Introducción al Cómputo

Convención de pasaje de argumentos

Ámbito de Variables

Constructores/destructores

Convención de pasaje de argumentos

Un mecanismo:

```
void fun(int a) {
    cout << "a vale: " << a << " su memoria esta en: " << &a << '\n';
    a = 64;
    cout << "a vale: " << a << '\n';
}
int main() {
    int b = 32;
    cout << "b vale: " << b << " su memoria esta en: " << &b << '\n';
    fun(b);
    cout << "b vale: " << b << '\n';
    return 0;
}
```

```
b vale: 32 su memoria esta en: 0xffffcc1c
a vale: 32 su memoria esta en: 0xffffcbf0
a vale: 64
b vale: 32
```

Pasaje de argumentos por valor

Una vieja: `std::vector` como argumento

ejem6_5.cpp

```
#include "icom_helpers.h"

void func(vector<int> a) {
    cout << "En func &a[0] = " << &a[0] << '\n';
    cout << "En func &a    = " << &a << '\n';
    a[1] = 32;
}

int main(void)
{
    vector<int> v {0, 1, 2, 3, 4, 5, 6};
    cout << "En main &v[0] = " << &v[0] << '\n';
    cout << "En main &v    = " << &v << '\n';

    func(v);
    cout << "En main v[1] = " << v[1] << '\n';

    return 0;
}
```

```
$ ./ejem6_5.exe
En main &v[0] = 0x600000440
En main &v    = 0xffffcbd0
En func &a[0] = 0x600000470
En func &a    = 0xffffcbf0
En main v[1] = 1
```

Convención de pasaje de argumentos

Otro mecanismo:

```
void fun(int &a) {
    cout << "a vale: " << a << " su memoria esta en: " << &a << '\n';
    a = 64;
    cout << "a vale: " << a << '\n';
}
int main() {
    int b = 32;
    cout << "b vale: " << b << " su memoria esta en: " << &b << '\n';
    fun(b);
    cout << "b vale: " << b << '\n';
    return 0;
}
```

```
b vale: 32 su memoria esta en: 0xffffcc1c
a vale: 32 su memoria esta en: 0xffffcc1c
a vale: 64
b vale: 64
```

Pasaje de argumentos por referencia

Otra vieja: `std::vector` como argumento

ejem6_5.cpp*

```
#include "icom_helpers.h"

void func(vector<int> &a) {
    cout << "En func &a[0] = " << &a[0] << '\n';
    cout << "En func &a    = " << &a << '\n';
    a[1] = 32;
}

int main(void)
{
    vector<int> v {0, 1, 2, 3, 4, 5, 6};
    cout << "En main &v[0] = " << &v[0] << '\n';
    cout << "En main &v    = " << &v << '\n';

    func(v);
    cout << "En main v[1] = " << v[1] << '\n';

    return 0;
}
```

```
$ ./ejem6_5_1
En main &v[0] = 0x600000440
En main &v    = 0xffffcbf0
En func &a[0] = 0x600000440
En func &a    = 0xffffcbf0
En main v[1] = 32
```

Otra vieja: Arrays nativos como argumentos

ejem6_2.cpp

```
#include "icom_helpers.h"

void func(int a[]) {
    cout << "En func &a[0] = " << &a[0] << '\n';
    cout << "En func a     = " << a << '\n';
}

int main(void)
{
    int v[20] = {0};

    cout << "En main &v[0] = " << &v[0] << '\n';
    cout << "En main v     = " << v << '\n';

    func(v);

    return 0;
}
```

```
$ ./ejem6_2
En main &v[0] = 0xffffcbd0
En main v     = 0xffffcbd0
En func &a[0] = 0xffffcbd0
En func a     = 0xffffcbd0
```

Convención de pasaje de argumentos

- En C++, siguiendo a C, por defecto todos los pasajes de argumentos se realizan por valor.
  - Este tipo de convención garantiza que la función llamada no puede modificar el valor de la variable original (ojo con los arreglos nativos!).
  - Costoso cuando se pasan tipos “gordos”, ya que siempre se realiza una copia del objeto.

- C++ agrega un nuevo mecanismo, el pasaje de argumentos por referencia.
  - Mucho mas eficiente cuando se pasan tipos “gordos”, ya que lo que se pasa en definitiva es una dirección de memoria (la de la variable original).
  - Da acceso en la función llamada a la memoria de la variable/objeto original. Puede ser peligroso. Cuando se quiere proteger ese acceso, la técnica es hacer que reciba una referencia constante:

```
void fun(const array<int, 1000000> &a)
{
    // ...
}
```

Ámbito de variables

El ámbito (scope) de una variable define:

- Visibilidad: dónde es visible.
- Tiempo de vida: cuándo se crea y cuándo deja de existir.

Existen principalmente 2 ámbitos para variables:

- Variables locales, automáticas o de stack.
- Variables globales.

Variables Locales

A las variables/objetos con ámbito local también se las refiere como variables locales, automáticas o de stack.

- **Visibilidad**: bloque de código en el que se la define. Debe ser declarada antes de ser usada.

- **Tiempo de vida**: el tiempo que dure la ejecución del bloque que la definió. Se crean cuando la variable se declara, dejan de existir cuando termina el ámbito en donde fue declarada. Su lugar de vida es una zona llamada “stack” o “pila”.

Variables Locales

```
void f1(int a1, double a2) // a1 y a2 locales a f1
{
    int a, b, c;     // variables locales a f1
    …
    …
    while(…) {
        int d;       // variable local al bloque
        …
    }
}
void f2(char a1)      // a1 local a f2
{
    int a;           // variables locales a f2
    …
}
```

Variables Locales

```
void f() {
    int a;
    cout << "a de f() esta en " << &a << '\n';
}
void g() {
    int b;
    cout << "b de g() esta en " << &b << '\n';
    f();
}

int main()
{
    f();
    g();
    return 0;
}
```

```
$ ./test
a de f() esta en 0x28cd14
b de g() esta en 0x28cd14
a de f() esta en 0x28ccf4
```

Ámbito Global

Las variables con ámbito global son aquellas que se declaran fuera de cualquier función.

- **Visibilidad**: Son visibles desde cualquier función.
- **Tiempo de vida**: el tiempo que dure la ejecución del programa.

Variables globales

```
int a = 1;  // variables globales
int b = 4;

void f() {
    int a;
    a = 5;
    b = 8;
    cout <<"en f, a: " << a << " b: " << '\n';
}
int main()
{
    cout << "antes de f, a: " << a << " b: " << b << '\n';
    f();
    cout << "después de f, a: " << a << " b: " << b << '\n';
    return 0;
}
```

```
$ ./test
antes de f, a: 1 b: 4
en f, a: 5 b: 8
después de f, a: 1 b: 8
```

Variables globales

- Se debe minimizar el uso de variables globales.
- Las funciones deberían enterarse del problema que tienen que resolver sólo a través de sus argumentos.
- Una función que utiliza variables globales es muy difícil de reutilizar en otro contexto.
- Usar variables locales, sumado a la convención de pasajes de argumentos por valor o referencias constantes, otorga gran controlabilidad de modificación de una variable.

Variables estáticas

- Una variable/objeto puede declararse como estática.

```
static double valor;
```

- Aplicadas sobre variables/objetos de ámbito local: Le modifican dónde son alojadas. En lugar del stack, se crean en la zona de «datos», modificando también su tiempo de vida.

Variables estáticas

```
#include <time.h>
#include <stdio.h>

void f()
{
    static int s = 0;
    int b = 0;
    s++;
    b++;
    cout << "s: " << s << " b: " << b  << '\n';
}

int main() {
    f(); f(); f(); f();
    return 0;
}
```

Ámbito en instancias de UDT

- Se manejan de manera similar a las instancias de tipos nativos.

- C++ provee un mecanismo que permite al programador que una instancia de un UDT sea inicializada correctamente en su declaración. Esto se realiza a través de la definición de uno o más "**constructores**".

- De idéntica forma, C++ provee un mecanismo para que una instancia de un UDT pase por una fase de cleanup/destrucción cuando se termina el ámbito en donde fue declarada. Esto se realiza a través de la definición de un "**destructor**".

Constructores/destructores

- Constructor:
  - Es llamado automáticamente cuando se declara una instancia del tipo.
  - Es como un método. Se caracteriza por tener el mismo nombre del tipo, y por no retornar nada.
  - Puede haber varios constructores (igual que métodos con el mismo nombre) siempre que se diferencien por cantidad y/o tipo de argumentos.
- Destructor:
  - Es llamado automáticamente cuando una variable sale de ámbito.
  - Es como un método. Se caracteriza por tener el mismo nombre del tipo con el caracter ~ adelante, por no retornar nada y por no recibir argumentos.

Constructores/destructores

ejem7_1.cpp

```
#include "icom_helpers.h"

struct T {
    int x, y;
    T() {
        x = y = 0;
        cout << "Construccion con T()\n";
    }
    T(int x_, int y_) {
        x = x_;
        y = y_;
        cout << "Construccion con T(int, int)\n";
    }
    ~T() {
        cout << "Destruccion (" << x << ", "
             << y << ")\n";
    }
    void print() {
        cout << "(" << x << ", " << y << ")\n";
    }
};
```

```
int main() {
    T t1;
    t1.print();
    cin.get();

    {
        T t3(3,4);
        t3.print();
        cin.get();

    }             // sale de scope t3
    cin.get();
    T t2(1,2);
    t2.print();
    cin.get();

    return 0;
// salen de ambito t2 y t1
}
```

Constructor de copia (Copy constructor)

- Existe un constructor especial que es activado cuando se quiere inicializar un objeto a partir de otro del mismo tipo (UDT). Esto se produce cuando:
  - Se realiza una inicialización explícita:
    `T a = b;` // ó `T a(b);` donde `b` es del tipo `T`.
  - Se pasan argumentos por valor a funciones:

```
T a;
...
f(a);  // f(T x) {...}
```

  - En el retorno de funciones:

```
T f() {
    T a;
    …
    return a;                    // -fno-elide-constructors
}
```

Copy constructor: forma explícita

- La forma explícita de definir un constructor de copia es a través de un constructor con la firma:

```
struct T {
    T(const T &x) { … }
};
```

ejem7_2.cpp

```
struct A {
    A()             { cout << "Construyendo " << this << endl; }
    A(const A &x)   { cout << "Haciendo una copia de " << &x << " en " << this << endl; }
    ~A() { cout << "Destruyendo " << this << endl; }
};

int main() {
    A a;
    cout << "a esta en " << &a << endl;
    A b(a);
    cout << "b esta en " << &b << endl;
    return 0;
}
```

```
Construyendo 0xffffcc0f
a esta en 0xffffcc0f
Haciendo una copia de 0xffffcc0f en 0xffffcc0e
b esta en 0xffffcc0e
Destruyendo 0xffffcc0e
Destruyendo 0xffffcc0f
```

Copy constructor: forma implícita

- Cuando se diseña un UDT, si no se especifica un “copy constructor” el compilador generará uno automáticamente que hará una copia miembro a miembro de la instancia a copiar.
- Hay ocasiones donde un constructor de copia automático es suficientemente bueno.
- Hay ocasiones donde un constructor de copia automático es absolutamente malo.