Auxiliar N°1
C++

Auxiliar: Pablo Pizarro R. @ppizarror
Estudiante MSc. Ingeniería Estructural
Universidad de Chile

Introducción

- Introducción a C++
- Principales componentes del lenguaje
  - Variables
  - Condicionales
  - Ciclos
  - Funciones
  - Bibliotecas
  - Clases
- Testing

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

- Este lenguaje de programación lo utilizaremos para:
  - Tarea 1
  - Tarea 3, uso de CUDA/OPENCL
  - Sus proyectos semestrales
- Proyecto semestral (ejemplo)
  - “Implementación plataforma análisis estructural mediante elementos finitos de membrana”
  - Proyecto en C++/CUDA, uso cálculo paralelo en GPU
  - https://github.com/ppizarror/FNELEM-GPU

Introducción - Sobre C++

- Creado en 1979, por Bjarne S.
- Multiparadigma
- Tipado fuerte
- Eficiente

Introducción – El clásico HELLO-WORLD

01_helloworld.cpp

```
#include <iostream>

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

Importación de librerías

Función principal, retorna 0 (sin errores) o cualquier cosa (error)

Pasamos “Hello World!” al I/O de std::cout de la salida estándar

Si no añadimos esto el compilador lo hará solo

Variables

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

02_variables.cpp

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

    int d;
    d = 6;

    // Carácteres
    char e;
    e = 'A';

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

    return 0;
}
```

If/Else

```
if (boolean) {
        // código
} else if (boolean) {
        // código
} else {
        // código
}
```

If/Else

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;
}
```

While / For

```
while (condition) {
        // código
}

for (init; condition; increment){
        // código
}
```

While / For

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;
}
```

Funciones

```
return_type function_name(parameters …) {
        // código
}

int main(){
        return 0;
}
```

Funciones

05_functions.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 Listas de elementos

Strings

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;
}
```

Nombre vs Puntero vs 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

07_pointer.cpp

```cpp
#include <iostream>

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

// Pointers
int *aPtr;
aPtr = &a; // Get memory address

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

08_reference.cpp

```cpp
#include <iostream>

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

// References
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

```
#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;
};
```

09_complex_v1.cpp

```
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;
}
```

Parte pública

Parte privada

Métodos de la clase

Clases – El mismo ejemplo, pero separado

10_complex_v1_sep.h

```
class Complex {
public:
    Complex(float real, float imag);

    float getImaginaryPart() const;

    float getRealPart() const;

private:
    float m_realPart;
    float m_imaginaryPart;
};
```

10_complex_v1_sep.cpp

```
#include "10_complex_v1_sep.h"

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

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

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

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 – Constructor / Destructor

11_complex_v2_sep.h

```
class Complex {
public:
    explicit Complex(float real);

    Complex(float real, float imag);

    Complex();

    ~Complex();

    float getImaginaryPart() const;

    float getRealPart() const;

private:
    float m_realPart;
    float m_imaginaryPart;
};
```

11_complex_v2_sep.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 called."
<< std::endl;
}
```

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

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

Destructor

Clases – Constructor / Destructor

11_main.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

12_complex_v3.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

12_complex_v3.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

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

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>
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

13_stack.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 << "On start, size of stack =
" << stack.size() << std::endl;

    stack.push(c1);

    std::cout << "On progress, size of
stack = " << stack.size() << std::endl;

    Complex c2 = stack.pop();

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

    std::cout << "On end, size of stack = "
<< stack.size() << std::endl;
    return 0;
}
```

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

14_tests.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

```
115     assert(m1->get(0, 0) == 4);
116     assert(m1->get(1, 1) == 6);
117
118     // Destroy
119     delete m1;
120     delete m2;
121 }
122
123 void __test_fematrix_substract() {
124     test_print_title("FEMATRIX", "test_fematrix_substract");
125     FEMatrix *m1 = new FEMatrix(3, 3);
126     m1->fill_ones();
127     FEMatrix *m2 = new FEMatrix(3, 3);
128     m2->fill_ones();
129     m2->set(0, 0, 3);
130     m2->set(1, 1, 5);
131     FEMatrix *m3 = *m2 - *m1;
132     FEMatrix *m4 = -*m3;
133     assert(m3->get(0, 0) == 2);
134     assert(m4->get(0, 0) == -2);
135     delete m1;
136     delete m2;
137     delete m3;
138     delete m4;
139 }
140
141 void __test_fematrix_transpose() {
142     test_print_title("FEMATRIX", "test_fematrix_transpose");
143     FEMatrix *m1 = new FEMatrix(2, 3);
144     m1->set(0, 0, 1);
145     m1->set(0, 1, 1);
146     m1->set(0, 2, 5);
147     m1->set(1, 0, 9);
148     m1->set(1, 1, 10);
149     m1->set(1, 1, -1);
150     m1->disp();
```

Un test por cada característica de su código

Realizar asserts

Recordar ir borrando la memoria

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.

Buenas prácticas

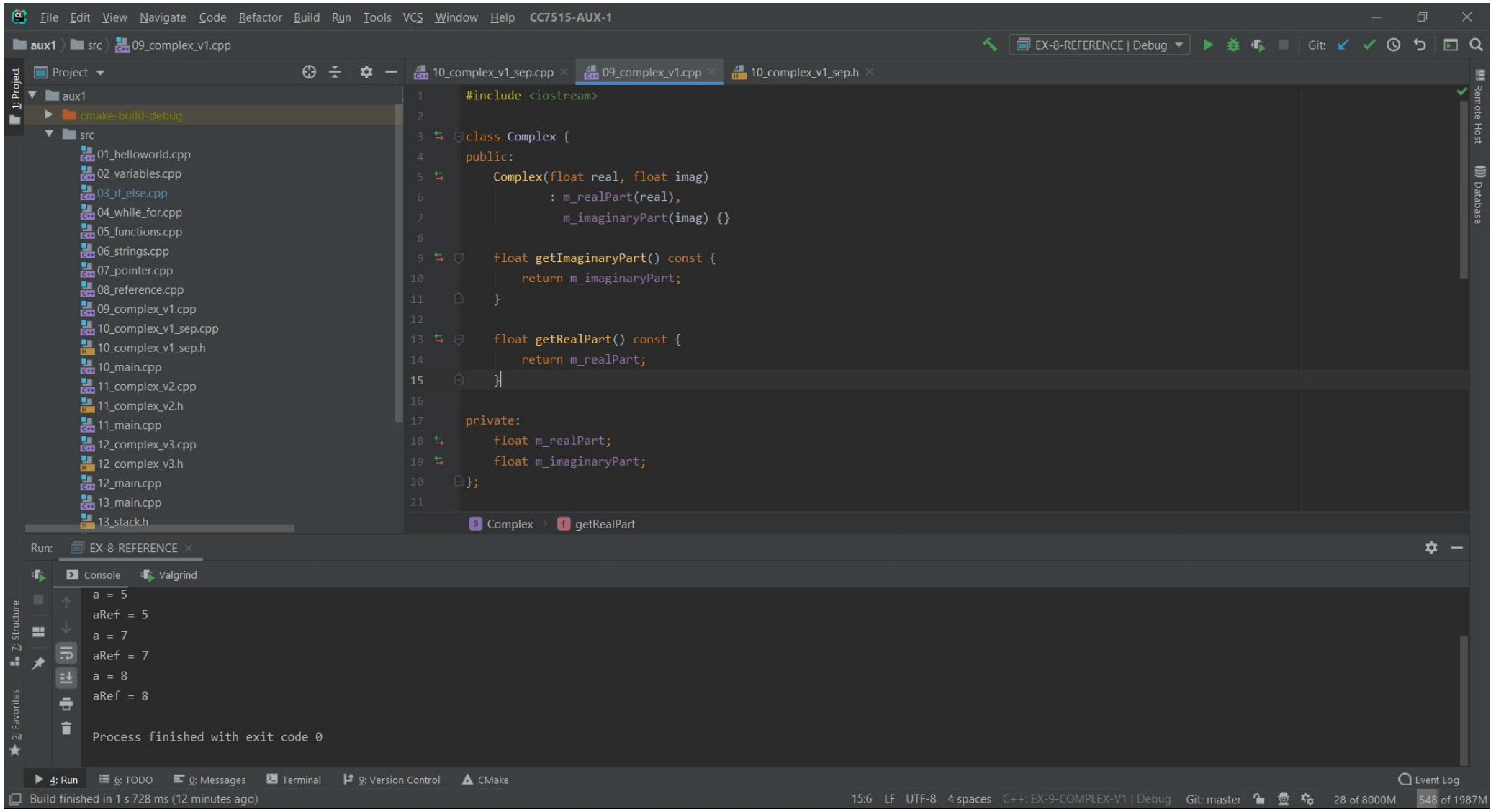
- Crear tests para todas las funcionalidades.
- Siempre pensar qué vas a hacer antes de programar.
- Después de programar, preguntarte si estás satisfecho con lo programado.
- Usar *debuggers* en vez de *print debugging*.

Recomendación IDE

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

Recomendación IDE - CLion



Finales

¿Cómo se ejecutan los códigos?, ¿Cómo se compila?

- Para compilar crear un makefile

```
all: helloworld variables if_else while_for functions strings ...

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

functions:
g++ 05_functions.cpp -o 05_functions

strings:
g++ 06_strings.cpp -o 06_strings
```

Finales

¿Cómo se ejecutan los códigos?, ¿Cómo se compila?

- Para compilar crear un makefile, o bien usar un Cmakelists.txt

```
# 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)
add_executable(EX-5-FUNCTIONS src/05_functions.cpp)
add_executable(EX-10-COMPLEX-V1-SEP src/10_main.cpp ${AUX1_EX10})
```

Flags al compilador, permite definir versión mínima

Nombre del proyecto

Define un set u conjunto de archivos de un "ejecutable"

Ejecutable, requiere un nombre, un archivo principal (el que tiene el main) y el set de archivos que requiere el ejecutable

Bibliografía

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