

Introducción a C++ y Code::Blocks

Práctica Imperativo Clase 1

Brian Curcio - Emi Höss

Departamento de Computación, FCEyN, Universidad de Buenos Aires.

28 de mayo de 2012

Menu de hoy

- ▶ Funcional Vs. Imperativo (Intérprete Vs. Compilador).
- ▶ Transformación de estados.
- ▶ Sintaxis de C++.
- ▶ Entorno Code::Blocks (ejecución, compilación y debugging).
- ▶ Ejercicios en vivo y en directo.

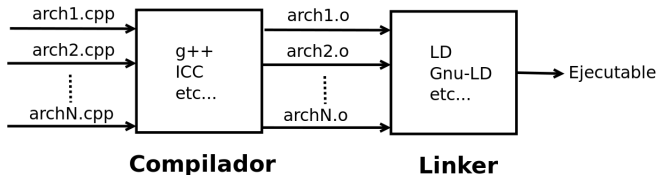
Imperativo

En la teórica ya vieron algunas diferencias con funcional:

- ▶ Los programas no son necesariamente funciones.
- ▶ Existe el concepto de **Variable** y de **cambio de estado**.
- ▶ Se amplía un poco el lenguaje de especificación (Vale, Implica, @, etc)

Interprete Vs. Compilador

- ▶ En funcional teníamos a Hugs que ejecutaba programas escritos en haskell.
- ▶ En C++ tenemos una serie de programas que toman nuestro código fuente y lo transforman en un archivo ejecutable.



Lenguaje C++

- ▶ Fue creado en la década del 80 por **Bjarne Stroustrup** (no, ni idea de como se pronuncia).
- ▶ La idea principal fue extender el lenguaje C para poder trabajar con objetos.
- ▶ Lenguaje amado y odiado por muchos.
- ▶ Lo que vamos a usar en la materia es solo una pequeña (pequeñiiiiisima) porción de todo lo que ofrece.
- ▶ Más adelante en la carrera van a sacarle el jugo (Algo II, Algo III, Métodos, etc) ahí van a entender porqué hay gente que lo odia.

Tipos Básicos

En C++ los tipos se escriben con minúscula.

- ▶ `char`
- ▶ `bool`
- ▶ `int` (`int`, `short int`, `long int`, `long long int`)
- ▶ `float` (`float`, `double`, `long double`)
- ▶ `void` (vacío)

Declaración de variables

- ▶ Cuando declaremos una variable obligatoriamente tenemos que indicar el tipo, y opcionalmente inicializarla, por Ej.:

```
int a;  
int a=3;  
char a='a';  
bool verdadero = true;
```

- ▶ Solo podemos usar la variable dentro del **scope** en el que fue declarada. Esto es, entre la anterior llave abierta, y la próxima llave cerrada.

Operadores

- ▶ Cuando trabajamos con **booleanos** tenemos los operadores que ya conocemos de haskell: `!=`, `==`, `&&`, `||`, `!`,
- ▶ Cuando trabajamos con **enteros** tenemos:
`!=`, `==`, `>=`, `<=`, `+`, `-`, `*`, `/`, `%(modulo)`,
- ▶ **const** Se suele colocar antes o despues de una variable o funcion, indicando que no debe modificarse la variable o lo que la función devuelva.

Arreglos

- ▶ Los arreglos son similares a las listas, pero permiten acceder directamente a cada uno de sus elementos sin tener que pasar por todos los anteriores.
- ▶ Tienen longitud fija, la misma debe indicarse en el momento en que se declaran.
- ▶ Para acceder a una posición donde queremos guardar o leer un dato ponemos el subíndice entre corchetes. Veamos ejemplos:

```
char b[100];    // Declaro un arreglo de char de
                // nombre b y de 100 posiciones.

b[13]='a';      // En el lugar 13 guardo la a.

b[1000] = 'c';  // Ojo!! nadie chequea que
                // me pase con los subindices!! y
                // de paso recordemos que se empieza
                // a numerar desde 0, o sea que el rango
                // va desde b[0] a b[99].

int a[] = {4,8,15,16,23,42}; // Otra forma, declaramos
                             // e inicializamos.

int num = a[0]              // En num tenemos al 4
```

- Al igual que Haskell y el lenguaje de especificación tenemos la estructura IF THEN ELSE.

```
int maximo(int a, int b)
{
    int res;

    if(a>b)
    {
        res = a;
    }
    else
    {
        res = b;
    }
    return res;
}
```

- También tenemos el WHILE y el FOR como otras dos estructuras de control para hacer ciclos.

```
void inicializarLista(bool listaDeTrues[], int tam)
{
    int i = 0;
    while(i < tam)
    {
        listaDeTrues[i] = true;
        i++;
    }
}

void inicializarLista(bool listaDeTrues[], int tam)
{
    for(int i = 0; i < tam; i++)
    {
        listaDeTrues[i] = true;
    }
}
```

Funciones

- ▶ Al igual que haskell, cuando definimos una función tenemos que indicar la aridad.

```
int sumar(int a, int b){...}  
bool espar(const int &a){...}  
void incrementar(int &a){...}
```

- ▶ Casi toda línea de código debe terminar con un **punto y coma**, esto divide el código en distintas partes que se ejecutan secuencialmente.

Funciones

- ▶ Existen dos maneras de declarar un parametro:
- ▶ Por **copia** es el caso general, cuando un parametro es pasado por copia cualquier modificación al mismo se pierde fuera del **scope** de la función.
- ▶ Por **referencia** es cuando se agrega un **&** pegado antes del nombre del mismo, en este caso se reflejan las modificaciones a un parametro fuera del **scope** de la función.

Funciones (Cont.)

- El operador `return` indica que en esa línea finaliza la función y no se ejecuta más nada. Puede estar seguida de un variable o expresión del mismo tipo de lo que devuelve la función. Cuando la función devuelve `void` no se pone nada.

```
int sumar(int a, int b){  
    return a+b;  
}  
  
int sumar1(int a, int b){  
    int res;  
    res = a+b;  
    return res;  
}
```

Funciones (Cont.)

```
► void sumarleUno(int &a){  
    a = a + 1;  
    return;  
}  
void sumarleUno(int &a){  
    a = a + 1;  
}  
void sumarleUno(int &a){  
    a++;  
}  
void sumarleUno(int &a){  
    a+=1;  
}
```


Todo programa en C++ tiene que tener una función llamada **Main**. Es una función como cualquier otra pero indica el "*Entry Point*" del programa, es decir, desde donde tiene que empezar a ejecutar.

- ▶ `int main(int argc, char** argv){...}`
- ▶ `int main(int argc, char* argv[]){...}`
- ▶ `int main(){...}`



¿Por qué devuelve Int?
¿Qué serán esos parámetros raros?
¿Los vamos a usar?

El int se usa para indicar si hubo error y los parámetros son para pasarle cosas al main, los vas a usar mucho en Orga 2, ahora ni te calentés.



Bibliotecas (Library)

- ▶ Muchas veces vamos a necesitar incluir *bibliotecas* en nuestro programa.
- ▶ Una *biblioteca* es un archivo donde hay definidas funciones que podemos usar, así nos ahorramos tener que estar escribiéndolas de nuevo cada vez.
- ▶ Con la instalación del compilador ya vienen varias.
- ▶ Para incluir una *biblioteca* usamos la directiva `#include<...>`, y si queremos incluir una *biblioteca* o archivo que está en el mismo directorio del proyecto principal tenemos que usar `#include"../../"`.

Por ejemplo para mostrar algún un mensaje por pantalla necesitamos el operador << que está en la *biblioteca* `iostream`:

```
#include <iostream>

using namespace std;

int main(int argc, char* argv[]) {
    cout << "Hola mundo!" << endl;
    return 0;
}
```



¿Qué es Namespace?
¿Qué es COUT, << y endl?
¿Qué más tiene iostream?

Es para no tener que andar escribiendo
std:: a cada rato, te va a quedar más
claro cuando veamos clases.

COUT es para mostrar algo por la pantalla
y endl es para que ponga un enter.
También tenés CIN para ingresar algo
por el teclado, después vas a ver ejemplos



Usando funciones

- ▶ Al igual que Haskell podemos llamar funciones desde otras funciones.
- ▶ En C++ tenemos dos formas de pasar parámetros a las funciones, por **referencia** o por **copia**.
- ▶ Por copia significa que a la función se le pasa otra variable nueva con el valor de la original.
- ▶ Por referencia significa que se le pasa una referencia (valga la redundancia) a la variable, si la función le cambia el valor se lo está cambiando a la variable original, puede ser un comportamiento deseado o no por lo que hay que tener cuidado.
- ▶ En C++ se indica con un **&** delante del nombre de la variable.
- ▶ Mejor veamos un ejemplo sino no se entiende nada.

Usando funciones

```
void decrementar(int &a){  
    a = a-1;  
}  
void incrementar(int a){  
    a++;  
}  
int main(int argc, char* argv[]){  
    int a = 10;  
  
    incrementar(a);  
    decrementar(a);  
    incrementar(a);  
  
    cout << a << endl;  
    return 0;  
}
```

¿Cuál es el valor que se muestra por pantalla? ¿Donde “vive” cada variable?

Usando funciones

¿Cuándo vamos a usar pasaje por referencia?

- ▶ En Algo 1 cuando lo sugiera la especificación, generalmente si el problema tiene algún parámetro de entrada que se modifica.
- ▶ En Algo 2 va a ser útil para cuando tengan que trabajar con estructuras muuuuuuy grandes.

Usando funciones

Supongamos que tenemos el archivo *cuadruple.cpp* con este código:

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    cout << "El cuadruple de 2 es: " << cuadruple(2) << endl;
    return 0;
}

int cuadruple (int a)
{
    return 4*a;
}
```

¿Por qué no funciona?

Usando funciones

- ▶ Cuando el compilador empieza a trabajar con nuestro código y llega a *cuadruple(2)* no sabe quién es esa función.
- ▶ La solución que primero viene a la cabeza es declararla al comienzo:

Usando funciones

```
#include <iostream>
using namespace std;

int cuadruple (int a)
{
    return 4*a;
}

int main(int argc, char* argv[])
{
    cout << "El cuadruple de 2 es: " << cuadruple(2) << endl;
    return 0;
}
```

Anda, pero no queda muy bueno. ¿Por qué?

Usando funciones

- ▶ Si hacemos esto tendríamos que preocuparnos de como ir acomodando todo para que funcione.
- ▶ La solución es declarar los **prototipos** de las funciones, o sea, al comienzo solo declaramos la función pero no la implementamos, de esta forma el compilador cuando llegue a *cuadruple*(2) ya sabe que *cuadruple* es una función que toma un *int* y devuelve un *int*, nos quedaría así:

Funciones

```
#include <iostream>
using namespace std;

int cuadruple (int a);

int main(int argc, char* argv[])
{
    cout << "El cuadruple de 2 es: " << cuadruple(2) << endl;
    return 0;
}

int cuadruple (int a)
{
    return 4*a;
}
```

Va tomando color, pero ¿está bueno que este todo en el mismo archivo? ¿Qué pasa si quiero usar *cuadruple* desde otro lado?

- ▶ La solución más elegante es separar en archivos diferentes la implementación de las funciones de su declaración.
- ▶ El archivo con las declaraciones lleva la extensión **h** (de Headers) y el de las implementaciones **cpp**.
- ▶ Luego incluimos el **.h** en todos los archivos que usen las funciones declaradas ahí.
- ▶ Para eso usamos la directiva `#include "archivo.h"`.
- ▶ También vamos a separar la función principal (main) de las otras funciones.
- ▶ Usando el ejemplo anterior nos quedarían 3 archivos *main.cpp*, *funciones.h* y *funciones.cpp*.

main.cpp

```
#include <iostream>

#include "funciones.h"

using namespace std;

int main(int argc, char* argv[])
{
    cout << "El cuadruple de 2 es: " << cuadruple(2) << endl;
    return 0;
}
```

funciones.cpp

```
int cuadruple (int a)
{
    return 4*a;
}
```


funciones.h

```
#ifndef FUNCIONES_H
#define FUNCIONES_H

int cuadruple (int a);

#endif
```



¿iWTFi? qué es todo eso
de ifndef?

Es para evitar las multiples definiciones
de los headers, lo que hace es decirle al
compilador "Si alguien ya me definio no
me vuelvas a incluir"



Code::Blocks

- ▶ CodeBlocks es un Entorno Integrado de Desarrollo (**IDE**).
- ▶ No es un compilador, interactua con el compilador.
- ▶ Nos provee de muchas facilidades a la hora de programar y buscar errores.

¡¡Vamos al Code::Blocks!!








Crear un nuevo proyecto

- ▶ File → New → Project.
- ▶ Elegimos “*Console application*”.
- ▶ Como lenguaje usamos C++.
- ▶ Primero va el título del proyecto, después el directorio donde lo vamos a guardar, lo demás se llena solo.
- ▶ En la siguiente pantalla tienen que estar marcadas las dos opciones.
- ▶ Le damos a “*Finish*” y listo. A la izquierda tenemos un árbol con nuestros archivos.


Incorporar archivos a nuestro proyecto

- ▶ Copiamos los archivos al directorio de nuestro proyecto.
- ▶ Project → Add Files.
- ▶ Marcamos los archivos a agregar.
- ▶ Tildamos “*Debug*” y “*Release*”.

Sacando los bichos (vulgarmente conocido como Debuggear)

- ▶ Para poner un “*Breakpoint*” tenemos que hacer click a la derecha de los números de línea (los que están a la izquierda del código).
- ▶ En “*BuildTarget*” ponemos “*Debug*”.
- ▶ Hacemos un “*rebuild*” → 
- ▶ Para arrancar → 
- ▶ Para avanzar a la próxima instrucción → 
- ▶ Para meternos dentro de una función → 
- ▶ Para salir de una función → 

Sacando los bichos (vulgarmente conocido como Debuggear)

- ▶ Para ver el valor de las variables hacemos click en →  y tildamos “*Watches*”.
- ▶ Haciendo click con botón derecho y seleccionando “*Add Watch*” podemos agregar variables a observar.
- ▶ Se puede cambiar el valor de las variables durante la ejecución, solo hay que hacer botón derecho sobre la variable en cuestión.

Fin de esta clase “*Especial*” de taller =)



¿CONSULTAS?