Introducción a C++

Miguel Ángel Martín López
Grupo Universitario de Informática

Hora del Código
6 de Noviembre 2024

Contenidos

¿Qué es C++?

- Lenguage de programación multiparadigma (puedes hacer muchas cosas)
- Principales aplicaciones:
  - Desarrollo de Sistemas Operativos.
  - Videojuegos y Motores Gráficos (OpenGL)
  - Simulaciones Físicas
  - Desarrollo de Aplicaciones
  - ⋮
- Lenguage Compilado → con g++
- Control **casi** total de los recursos del ordenador.
- Compatibilidad con el Lenguaje `C`.

Hello World!

```cpp
#include <iostream>

int main(int argc, char **argv)
{
    // Comentarios de una linea
    std::cout << "Hola Mundo!" << std::endl;
    return 0;
    /*
        Comentario
        multilinea
    */
}
```

Y para compilar...

Usamos g++ para compilar. Un uso del compilador sería:

- `g++ archivo.cpp -o nombre_ejecutable`

Otras opciones que usar al compilar:

- `-g` para habilitar depuracion (con `gdb`)
- `-O0 -O1 -O2 -O3` niveles de optimización (0 por defecto)
- `-lm` si usamos la biblioteca matemática.
- `-l` en general, para enlazar otras librerias.

Tipos de datos básicos

- `int`: entero de 32 bits (4 bytes)
- `char`: caracter de 8 bits (1 byte)
- `short int` (`short`): entero de 16 bits (2 bytes)
- `long int` (`long`): entero de 64 bits (8 bytes)
- `bool`: true o false (1 byte)
- `float`: numero en coma flotante de 32 bits (4 bytes)
- `double`: numero en coma flotante de 64 bits (8 bytes)
- `unsigned`: para numeros enteros, numeros sin signo, desde 0 hasta $2^n - 1$.

Y auto?

La palabra reservada `auto` es especial.

- C++, al ser un lenguaje de tipado fuerte, las variables **no** pueden cambiar de tipo durante una ejecución.
- Solo es posible bajo ciertas condiciones, pero se escapan a este taller :(
- `auto` permite que el compilador detecte automáticamente el tipo que debería tener la variable, en función del contexto cercano.
  - `auto x = 10; //x será un int`
  - `auto x = 'a'; //x será un char`
  - `auto x = 1.2; //x será un float`

Operadores arimético-lógicos

- Suma: +
- Resta: -
- Multiplicación: *
- Division: /
- Resto de división entera: %
- Operador AND: &&
- Operador OR: ||
- Operador NOT: !
- Igualdad: ==
- Desigualdad: !=

Operadores especiales de C++: Operadores de flujo

- Son operadores especiales para utilizar variables dentro de streams o flujos de datos.
- Los streams en C++ es un concepto algo más avanzado, pero conocemos 2 de ellos: `std::cout, std::cin`
- `std::cin` sirve para introducir datos por teclado a través de la terminal. Un ejemplo de introducir dos enteros por teclado:

```
1 int num1, num2;
2
3 std::cin >> num1 >> num2;
4
5 // si introduzco por terminal lo siguiente:
6 // 189 12211
7 // num1 = 189 y num2 = 12211
```

Condicionales

Necesidad de una condición verdadera o falsa: `true`, `false`.
Para ello usamos la palabra reservada `if`.
Sintaxis:

```
1 if(condicion){
2     // codigo que se ejecuta si se cumple la condicion
3 }
4 // si no se cumple continua por aqui
```

if-else

`else` es otra palabra reservada que nos permite ejecutar una determinada sección de código si **no** se cumple la condición dada por un if.

```
1 if(condicion){
2     // Codigo que se ejecuta si se cumple la condicion
3 } else {
4     // Codigo que se ejecuta si NO se cumple la
          condicion
5 }
```

Bucle while

- Repetir una acción mientras se cumpla una determinada condición.
- Sintaxis:

```
while(condicion){
    // Codigo que ejecuta cada vez que se cumpla la
       condicion
}
// Cuando no se cumpla, sale del bucle y continua hacia
    abajo
```

Bucle do-while

- Similar al bucle while, pero se hace como mínimo 1 vez.
- Con diferencia, es el que menos se usa.
- Sintaxis:

```
do{
    // Codigo que se ejecuta al menos una vez
} while(condicion);
```

Bucle for

- Es el bucle más usado. Permite realizar una acción ya sea por una condición, o un número controlado de veces.
- Sintaxis algo más compleja, pero más útil:

```
for(accion_inicio ; condicion ; accion_final_iteracion){
    // Código a ejecutar
}
```

- `accion_inicio` es una acción que se realizará antes de entrar en el bucle
- `condicion`, una condicion que se evaluara con cada iteracion. Si se cumple, volvemos al bucle, si no salimos.
- `accion_final_iteracion`, una acción que se realiza al final de una iteración. Se realiza antes de comprobar la condicion.

¿Qué es una función?

- Es un fragmento de código separado al que se le puede llamar.
- Es como una especie de mini-programa dentro de nuestro código.
- Sirve para dividir una tarea compleja en partes más pequeñas y diferenciadas.
- Ejemplo:

```
1  // Funcion que imprime hola mundo
2  void hello_world(){
3      std::cout << "Hello World!" << std::endl;
4  }
5
6  //Funcion que devuelve true si
7  //el parametro x es multiplo de 3
8  bool es_multiplo_de_tres(int x){
9      return x % 3 == 0;
10 }
```

- void → no devuelve nada
- otra cosa → devuelve el tipo que indique la funcion

Primer Ejercicio

Programar una calculadora que muestre el resultado por terminal:

- Debe permitir las siguientes operaciones: Suma, Resta, Multiplicación y División.
- Cada cálculo debe hacerse en una función, y debe haber otra funcion que imprima el resultado por pantalla. (Una función suma, una función resta, etc...)
- El programa pedirá por pantalla que se introduzca una operación que se procesará con `std::cin`.
- El tipo de los numeros será `double`.

Structs o estructuras

Un struct es una serie de tipos de datos que están juntos en memoria. Podemos pensarlo como un tipo de dato compuesto.

```
1  // Declaración de un struct
2  struct ejemplo{
3      int x;
4      char algo;
5      double me_cago_en_todo;
6  };
7  // Creación
8  ejemplo un_ejemplo = {123, 'h', 10.45};
9  // Acceso a un elemento concreto
10 std::cout << un_ejemplo.x << std::endl;
11 std::cout << un_ejemplo.me_cago_en_todo << std::endl;
12 // Imprimirá 123
13 //           10.45
```

Clase

Una clase es muy similar a un struct, con la diferencia de que también permite tener funciones dentro. Ejemplo:

```cpp
#include <string>

class persona {
    int edad;
    std::string nombre; // Cadena de caracteres

    public:
        persona(int e, std::string n) : edad(e), nombre(n){};

        void saludar(){
            std::cout << "Hola! Me llamo " << nombre << "!" << std::endl;
        }
        int getEdad(){
            return edad;
        }
};
```

Objetos

Un objeto es la instanciación de una clase, es decir, darle los valores que queramos. Un ejemplo:

```
1 persona pedro(23, "Pedro");
2 // La variable pedro es un objeto
3
4 //otra forma de crear objetos
5 persona* juan = new persona(19, "Juan");
6
7 // Qué diferencia hay entre ellos?
```

Métodos de objetos

```
// Para ejecutar la funcion saludar:
pedro.saludar();
// Hola! Me llamo Pedro!

juan->saludar();
// Hola! Me llamo Juan!

/*
La diferencia está entre que
Pedro es un objeto en la pila
y Juan es un objeto en el heap.
*/
```

Ejemplo para entender punteros

- Yo tengo una casa, y esa casa tiene una dirección. (calle menganito...)
- Si la casa es una variable, su dirección es un puntero a la casa.

- En programación, las variables tienen una **dirección de memoria**. Podemos acceder a esa dirección con ciertos operadores.
- Si x es una variable entera, podemos acceder a su dirección de memoria así:

```
1 int x = 3;
2 int *p = &x;
3 // La variable p tiene la
4 // dirección de memoria de x
5 // Decimos que es de tipo int*
6 // que significa que apunta a un entero
```

Punteros en todo

Cada variable, da igual el tipo que tenga, tiene una dirección de memoria, por muy compleja que sea. Los operadores tienen los siguientes nombres:

- El operador de indirección, `*` tiene 2 usos:
  - para declarar variables puntero `int *p;` por ejemplo.
    También para otros tipos complejos, como el struct que vimos antes:
    `ejemplo *e;`
  - para acceder al valor de la variable a la que apunta la dirección de memoria en la variable puntero. (En el ejemplo de la casa es como mirar por la ventana).
    `int num = *p;` Aquí accedemos al valor al que apunta el puntero p.
  - **IMPORTANTE**: la variable no es `*p`, todo junto, la variable es 'p'. El '`*`' indica que es un puntero en su declaración
- El operador de referencia, &, que sirve para acceder a la dirección de memoria de una variable.

Vectores clásicos

Podemos hacer vectores del tipo que nosotros queramos, pero solo puede de ser un tipo. Por ejemplo:
`int vector[10];` Aquí creamos un vector de 10 elementos.
Podemos hacer también vectores de structs u objetos. Por ejemplo, un vector de personas:
`persona grupo[20];` Grupo de 20 personas.
Otra cosa interesante que se pueden hacer con los vectores es meter vectores dentro de vectores para formar matrices. Un ejemplo con enteros:
`int matriz[10][100];` Aquí tendremos una matriz de 10 filas y 100 columnas.
Ahora juntando punteros con vectores... ¿sabríais decir que es la variable `x` en el siguiente fragmento?
`char *(*(*x[][8])())[];` o el siguiente código:
`int *(*(*(**x[])(char*, int*(*)(char*)))[])(char**, char*(*)());`
(No lo intenteis entender, yo tampoco lo entiendo)

Los vectores que sí se usan

Usaremos la clase `std::vector` para vectores de tamaño indefinido, o si es necesario modificar su capacidad a lo largo de un programa.
Es, probablemente, la clase más usada de C++, junto con la clase `std::string`.
Como esta clase tiene muchos métodos, es mejor mirar la documentación oficial de C++ en la siguiente página: Documentacion C++ Un ejemplo de uso de la clase vector:

```
1 std::vector<double> *notas_fmat = new vector<double>();
2 for(int i = 0; i < 10; i++){
3     notas_fmat->push_back(i*0.1);
4 }
```

Índices en vectores

- Los índices en C++ empiezan en 0. El último indice válido es $n - 1$, si n es el tamaño del vector.
- Para acceder a un vector se usa la siguiente sintaxis: `v[indice]`.
- Ejemplo: imprimir todo el contenido de un vector

```
1 // vector.size() nos da el tamaño del vector
2 for(int i = 0; i < vector.size(); i++){
3     std::cout << vector[i] << std::endl;
4 }
```

- Otra sintaxis similar:

```
1 for(auto &elemento : vector){
2     std::cout << elemento << std::endl;
3 }
```

¡GRACIAS POR VENIR!