

Introducción a C++

Algoritmos y Estructuras de Datos II

DC-FCEyN-UBA

20 de Agosto de 2014

Agenda

- ▶ Java vs C++
- ▶ Tipos, variables y expresiones
- ▶ Funciones
- ▶ Estructura de un programa C++
- ▶ Compilación y linking
- ▶ Estructuras de control
- ▶ Estructuras de datos compuestas
- ▶ Instrucciones de preprocesador



```
1      // Archivo hola.cpp
2
3      #include <iostream>
4
5      using namespace std;
6
7      int main( ) {
8
9          cout << "Hola Mundo" << endl;
10         return 0;
11     }
```

- ▶ Línea 1: Comentario
- ▶ Línea 3: Directiva para incluir una librería estándar
- ▶ Línea 5: Declaración de uso de la librería estándar
- ▶ Línea 7: Definición de la función main
- ▶ Línea 9: Cuerpo de la función main
- ▶ Línea 10: Sentencia de retorno del programa

Java vs C++

Java vs C++: Donde y como se ejecuta?

Java: Corre en una maquina virtual (JVM).

C++: Se ejecuta como codigo de maquina nativo (compilado especialmente para cada arquitectura).

Java vs C++: Manejo de memoria

Java: Garbage Collection.

C++: Hecho a mano mediante new/delete (lo vemos con mas detalle en la clase de memoria dinamica).

Java vs C++: Pasaje de parametros

Java: Todos los tipos basicos (int, char, boolean y referencias) se pasan por valor. Todos los tipos no basicos, se pasan por referencia.

C++: Punteros, referencias y pasajes por valor soportado para todos los tipos. Lo vamos a ver mas adelante.



Tipos de datos primitivos

- ▶ `int` valor entero de 4 bytes
- ▶ `char` valor entero de 1 byte
- ▶ `bool` valor de verdad de 1 byte (`true` o `false`)
- ▶ `float` valor de punto flotante de 4 bytes
- ▶ `double` valor de punto flotante de 8 bytes

Los tamaños de los datos en realidad dependen del sistema para el que se compila el programa.

Hay otros tipos de datos y modificadores para alterar su precisión.

Por ejemplo: `unsigned`, `short`, `long`

Declaración de variables

La declaración de variables debe respetar el esquema:

```
tipo identificador;
```

De forma opcional puede inicializarse en el lugar de la declaración.

Ejemplos:

```
int i;
```

```
float pi = 3.14;
```

```
bool p = pi > 3.0;
```

```
char a('a'), b('b'), c('c');
```

Operadores

Las operaciones aritméticas básicas están incluidas en forma de operadores.

```
int a, b;  
a = 5; // operador de asignacion con un literal  
b = a; // operador de asignacion con una variable  
  
b = (2 * a + b / 3) % (a - 1); // aritmeticos  
  
bool p;  
p = a == b && !(a > 0 || b <= 0); // logicos  
  
a++; --b; // de incremento y decremento
```

- Los operadores son funciones y luego vamos a ver como redefinirlos

Scope de las variables

Las variables sólo pueden ser accedidas dentro del bloque `{}` en el que son declaradas. Una vez que se llega al `}` que cierra el bloque la variable es destruida y la memoria necesaria liberada.

```
int global;  
  
int main() {  
    int i = 1;  
    global = i + 1;  
    return 0;  
}
```

Estructuras de control

Los programas normalmente no están limitados a una ejecución secuencial de instrucciones, sino que durante el proceso se pueden realizar bifurcaciones y repeticiones de bloques de código.

Un bloque de código está delimitado por símbolos `{}` y define el scope de las variables declaradas dentro de él.

```
int main() {  
    int a;  
  
    // Nuevo bloque  
    {  
        int b = a; // Aqui a es visible  
    }  
    // Aqui b ya no es visible  
}
```

Estructuras de control

If-then-else

Las estructuras condicionales de tipo if-then-else permiten ejecutar distintos bloques de código a partir de si un valor booleano es verdadero o falso;

```
int a = 0;

if (a != 0) {
    a--;    // si la condicion resulta verdadera
} else {
    a++;    // si la condicion resulta falsa
}

if (a != 0)
    a = 0; // si la condicion resulta verdadera
```

Estructuras de control

While

La estructura de tipo `while` permite ejecutar **cero** o más veces un bloque de código mientras una condición booleana se mantenga verdadera.

```
int a = 0;

while (a > 0) {
    a++;
}
```



for

La estructura for permite realizar una inicialización de variables e indicar un procedimiento para “incrementar” la iteración. Mientras una condición booleana se mantenga verdadera un bloque de código puede ejecutarse **cero** o más veces.

```
for (int i = 0, j = 0; i <= 10; i++, j++) {  
    cout << i+j << endl;  
}
```

Equivalencia con el While:

```
int i = 0;  
int j = 0;  
while(i <= 10) {  
    cout << i+j << endl;  
    i++;  
    j++;  
}
```




Qué es una función ?

Una función es una secuencia de sentencias con un nombre que puede ser invocada en algún punto de un programa.

La creación de una función consta de dos pasos:

- Declaración: Consiste en declarar el nombre de la función, el tipo de retorno, y los tipos de sus parámetros, pero sin incluir el cuerpo de la función misma. Termina con ';'

```
tipo nombreF (tipo parámetro1, ..., tipo parámetroN) ;
```

- Definición: En la definición damos comportamiento a la función

```
tipo nombreF (tipo parámetro1, ..., tipo parámetroN) {  
    ...  
    sentencias  
    ...  
}
```



Las funciones pueden devolver uno o ningún valor.
Para devolver un valor, existe el *keyword* return
Devolver un valor

```
int max(int n1, int n2) {  
    if (n1 > n2) {  
        return n1;  
    } else {  
        return n2;  
    }  
}
```

Si queremos decir que la función no retornará ningún valor,
usamos el tipo especial void
No retornar nada

```
void printDash() {  
    cout << setfill('-') << setw(80) << 'A' << endl;  
}
```



Sobrecarga

En C++ dos funciones pueden tener el mismo nombre siempre y cuando el tipo y número de parámetros difiera. Esto se conoce como *overloading/sobrecarga*.

```
int add(int a, int b) {  
    return a + b;  
}  
  
float add(float a, float b) {  
    return a + b;  
}
```

Ejemplo de funciones

¿Cómo podemos escribir una función que calcule el factorial?

```
int factorial(int n) {  
    int fact = 1;  
    while (n > 1)  
        fact *= n--;  
    return fact;  
}
```

```
int factorial(int n) {  
    if (n <= 1) return 1;  
    else return n * factorial(n-1);  
}
```



La directiva `#include` inserta el contenido de un archivo en el punto donde se utiliza. En general se utiliza para incluir las declaraciones de la funcionalidad exportada por un “módulo”. Si bien puede utilizarse con otros fines, como por ejemplo inclusión de código directamente dentro de una función, tales usos no son recomendados.

```
// busca el archivo en los directorios de headers standard
#include <archivo>

/* primero busca el archivo en el directorio actual
si no lo encuentra lo busca en el standard */
#include "archivo"
```

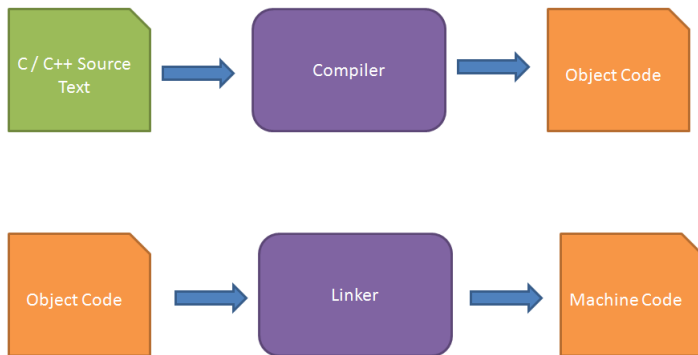
Standard input/output

`iostream` En el encabezado `iostream` de las librerías standard de C++ se encuentran declarados los objetos necesarios para escribir y leer de la consola. `[cout]` Permite escribir a la consola utilizando el operador `<<`

```
cout << "Le podemos pasar numeros -> " << 42 << endl;
```

`[cin]` Permite leer de la consola utilizando el operador `>>`

```
int n;  
cin >> n; // lee numeros de la consola hasta un enter
```



El compiler compila cada archivo por separado y genera un .o para cada uno.

El linker toma todos los .o y genera un solo ejecutable.



¿Qué es el compilador ?

El compilador es un programa que toma como entrada archivos con código fuente, cada uno representa una unidad de compilación, y los traduce a código objeto (lenguaje de máquina).

```
g++ -c hola.cpp -o hola.o
```

¿Qué es el linker ?

El linker es un programa que toma archivos con código objeto y los une en un único programa ejecutable (cuyo punto de entrada es la función main).

```
g++ hola.o -o hola
```

Alternativamente pueden hacerse ambos pasos simultáneamente:

```
g++ hola.cpp -o hola
```




¿Porqué compilar y luego linkear ?

La ventaja de hacer la compilación de cada unidad de compilación y luego el *linking* es que no es necesario recompilar todo los archivos con código fuente si sólo cambia un subconjunto estricto.

¿Qué son los makefiles ?

Los *Makefiles* son archivos con instrucciones para el proceso de compilación. La herramienta que interpreta estos archivos se llama *make*. Make detecta automáticamente si un archivo fue modificado desde la última compilación. Muchos proyectos *open source* distribuyen la configuración para la compilación en makefiles u otros formatos similares.

IDE's

Otra forma de manejar de forma automática el proceso de compilación es utilizando un *Integrated Development*



- ▶ punteros valor de 4 bytes con una dirección de memoria
- ▶ arreglos puntero al primer elemento de un bloque consecutivo
- ▶ referencias puntero con la interfaz del tipo apuntado

Nuevamente los tamaños de los datos en realidad dependen del sistema para el que se compila el programa.



¿Cómo se declara una variable de tipo puntero?

```
int* pi;  
char * pc1, *pc2;  
char * pc1, *pc2, **pc3, pc4;
```

¿Cómo se obtiene la dirección de memoria de una variable para asignarla a un puntero?

```
int a = 42;  
int* pa = &a; // operador de referencia &
```

¿Cómo se recupera el valor apuntado por un puntero?

```
a = *pa; // operador de derreferencia *
```

Más sobre punteros en la clase de memoria dinámica.



Los arreglos son secuencias de valores de un tipo dado consecutivas en memoria. El tamaño de los arreglos se fija en su construcción.

¿Cómo declaro una variable de tipo arreglo?

```
int    ai[4];  
bool   ab[200];  
char   ac[] = {'h', 'o', 'l', 'a'};
```

¿Cómo accedo al i-ésimo elemento del arreglo?

```
int    i0 = ai[0];  
char   c3 = ac[3];
```

Los arreglos en C++ no guardan su tamaño, por lo que es responsabilidad del programador conocerlo. De acceder a un elemento fuera del arreglo el comportamiento del programa puede volverse impredecible.

Puede verse como un puntero.



Las referencias mantienen internamente un puntero a un dato, pero en lugar de requerir el uso del operador de derreferencia permiten utilizar la misma interfaz que el tipo referenciado.

¿Cómo declaro una variable de tipo referencia?

```
int i;  
int& ri(i);  
char c, &rc(c);
```

¿Cómo utilizo una referencia?

```
int n = 2 * ri; // igual que el tipo referenciado  
rc = 'W';
```

Las referencias siempre deben apuntar a un dato válido, por lo que no pueden construirse si no se conoce la variable a referenciar. Una vez creada la referencia no puede ser alterada para referenciar a otra variable.

Pasaje de parámetros por referencia



Al utilizar punteros, arreglos o referencias como parámetros de una función sólo se copia el “puntero”, por lo que cambios en un parámetro son visibles al retornar de la invocación de la función.

```
void zero(int* a) {
    *a = 0;
}

void one(int& a) {
    a = 1;
}

void func() {
    int a = 3;
    zero(&a);
    cout << a << endl;
    one(a); // La referencia se inicializa automaticamente
    cout << a << endl;
}
```



Una estructura de datos es un conjunto de elementos agrupados con un nombre.

```
struct Student {  
    int    lu;  
    char  name[100];  
};
```

¿Cómo declaro una variable de tipo Student?

```
Student tito;
```

¿Cómo accedo a los miembros de una estructura?

```
tito.lu = 12304;  
cout << tito.name << endl;
```



¿Cómo declaro un puntero a una estructura?

```
Student* pStudent = &tito;
```

¿Cómo accedo a los miembros de un struct si tengo un puntero?

```
(*pStudent).lu = 12304;  
cout << pStudent->name << endl; // operador ->
```


Otros tipos de datos

El *keyword* `typedef` permite crear un sinónimo de tipos, lo que es particularmente útil cuando el identificador del tipo es largo o posee constantes que no queremos replicar.

```
typedef unsigned int nat;  
typedef char const * const chptr;
```

```
nat n = 0;  
chptr string = "hola";
```



Una clase es una estructura de datos que puede contener datos y funciones. Además, las clases permiten definir la visibilidad de sus miembros.

```
class Rectangle {  
    private:  
        int x, y;  
    public:  
        void set(int x, int y);           // Declaracion  
        int area() { return x * y; }     // Ambas  
};  
  
void Rectangle::set(int a, int b) {      // Definicion  
    this->x = a;  
    this->y = b;  
}
```

El *keyword* `this` es un valor especial con un puntero a la representación de la clase, este valor sólo tiene sentido utilizarlo dentro de la definición de un método de la clase.



¿Cómo se invocan los métodos de una clase?

```
int main(int argc, char** argv) {  
    Rectangle r;  
    r.set(10, 20);  
    Rectangle* pr = &r;  
    cout << pr->area() << endl;  
}
```

Al igual que para acceder a los miembros de una estructura el ‘.’ y el operador ‘->’ sirven para invocar los métodos sobre una instancia de la clase.

Clases III

Sobre una instancia constante sólo se pueden invocar métodos que garanticen que el estado interno de la clase no será alterado. Para ello se utiliza el *keyword* `const` al final de la declaración de los métodos constantes.

```
class Rectangle {  
    private:  
        int x, y;  
    public:  
        void set(int x, int y);  
        int area() const { return x * y; }  
};
```