C++ refresh and CMake

XXI Seminar on Software for Nuclear, Subnuclear and Applied Physics
9th June 2024

Carlo Mancini Terracciano
carlo.mancini-terracciano@uniroma1.it

INFN

SAPIENZA
UNIVERSITÀ DI ROMA

Foreword

- WiFi:

  - Ssid: Porto Conte Congressi

  - Pass: contecongress22

Outline

- Static and dynamic allocations

- Classes and inheritance

- Some C++ features largely used in Geant4

- An example of CMake usage

Plan...

Just an introduction

- This is not a C++ course

- Just few information useful to understand the Geant4 examples

- For a complete course: http://www.roma1.infn.it/people/rahatlou/index.php?link=Didattica&sublink=ppp

Few things about C++

- A general-purpose programming language

- Has imperative, **object-oriented** and generic programming features

- Provides facilities for low-level memory manipulation

- In 1983, "C with Classes" was renamed to "C++"
(++ being the increment operator in C)

- Initially standardised in 1998
(current standard is C++23 but the most used is C++17)

Memory management in C++

- Two ways of allocating memory

- Static Allocation:

  - Allocated on the stack

  - Automatic handling (allocation and deallocation)

- Dynamic Allocation:

  - Allocated on the heap using operators (**new** and **delete**)

  - Manual handling required

Stack Memory in C++

- Automatically managed memory that stores local variables and function call details

  - Fast access

  - Memory is managed automatically (pushed and popped)

  - Limited in size, leading to potential stack overflow

  - Local variables only exist while the function that created them is running

Stack allocation example
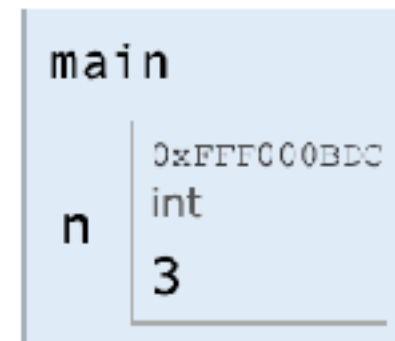
- Allocating on the stack is easy

```
C++ (C++20 + GNU extensions)
      known limitations

  1   int main()
  2   {
  3      int n;
  4      n=3;
→ 5   }

      Edit this code
```

Stack   Heap

```
main

         0xFFFC00BDC
         int
     n
         3
```

C/C++ details: [show memory addresses ∨]

https://pythontutor.com/render.html#mode=display

Static allocation example

- Allocated on the stack

- Stack deallocation is automatic when the variable goes out of scope

- When the function returns then memory for **m** is freed

```
C++ (C++20 + GNU extensions)
    known limitations

 1  void function()
 2  {
 3      int m=4;
 4      return;
 5  }
 6
 7  int main()
 8  {
 9      function();
10  }
```

Stack   Heap

main

function()
                OxFFFC0CBCC
                int
      m   4

C/C++ details: show memory addresses ∨

```
C++ (C++20 + GNU extensions)
    known limitations

 1  void function()
 2  {
 3      int m=4;
 4      return;
 5  }
 6
 7  int main()
 8  {
 9      function();
10  }
```

Stack   Heap

main

C/C++ details: show memory addresses ∨
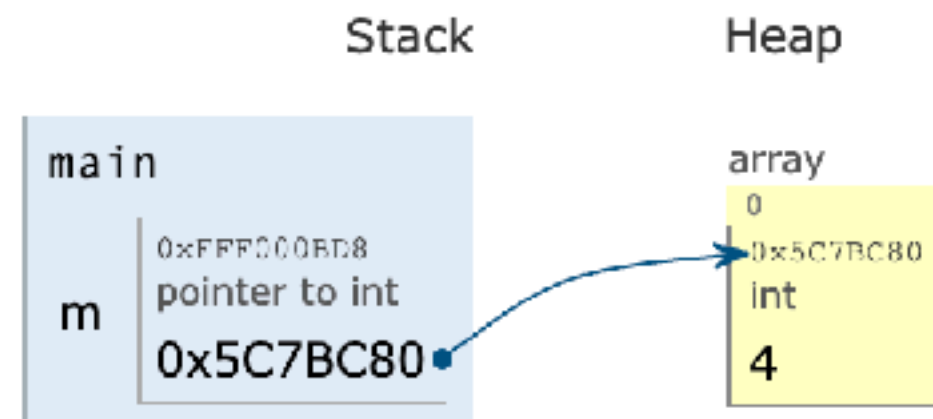
9

Heap memory

- Dynamically allocated memory that must be manually managed

  - Accessed via pointers

  - Slower to access compared to stack memory

  - Size only limited by the system's available memory

  - Requires explicit deallocation
    (using **delete** or **delete[]** in C++) to avoid memory leaks

Dynamic allocation example

```
C++ (C++20 + GNU extensions)
        known limitations

1   int main()
2   {
3     int *m=new int();
4     *m=4;
5   }
```

Edit this code

Stack

main

m | 0xFFFCC0BD8
pointer to int
0x5C7BC80

Heap

array
0
0x5C7BC80
int
4

C/C++ details: show memory addresses ∨

- Allocated on the heap

- With the operator **new**

- The result is a pointer to a location of memory

- The pointer is allocated in the stack

Stack and heap key differences

	Stack	Heap

Lifetime

Stack
- automatic lifetime, allocated and deallocated automatically

Heap
- manual management

Memory Management

Stack
- not need to be managed by the developer

Heap
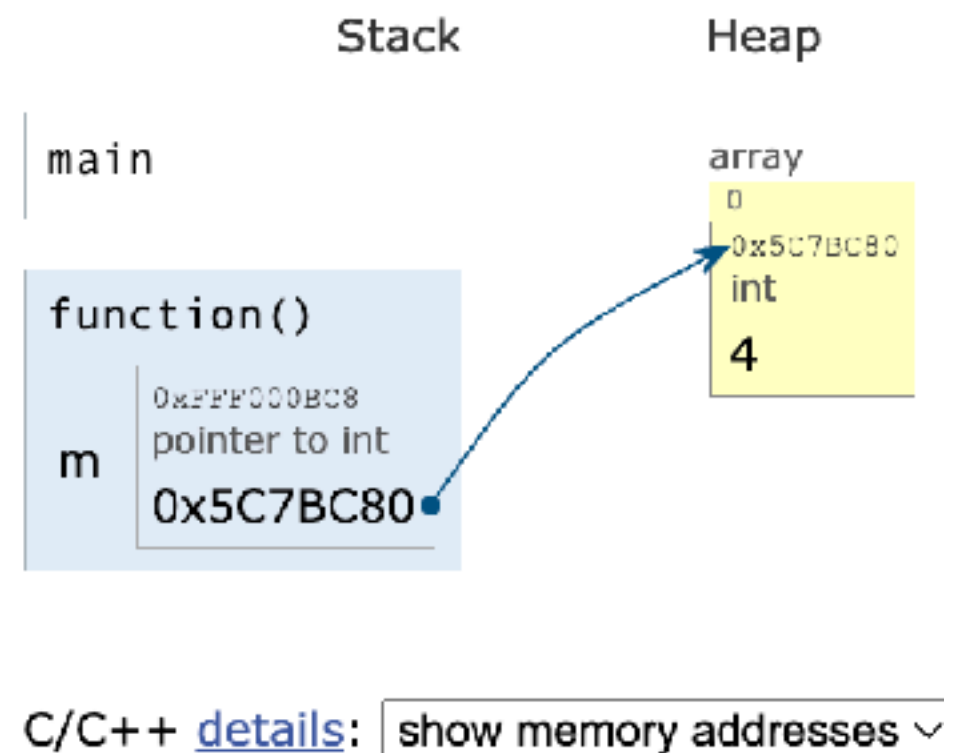- manual management to prevent leaks

Dynamic allocation example

```
C++ (C++20 + GNU extensions)
     known limitations

 1   void function()
 2   {
 3       int *m=new int(4);
 4       return;
 5   }
 6
 7   int main()
 8   {
 9       function();
10   }
```

Stack

Heap

```
main
```

```
function()

         0xFFF000BC8
  m      pointer to int
         0x5C7BC80
```

```
array
   0
  0x5C7BC80
   int
   4
```

C/C++ details: show memory addresses ⌄

- The memory in the heap is not freed when the function ends

- You have to **delete**

Stack and heap use cases

Stack

- small data that won't exceed the stack's limit and has a short lifetime

- Use where possible for simplicity and speed

Heap

- Ideal for large amounts of data or data that needs to persist beyond the execution of a function

- Use for large, complex data structures or when dynamic allocation is needed

- Always ensure that every **new** has a corresponding **delete**

Smart pointers

- Smart pointers are template classes in the C++ Standard Library that manage the lifetime of dynamically allocated objects

- They ensure automatic and appropriate destruction of dynamically allocated objects, helping to prevent memory leaks and dangling pointers

- Automatic Resource Management: Automatically deallocates memory when it's no longer needed.

- Memory Leak Prevention: Reduces the risks of memory leaks by ensuring proper deallocation

Most used smart pointers

- **std::unique_ptr**: Owns and manages another object through a pointer and destroys that object when the unique_ptr goes out of scope.

  - Use for exclusive ownership of dynamically allocated resources

- **std::shared_ptr**: Maintains reference counting for shared ownership of an object.

  - The object is destroyed when the last **shared_ptr** pointing to it is destroyed or reset

  - Use when you need multiple owners of the same resource

Example of smart pointers usage

```cpp
#include <memory>

// Automatically deleted when ptr goes out of scope
std::unique_ptr<int> ptr(new int(10));

// Better implementation:
auto ptrB = std::make_unique<int>(10);

// Reference-counted, safe for shared use
std::shared_ptr<int> shared1(new int(20));

// Better implementation:
auto sharedB1 = std::make_shared<int>(20);
```

Classes

- Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members

Like Plato's ideas (the idea of apple), classes have generic attributes (e.g. color). Each instance (this Golden Delicious apple) of the class have a specific attribute (e.g. yellow)

```
class Apple {
public:
    void setColor(color);
    color getColor();

private:
    color fColor;
    double fWeight;
};
```

Example of class usage

```cpp
#include <iostream>
using std::cout;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y)
{
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
```

Idea of rectangle

An instance
of rectangle

Example of class usage

```cpp
#include <iostream>
using std::cout;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y)
{
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
```

Declaration

Namespace

Implementation

Usage of the methods

Example of class usage

```cpp
#include <iostream>
using std::cout;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y)
{
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
```

Hyperuranion
(ὑπερουράνιος τόπος)
literally: "place beyond heaven"

"Real" world

What if I want to protect the rectangle
properties (the dimensions), once instantiated?

Constructors

```cpp
#include <iostream>
using std::cout;

class Rectangle {
    int width, height;
  public:
    Rectangle(int x, int y);
    int area() {return width*height;}
};

Rectangle::Rectangle(int x, int y)
{
  width = x;
  height = y;
}

int main () {
  Rectangle rect(3,4);
  cout << "area: " << rect.area();
  return 0;
```

Using the constructor and removing the setting method

Constructors

```cpp
#include <iostream>
using std::cout;

class Rectangle {
    int width, height;
  public:
    Rectangle(int x, int y);
    int area() {return width*height;}
};

Rectangle::Rectangle (int x, int y) :
width(x), height(y) { }


int main () {
  Rectangle rect(3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Better implementation!

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class

- This process, known as inheritance, involves a base class and a derived class

- The derived class inherits the members of the base class, on top of which it can add its own members
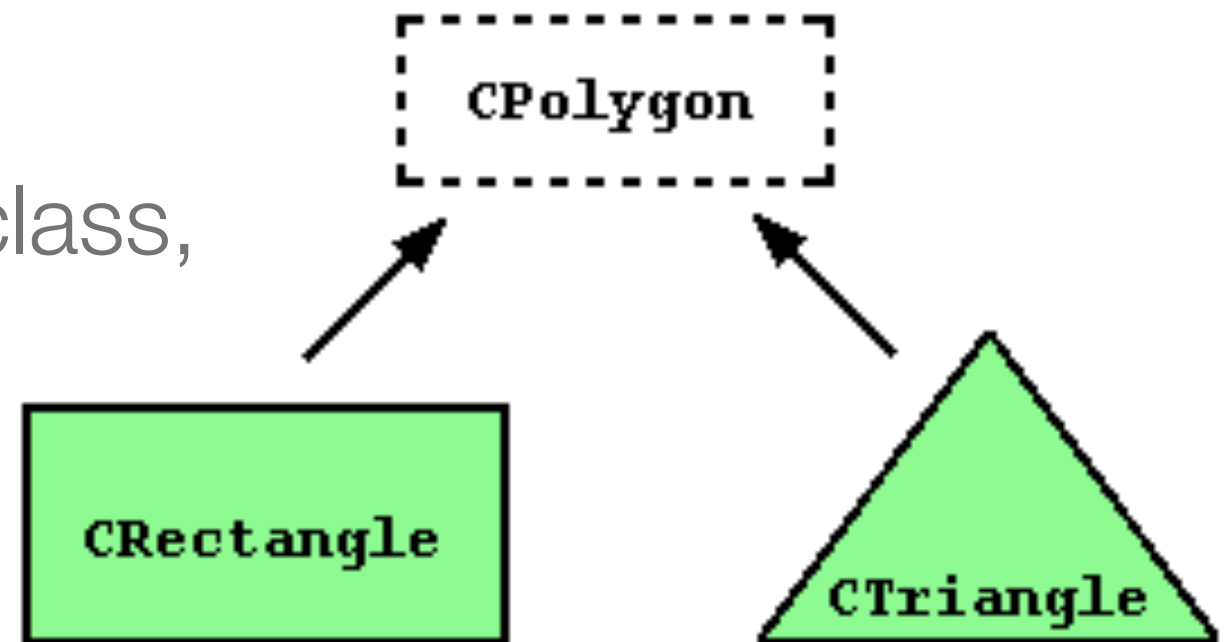
Inheritance, an example

```cpp
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};
```

```cpp
class Rectangle: public Polygon
{
 public:
  int area ()
  {
   return width*height;
  }
};
```

```cpp
class Triangle: public Polygon
{
 public:
  int area()
  {
    return width*height/2;
  }
};
```

Protected and not private!

- The protected access specifier used in class Polygon is similar to private. Its only difference occurs in fact with inheritance:

- When a class inherits another one, the members of the derived class can access the protected members inherited from the base class, but not its private member

- By declaring width and height as protected instead of private, these members are also accessible from the derived classes Rectangle and Triangle, instead of just from members of Polygon

- If they were public, they could be accessed just from anywhere

Public inheritance?

	Mother class members access specifiers		Daughter class members access specifiers
Public inheritance	Public	→	Public
	Protected	→	Protected
Protected inheritance	Public	→	Protected
	Protected	→	Protected
Private inheritance	Public	→	Private
	Protected	→	Private

Let's use the classes…

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

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

have a look at the example
https://github.com/carlomt/inheritance_example
for more details

CMake

- a 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, files, and libraries

- https://cliutils.gitlab.io/modern-cmake/

- use a version of CMake that came out after your compiler

- since CMake will dumb itself down to the minimum required version in your CMake file, installing a new CMake, even system wide, is pretty safe