Auxiliar Nº2 – Introducción C++

CC7515 - Computación en GPU

Profesora:

Nancy Hitschfeld

Auxiliares:

Pablo Pizarro R. Sergio P. Salinas

Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

16 de marzo de 2022

Contenidos

Introducción a C++ – ¿Por qué estudiamos este lenguaje?

- Este lenguaje de programación lo usaremos para todas las tareas del semestre
- Creado en 1979, por Bjarne Stroustrup
- Multiparadigma (*imperativo, funcional, lógico, declarativo, orientado a objetos*)
- Tipado fuerte
- **Eficiente**

Introducción a C++ – El clásico HELLO-WORLD

Código 1: 01_helloworld.cpp

```cpp
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Introducción a C++ – El clásico HELLO-WORLD

Código 2: 01_helloworld.cpp

```cpp
#include <iostream> // Importación de librerías

int main() { // Función principal, retorna 0 (sin errores) o cualquier cosa (error)
  std::cout << "Hello World!" << std::endl; // Pasamos “Hello World!” al I/O de std::cout de la salida estándar
  return 0; // Si no añadimos esto el compilador lo hará solo
}
```

Componentes – Variables

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point

Código 3: 02_variables.cpp

```cpp
int main() {
  // Números
  int a = 1;
  float b = 2.3;
  double c = 4.5;

  int d;
  d = 6;

  // Caracteres
  char e;
  e = 'A';

  // Booleanos
  bool f = false;
  bool g = true;

  return 0;
}
```

Componentes – If/Else

```
if (boolean) {

   // código

} else if (boolean) {

   // código

} else {

   // código

}
```

Componentes – If/Else

Código 4: 03_if_else.cpp

```cpp
#include <iostream>

int main() {
  // Comparaciones
  int C = 5;
  if (C < 5) {
    std::cout << "C es menor que 5" << std::endl;
  } else if (C > 5) {
    std::cout << "C es mayor que 5" << std::endl;
  } else // C == 5
  {
    std::cout << "C es 5" << std::endl;
  }
  return 0;
}
```

Componentes – While/For

```
while (condition) {

   // código

}

for (init; condition; increment){

   // código

}
```

Componentes – While/For

Código 5: 04_while_for.cpp

```cpp
#include <iostream>

int main() {
  // While
  int i = 0;
  while (i < 5) {
    std::cout << "i = " << i << std::endl;
    i++;
  }

  // For
  for (int j = 0; j < 5; j++) {
    std::cout << "j = " << j << std::endl;
  }

  return 0;
}
```

Componentes – Funciones

```
return_type function_name(parameters ...) {

   // código

}

int main(){

   return 0;

}
```

Componentes – Funciones

Código 6: 05_functions.cpp

```cpp
#include <iostream>

// Declara una función (el encabezado), esto podría hacerse en un .h
double square(double num);

double square(double num) {
  return num * num;
}

int main() {
  double i = 4;
  double ii;

  ii = square(i);
  std::cout << "Valor de i = " << i << std::endl;
  std::cout << "i^2 = i^i = " << ii << std::endl;

  return 0;
}
```

Biblioteca std

- Biblioteca que contiene todas las herramientas básicas para programar:
  - std::cout/std::cin — Entrada/Salida estándar
  - std::string — Manejo de strings
  - std::vector — Vector de elementos
  - std::list — Listas de elementos
  - std::queue — Queue
  - std::set — Set ordenado
- Muchos más, ver https://cplusplus.com y https://cppreference.com

Biblioteca std – Strings

Código 7: 06_strings.cpp

```cpp
#include <iostream>
#include <string>

int main() {
  std::string str1 = "Hello";
  std::string str2, str3;
  int len;

  std::cout << "Write a word: ";
  std::cin >> str2;

  // Copia str1 en str3
  str3 = str1;
  std::cout << "str3 : " << str3 << std::
    ↪ endl;

  // Concatena str1 y str2
  str3 = str1 + str2;
  std::cout << "str1 + str2 : " << str3
    ↪ << std::endl;

  // Largo total de str3 después de
    ↪ concatenar
  len = str3.size();
  std::cout << "str3.size() :  " << len <<
    ↪ std::endl;

  return 0;
}
```

Puntero – Referencia

- `int a;`
  - Variable simple
- `int *b;`
  - Variable que puede guardar una dirección de memoria
- `int &c = a;`
  - Alias de una variable. No puede ser `NULL`

Punteros

Código 8: 07_pointer.cpp

```cpp
#include <iostream>

int main() {
  // Variables
  int a = 5;
  int b = 10;

  // Punteros
  int *aPtr;
  aPtr = &a; // Obtiene dirección de memoria

  std::cout << "a = " << a << std::endl;
  std::cout << "Dirección de a = " << &a << std::endl;
  std::cout << "aPtr = " << aPtr << std::endl;
  std::cout << "*aPtr = " << *aPtr << std::endl;

  return 0;
}
```

Referencias

Código 9: 08_reference.cpp

```cpp
#include <iostream>

int main() {
  // Variables
  int a = 5;

  // Referencias
  int &aRef = a;
  std::cout << "a = " << a << std::endl;
  std::cout << "aRef = " << aRef << std
    ↪ ::endl;

  // Modifica variable
  a = 7;
  std::cout << "a = " << a << std::endl;
  std::cout << "aRef = " << aRef << std
    ↪ ::endl;

  // Modifica referencia
  aRef = 8;
  std::cout << "a = " << a << std::endl;
  std::cout << "aRef = " << aRef << std
    ↪ ::endl;

  return 0;
}
```

Clases

- Similar a Java
- Separación público/privado por bloques
- Uso de templates
- Se puede definir la declaración e implementación en archivos distintos:
  - `archivo.h` Declaración
  - `archivo.cpp` Implementación

Clases

Código 10: 09_complex_v1.cpp

```cpp
#include <iostream>

class Complex {
  public:
  Complex(float real, float imag)
  : m_realPart(real),
  m_imaginaryPart(imag) {}

  float getImaginaryPart() const {
    return m_imaginaryPart;
  }

  float getRealPart() const {
    return m_realPart;
  }

  private:
  float m_realPart;
  float m_imaginaryPart;
};

int main() {
  auto *c1 = new Complex(3, 5); //
    ↪ Complex *c1 = new Complex
    ↪ (3, 5);
  Complex c2(2, 7);
  std::cout << "c1 = (" << c1->
    ↪ getRealPart() << ", " << c1->
    ↪ getImaginaryPart() << ")" <<
    ↪ std::endl;
  std::cout << "c2 = (" << c2.
    ↪ getRealPart() << ", " << c2.
    ↪ getImaginaryPart() << ")" <<
    ↪ std::endl;

  return 0;
}
```

Clases – El mismo ejemplo, pero en header e implementación

Código 11: 10_complex_v1_sep.h

```
1  class Complex {
2    public:
3    Complex(float real, float imag);
4
5    float getImaginaryPart() const;
6
7    float getRealPart() const;
8
9    private:
10   float m_realPart;
11   float m_imaginaryPart;
12 };
```

Código 12: 10_complex_v1_sep.cpp

```
1  #include "10_complex_v1_sep.h"
2
3  Complex::Complex(float real, float imag
       ↪ )
4  : m_realPart(real),
5  m_imaginaryPart(imag) {}
6
7  float Complex::getImaginaryPart()
       ↪ const {
8    return m_imaginaryPart;
9  }
10
11 float Complex::getRealPart() const {
12   return this->m_realPart;
13 }
```

Creando el objeto y llamándolo desde otro archivo

Código 13: 10_main.cpp

```cpp
#include <iostream>
#include "10_complex_v1_sep.h"

int main() {
  auto *c1 = new Complex(3, 5);
  Complex c2(2, 7);
  std::cout << "c1 = (" << c1->getRealPart() << ", " << c1->getImaginaryPart() << ")" <<
    ↪ std::endl;
  std::cout << "c2 = (" << c2.getRealPart() << ", " << c2.getImaginaryPart() << ")" <<
    ↪ std::endl;
  return 0;
}
```

Clases – Constructor / Destructor

- Constructor inicializa los atributos del objeto
- Destructor realiza tareas de limpieza al momento de eliminar un objeto
- Ambos existen por default
  - SIEMPRE REDEFINIR EL DESTRUCTOR!

Clases – Definiendo constructor y destructor

Código 14: 11_complex_v2.h

```
1  class Complex {
2    public:
3    explicit Complex(float real);
4
5    Complex(float real, float imag);
6
7    Complex();
8
9    ~Complex();
10
11   float getImaginaryPart() const;
12
13   float getRealPart() const;
14
15   private:
16   float m_realPart;
17   float m_imaginaryPart;
18 };
```

Clases – Definiendo constructor y destructor

Código 15: 11_complex_v2.cpp

```cpp
#include <iostream>
#include "11_complex_v2.h"

Complex::Complex(float real, float imag
    ↪ )
: m_realPart(real),
m_imaginaryPart(imag) {}

Complex::Complex(float real)
: m_realPart(real),
m_imaginaryPart(0) {}

Complex::Complex()
: m_realPart(0),
m_imaginaryPart(0) {}

Complex::~Complex() {
  std::cout << "Destructor llamado" <<
    ↪ std::endl;
}

float Complex::getImaginaryPart()
    ↪ const {
  return m_imaginaryPart;
}

float Complex::getRealPart() const {
  return this->m_realPart;
}
```

Clases – Definiendo constructor y destructor

Código 16: 11_main.cpp

```cpp
#include <iostream>
#include "11_complex_v2.h"

int main() {
  auto *c1 = new Complex(3, 5);
  Complex c2(2);
  Complex c3;
  std::cout << "c1 = (" << c1->
    ↪ getRealPart() << ", " << c1->
    ↪ getImaginaryPart() << ")" <<
    ↪ std::endl;
  std::cout << "c2 = (" << c2.
    ↪ getRealPart() << ", " << c2.
    ↪ getImaginaryPart() << ")" <<
    ↪ std::endl;
  std::cout << "c3 = (" << c3.
    ↪ getRealPart() << ", " << c3.
    ↪ getImaginaryPart() << ")" <<
    ↪ std::endl;

   std::cout << "----------------" << std::
     ↪ endl;
  std::cout << "Destrucción manual de
     ↪ c1" << std::endl;
  delete c1;
  std::cout << "c1 se destruyó" << std::
     ↪ endl;
  std::cout << "----------------" << std::
     ↪ endl;

  std::cout << std::endl;

  std::cout << "Final de main():" << std
     ↪ ::endl;
  std::cout << "----------------" << std::
     ↪ endl;

  return 0;
}
```

Clases – Overloading

- Permite usar operadores nativos con funcionalidad personalizada
- Simplifica lectura y escritura de código, Ej:

```
numC = numA.multliplyBy(numB);
numC = numA * numB;
```

Clases – Overloading

Código 17: 12_complex_v3.h

```
1  #include <iostream>
2
3  class Complex {
4    friend std::ostream &operator<<(std::ostream &out, const Complex &complex);
5
6    public:
7    Complex(float real, float imag);
8
9    Complex operator+(Complex &complex) const;
10
11   float getImaginaryPart() const;
12
13   float getRealPart() const;
14
15   private:
16   float m_realPart;
17   float m_imaginaryPart;
18 };
```

Clases – Overloading

Código 18: 12_complex_v3.cpp

```cpp
#include <iostream>
#include "12_complex_v3.h"

Complex::Complex(float real, float imag
    ↪ )
: m_realPart(real),
m_imaginaryPart(imag) {}

std::ostream &operator<<(std::ostream
    ↪ &out, const Complex &
    ↪ complex) {
  out << "(" << complex.getRealPart()
    ↪ << ", " << complex.
    ↪ getImaginaryPart() << ")";
  return out;
}

Complex Complex::operator+(Complex
    ↪  &complex) const {
  return {this->m_realPart + complex.
    ↪ m_realPart,
    this->m_imaginaryPart + complex.
    ↪ m_imaginaryPart};
}

float Complex::getImaginaryPart()
    ↪ const {
  return m_imaginaryPart;
}

float Complex::getRealPart() const {
  return this->m_realPart;
}
```

Clases – Overloading

Código 19: 12_main.cpp

```cpp
#include <iostream>
#include "12_complex_v3.h"

int main() {
   Complex c1(3, 5);
   Complex c2(2, 8);
   Complex c3(0, 0);

   std::cout << "c1 = " << c1 << std::endl;
   std::cout << "c2 = " << c2 << std::endl;
   std::cout << "c3 = " << c3 << std::endl;
   c3 = c1 + c2;

   std::cout << "Ahora c3 = c1 + c2 = " << c3 << std::endl;
   return 0;
}
```

Clases – Templates

- Distintas primitivas usadas por la misma estructura
- Se pasa como argumento el tipo (`Class<T>`)
- Evita tener que redefinir los mismos métodos para cada tipo
- Se definen en los archivos .h

Clases – Templates

Código 20: 13_stack.h

```
#include <vector>

template<class T>
class Stack
{
  public:
  Stack();
  T pop();
  void push(T &elem);
  T top() const;
  int size() const;

  private:
  std::vector<T> m_stack;
};

template<class T>
Stack<T>::Stack(){}

template<class T>
T Stack<T>::pop()
{
  T elem = top();
  m_stack.pop_back();
  return elem;
}

template<class T>
T Stack<T>::top() const
{
  return m_stack.back();
}

template<class T>
T Stack<T>::top() const
{
  return m_stack.back();
}

template<class T>
void Stack<T>::push(T &
    ↪ elem)
{
  m_stack.push_back(
    ↪ elem);
}

template<class T>
int Stack<T>::size() const
{
  return m_stack.size();
}
```

Clases – Templates

Código 21: 13_main.cpp

```cpp
#include <iostream>
#include "13_stack.h"
#include "12_complex_v3.h"

int main() {
  Stack<Complex> stack;
  Complex c1(1.1, 2.2);

  std::cout << "Inicio, tamaño del stack = " << stack.size() << std::endl;
  stack.push(c1);
  std::cout << "Progreso, tamaño del stack = " << stack.size() << std::endl;
  Complex c2 = stack.pop();

  std::cout << "c1 = " << c1 << std::endl;
  std::cout << "c2 = " << c2 << std::endl;

  std::cout << "Final, tamaño del stack = " << stack.size() << std::endl;
  return 0;
}
```

Contenidos

Tests

- El uso de tests es vital en C++
- Permite sanar errores asociados a memoria, punteros, etc.
- Pasos:

  1. Iniciar variables
  2. Probar funcionalidad
  3. Comprobar resultados correctos
  4. Liberar recursos

Tests

Código 22: 14_tests.cpp

```cpp
#include <cassert>
#include "12_complex_v3.h"

int main() {
  // Inicializa
  Complex s(1, 1);
  Complex d(0, 1);

  // Experimento
  s = s + d;

  // Assert
  assert(s.getRealPart() == 1 and s.getImaginaryPart() == 2);
}
```

Tests

- Recomendación: Testear sus códigos y tests con valgrind, herramienta de *Dynamic analysis*
- Valgrind es soportado nativamente en Linux, en Windows se puede usar mediante WSL (Windows Subsystem for Linux)
- Herramienta soportada nativamente por CLion

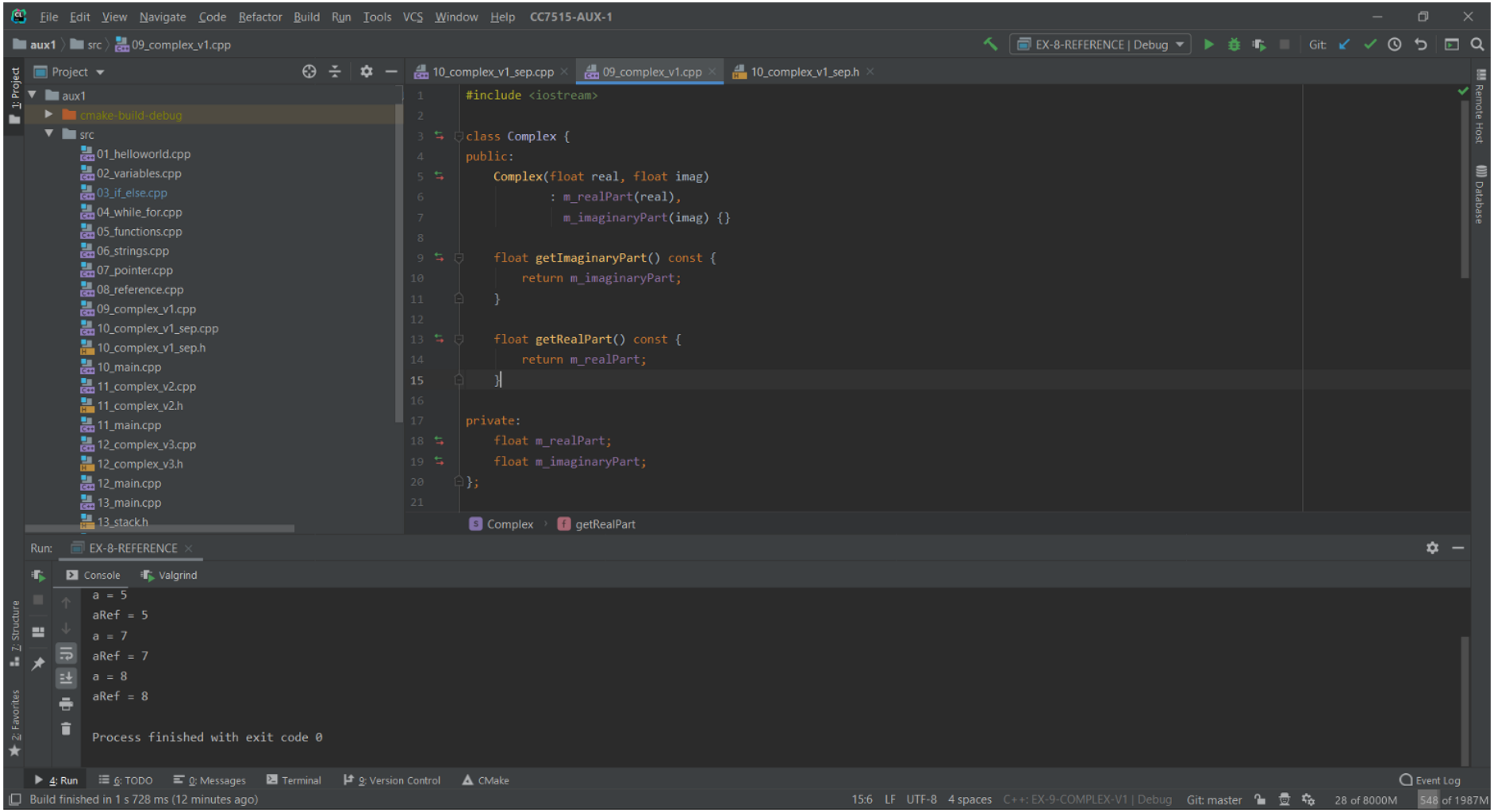
Tests – Buenas prácticas

- Crear tests para todas las funcionalidades
- Siempre pensar qué vas a hacer antes de programar
- Después de programar, preguntarte si estás satisfecho con lo programado
- Usar debuggers en vez de print debugging

CLion

- https://www.jetbrains.com/clion/
- Permite tener múltiples compiladores
- Herramienta de *debugging*
- Soporte con *valgrind* y otras herramientas
- Incorpora uno de los mejores *Intelli-Sense*
- Fácil de usar
- Permite construir el *make* de manera muy fácil

CLion


File Edit View Navigate Code Refactor Build Run Tools VCS Window Help CC7515-AUX-1
aux1 src 09_complex_v1.cpp
EX-8-REFERENCE | Debug
Git:
Project
aux1
cmake-build-debug
src
01_helloworld.cpp
02_variables.cpp
03_if_else.cpp
04_while_for.cpp
05_functions.cpp
06_strings.cpp
07_pointer.cpp
08_reference.cpp
09_complex_v1.cpp
10_complex_v1_sep.cpp
10_complex_v1_sep.h
10_main.cpp
11_complex_v2.cpp
11_complex_v2.h
11_main.cpp
12_complex_v3.cpp
12_complex_v3.h
12_main.cpp
13_main.cpp
13_stack.h
10_complex_v1_sep.cpp
09_complex_v1.cpp
10_complex_v1_sep.h
#include <iostream>
class Complex {
public:
Complex(float real, float imag)
: m_realPart(real),
m_imaginaryPart(imag) {}
float getImaginaryPart() const {
return m_imaginaryPart;
}
float getRealPart() const {
return m_realPart;
}
private:
float m_realPart;
float m_imaginaryPart;
};
Complex getRealPart
Run: EX-8-REFERENCE
Console Valgrind
a = 5
aRef = 5
a = 7
aRef = 7
a = 8
aRef = 8
Process finished with exit code 0
4: Run 6: TODO 0: Messages Terminal 9: Version Control CMake
Event Log
Build finished in 1 s 728 ms (12 minutes ago)
15:6 LF UTF-8 4 spaces C++: EX-9-COMPLEX-V1 | Debug Git: master
28 of 8000M 548 of 1987M

Compilación

- ¿Cómo se ejecutan los códigos?, ¿Cómo se compila?
- Para compilar crear un **makefile**

Código 23: Makefile

```
all: helloworld variables if_else while_for functions strings pointer reference complexv1
     ↪ complexv1sep complexv2 complexv3 stack tests

helloworld:
g++ 01_helloworld.cpp -o 01_helloworld

variables:
g++ 02_variables.cpp -o 02_variables

if_else:
g++ 03_if_else.cpp -o 03_if_else

while_for:
g++ 04_while_for.cpp -o 04_while_for
```

Cmake

Código 24: Makefile con CMakelists

```
# Propiedades del make
cmake_minimum_required(VERSION 3.10)
project(CC7515-AUX-1)
set(CMAKE_CXX_STANDARD 11)

# Crea un set, contiene varios archivos que permiten compilar al principal
set(AUX1_EX10
src/10_complex_v1_sep.cpp)

set(AUX1_STACK
src/13_stack.h
src/12_complex_v3.cpp)

# Define ejecutables
add_executable(EX-1-HELLO-WORLD src/01_helloworld.cpp)
add_executable(EX-2-VARIABLES src/02_variables.cpp)
add_executable(EX-3-IF-ELSE src/03_if_else.cpp)
add_executable(EX-4-WHILE-FOR src/04_while_for.cpp)
...
```

Bibliografía

- http://www.cplusplus.com/doc/
- https://stackoverflow.com
- Deitel&Deitel, How to Program C++, 8th Edition

Gracias por su atención