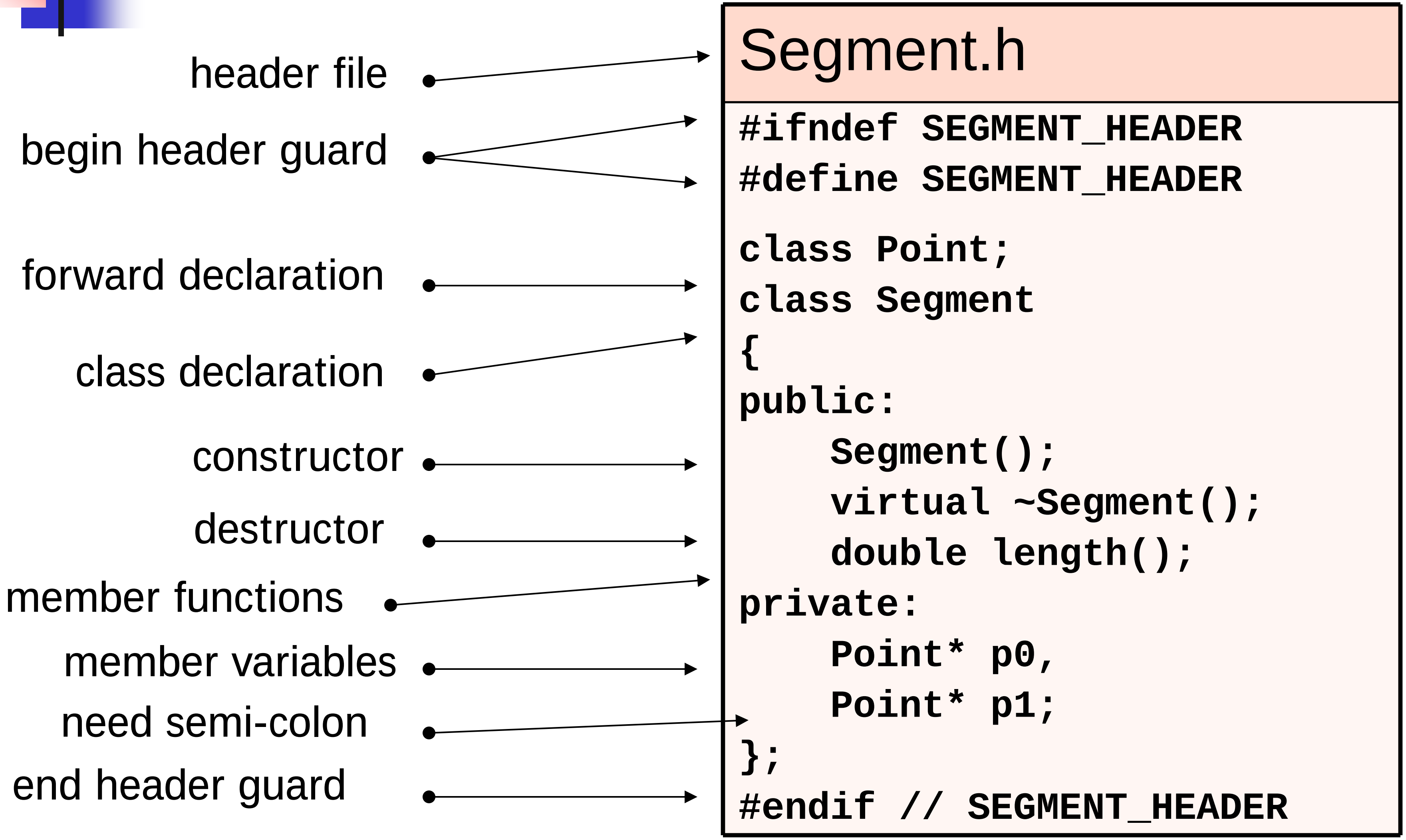
Main program

All headers (.hh) here

All implementations (.cc) here

.mac and .in are macro files, not compiled

How a header file looks like



header file

begin header guard

forward declaration

class declaration

constructor

destructor

member functions

member variables

need semi-colon

end header guard

Segment.h

```
#ifndef SEGMENT_HEADER
#define SEGMENT_HEADER

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

Forward declaration

Gui.hh

```
Class Gui
{
//
};
```

Controller.hh

```
//Forward declaration
class Gui;

class Controller
{
//...
private:
    Gui* myGui;
//...
};
```

- In header files, only include what you must
- If only pointers to a class are used, use forward declarations (than put the real #include in the .cc)

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 …
}
```

Segmentation fault (core dumped)

Typical causes:

```
int intArray[10];
intArray[10] = 6837;
//Remember: in C++ array index starts from 0!
```

Access outside of array bounds

```
Image* image;
image->SetAllPixels(colour);
```

Attempt to access a NULL or previously deleted pointer

These errors are often very difficult to catch and can cause erratic, unpredictable behaviour

More C++

Standard Template Library (STL)

Containers

- **Sequence**
  - **vector**: array in contiguous memory
  - **list**: doubly-linked list (fast insert/delete)
  - **deque**: double-ended queue
  - stack, queue, priority queue
- **Associative**
  - **map**: collection of (key,value) pairs
  - **set**: map with values ignored
  - multimap, multiset (duplicate keys)
- **Other**
  - **string**, basic_string
  - **valarray:**for numeric computation
  - bitset: set of N bits

Algorithms

- **Non-modifying**
  - find, search, mismatch, count, for_each
- **Modifying**
  - copy, transform/apply, replace, remove
- **Others**
  - unique, reverse, random_shuffle
  - sort, merge, partition
  - set_union, set_intersection, set_difference
  - min, max, min_element, max_element
  - next_permutation, prev_permutation

std::vector

use std::vector,

not built-in C-style array,

whenever possible

```
#include <vector>
void FunctionExample()
{
  std::vector<int> v(10);
  int a0 = v[3];     // unchecked access
  int a1 = v.at(3);  // checked access
  v.push_back(2);    // append element to end
  v.pop_back();      // remove last element
  size_t howbig = v.size();  // get # of elements
  v.insert(v.begin()+5, 2);  // insert 2 after 5th element
}
```

Dynamic management of arrays having size is not known a priori!

std::string

Example:

```
#include <string>

void FunctionExample()
{
  std::string s, t;
  char c = 'a';
  s.push_back(c);  // s is now “a”;
  const char* cc = s.c_str();  // get ptr to “a”
  const char dd[] = ‘like’;
  t = dd;  // t is now “like”;
  t = s + t;  // append “like” to “a”
}
```

std::atomic

- Meant for MT-programming. Avoid/reduce competition (= data race) when many cores handle the same variables
- **`count++`** means:
  1. read count value into a register
  2. increment register value
  3. write register back into count
- What if an other core checks/modifies **count** while the work is in progress?
  - Unpredictable behaviour
- Want the **block of instructions to be "atomic"** (i.e. inseparable, seen as a single operation), before an other core can access the variable

```
std::atomic<int> count;
count++; //working now
```

Backup

C++ “rule of thumb”

Uninitialized pointers are bad!

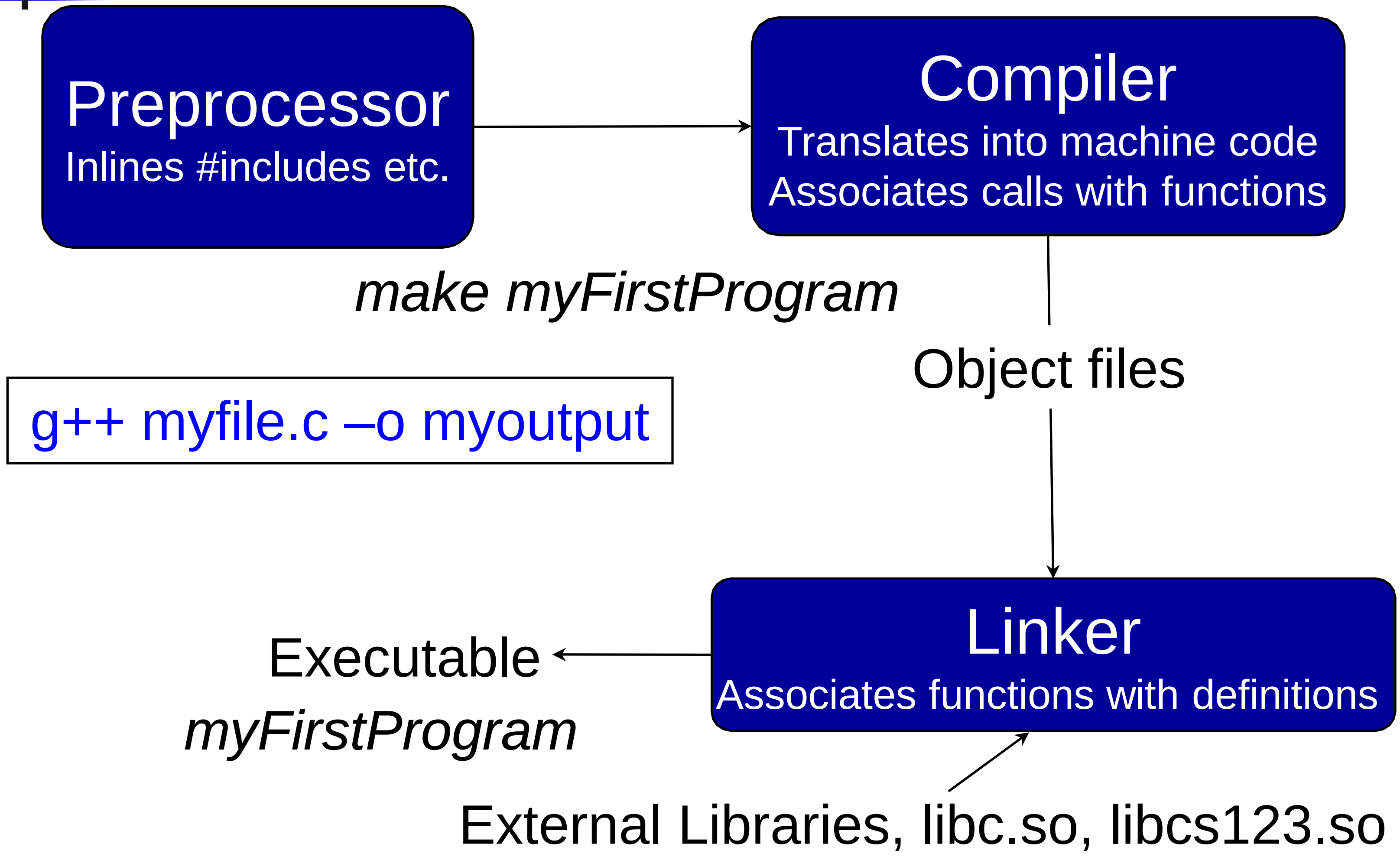
```
int* i;

if ( someCondition ) {
        …
        i = new int;
} else if ( anotherCondition ) {
        …
        i = new int;
}

*i = someVariable;
```

“null pointer exception”

Compilation



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

Getting started – 1

```
// my first program in C++
#include <iostream>
int main ()
{
  std::cout << "Hello World!";
  return 0;
}
```

// This is a **comment** line

#include <iostream>

- directive for the **preprocessor**
- used to include in the program external libraries or files

int main ()

- beginning of the definition of the **main function**
- the **main function** is the point by where all C++ programs start their execution
- all C++ programs **must have** a main function
- body enclosed in braces **{}**
- it returns a **"int"** variable (usually returning 0 means "all right")

Getting started – 2

```
// my first program in C++
#include <iostream>
int main ()
{
  std::cout << "Hello World!";
  return 0;
}
```

std::cout << "Hello World";

- C++ statement
- **cout** is declared in the iostream standard file within the std namespace, used to print something on the screen
- it belongs to the “std” set of C++ libraries → require std::
- cin used to read from keyboard
- **semicolon** (;) marks the end of the statement

return 0;

- the return statement causes the main function to finish

Namespace std

```
#include <iostream>
#include <string>
…
std::string question = “What do I learn this week?”;
std::cout << question << std::endl;
```

Alternatively:

```
using namespace std;
…
string answer = “How to use Geant4”;
cout << answer << endl;
```

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 - 1

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

```
for (n=10; n>0; n--)
 {
   cout << n << ", ";
   if (n==3)
    {
       cout << "countdown aborted!";
       break;
    }
  }
```

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

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

reads until file is over

Control structures - 2

```
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";
```

```
while (n>0) {
  cout << n << ", ";
  --n;
}
```

```
do {
  cout << "Enter number (0 to end): ";
  cin >> n;
  cout << "You entered: " << n << endl;
} while (n != 0);
```