Auxiliar 1

Introducción a objetos en C++

Vicente González y Diego García

CC7515-1 — Computación en GPU

Introducción a C++

El lenguaje

- Desarrollado por Bjarne Stroustrup en 1983 como una extensión del lenguaje C.
- Lenguaje de bajo nivel con control de memoria necesarias para aprovechar la arquitectura de la GPU.
- Se recomienda usar el estándar de C++ moderno (C++ 11 en adelante).

Hello world

```cpp
// file: main.cpp
#include <iostream>

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

Para compilar:

```
$ g++ main.cpp -o main
$ ./main
```

Actividad

Hello CMake

- Crear un pequeño proyecto
- Ejecutables y librerias
- Aprender lo básico de clases en C++
- Uso de CMake y Google Test
- Tips del auxiliar para trabajar más cómodamente :D

Proyecto

Vamos a crear un proyecto "típico" de C++, con las siguientes carpetas:

- `src`: Donde van todos los archivos .cpp y .h para compilar el proyecto.
- `include`: Reservado para headers "públicos" cuando se hacen librerías o usan.
- `extern`: Dependencias del proyecto.
- `test`: Donde dejar los tests del proyecto.
- `build`: Donde van a ir nuestros ejecutables y cositas de CMake.

Clone la repo https://github.com/Seivier/hello-cmake.git

Compilación

CMake o Makefile?

- CMake
  - Estándar moderno de C++
  - Complejo y estructurado
  - Multiplataforma
  - Relativamente intuitivo
  - Basado en macros y funciones
- Makefile
  - Pensado como una *build tool* de Linux
  - Sencillo y directo
  - Basado en reglas
  - Ideal para automatizar comandos

CMake

Algunas funciones utiles:

- `project(NAME)`: Inicia un proyecto con un nombre `NAME`
- `add_executable(NAME FILES)`: Añade un ejecutable con nombre `NAME` y archivos `FILES`
- `add_library(NAME TYPE FILES)`: Añade una librería con nombre `NAME`, tipo `TYPE` y archivos `FILES`
- `target_link_libraries(NAME LIBRARIES)`: Añade las librerías `LIBRARIES` al ejecutable (o libreria) `NAME`
- `add_subdirectory(PATH)`: Añade un subdirectorio al proyecto

Más en https://cmake.org/cmake/help/latest/guide/tutorial/index.html

```
$ cmake -S . -B build
# S es para el codigo fuente
# B es donde dejar los archivos generados

$ cmake --build build
# build construye en ejectable o libreria

$ ./build/src/main
# con esto se ejecuta la aplicación
```

P1 — Hello world

Cree su primer proyecto con CMake, para ello cree la carpeta `src`, en esta carpeta es donde debe ir el código de su proyecto.

Dentro de `src` cree el archivo `main.cpp` donde coloque su función `main` y haga un script que imprima lo siguiente:

```
Hello CC7515!
```

Por ultimo añada el archivo `CMakeLists.txt` donde inicie su proyecto y añada el ejecutable.

El ejecutable deberia estar en `build/src/`.

Clases en C++

Introducción a Clase

- En C++, un objeto es una instancia de una clase.
- Una clase es una estructura de datos que define un conjunto de atributos y métodos que operan sobre esos atributos.
- Los objetos se pueden crear dinámicamente en tiempo de ejecución usando punteros y el operador `new`.
- Los objetos se eliminan usando el operador `delete` para liberar la memoria asignada a ellos.
- Se pueden usar tanto `class` como `struct`.

Clases abstractas

Declaración

```
struct Pokemon {
  virtual void speak() const = 0;
  virtual const std::string& name() const = 0;
  virtual int hp() const = 0;
  virtual int maxHp() const = 0;
  virtual const Type& type() const = 0;
};
```

Una *interface* en C++ !!!

Declaración

```
class APokemon {
public:
  // ...
  int hp() const override;
private:
  // ...
  int _hp; // o mHp;
};
```

Una clase *abstracta* !!

Usualmente la declaración va en un .h y la implementación en un .cpp

Constructores

Tipos de constructores y destructores


```
// Constructor por defecto
Rectangulo(): ancho(0), alto(0) {}

// Constructor con parámetros
Rectangulo(double ancho, double alto) {
    this->ancho = ancho;
    this->alto = alto; }

// Constructor copia
Rectangulo(const Rectangulo& r) {
    this->ancho = r.ancho;
    this->alto = r.alto; }

// Destructor
~Rectangulo() {
    std::cout << "Se ha destruido un rectangulo." << std::endl;
}
```

Tipos de constructores y destructores

IMPORTANTE!

Usted no puede usar una clase abstracta como miembro de la clase, un lugar de eso deber usar un puntero

Biblioteca estándar

La STL

Conjunto de funciones, objetos y clases que proporcionan una amplia variedad de características y funcionalidades para el lenguaje C++. Por ejemplo:

- Entrada/salida: operaciones de entrada y salida, como leer o escribir en archivos o en la consola.
- Contenedores: estructuras de datos para almacenar y manipular colecciones de objetos, como vectores, listas, mapas, etc.
- Algoritmos: funciones para realizar operaciones comunes en contenedores, como ordenar, buscar, mezclar, etc.
- Tipos de datos: tipos de datos comunes, como cadenas de caracteres, booleanos, números, etc.
- Funciones matemáticas: funciones matemáticas comunes, como seno, coseno, exponencial, etc.

La Biblioteca estándar de C++ está disponible en cualquier compilador que cumpla con el estándar de C++. Para usarla, se debe incluir el archivo de cabecera correspondiente.

La STL

- `#import <iostream>`: para entrada/salida de consola
- `#import <vector>`: para el uso de vectores dinámicos
- `#import <string>`: para el uso de cadenas de texto
- `#import <algorithm>`: para el uso de algoritmos de ordenación, búsqueda, etc.
- `#import <unordered_map>` y `#import <map>`: para el uso de mapas y diccionarios
- `#import <set>`: para el uso de conjuntos
- `#import <cmath>`: para el uso de funciones matemáticas como `sqrt()`, `cos()`, `sin()`, etc.
- `#import <chrono>`: para el uso de medidas de tiempo
- `#import <memory>`: para el uso de puntero inteligentes
- etc

Más en https://en.cppreference.com/w/cpp/standard_library

Memory

- Existen 3 tipos de punteros “inteligentes” en la STL
- Funcionan como puntero normales desde afuera
- El uso correcto de estos garantiza el despreocuparse por el manejo de memoria
- `std::shared_ptr<T>`: Permite crear un puntero compartido, va contando las referencias y autodestruye cuando el contador llega a 0
- `std::unique_ptr<T>`: Permite crear un puntero único, el cual no permite compartir la información que contiene con otros punteros
- `std::weak_ptr<T>`: Permite crear un puntero débil, se comporta de manera similar a un puntero clásico y sirve para referencia cíclicas

Memory

```
class C {
public:
    int x, y, z;
    C(int x, int y, int z);

}

int main() {
    std::unique_ptr<C> p;
    // make_* llama al constructor
    p = std::make_unique<C>(10, 20, 30);
    std::cout << p->x << std::endl; // "10"
}
```

String

- Permite manejar strings de manera cómoda
- Funciona como cualquier string de otro lenguaje tipado

```
struct Pokemon {
    // ...
    virtual const std::string& name() const = 0;
}
```

Vector

- Los vectores de la STL son arreglos típicos de otros lenguajes
- Permiten manipular arreglos de tamaño dínamico alojados en el heap
- Usando esto y los punteros inteligentes puede evitar tratar con `new` y `delete` casi en su totalidad

Overloading

Overloading de operadores

El **overloading de operadores** es una funcionalidad en C++ que permite a los desarrolladores definir una función que se comporta como un operador. Por ejemplo, para sumar dos números complejos, podemos sobrecargar el operador + de la siguiente manera:

```
class Complejo {
public:
    Complejo operator+(const Complejo& c) const {
        return Complejo(real + c.real, imag + c.imag);
    }
private:
    double real;
    double imag;
};
```

Overloading de operadores

En este caso, estamos sobrecargando el operador + para que sume dos números complejos. El operador se define dentro de la clase `Complejo` y se utiliza el operador + para definir la función.

P2 — Añadirle clase

Hagamos algo más complejo, cree una clase `Ball` que represente una bola. Una bola tiene:

- Masa
- Posición
- Radio

Para esto añado los archivos `ball.cpp` dentro de `src` y `ball.h` dentro de `include`.

P2 — Añadirle clase

Adicionalmente, la bola debe cumplir con los siguientes requerimientos:

- A una bola se le puede aplicar una fuerza
- Una bola se debe mover en base a las fuerzas que existen sobre ella

Antes de implementar estas funcionalidades usted considera que es necesario contar con la ayuda de una librería matemática. Su auxiliar favorito le recomienda usar GLM.

P2 — Añadirle clase

Para añadir esta libreria se le recomienda crear la carpeta `extern`, en esta carpeta es donde van todas las librerías extenras que necesita su proyecto, use los súbmodulos de git para añadir la dependencia:

```
cd extern
git submodule add <link>
```

También es recomendado añadir un archivo `CMakeLists.txt` tanto dentro de `src` como de `extern`, para segmentar las funcionalidades, usted puede cargar estos archivos usando `add_subdirectory` en CMake.

Otra función que se les será útil es `target_include_directories`.

Tests

Testing con Asserts en C++

En C++, se pueden hacer pruebas unitarias utilizando asserts.

- Un assert es una macro que verifica una expresión y termina el programa si la expresión es falsa.
- Los asserts son útiles para detectar errores lógicos en el programa, como valores inválidos de parámetros o errores de cálculo.

Testing con Asserts en C++

Para utilizar los asserts, se debe incluir la biblioteca `cassert` y luego utilizar la macro `assert` con la expresión a evaluar:

```
#include <cassert>

int dividir(int a, int b) {
    assert(b != 0);
    return a / b;
}
```

Si la expresión `b != 0` es falsa, el programa terminará en ese punto y mostrará un mensaje de error.

Google Test

Google Test es un framework de testing más sofisticado

- Diversos tipos de asserts
- Permite el uso de clases para crear suites de tests
- Similar a `jUnit` de Java o `mUnit` de Scala
- Su uso no es obligatorio pero muy recomendado

Google test

Para usarlo junto a CMake las siguientes funciones son necesarias:

- `enable_testing()`: Habilita el testing en el proyecto
- `include(GoogleTest)`: Incluye las funciones de Google Test
- `gtest_discover_tests(NAME)`: Añade los tests dentro del ejecutable `NAME`

Más detalles en: https://google.github.io/googletest/quickstart-cmake.html

O puede ver el siguiente video: https://youtu.be/pxJoVRfpRPE?si=hqbv1whpbNtKcrzl

Google test

Luego para compilar y correr los tests:

```
$ cmake -S . -B build
$ cmake --build build
$ cd build && ctest
```

P3 — Testing

Muy bien, ahora para corroborar que todo funcione correctamente, haga tests para comprobar dichas funcionalidades.

Cree la carpeta `test` esta carpeta es similar a la de `src` solo que contiene el código que es exclusivo al testing.

En el curso usaremos la libreria de testing de Google: [googletest] (https://github.com/google/googletest).

Añada esta dependencia **dentro** de su carpeta `test`.

Esqueleto

```
cmake_minimum_required(VERSION 3.20)

project(hello-cmake)

add_subdirectory(extern)
add_subdirectory(src)
if (BUILD_TESTING)
  add_subdirectory(test)
endif()
```

Esqueleto

En `src`:

```
add_library(ball STATIC ball.cpp)
target_include_directories(ball PUBLIC ${PROJECT_SOURCE_DIR}/
include)

add_executable(hello-cmake main.cpp)
target_link_libraries(hello-cmake PUBLIC ball)
```

En `extern`:

```
add_subdirectory(glm)
```

Esqueleto

En `test`:

```
add_subdirectory(extern)
add_executable(tests test.cpp)
target_link_libraries(tests PRIVATE GTest::gtest_main ball)

include(GoogleTest)
gtest_discover_tests(tests)
```

Puede ver en detalle el proyecto en la rama `reference`

Bibliografía

Bibliografía

Stroustrup, B. (2018). A Tour of C++, Second Edition.