Introducción a C++

Curso de Técnicas Experimentales Avanzadas en Física Nuclear

Master Inter-universitario de Física Nuclear, curso 2017-2018

A. Tolosa Delgado

Instituto de Física Corpuscular (CSIC-Universitat de València)

Índice

Introducción C++

Código fuente: conjunto de instrucciones (lenguaje humano)
--El código fuente es texto plano!

--Interpretado (ROOT) o compilado (ROOT, g++, clang, etc)

Debug & benchmarking: más fácil con un IDE, como Kdevelop

Programación estructurada: dividir las tareas en tareas más pequeñas y simples

No hay que reinventar la rueda, comprobar web antes!

http://www.cplusplus.com/doc/tutorial/
http://www.cplusplus.com/reference/
Foros: Stackexchange, Stack Overflow, etc

Pensar antes de programar (el 90% del esfuerzo es debug)

Hello world!

```
// File: main.cpp
1: #include <iostream>
2: int main()
3: {
4:    std::cout << "Hello World! \n ";   // Uso de “;”
5:    return 0;                                  // Uso de “;”
6: }
```

Directiva de precompilador : #include, #define, #ifdef, #pragma

Función “main” en todo programa externo (pero no en ROOT!)

Uso de “{}” para marcar un entorno/scope (función, bucle, etc)

Compilar (g++):

g++ main.cpp -o myFirstProgram

Hello world! Comentarios, variables

```
// Compilar: g++ main.cpp
// Ejemplo
// Autor
// Fecha


1: #include <iostream>


2: int main ( /* esta función main no tiene argumentos */  )
3: {
4:    double x ( 5.0 ) ;  // ejemplo de inicialización de variable
5:    int y = 3;      // ejemplo de asignación
6:    std::cout << x / y << std::endl ;
7:    return 0;
8: }
```

Variables

a) Una variable (u objeto) es una reserva de espacio en la memoria.

b) Se puede guardar información y modificarla más tarde.

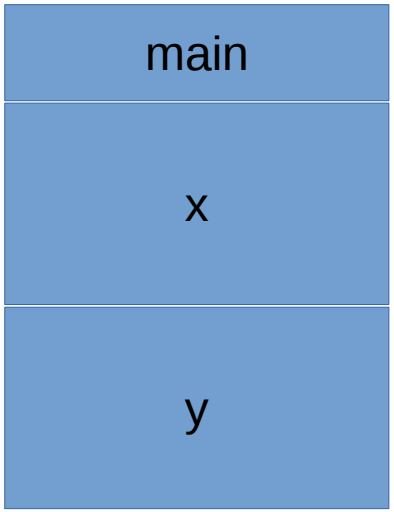
c) La memoria del ordenador = muchas cajitas, una detrás de otra.

d) Cada cajita tiene una dirección de memoria.

e) Cuando se define una variable se reservan las cajitas necesarias

f) El nombre de una variable evita trabajar con direcciones de memoria

En el código anterior, las cajitas de memoria en la línea 5:



Cuando se dice el tipo de variable/objeto, el compilador reserva la memoria suficiente y utiliza la información de una determinada forma

Ejemplo: int z(12345) reservará 4B, y escribirá en memoria 0...00001100000111001

Variables tipo “Plain Old Data” (POD)

El tamaño de los enteros es dependiente del compilador!

Type	Bits	Range
int	16	-32768 to -32767
unsigned int	16	0 to 65535
signed int	16	-31768 to 32767
short int	16	-31768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to -32767
long int	32	-2147483648 to 2147483647
unsigned long int	32	-2147483648 to 2147483647
signed long int	32	0 to 4294967295
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 3.4E+4932
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127

El tipo de variable dice:

a) El tamaño que ocupa en memoria (siempre el mismo)

b) La información que puede tener

c) Qué se puede hacer con esa información

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Control de flujo

If...else, while, do...while, switch, break, for (incremento & rango)

-See http://www.cplusplus.com/doc/tutorial/control/

```
1: #include <iostream>
2: int main() {

4:    int start(5), stop(15), step(3), loopCounter(0);  // inicializar siempre los POD!


   // for ( initialization  ; condition ;       increase                  ) { statements; }
5:    for ( int val = start; val < stop ; val += step , ++loopCounter )
6:    {
7:        std::cout << “Step: “ << loopCounter << “\t Val: “ << val << std::endl;
8:    } // end loop for, val


9:    return 0;
10: }
```

Variables puntero

Un puntero es un tipo de variable que guarda una dirección de memoria

--**Se debe especificar a qué tipo de variable apunta**

```
1: #include <iostream>
2: int main()
3: {
4:    int myVariable(5);          // inicializar siempre los POD!
5:    int * myPtr = 0;            // inizializar siempre los punteros!!
6:    myPtr = & myVariable        // & = operador dirección memoria
7:    *myPtr = 666;               // * = operador acceso a memoria (“indirection”)
8:    std::cout << myVariable << std::endl;
9:    return 0;
10: }
```

Variables puntero

Hay varios espacios de memoria:

-Global name space: las variables serán accesibles por cualquier función en cualquier momento

-Stack: variables locales, se limpian cuando se acaba el entorno, “{}” (bucle, función, etc)

-Free store: las variables definidas aquí persisten hasta que explícitamente se libera espacio (o el programa termina). Para reservar espacio se necesita la función “**new**”

```
1: int  x(3);                    // variable global
2: int main() {
3:   double * y = new double(44);    // variable local, puntero
4:    if ( y != NULL ){ std::cout << x + (*y) << std::endl ; }
5:    return 0;
6: }
```

Funciones

-Función: parte del programa que actúa sobre datos y que retorna un valor

--Todo programa de C++ debe tener una función “main”, que es llamada automáticamente en primer lugar

--Ésta puede llamar a otras funciones

--Sintaxis para la definición de una función:

```
ReturnType  functionName  ( type parameterName, etc...)
{
      statements;
      return [ReturnType];
}
```

```
1: double myDivision( double x , double y  )
2: {
3:    return  x / y;
4: }
```

Funciones

Pasar argumentos por **valor**, **referencia**, **puntero**

```
1: #include <iostream>
2: double myDivision( double x , double & y , double * result )
3: {
4:    if( result ) *result = x / y;
4:    else         std::cerr << “Null pointer” << std::endl ;
5:    return   x / y;
6: }

7: int main() {
8:   double x(10), y(7);
8:   double divisionRes(0);
9:   std::cout << myDivision( x , y , & divisionRes ) << std::endl;
10:    return 0;
11: }
```

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Funciones. Sobrecarga (polimorfismo)

Diferentes argumentos y/o retorno. El compilador elegirá la función correcta dependiendo de los argumentos.

```
1: double myDivision( double x , double y , double * result )  {
2:    if( result ) *result = x / y;
3:    else          std::cerr << “Null pointer” << std::endl ;
4:    return   x / y;
5: }

6: float myDivision( float x , float y , float * result )
7: {
8:    if( result ) *result = x / y;
9:    else          std::cerr << “Null pointer” << std::endl ;
10:    return   x / y;
11: }
```

Clases y objetos

Una clase es una colección de distintas variables y una colección de funciones asociadas a esas variables.

¿Cuándo hay que definir una nueva clase?

Los programas resuelven problemas reales. Los problemas complejos pueden resolverse usando los tipos “int” y “char”, pero suele ser más fácil escribir y entender el código si se crean representaciones de lo que se está tratando.

Por ejemplo, podemos definir un tipo “pieza”, que incluya su geometría y composición, y definir un “detector” como suma de muchas “piezas”. Hacer una simulación de un detector será así más fácil que usar “int” y “char”.

En general **HAY QUE USAR LAS CLASES DE ROOT/GEANT4**

Clases y objetos. Ejemplo (myIsotope.hpp)

```
1: #include <iostream>
2: #include <string>
3: class myIsotope
4: {
5:   public:
6:     myIsotope(): isoZ(0), isoName("") {}
7:     ~myIsotope(){ std::cout << "Bye...\n"; }
8:    std::string  GetName  (   )          { return isoName; }
9:    void SetName ( std::string & iN ){ isoName  = iN ;  }
10:    int   GetZ(          ){ return isoZ; }
11:    void SetZ( int iZ ){   isoZ = iZ;  }
10:  private:
11:     int isoZ;
12:     std::string isoName;
13: } ;   // importante terminar con ";" !!
```

Constructor

Destructor

Métodos de la clase (funciones)

Miembros de la clase (datos/variables)

Clases y objetos. Ejemplo (myIsotope.hpp)

```
1: #include <iostream>
2: #include <string>
3: class myIsotope
4: {
5:  public:
6:    myIsotope(): isoZ(0), isoName("") {}
7:    ~myIsotope(){ std::cout << "Bye...\n"; }
8:    myIsotope(int iZ): isoZ(iZ), isoName("") {}
9:    myIsotope(std::string & iN): isoZ(0), isoName(iN) {}
10:   myIsotope(std::string & iN, int iZ): isoZ(iZ), isoName(iN) {}
11:   myIsotope( myIsotope & inIso ): isoZ(inIso.GetZ() ), isoName( inIso.GetName() ) {}
12:   myIsotope   Clone();
...: } ;  // importante terminar con ";" !!
```

Constructor por defecto

Destructor

Constructor sobrecargado

Clases y objetos. Acceso a los miembros de la clase

Un objeto es una representación individual de una “clase”.

Declarar una clase dice al compilador cuánta memoria necesita reservar para cada objeto, y qué puede hacerse con esa información (métodos)

```
1: int x(5);
2: std::string aName( “Jorge” ); //std::string mejor que “char”
3: myIsotope galium( “Galium” , 31 );
4: myIsotope cooper( “Cooper” );
5:                  cooper.SetZ( 29 ); // Z es privada!, ~~cooper.Z = 3~~;
6: std::cout << cooper.GetZ() ;
7: myIsotope * uranium = new myIsotope( “Uranium” , 92 );
8: std::cout << uranium->GetZ() ;
9: std::cout << (*uranium).GetZ() ;
```

Clases y objetos. ¿Dónde definir una clase?

Es recomendable definir una clase en un fichero separado, e incluirlo en el programa principal (o en el intérprete) con la directiva **#include "myIsotope.hpp"**
En general, se suele **separar la declaración** de la clase en un fichero ".hpp" o ".h", **y la definición** de los métodos (funciones de la clase) en un fichero ".cpp,", ".cxx", ".C". Si la clase se usa en el input-output de ROOT (*diccionario*), la separación es obligatoria.

myIsotope.hpp

```
1: #include <iostream>
2: #include <string>
3: #ifndef  __myIsotope_hpp__
4: #define __myIsotope_hpp__
5: class myIsotope {
6:   public:
7:     myIsotope();
8:     ~myIsotope();
9:    std::string  GetName  (   );
10:    void SetName ( std::string & iN );
11:   int   GetZ(          );
12:   void SetZ( int iZ );
13:  private:
14:     int isoZ;
15:     std::string isoName;  } ;
16: #endif
```

myIsotope.cxx

```
1: #ifndef  __myIsotope_cxx__
2: #define __myIsotope_cxx__
3: #include "myIsotope.hpp"
4: myIsotope::myIsotope(): isoZ(0), isoName("") {}
5: myIsotope::~myIsotope() { std::cout << "Bye...\n";}
6:    std::string  myIsotope::GetName ()
                                  { return isoName; }
7:    void myIsotope::SetName ( std::string & iN )
                                  { isoName  = iN ;  }
8:   int    myIsotope::GetZ(          ){ return isoZ; }
9:   void myIsotope::SetZ( int iZ ){   isoZ = iZ;  }
10: #endif
```

Arrays y contenedores. Arrays (C++)

Array: es una colección de objetos. Se declara usando "[]", ie,

```
int arrayExample[5] = {0, 1, 2, 3, 4};
myIsotope isoArray[3];
```

-Para acceder al elemento n-ésimo se usa "[n]"

```
std::cout << arrayExample[1] ;
```

-El tamaño del array tiene que estar definido para el precompilador

```
int arrayLength = 8;
int anotherArray [ arrayLength ]; // no compilará!
```

-Se pueden declarar arrays multidimensionales

```
int matrixExample[2][2] = { {0, 1} , {2, 3} }; // los {} interiores se ignoran
```

Dificultades:

-¿Y si queremos añadir nuevos elementos al array?

>> Habría que crear un nuevo array con el nuevo elemento.

- ¿Y si accedemos a un elemento que no existe?

>> No tienen límites, intentar acceder a "arrayExample[100]" es posible pero el valor será absurdo

Arrays y contenedores. std::vector

std::vector: funciona de una forma similar a un array, pero resuelve los problemas anteriores (tiene límites y no hay que preocuparse por el tamaño)

```
std::vector< int > vExample; // 0 elementos
std::vector< int > vOtherExample ( 5 ); // 5 elementos, iniciados por defecto
std::vector< int > vAnotherExample (5 , -1 ); // 5 elementos, iniciados a "-1"
```

-Para acceder al elemento n-ésimo se usa el método"at(n)"

```
std::cout << vOtherExample.at(1) ; // at() checkea si el elemento existe
```

-Se pueden añadir elementos nuevos

```
vOtherExample.push_back( 66.6 );
```

-Si creemos que el vector puede ser muy grande, podemos reservar memoria, lo que disminuirá el tiempo necesario para añadir nuevos elementos

```
vOtherExample.reserve(50);
```

Ventajas std::vector frente a un array de C/C++

>> No hay que preocuparse por la gestión de memoria

>> Métodos propios que facilitan su manejo

http://www.cplusplus.com/reference/vector/vector/

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Arrays y contenedores. std::vector (looping)

-Como un array

```
for( int i=0; i< vExample.size() ; i++ )
{
      std::cout << vExample.at(i) << std::endl;
}
```

-Como un contenedor, con **iteradores** (C++)

```
for( std::vector< int >::iterator vit = vExample.begin();  vit != vExample.end(); ++vit )
{
      std::cout << *vit << std::endl;
}
```

-Como un contenedor, con **iteradores** (**C++11**)

```
for( auto vit : vExample )
{
      std::cout << vit << std::endl;
}
```

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Arrays y contenedores. Iteradores (looping)

-Los iteradores son una forma flexible de acceder a cada uno de los elementos de un contenedor (representan elementos individuales de un contenedor)

-Todos los contenedores tienen un método que devuelve el iterador al primer elemento (“begin()”) y al último (“end()”)

-Los iteradores de std::vector pueden ser avanzados con los operadores “+” o “-”, ie

```
std::vector< int >::iterator vit = vExample.begin();

vit = vit + 3; // avanza 3 posiciones, sólo std::vector
```

-En general, para avanzar iteradores se puede usar la función “**advance**”/”next”

```
std::advance( vit, 3 ) ;
```

WARNING: advance/next/prev **no comprueban si pasan los límites del contendor.** Debe comprobarse como un paso extra. La función **std::distance()** puede ser útil.

-Se pueden eliminar elementos de los contenedores con la función “**erase**()”. Los elementos pueden eliminarse de uno en uno, o especificando un rango, eg

```
vExample.erase(vit); // elimina el elemento correspondiente a “vit”

vExample.erase( vit, vit + 2 ); // elimina 2 elementos
```

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Arrays y contenedores. std::map

-Un mapa es una lista asociativa, como un diccionario. Los elementos están ordenados en el contenedor según una clave (*key*).

-Cada elemento de este contenedor tiene dos partes

→ Una "***clave***", que identifica al elemento del contenedor.

→ Un "**objeto mapeado**", asociado a cada "clave"

-**Ejemplo**: queremos ordenar por tiempo un conjunto de datos.

```
// std::map < key , mapped >

   std::map <  int  , double   >  gammaMap;
```

-Para insertar los elementos podemos usar las funciones insert/emplace

```
   gammaMap.emplace( 6 , 1460 );  // el retorno permite confirmar si se insertó
```

-Podemos buscar un elemento por la "clave", usando la función "find"

```
gammaMap.find ( 6 );
```

Esta función devuelve gammaMap.end() en caso de que "6" no exista, o un puntero al iterador con clave "6".

CSIC IFIC INSTITUT DE FÍSICA CORPUSCULAR VNIVERSITAT DE VALÈNCIA

Arrays y contenedores. std::map (looping)

-Para recorrer el **mapa** se utilizan **iteradores**,

```
for( std::map<int,double>::iterator mit = gammaMap.begin();
                                          mit != gammaMap.end();
                                          ++mit )
{     std::cout << “Time: “ << mit->first << “ “ << “Energy: “ <<  mit->second;        }
```

-Las funciones “lower_bound”/”upper_bound” pueden ser usadas para **seleccionar un rango** de elementos de un mapa

```
for( std::map<int,double>::iterator mit = gammaMap.lower_bound( 10 );
                                          mit != gammaMap.upper_bound( 20 );
                                          ++mit )
{     std::cout << “Time: “ << mit->first << “ “ << “Energy: “ <<  mit->second;        }
```

Bibliografía

Tutoriales:
http://www.cplusplus.com/doc/tutorial/
“Teach Yourself C++ in 21 Days” - Jesse Liberty
“Lecture on C++ and ROOT for physicists” - Deepak Samuel

Referencia (sobre uso)
http://www.cplusplus.com/reference/

Apéndice. Compilar en Linux con g++

Se distinguen dos etapas:
-Compilar: traducir el código humano en código máquina. Diferentes ficheros pueden compilarse por separado.
-Linkar: unir los fragmentos de código traducido y hacer un único ejecutable

Ventajas de compilar frente a interpretar:
-Más fácil localizar errores
-Programas más rápidos

Para más detalles, véase:
https://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm