Introducción al Cómputo

Vida de los objetos
Funciones miembros especiales
Repaso de conceptos

Operadores **`new`** y **`new[]`**

- Existen 2 formas de uso del operador new.

```
Type *pointer = new Type;
Type *pointer = new Type[numero_de_elementos];
```

La primera expresión es usada para alocar espacio para una simple instancia del tipo **`Type`**.

La segunda es usada para alocar un bloque (para manejarla como arreglo nativo) de elementos del tipo **`Type`**.

```
int *ptr = new int[5];
```

En este caso se aloca dinámicamente espacio para 5 elementos del tipo **`int`** y retorna un puntero al primer elemento de la secuencia.

Operadores **`delete`** y **`delete[]`**

- En general, la memoria alocada dinámicamente es necesaria solo durante períodos de tiempo dentro del programa, una vez que no es más necesaria, esta puede ser liberada para que vuelva a estar disponible.

- El operador **`delete`** cumple esa función de liberación

```
delete pointer;
delete [] pointer;
```

La primera expresión libera un elemento simple alocado con **`new`**.

Las segunda libera la memoria alocada para un arreglo nativo a través del operador **`new[]`**

Alocación dinámica ¿Por qué? ¿Para qué?

- Controlar el tiempo de vida de los objetos (variables).
  - Variables locales a las funciones, viven dentro de su scope.
  - Variables globales y estáticas, viven del comienzo al fin del programa.

- Modelo de programas que trabajan con “documentos”. Editores de texto, planillas de cálculo, navegadores, compiladores, etc.
  - No conocen al momento de compilación la cantidad de memoria que van a necesitar como para declarar las variables.

- Contenedores. Objetos que sirven para coleccionar otros objetos.
  - **`std::vector`**, **`std::list`**, **`std::map`**
  - **`std::string`**

`Vector`: Constructor y Destructor

```
struct Vector
{
    Vector(int s) {
        sz = s;
        elem = new double[sz];
    }

    ~Vector() {
        delete [] elem;
    }

    int sz;
    double *elem;
};


Vector e;  // Error de compilación!
           // El constructor necesita
           // un argumento
```

```
void f()
{
    Vector v(7);

    v.elem[3] = 3.14;

    cout << v.elem[2] << endl;

    // v sale de scope y se llama a ~Vector
}

void g()
{
    Vector v(3);

    v.sz = 8;
    v.elem = nullptr;

    // v sale de scope y se destruye
}
```

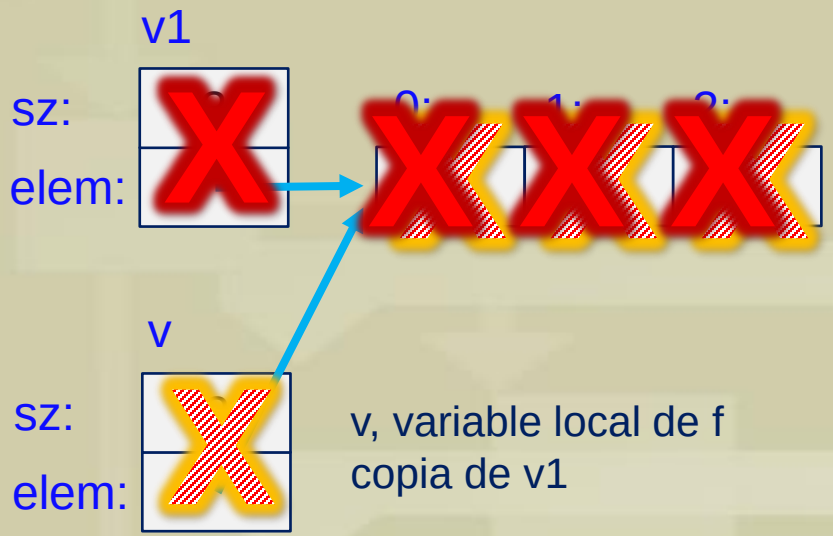
`class Vector`: Interface Pública

```
class Vector
{
public:
    Vector(int s) {
        sz = s;
        elem = new double[sz];
    }

    ~Vector() {
        delete [] elem;
    }

    double &operator[](int idx) {
        return elem[idx];
    }

    int size() const { return sz; }

private:
    int sz;
    double *elem;
};
```

```
// rv pasado por referencia
// v pasado por copia
void f(Vector &rv, Vector v)
{
    rv[0] = 3.14;

    cout << v[0] << endl;

    // v sale de scope y se llama a ~Vector
}

int main() {
    Vector v1(7);
    Vector v2(5);

    v1[0] = 2.71;
    cout << v1[0] << endl;

    cout << v1.size() << endl;
    f(v1, v2);

    return 0;
} // v1 y v2 salen de scope y se destruyen
```

Problemas con `Vector`

```
// v es pasado por copia
void f(Vector v)
{
    cout << v[0] << endl;
    v[1] = 8.1;

    // v1 sale de scope y se llama ~Vector
}

int main() {
    Vector v1(3);
    v1[0] = 1.1;

    f(v1);

    cout << v1[0] << endl;

    return 0;

    // v1 sale de scope y se llama ~Vector
}
```



Dos problemas con la copia default miembro a miembro:

- La copia no es tan copia
- La destrucción de la copia corrompe el original

Solución: Constructor Copia

`Vector`: Constructor Copia

```
class Vector
{
public:
    Vector(int s) {
        sz = s;
        elem = new double[sz];
    }
    ~Vector() {
        delete [] elem;
    }

    Vector(const Vector &v) {
        sz = v.sz;
        elem = new double[sz];
        for( int i = 0; i < sz; ++i )
            elem[i] = v.elem[i];
    }

private:
    int sz;
    double *elem;
};
```

```
int main()
{
    // Constructor Vector(int)
    Vector a(5);

    f(a); // Se copia el argumento con el
          // constructor copia

    // Constructor copia
    Vector b(a);
    Vector c{a};
    Vector d = a;

    Vector x(5);

    x = a;   // ?????

    return 0;
}
```

`Vector`: Asignación Copia

```
class Vector
{
public:
    Vector(int s);
    ~Vector();

    Vector(const Vector &v);

    Vector &operator=(const Vector &v) {
        double *p = new double[v.sz];
        for( int i = 0; i < v.sz; ++i ) {
            p[i] = v.elem[i];
        delete [] elem;
        elem = p;
        sz = v.sz;
        return *this;
    }

private:
    int sz;
    double *elem;
};
```

```
Vector carga_vector(int sz);

void f()
{
    // Constructor Vector(int)
    Vector a(5);

    // Constructor copia
    Vector b(a);
    Vector c{a};
    Vector d = a;

    Vector x(5);

    x = a;   // Asignación copia

    x = carga_vector(9); // Asignación copia

    // Constructor copia
    Vector z = carga_vector(123);

}
```

Vida de los objetos

- Constructor: inicializa un objeto, puede o no recibir argumentos.
  - Garantiza el buen estado de un objeto desde su creación.
  - Método con el mismo nombre de la clase.
  - No tiene valor de retorno.
- Destructor: se llama automáticamente cuando un objeto deja de existir.
  - Garantiza la liberación de recursos mantenidos en el objeto.
  - Método con el mismo nombre de la clase precedido por **~**.
  - No recibe argumentos. No tiene valor de retorno.
  - Si no está definido, el compilador genera uno por defecto que llama a los destructores de cada uno de sus miembros.

```
class X {
    public:
        X();    // constructor
        ~X();   // destructor
};
```

```
void f(){
    X a;                // constructor a
    X *p = new X();     // constructor *p
    delete p;           // destructor *p
}                       // destructor a
```

Vida de los objetos

- Acceso a miembros de los objetos a través de su interface.

- Copia de objetos:
  - Cuando se pasa un objeto a una función por valor.
  - Cuando se inicializa un objeto utilizando con otro del mismo tipo.
  - El compilador por defecto copia miembro a miembro.
  - Hay que definir un constructor copia si se quiere evitar el comportamiento por defecto.

- Asignación de objetos:
  - Cuando se asigna con el operador **=**.
  - El compilador por defecto asigna miembro a miembro.
  - Hay que definir un **`operator=`** si se quiere evitar el comportamiento por defecto.

- Usos avanzados en C++, fuera del scope de ICOM. Prohibición de construcción o destrucción de objetos. Prohibición de copia de objetos. Move de objetos. Swap de objetos.

Vida de los objetos

```
struct C {
    C();                              // constructor "default", no recibe argumentos
    C(T t1, ...);                     // constructor(es) con argumentos
   ~C();                              // destructor
    C(const C &c);                    // constructor copia
    C &operator=(const C &c);         // asignación copia
};
```

- Todos estos métodos especiales pueden estar definidos o no para una clase.
- Si no están definidos en la clase, el compilador los genera por default, aplicando la operación a cada dato miembro del objeto.
- La implementación por default de destructor, constructor copia u operador de asignación copia puede ser problemática.

Rule of Three

```
struct C {
    ~C();                           // destructor
    C(const C &c);                  // constructor copia
    C &operator=(const C &c);       // asignación copia
};
```

- Regla empírica para evitar problemas: Rule of Three (or Law of The Big Three):
  - Si una clase define uno o más de los siguientes, probablemente debería definir explícitamente todos los 3:
    - destructor
    - constructor copia
    - operador de asignación copia
- Link: https://en.wikipedia.org/wiki/Rule_of_three_(C%2B%2B_programming)

`MyString` – Ejercicio 7 de la práctica 9

Con fines didácticos y de aprendizaje se desea implementar el UDT **`MyString`** para representar strings. El tipo tiene que ser implementado de tal forma de poder soportar el uso siguiente:

```
MyString s1("hola mundo");
MyString s2(s1);
MyString s3;

s3 = s1 + s2;
s3[4] = 'X';
char c = s3[2];
```

- ¿Cúal sería la representación interna?
- ¿Qué constructores necesitaría?
- Evaluar la necesidad de los Big 3.
- ¿Qué interface pública necesitaría?
- ¿Qué operadores necesitaría?

`MyString` – Representación Interna

- ¿Qué miembros dato necesito para poder representar un string?
  - Conjunto indexado de caracteres para guardar los caracteres del string.
- ¿Sé **ahora** cuántos caracteres necesito?
  - Ahora significa cuando estoy escribiendo el código y compilando.
  - No. Así que voy a necesitar allocar un array nativo dinámicamente.
- Si la cantidad de caracteres es dinámica, ¿necesito saber cuántos hay?
  - En principio puedo usar la convención que los strings están terminados con un character 0. Así que cuando necesito saber cuántos hay, los puedo contar.
  - El programa se la va a pasar haciendo loops cada vez que quiera saber cuántos hay. Entonces me conviene tener ese dato a mano. Cache.

```
struct MyString {
    char *str;
    size_t sz;
};
```

`MyString` – Constructores

- ¿Qué constructores necesito proveer?
  - Me piden:

```
MyString s1("hola mundo");
MyString s2(s1);
MyString s3;
```

  - Construir con un string literal.
  - Construir con otro **`MyString`**.
  - Constructor por defecto, sin argumentos.

```
struct MyString {
    MyString();
    MyString(const MyString &);
    MyString(const char *);

    char *str;
    size_t sz;
};
```

MyString – Big 3

- ¿Necesito alguno de los Big 3?
  - Seguro voy a necesitar definir un destructor que no es el default para liberar la memoria allocada dinámicamente.
  - Entonces voy a tener problemas con las copias generadas por default.
  - Se cumple la regla de los Big 3. Hay que hacer todos...

```
struct MyString {
    MyString();
    MyString(const MyString &);
    MyString(const char *);
    ~MyString();
    MyString &operator=(const MyString &);

    char *str;
    size_t sz;
};
```

`MyString` – Interface Pública

- ¿Qué interface pública necesitaría?
  - Conocer el tamaño del string.
  - Acceso por índice a los caracteres. Usar el operador `[]` es una notación conveniente.

```
class MyString {
    public:
        MyString();
        MyString(const MyString &);
        MyString(const char *);
        ~MyString();
        MyString &operator=(const MyString &);
        size_t size() const;
        char &operator[](int idx);

    private:
        char *str;
        size_t sz;
};
```

`MyString` – Operadores

- ¿Qué operadores necesitaría?
  - Operador `[]` para acceso indexado a los caracteres.
  - Operador `+` con semántica de concatenación de strings.

```
class MyString {
    public:
        MyString();
        MyString(const MyString &);
        MyString(const char *);
        ~MyString();
        MyString &operator=(const MyString &);
        size_t size();
        char &operator[](int idx);
        MyString operator+(const MyString &);

    private:
        char *str;
        size_t sz;
};
```

`MyString` – Detalles

- Definición del método **`size()`** como:

```
size_t size() const;
```

- ¿Qué significa el **`const`** que aparece al final?
- A un objeto constante o referencia constante o puntero constante no se lo puede modificar. Para eso fue calificada su definición como **`const`** y el compilador garantiza que no se modifique.
- ¿Podría llamar a un método sobre un objeto (o referencia o puntero) constante que lo modifique?
- No. El compilador es el garante que eso no suceda.
- ¿Entonces no puedo llamar ningún método sobre un objeto (o referencia o puntero) constante?
- Acá es donde aparece la calificación **`const`** de un método.
- El compilador garantiza que en un método calificado **`const`** no se modifique el objeto.
- Por lo tanto sobre un objeto (o referencia o puntero) constante solo se pueden llamar métodos calificados **`const`**.
- Por ejemplo:

```
size_t size() const { return sz; }
```

- No modifica el estado del objeto. Retorna el miembro **`sz`** por copia.

- ¿Y qué pasa con: **`char &operator[](int idx);`** ?

`MyString` – Detalles

- ¿Y qué pasa con: `char &operator[](int idx);` ?

```
char &operator[](int idx) { return str[idx]; }
```

- Retorna una referencia a un caracter del objeto. A través de esa referencia se puede modificar el objeto. Por lo tanto no se puede aplicar el operador `[]` sobre un objeto (o referencia o puntero) constante.
- ¿Entonces no puedo llamar al operador `[]` sobre un objeto (o referencia o puntero) constante?
- Así como está, NO.
- Necesitaría calificar `const` al operador [].
- Pero si lo califico `const` no voy a poder retornar una referencia un caracter miembro...
- Solución:

```
char &operator[](int idx) { return str[idx]; }
char operator[](int idx) const { return str[idx]; }
```

- Definir 2 operadores `[]`. Sobrecarga u overloading.
- Uno retorna una referencia a través de la cual se puede modificar indirectamente el objeto.
- El segundo retorna una copia, por lo que lo puedo calificar `const`.