Programación II

Tema 8 Repaso general

jtorresmat@nebrija.es

Índice

Punteros clásicos

Puntero es un tipo de datos más (*tipo**) con el que apuntamos a la dirección de una variable.

- Puntero = dirección de memoria.
- Una variable = dirección de memoria y su contenido.

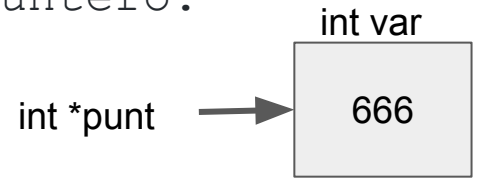
Operadores para trabajar con punteros:

- Operador dirección: &
- Operador indirección: *
- Operador acceso-> (equivalente al operador .)

Punteros clásicos

- Visualización del concepto puntero:

```
int var=666;
int *punt= &var;
```



- Inicialización puntero
  - A un puntero podemos asignarle la dirección de otra variable del mismo tipo (caso dibujo bloque)
  - También podemos reservar un espacio nuevo de memoria (memoria dinámica) => deberemos liberar la reserva una vez que no la necesitemos (para evitar memory leaks)

```
float *var= new float{1.23};
cout<<*var; //muestra 1.23
cout<< var; //muestra 0x.... (la dirección)
```

Punteros inteligentes

Problemas con 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?

Punteros inteligentes

- Son tipos de datos abstractos que **encapsulan** los punteros “en bruto” en su interior.
- Son plantillas que permiten el manejo de punteros clásicos, 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 => No nos tenemos que acordar ni tenemos que decidir quién lo hace.

Punteros inteligentes

- std::unique_ptr

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/smartpointers.md

- std::shared_ptr

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/smartpointersii.md

Índice

Repaso Herencia

- Campus Virtual apartado: Tema 6_Sesión 11 accesos herencia
  - Ejemplo diferencia entre herencia privada y protegida

Repaso Herencia

- Herencia

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/herencia.md

- Herencia, constructores y destructores

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/herenciaII.md

- Herencia, miembros públicos, privados y protegidos

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/herenciaIII.md

Índice

Repaso Polimorfismo

- Polimorfismo

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/polimorfismo.md

- Polimorfismo, clases abstractas

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/polimorfismoii.md

- Polimorfismo, downcasting

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/downcasting.md

Índice

Repaso Clases

- Clases:
  - Funciones miembro
  - Variables miembro
- Privado y público
- Constructores (sobrecarga)
  - Por defecto
  - Con parámetros
  - Copia
- Destructor (no sobrecarga)

Rectangulo
private: double ladoLargo; double ladoCorto; point vertice;

public:
Rectangulo(void);
Rectangulo(double ladoL,double ladoC);
double getLargo(void);
double getCorto(void);
point getVertice(void);
void setLargo(double lado);
void setCorto(double lado);
void setVertice(point vert);
double getArea(); //….

Constructor y destructor

- Constructor:
  - El/los constructores de una clase son unos métodos especiales que sirven para inicializar un objeto al mismo tiempo que se declara.
  - Características:
    - Tienen el mismo nombre de la clase
    - No tienen valor de retorno (tampoco **void**)
    - Suelen ser públicos
    - Es el primer método que se llama al declarar un objeto de esa clase
    - Se pueden sobrecargar
    - Si no definimos ninguno nos dan uno de “oficio”

Repaso Clases

- Destructor:
  - Definición: función miembro de una clase a la que se le llama cuando el objeto va a dejar de existir.
    - Si el objeto es local-> Cuando acabe su ámbito
    - Si el objeto es global-> Cuando termine el programa
    - Si objeto creado con ***new*** -> Con el ***delete***
  - Características:
    - Se llama *~NombreClase()*
    - Es único (no admite sobrecarga)
    - No tiene argumentos ni valor de retorno
    - Si no se define uno, el compilador asigna uno "por defecto"
  - Ejemplo

Clases

- Clases:
  - Funciones miembro
  - Variables miembro

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clases.md
- Privado y público

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clasesII.md
- Constructores

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clasesIII.md

Índice

Excepciones

- Gestión de errores con excepciones:
  - Donde podemos prever una situación de error lanzamos la excepción
    - throw{};
  - Donde vayamos a gestionar el tratamiento de la excepción
    - try{}catch(){}
- Ejemplo

Excepciones

- Gestión de errores con excepciones

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/excepciones.md

Índice

Repaso Sobrecarga de operadores

- La sobrecarga de operadores nos permite ampliar sus capacidades (por ejemplo con + podremos "sumar" más cosas, clases persona, etc)
- La declaración y definición es parecida a la de una función.
- Sintaxis:

```
"tipo" operator "símbolo"(operadores);
```

valor de retorno

palabra reservada

operador

los que necesite la operación...

Repaso Sobrecarga de operadores

- También se puede sobrecargar un operador dentro de una clase.
    - Operadores binarios y unarios
- Sintaxis:

```
"tipo" nombre clase:: operator "símbolo"(operadores);
```

Repaso Sobrecarga de operadores

- Sobrecarga de “<<” y “>>”
  - “<<”:

```
std::ostream &operator <<(std::ostream & os,"UnTipo" const & “etigueta”){
  os << "adasdasd " << ... << "\n";
  os << "asdsad " << ... << "\n";
  return os;
}
```

  - “>>”:

```
std::istream &operator >>(istream &is,"UnTipo" &”etigueta”)
{
    is>>”etigueta”.sdasad>>”etigueta”.sdasad2;
    i.ignore();

    return i;
}
```

  - Si se quieren encapsular hay que usar friend

Repaso Sobrecarga de operadores

- Sobrecarga operadores aritméticos binarios y unarios:

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/sobrecargaopar.md

- Sobrecarga operadores lógicos:

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/sobrecargaoplog.md

- Sobrecarga operadores de flujo:

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/sobrecargaopos.md

Repaso Sobrecarga de operadores

Ejemplo: implementar una clase que represente un polinomio. Se debe sobrecargar los operadores <<, >> y +

Para que en el programa principal podamos introducir los coeficientes por pantalla, sumar 2 polinomios y mostrar su resultado.

Índice

- Punteros
  - Clásicos
  - Inteligentes
- Herencia
- Polimorfismo
- Clases
- Excepciones
- Sobrecarga
- Plantillas (funciones y clases)

Repaso Plantillas

- Funciones templatizadas:

```
template<typename T>
void "etiqueta"(T const & a){
    std::cout << a << "\n";
}
```

nombre del “comodín”

declaración de la función poniendo el tipo comodín donde sea necesario

Repaso Plantillas

- Funciones templatizadas:
  - Podemos tener varios “comodines”

```
template<typename T, typename K>
K “etiqueta”(T const & a, K const & b){
    return a*b;
}
```

Repaso Plantillas

- Clases templatizadas:

```
template<typename T>
class "etiqueta"{
    public:
        T getMiVar();
        void setMiVar(T in);
    private:
        T miVar;
}
```

nombre del "comodín"

declaración de la clase poniendo el tipo comodín donde sea necesario

En la definición de la clase el mismo concepto

Repaso Plantillas

- Funciones templatizadas

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/funcionestempl.md

- Clases templatizadas

  https://github.com/Nebrija-Programacion/Programacion-II/blob/master/temario/clasescionestempl.md