Programación II

Tema 1

jtorresmat@nebrija.es

Índice

Introducción (¿qué conocimientos se os presupone?)

- `Programación I (`https://github.com/Nebrija-Programacion/Programacion-I`):`

  Variables (tipos de datos simples y estructuras)

  Estructura básica de un programa en C++:

  - Bibliotecas
  - Espacio de nombres (ámbitos)
  - Comentarios
  - Operadores
  - Entrada y salida de datos por consola

Introducción

Control de flujo:

- if, if-else, if-else if-else
- switch-case
- while, do-while
- for

Tipos de datos contenedor:

- std::string
- std::array
- std::vector
- std::set

Introducción

Funciones:

- Valor de retorno
- Sin parámetros
- Con parámetros:
  - Paso por copia
  - Paso por referencia (constante y no constante)

Otros tipos de datos:

- Funciones Lambda
- ~~Punteros:~~
  - ~~Clásicos~~

Introducción

Características de C++:

- Hereda sintaxis de C
- Orientado a objetos
  - Abstracción / Encapsulado / Herencia / Polimorfismo
- Permite sobrecarga de operadores
- Permite funciones Lambda
- Fuertemente tipado
- Lenguaje programación compilado (JavaScript y python son interpretados)
- Es muy eficiente (se compila para hardware específico -> Siendo de alto nivel está cerca de lenguajes de bajo nivel)
- Constantes actualizaciones (ISO14882:2020 y la 2023 está en camino…)
- Contra: ***¡Tocaba y toca seguir estudiando…! (ya os lo advertí…)***

Índice

Punteros clásicos

- Una variable es una etiqueta que asocia un valor a un nombre **tipo** *etiqueta;*
- Un puntero es un tipo de datos con el que asociamos un nombre a un valor. Siendo ese valor una dirección de memoria.
- Declaración:

```
tipo* etiqueta;
```

Punteros clásicos

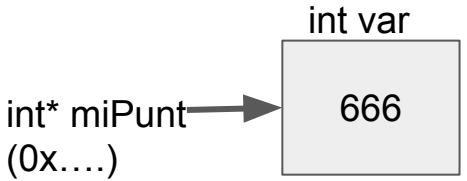
- Declaración:

```
tipo* etiqueta;
```

- Ejemplo:

```
int* miPunt;

//miPunt tendrá la dirección de un
valor entero
```



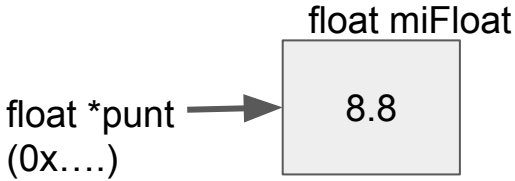
Punteros clásicos

- Inicialización:

```
float miFloat{8.8};

float* punt{&miFloat};

//Como en miPuntero vamos a guardar la dirección de un float, lo podemos inicializar con la dirección de una variable float -> Operador &
```



Punteros clásicos

- Inicialización:

```
float* miFloat{nullptr};

//Es recomendable inicializar el
puntero a nullptr para evitar apuntar
a un sitio desconocido (si no
inicializamos una variable su
contenido será impredecible)
```

Punteros clásicos

- Inicialización:

```
float* miFloat{new float};

//Con new le decimos al sistema operativo que nos "deje" una posición de memoria para un tipo float
```

- Con operador new reservamos dinámicamente memoria

Punteros clásicos

- Operadores relacionados con punteros:
  - Operador dirección &
    - Nos da la dirección de una variable
  - Operador indirección *
    - Lo usamos para declarar un puntero
    - Acceso al contenido de un puntero
  - Ejemplo
  - Otro ejemplo

Punteros clásicos

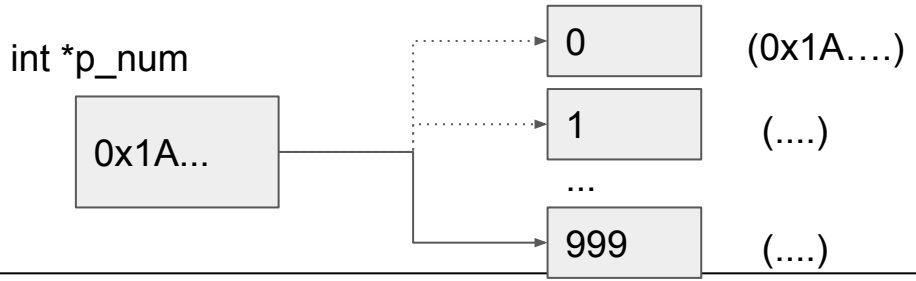
- Operadores new/delete:
  - new: reserva dinámica de memoria, pedimos al Sistema Operativo que nos de memoria
  - delete: liberamos esa memoria cuando ya no la necesitamos. Si no la liberamos podríamos agotarla (memory leaks)

Punteros clásicos

- new/delete:
  - Ejemplo pérdida información al liberar memoria
  - Es muy importante liberar la memoria porque si no podríamos “agotar” la memoria de nuestro sistema
    - Ejemplo

Punteros clásicos

- El ejemplo anterior de “agotamiento” de memoria tiene 2 problemas
  - Estamos perdiendo la información en la memoria de nuestro sistema
  - No vamos a poder liberarla (memory leak)



Punteros clásicos

- Resumen:
  - Puntero es un tipo de datos que almacena una dirección de memoria.
  - El puntero puede ser de cualquier tipo

    tipo* etiqueta;
  - Pero... debe ser del tipo a donde apunta
  - Si hemos reservado con new, tiene que haber un delete (para evitar memory leaks)

Punteros clásicos

- Podemos tener contenedores de punteros
- Podemos tener paso de punteros por parámetro en nuestras funciones
- Podemos tener retorno de punteros en nuestras funciones
- Podemos usarlos en nuestras funciones lambda
- …

Punteros clásicos

- Pero… este año no vamos a utilizarlos (al menos en Progra I y II)
- Porque el estándar de C++ moderno recomienda el uso de los punteros inteligentes frente a los punteros clásicos

Github

- https://github.com/Nebrija-Programacion/Programacion-I/blob/master/temario/variables/punteros.md

Índice

- Introducción: Programación I
- Punteros
    - Clásicos
    - Inteligentes

Smart pointers

Problemas punteros clásicos:

- Cada “new” con su “delete” -> Cuidado con las fugas de memoria!! (Memory leaks)
- ¿Posesión? -> Si una función devuelve un puntero... ¿quién se encarga de liberar memoria?

Smart pointers

Usando "smart pointers" nos quitamos los problemas:

- Son tipos de datos abstractos que **encapsulan** los punteros "en bruto" en su interior.
- Son plantillas que permiten el manejo de punteros clásico, mediante la **sobrecarga** de los operadores "*****" y "**->**". Cuando dejan de usarse se libera la memoria inmediatamente a través de su **destructor**.
- Por lo que no es necesario hacer delete. Y no nos tenemos que acordar ni tenemos que decidir quién lo hace.

Al igual que los tipos contenedor lo veremos en las próximas lecciones… de momento acto de fe :)

Smart pointers

Usando “smart pointers” nos quitamos los problemas:

- Realizan una gestión automatizada de la memoria, liberándola automáticamente cuando la memoria reservada ya no es necesaria.
- El estándar de C++ (C++20) recomienda su uso ISO/IEC 14882:2020.

Smart pointers

Tipos de punteros inteligentes C++11(**#include <memory>**)

- **std::unique_ptr** : Puntero inteligente que tiene un recurso de reserva dinámica.
- **std::shared_ptr** : Puntero inteligente que tiene un recurso de reserva dinámica compartido.
- **std::weak_ptr** : Como shared_ptr pero sin incrementar contador
- Nos centraremos sólo en ***unique_ptr*** y ***shared_ptr***

Índice

- Punteros
  - Clásicos
  - Smart pointers:
    - std::unique_ptr
    - std::shared_ptr

Smart pointers: unique_ptr

- std::unique_ptr
  - Es un template que está definido en la cabecera <memory> (#include <memory>)
  - Es un tipo de datos que define una referencia a un objeto que solo existe en el ámbito de la variable, y que además no puede ser referenciado por ninguna otra variable.

Smart pointers: unique_ptr

- std::unique_ptr
  - Ejemplo
  - Para acceder a su contenido usamos los operadores * o ->
    - Ejemplo uso *
    - Ejemplo uso ->
  - ¿Que habéis notado después de estos ejemplos? -> No hacemos deletes… No es necesario!! lo gestiona el tipo:)

Smart pointers: unique_ptr

- std::unique_ptr
  - Ejemplo ciclo de vida
    - Antes de pasar al editor… ¿Qué hace el programa?
    - Depuramos en el entorno de desarrollo para verlo.

Smart pointers: unique_ptr

- *std::unique_ptr* características:
  - Es propietario del objeto al que apunta.
  - Cuando se acaba su ámbito (scope) se libera la memoria.
  - Útil cuando necesitamos un recurso de reserva de memoria dinámica temporal

Smart pointers: unique_ptr

- *std::unique_ptr* características:
  - El objeto referenciado no puede estar referenciado por ninguna otra variable. Es decir, no puede haber varios punteros apuntando al mismo espacio de memoria.

Smart pointers: unique_ptr

- *std::unique_ptr* características:
  - Ejemplos uso unique_ptr “tipo teoría de examen”.
    - Ejemplo 1
    - Ejemplo 2
    - Ejemplo 3
    - Ejemplo 4
    - Ejemplo 5

Índice

- Introducción gestión de memoria
- Punteros
  - Clásicos
  - Smart pointers:
    - std::unique_ptr
    - std::shared_ptr
- Ejemplos

Smart pointers: shared_ptr

- std::shared_ptr
  - Es un template que está definido en la cabecera <memory> (#include <memory>)
  - Es un tipo de datos que define una referencia a un objeto que solo existe en el ámbito de la variable, y que puede ser referenciado por otras variables.

Smart pointers: shared_ptr

- std::shared_ptr
  - Ejemplo
  - Para acceder a su contenido usamos los operadores * o ->
    - Ejemplo uso *
    - Ejemplo uso ->
  - ¿Que habéis notado después de estos ejemplos? -> No hacemos deletes… No es necesario!! lo gestiona el programa :)

Smart pointers: shared_ptr

- std::shared_ptr
  - Ejemplo ciclo de vida
    - Antes de pasar al editor… ¿Qué hace el programa?
    - Depuramos en el entorno de desarrollo para verlo.

Smart pointers: shared_ptr

- *std::shared_ptr* características:
  - Como unique_ptr, pero… permitiendo múltiples referencias.
  - Tiene un contador que se decrementa cada vez que un shared_ptr que apunta al mismo recurso sale del alcance. Y se incrementa cada vez que se comparte
  - Útil cuando quieres “compartir” tu dato de memoria dinámica (cómo punteros clásicos)

Smart pointers: shared_ptr

- *std::shared_ptr* características:
  - Al ser shared_ptr, puede tener varias referencias.
  - Puede existir varios punteros apuntando a la misma zona de memoria
  - Si tuviera varias referencias, el tipo shared_ptr “sabe” cuál es el momento oportuno para borrar. Ejemplo

Smart pointers: shared_ptr

- *std::shared_ptr* características:
  - Ejemplos uso shared_ptr "tipo teoría de examen".
    - Ejemplo 1
    - Ejemplo 2
    - Ejemplo 3
  - Ejemplo contador de shared_ptr

Smart pointers: Github

- Punteros inteligentes:
  - *unique_ptr*
  - *shared_ptr*

Índice

- Introducción gestión de memoria
- Punteros
    - Clásicos
    - Smart pointers
- Ejemplos

Ejemplos

Ejemplo: basándonos en este ejercicio vamos a ver ventajas de trabajar con punteros.

código

Ejemplos

Ejemplo: implementar una estructura para albergar un número complejo. E implementar una función que devuelva un puntero inteligente a un número complejo con el resultado de la suma del número complejo pasado por parámetro de entrada + (10+10i)

“puntero inteligente” funSuma(Complejo C1);