Lenguaje de programación C++

Manejo de memoria y punteros.
Sobrecarga de operadores

Repaso de punteros

Hemos dicho que C++ mantiene la sintaxis y filosofía de C. Por lo tanto, el manejo de la memoria es similar.

C++ introduce una palabra reservada, ***nullptr*** (la dirección 0)

```
int * i {nullptr} forma de inicializar el puntero a null

void func (int* i) // func(nullptr)
```

```
int *i = new int;
delete i;
```

I aloca un entero y es lo que se libera

```
int* p = new int[30];
delete[] p;
```

p señala la primera posición del arreglo de 30 intereos. Se requiere liberar todas las posiciones.

Repaso de punteros

Este también es un error común:

```
char* cadena(){
    char cad[25];
    strcpy(cad, “Hola”);
    return cad;
}
```

Si vemos, se está retornando un puntero a una variable local, al finalizar la función, esta variable se elimina y la posición es nula.

Asignar un valor en la posición de memoria ya eliminada.

```
int *i = new int;
delete i;

*i = 1;
```

Pasaje de parámetros

Podemos distinguir dos opciones que dependerán del uso, el prototipo y/o la necesidad de lo que se programa.

- paso el valor (la variable que es copia en la llamada a la función)

```
void func_param (int x, int y){ /* …*/}

func_param (entero_1, entero_2);
```

La función toma dos enteros, al invocarse, se generan copias de esas variables y los valores originales de entero_1 y entero_2 no se alteran.

Pasaje de parámetros

```
#include<iostream>
using namespace std;

void referencia(int& x){
  x = 25;
}

int main (){
  int y = 5;

  cout<< "y = "<< y << endl;
  referencia(y);
  cout<< "y = "<< y << endl;
  return 0;
}
```

Aquí usamos la referencia al puntero, si compilamos y ejecutamos el programa, el resultado es el siguiente:

```
$ g++ -Wall -oreferencia referencia.cpp
$ ./referencia
y = 5
y = 25
```

Vemos que hemos podido cambiar al valor de la variable original.

Pasaje de parámetros

Vemos al puntero usado como referencia.
El efecto es igual, el parámetro no es una copia y cambia el valor de la variable original.

```
$ g++ -Wall -opuntero puntero+.cpp
$ ./puntero
y = 10
y = 20
y = 40
```

```
#include<iostream>
using namespace std;

void duplica(int* x){
  *x *= 2;
}

int main (){
  int y{10};
  int* p{nullptr};

  cout<< "y = "<< y << endl;
  duplica(&y);
  cout<< "y = "<< y << endl;
  p = &y;
  duplica(p);
  cout<< "y = "<< y << endl;

  return 0;
}
```

Retorno de valores

Similar a los parámetros de entradas, puedo retornar el valor de una variable

Su referencia, a lo cual tenemos que pensar como en un alias de la variable, y por lo tanto no se genera la copia.

O el puntero, solo se tiene que tener en cuenta que debe hacerse la alocación y la debida asignación antes de retornar y no el puntero a una variable local.

Constantes

- Similar a las variables ordinarias, tienen un tipo y nombre.
- Una vez inicializadas no pueden cambiar su valor
- Para declarar una constante se utiliza la palabra reservada ***const***
- Puede ser aplicado a:
  - Variables
  - Argumentos de funciones y retornos
  - Punteros
  - Atributos de las clases
  - Funciones de las clases

Constantes

Ejemplos

Una variable constante:

```
/*...*/
const float pi{3.1416};

pi = 3.141592 // error al compilar
```

Puntero a una constante:

```
/*...*/
const int* x;
```

y un puntero constante

```
char a = 'a';
char * const cp_a = &a;
```

El contenido de la dirección podría cambiar, pero no la dirección del puntero.

Argumentos y retornos constantes

- Si un atributo de una clase es declarado constante, debe ser inicializado en el constructor, no puede estar inicializado en su declaración.
- La variable retornada por una función puede ser constante.

```
const int f ();
```

- El método de una clase puede ser declarado constante pero este método no podrá modificar a los valores de los atributos del objeto.

valor_retorno nombre_función () ***const***;

Sobrecarga de operadores

La sobrecarga es una de las características de la Orientación a Objetos. Nos permite redefinir funciones para que en determinados contextos actúen de una manera u otra.

¿Por qué querría sobrecargar a un operador? Principalmente, mantendría una coherencia dentro del uso del lenguaje, si puedo comparar un string con el operador ==, podría también sobrecargar a este para que la funcione con objetos de mi diseño, por ejemplo, dos objetos personas

Operadores

Recordemos que un operador es un símbolo que le indica al compilador una operación matemática, lógica u de otro tipo que se defina. Los ya conocidos operadores binarios “+”, “-”, “!=”, “=”, etc. Otros incorporados por el lenguaje como son los operadores de flujo “<<” o “>>” son ejemplos de operadores.

C++ nos permite redefinirlos, sobrecargarlos dentro de nuestras clases para que extiendan su funcionamiento su operatividad a los objetos propios.

Sobrecarga de operadores

Veamos el ejemplo ya usado de un objeto Punto. La suma de dos puntos A y B daría como resultado un tercer punto C donde A + B = C {(A.x + B.x); (A.y + B.y)}

```
class Punto{
  int x, y;

  public:
  Punto(int xx, int yy): x(xx), y(yy){}
  Punto operator+(const Punto& b) const;
  int get_x() {return x;}
  int get_y() {return y;}

};

Punto Punto::operator+(const Punto& b) const{
  return Punto(x + b.x, y + b.y);
}
```

```
int main(){
  Punto a(5, 3);
  Punto b(2, 2);

  Punto c = a + b; //(7, 5)
  cout << c.get_x() << " - "
       << c.get_y() << endl;
  return 0;

}
```

Funciones ***friends***

- Se definen en la clase pero no son miembros de ella.
- Tienen acceso a los atributos (aunque sean privados) por lo tanto no pueden invocarse utilizando el nombre del objeto y un operador de acceso (. o ->).
- Una función amiga se llama como una función global.
- Si bien tiene acceso a los atributos privados, y al no ser miembro, requiere que reciba los objetos como parámetros para que pueda manipularlos.
- Una función amiga no puede heredarse.
- Una función puede ser amiga de varias clases.
- Una función puede ser miembro de una clase y amiga de otra.

Ejemplos

Sobrecarga de un operador para el mismo objeto

```
#include<iostream>
using namespace std;

class Punto{
  int x, y;

  public:
  Punto(int xx, int yy):x(xx), y(yy){}
  friend Punto operator+(const Punto& a,
                         const Punto& b);
  int get_x() {return x;}
  int get_y() {return y;}
};

Punto operator+(const Punto& a, const Punto& b){
  return Punto(a.x + b.x, a.y + b.y);
}
```

```
int main(){
  Punto a(5, 3);
  Punto b(2, 2);

  Punto c = a + b; //(7, 5)
  cout << c.get_x() << " - "
       << c.get_y() << endl;
  return 0;
}
```

Ejemplo: Función friend compartida entre objetos

```
#include<iostream>
using namespace std;

class Camion;

class Auto{
  int pasajeros, velocidad;

  public:
  Auto(int cant, int vel):pasajeros(cant), velocidad(vel){}
  friend int operator-(const Auto& a, const Camion& c);

};
class Camion{
 int peso, velocidad;
 public:
 Camion(int p, int v):peso(p), velocidad(v){}
 friend int operator-(const Auto& a, const Camion& c);
};
```

Ejemplo: Función friend compartida entre objetos

```
int operator-(const Auto& a, const Camion& c){
  return a.velocidad - c.velocidad;
}

int main(){
  Auto a(5, 230);
  Camion c(1500, 180);

  cout << a - c << endl; // 50
  return 0;

}
```

Si compilamos vemos:

```
$ g++ -Wall -std=c++11 -ofriendCompartido friendCompartido.cpp
$ ./friendCompartido
50
[pablo@deskPC DOO]$
```

Ejemplo: función miembro y amiga

No pondré el ejemplo completo, sino una modificación al anterior en una situación intermedia. C++ permite lo siguiente:

```
class Auto{
  int pasajeros, velocidad;

  public:
  Auto(int cant, int vel):pasajeros(cant), velocidad(vel){}
  int operator-(const Camion& c) const;

};
class Camion{
 int peso, velocidad;
 public:
 Camion(int p, int v):peso(p), velocidad(v){}
 friend int Auto::operator-(const Camion& c)const;
};

int Auto::operator-(const Camion& c)const{
  return velocidad - c.velocidad;
}
```

Clases Amigas

Al concepto que vimos, lo podemos extender a las clases, teniendo en cuenta algunas salvedades:

- La amistad no puede transferirse, si A es amigo de B, y B es amigo de C, esto no implica que A sea amigo de C.
- La amistad no puede heredarse. Si A es amigo de B, y C es una clase derivada de B, A no es amigo de C.
- La amistad no es simétrica. Si A es amigo de B, B no tiene por qué ser amigo de A.

Ejemplo

```
#include<iostream>
using namespace std;

class Amigable {
private:
 int secreto;
 friend void FuncionAmiga(Amigable& o);
 friend class Amiga;
};
class Amiga {
public:
 void mirar(Amigable& o) {
  o.secreto = 120;
  cout << o.secreto << endl;
 }
};
void FuncionAmiga(Amigable& o) {
 o.secreto = 121;
 cout << o.secreto << endl;
}
```

```
int main() {
 Amiga amiga;
 Amigable amigable;
 amiga.mirar(amigable);
 FuncionAmiga(amigable);

 return 0;
}
```

```
$ g++ -Wall -std=c++11 -oamigable amigable.cpp
$ ./amigable
120
121
```

Resumen

- Como sintaxis, tenemos dos opciones:

```
retorno operador@ (parámetros);

friend retorno operador@ (parámetros)
```

Siendo @ el operador que se desea sobrecargar.

- Los operadores ***“=”*** y ***“&”*** ya se encuentran sobrecargados por default para todas las clases que se creen.

  Sin embargo esto último, la sobrecarga del = es útil, pero podría provocar un problema, la cual se soluciona haciendo nuestra propia sobrecarga.

Resumen

Es cuando dentro de los atributos mientros tenemos punteros. el dejar que se autoasignen los valores con el = traería un posible doble direccionamiento.

Al asignar atributo por atributo A b = A a, los datos de a pasan a b. Con punteros se haría una copia de direcciones y no de contenidos.

Sugerencias:

- Solo se debe sobrecargar un operador si su uso, su interfaz es obvia, natural, del mismo modo en que se utilizan para los tipos primitivos.
- Los objetos pasados por parámetros deben ser referencias

Resumen

- Los objetos pasados por parámetros no deben sufrir alteraciones por lo tanto deben ser pasados como constantes.
- Debe haber un retorno, sea el puntero this o uno de los objetos ingresados por referencia