Qué vimos? Objetos función: Functors

Un ***Functor*** es un objeto que actúa como una función. Básicamente es una clase la cual define el **`operator()`**.

```
class Duplicator {
   public:
      int operator()(int x) { return x * 2;}
};

Duplicator doubler;
int x = doubler(5);

cout << x << " " << doubler(32) << endl;
```

Functors: pueden mantener estado

Una ventaja de los functores respecto a los punteros a función tradicionales es que pueden mantener estado (contexto):

```
class Matcher {
      int target;
   public:
     Matcher(int m) { target = m; }
     bool operator()(int x) { return x == target;}
};

Matcher is5(5);
if (is5(n))     // igual que if (n == 5)
{ ....}
```

Functors: en algoritmos

Ejemplo holístico:

```
struct add_x {
    add_x(int x_) { x = x_; }
    int operator()(int y) const { return x + y; }

  private:
    int x;
};
add_x add42(42);  // crea una instancia del functor
int i = add42(8); // y lo llama ( i -> 50 )

vector<int> in { 1, 2, 3, 4, 5, 6 };
vector<int> out(in.size());
// Pasa un functor a transform, llama al functor para cada elemento
// de la secuencia 'in', y almacena el resultado en la secuencia 'out'
transform(in.begin(), in.end(), out.begin(), add_x(1));
```

Funciones de comparación en algoritmos

```
#include "icom_helper.h"

struct Terna {
    int i, j, value;
    bool operator<(const Terna &t2) const {
        return value < t2.value;
    }
};
int main() {
    std::vector<Terna> s{{1,2,3},{2,3,4},{3,4,2},{1,1,1}};

    std::sort(s.begin(), s.end());
    for (auto a : s)
       std::cout << a.value << " ";

    std::cout << '\n';

}
```

```
1 2 3 4
```

Funciones de comparación en algoritmos

```
#include "icom_helper.h"

struct Terna {
    int i, j, value;
};

bool cmp1(const Terna &t1, const Terna &t2) {
    return t1.value < t2.value;
}
int main() {
    vector<Terna> s{{1,2,3},{2,3,4},{3,4,2},{1,1,1}};

    sort(s.begin(), s.end(), cmp1);
    for (auto a : s)
        cout << a.value << " ";

    cout << '\n';
}
```

```
1 2 3 4
```

Alocación dinámica de memoria

- Hasta ahora toda la memoria necesaria estaba determinada antes de que el programa se ejecute definiendo todas las variables necesarias.

- Hay casos en donde la cantidad de memoria necesaria por un programa solo puede ser determinada en tiempo de ejecución (por ejemplo cuando depende del input del usuario). En estos casos el programa necesita alocar memoria dinámicamente.

- C++ introduce los operadores **`new`** y **`delete`** que permiten este mecanismo.

- Las variables creadas a través de **`new`** alocan memoria del heap y deben ser explícitamente destruidas con **`delete`**.

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.

Formas de chequear alocación correcta

- Un pedido de reserva de memoria puede fallar (por ejemplo porque no hay más recursos disponibles).

- La forma natural de fallar del operador **`new`**, o **`new[]`** es disparar una excepción del tipo **`bad_alloc`**.

```
int *ptr = new int[5];  // si falla dispara bad_alloc
```

- La otra forma de fallar se conoce como **`nothrow`** y termina retornando un puntero a 0 (**`nullptr`**) para indicar la falla.

```
int *ptr = new (nothrow) int[5];
if( ptr == nullptr) {
    // error handling
}
```

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[]`**

`new` y `delete` con UDTs

Hasta ahora, cuando necesitábamos instanciar un UDT, declarábamos variables (instancias de ese UDT) que automáticamente activaba algún constructor adecuado para asegurar el buen estado de la variable creada.

```
class A {
    public:
        A(int a_)          { a = a_; b = 0; }
        A(int a_, int b_) { a = a_; b = b_;}
        ~A()               { }
        int getA()         { return a; }
        int getB()         { return b; }
        ...

    private:
        int a, b;
};
...
A a1(5);
A a2(2, 4);
```

`new` y `delete` con UDTs

De la misma forma, cuando una instancia de UDT sale de ámbito, el compilador se encarga de activar el destructor de esa instancia (muy importante si hay alocación de recursos en la instancia).

```
class V {
    public:
        V(int n_)  { n = n_; elems = new int[n]; }
        ~V()       { delete [] elems; }
        ...
    private:
        int *elems, n;
};
...
{
    V v1(1000000);
    ...
}
```

new y **delete** con UDTs

- Cuando se crean instancias de UDTs a través de **new**, este operador, además de alocar la memoria para la instancia, activa el constructor de la clase sobre el objeto alocado para permitir una inicialización consistente.
- Del mismo modo, cuando se active el operador **delete** sobre una instancia creada con **new**, se activa el destructor del objeto y finalmente se libera su memoria.

```
class V {
    public:
        V(int n_)  { n = n_; elems = new int[n]; }
        ~V()       { delete [] elems; }
    private:
        int *elems, n;
};
...
V *pv1 = new V(1000000);  // se crea una instancia de V en el heap
...
delete pv1;               // se destruye la instancia
```