Unidad 1 - Programación de computadoras

Parte II - Repaso de C++

Laboratorio Avanzado

enero, 2021

Asignación dinámica de memoria

Introducción

Asignación dinámica de memoria se refiere a las técnicas de programación donde el programador es quien define cómo se hará la distribución de memoria, ya sea desde el inicio, o conforme vaya siendo requerido en la ejecución del programa. También es el programador el que se encarga de manejar las referencias que indican donde está la memoria asignada.
En contraposición, en la programación elemental, el programador solo indica que variables va necesitar y es el compilador el que se encarga de asignar la memoria requerida y de manejar las referencias a esta. Esto tiene la desventaja de que si no se conoce desde el inicio cuantas variables o cuantos números va a ser necesario utilizar, no se puede hacer la programación.

Punteros

Un puntero es un tipo especial de *variable* que se utiliza en programación. El valor que almacena es exclusivamente una *dirección de memoria*. Esta dirección de memoria puede ser la del *inicio* de una variable o de un bloque de instrucciones (subrutinas). Se dice que los punteros *apuntan* a otro elemento.
Si bien un puntero, por su naturaleza, no necesita estar asociado al tipo de valor que está almacenado en el espacio de memoria al cual apunta, los lenguajes de programación si lo hacen. Esto quiere decir que se van a tener *punteros a enteros*, *punteros a flotantes*, *punteros a funciones*, etc. siempre con un *apellido* que la naturaleza de aquello a lo que apuntan.
Un caso especial es cuando no se sabe que hay en el espacio de memoria al cual el puntero apunta. Estos son punteros sin tipo (*void*).

```cpp
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    /* Declaracion de una variable entera a */
    int     a = 123;
    /* Declaracion de un puntero p a una variable entera */
    int*    p = 0;

    /* La forma de obtener la direccion de una variable
     * es poner el simbolo & antes de ella. La siguiente
     * linea le asigna a p la direccion de a.
     */
    p = &a;

    /* Se imprimen los valores */
    cout<< "Valor de a: " <<  a << endl;
    cout<< "Valor de p: " <<  p << endl;
    /* Se puede acceder al valor de la variable a traves
     * de un puntero a ella. Se coloca un * antes del puntero
     */
    cout<< "Valor de a via *p: " <<  *p << endl;

    return 0;
}
```

codes/pointer.cpp

```
1 Valor de a: 123
2 Valor de p: 0x7ffe125a448c
3 Valor de a via *p: 123
```

codes/pointer.out

Notese que el valor del puntero *p* que es la dirección de memoria se muestra en hexadecimal. Es generalizada la práctica de mostrar las direcciones de memoria en hexadecimal por que son mas cortas y es posible siempre poner todos los dígitos, lo cual reduce confusiones. La longitud de una dirección de memoria depende directamente del *hardware* de la computadora donde se esté corriendo el programa. El valor de *p* puede cambiar en cada corrida del programa, ya que el sistema operativo asigna un espacio de memoria a la variable *a* al momento de ejecución.

Una buena práctica de programación es inicializar los punteros a cero, como se ve en el código de `pointer.cpp`. El valor de memoria cero o `NULL` es un valor especial que se usa para indicar que el puntero no esta asignado a ninguna dirección de memoria válida.
En necesario hacer notar que a los punteros que se definan en los programas que escribamos, deben apuntar a direcciones de memoria de variables del mismo programa. El sistema operativo no nos permitirá acceder a espacios de memoria asignados a otros programas por motivos de seguridad (al menos, eso es lo que debería suceder).

Estructuras de datos

Por estructuras de datos se entiende a la agrupación de varias variables en un bloque *lógico*. En la mayoría de casos, este bloque también ocupa un bloque continuo de memoria.
La estructura mas simple es agrupar *n* variables del mismo tipo, esto es lo que se conoce en C/C++ como *arreglos* (arrays).

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    /* Declaracion de un array 5 de enteros sin inicializar */
    int     a1[5];
    /* Declaracion de un array 5 de enteros inicializados (forma 1 ) */
    int     a2[5] = { 1, 2, 3 ,4 , 5};
    /* Declaracion de un array 5 de enteros inicializados (forma 2 ) */
    int     a3[]  = { 5, 4, 3 ,2 , 1};
    /* Declaracion de un array 5 de enteros inicializados (forma 3 ) */
    int     a4[5]  = { 0 };

    cout<< "i"  << setw(15)
        << "a1" << setw(5)
        << "a2" << setw(5)
        << "a3" << setw(5)
        << "a4" << endl;

    for( int i = 0; i< 5; i++)
        cout<< i     << setw(15)
            << a1[i] << setw(5)
            << a2[i] << setw(5)
            << a3[i] << setw(5)
            << a4[i] << endl;

    return 0;
}
```

codes/array01.cpp

```
1 i          a1    a2    a3    a4
2 0   681803968     1     5     0
3 1       32765     2     4     0
4 2           1     3     3     0
5 3           0     4     2     0
6 4   681803968     5     1     0
```

codes/array01.out

En el código `array01.cpp` se muestra como declarar arrays de enteros de 4 formas diferentes. Para el arrego `a1` se declara que ocupará 5 elementos y se asigna el espacio de memoria, pero *no se le asignan valores*. En la salida se nota que este toma valores pseudoaleatorios, que son básicamente lo que estaba en el lugar de memoria cuando fue asignada e interpretada ahora como enteros. Los demás arreglos son ejemplos de otras formas de declararlos con inicialización. En el caso de `a4` se debe hacer notar que funciona para asignar valor inicial cero a todos los elementos, lamentablemente este esquema no funciona para otros valores iniciales.

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    /* Declaracion de un array 5 de enteros inicializados */
    int     a2[5] = { 1, 2, 3 ,4 , 5};
    /* a2 contiene la direccion del inicio del bloque,
     * esta se puede asignar a un puntero del mismo tipo
     * del arreglo. */
    int*    p = a2;

    cout<< "a2: " << a2 << endl << endl;

    cout<< "i"   << setw(5)
        << "*p"  << setw(20)
        << "p++" << endl;

    /* Los punteros se pueden incrementar y obtener el
     * valor de donde apuntan. */
    for( int i = 0; i< 5; i++)
        cout<< i     << setw(5)
            << *p    << setw(20)
            << p++   <<  endl;

    return 0;
}
```

codes/array02.cpp

```
1 a2: 0x7ffd32c91750
2
3 i    *p                    p++
4 0     1       0x7ffd32c91750
5 1     2       0x7ffd32c91754
6 2     3       0x7ffd32c91758
7 3     4       0x7ffd32c9175c
8 4     5       0x7ffd32c91760
```

codes/array02.out

Los punteros guardan una relación especial con los arreglos como se puede ver en el código `array02.cpp`. Si se llama el nombre del arreglo sin ningún subíndice, este devuelve la dirección del inicio del bloque del arreglo.
Los punteros pueden ser manipulados con los operadores `++`, `--`, `+=`, `-=`, `+`, `-` y obtener otra dirección de memoria. Cada vez que un puntero se incrementa o decrementa lo hace en múltiplos enteros del tamaño en bytes del tipo de variable al cual apunta. Esto es lo que se conoce como *aritmética de punteros*. Por supuesto, si estas operaciones dan direcciones de memoria que están fuera del bloque definido por a2 en este caso, los resultados serán impredecibles.

El siguiente tipo de estructura que se estudiará es aquel donde los tipos de datos agrupados ya no son necesariamente los mismos. Para hacer esto en C/C++ se utiliza `struct`. Las estructuras definidas con `struct` ya no utilizan índices para identificar los datos, sino nombres. Con `struct` también se pueden definir nuevos tipos de datos.

```cpp
#include <cstdlib>
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    /* Declaracion de una nueva estructura accesible a
     * travez de la variable Examen */
    struct {
        char        Carnet[10];
        unsigned    Nota;
        bool        Aprobado;
    } Examen;

    /* Los _campos_ de la estructura se llaman por nombre
     * tanto para asignacion como para lectura */
    strcpy(Examen.Carnet, "199516766");
    Examen.Nota = 61;
    Examen.Aprobado = true;

    cout<< "Carnet: "   << Examen.Carnet    << endl
        << "Nota: "     << Examen.Nota      << endl;

    if( Examen.Aprobado )
        cout << "APROBADO" << endl;
    else
        cout << "REPROBADO" << endl;

    return 0;
}
```

codes/struct01.cpp

```
1 Carnet: 199516766
2 Nota: 61
3 APROBADO
```

codes/struct01.out

En el ejemplo `struct01.cpp` se define una nueva estructura, esta contiene 3 campos de diferente tipo de dato. La forma de acceder a esta nueva estructura es a través de la variable `Examen`. Para acceder a un campo determinado, se coloca un punto después del nombre de la variable y se coloca el nombre del campo.

```cpp
#include <cstdlib>
#include <iostream>
#include <iomanip>
#include <cstring>

using namespace std;

int main()
{
    /* Declaracion de una nueva estructura que sera
     * utilizada como un nuevo tipo de variable */
    struct Examen_t {
        char        Carnet[10];
        unsigned    Nota;
        bool        Aprobado;
    };

    /* Examen_t es un nuevo tipo, y puede ser usado
     * para definir variables individuales, o como
     * en este ejemplo, un arreglo */
    Examen_t Primer_parcial[3];

    /* La asignacion debe hacerce para cada elemnto
     * del arreglo. */
    strcpy(Primer_parcial[0].Carnet, "199516766");
    Primer_parcial[0].Nota = 61;
    Primer_parcial[0].Aprobado = true;

    strcpy(Primer_parcial[1].Carnet, "200516766");
```

codes/struct02.cpp

```
30      Primer_parcial[1].Nota = 51;
31      Primer_parcial[1].Aprobado = false;
32
33      strcpy(Primer_parcial[2].Carnet, "201516766");
34      Primer_parcial[2].Nota = 71;
35      Primer_parcial[2].Aprobado = true;
36
37      cout<< setw(12) << "Carnet"
38          << setw(7)  << "Nota"
39          << setw(13) << "Estado" << endl;
40
41      for( int i = 0; i < 3; i++ )
42      {
43          cout<< setw(12) << Primer_parcial[i].Carnet
44              << setw(7)  << Primer_parcial[i].Nota;
45
46          if( Primer_parcial[i].Aprobado )
47              cout << setw(13) << "APROBADO" << endl;
48          else
49              cout << setw(13) << "REPROBADO" << endl;
50      }
51
52      return 0;
53  }
```

codes/struct02.cpp

```
1        Carnet    Nota      Estado
2     199516766      61    APROBADO
3     200516766      51   REPROBADO
4     201516766      71    APROBADO
```

codes/struct02.out

En el ejemplo `struct02.cpp` se define un nuevo tipo de dato `Examen_t`. Este nuevo tipo puede ser utilizado como cualquier otro tipo de los predefinidos en C/C++. En este ejemplo en particular, se utiliza para construir un arreglo de variables del nuevo tipo definido.

La sintaxis de `struct` en general es la siguiente:

```
1 struct <struct_type> {
2
3         declaracion de campos
4
5 } <struct_vars>;
```

Donde `<struct_type>` se utiliza cuando se desea declarar un nuevo tipo de variable. `<stuct_vars>` se utiliza cuando se desea declarar variables con esta estructura.

```cpp
#include <cstdlib>
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    /* A continuacion se utiliza la forma completa de struct.
     * Sin embargo, es poco frecuente utilizala. */
    struct Examen_t {
        char        Carnet[10];
        unsigned    Nota;
        bool        Aprobado;
    } Examen;

    /* El tener la estructura como un tipo tambien
     * permite definir punteros a estas. */
    Examen_t* P2Examen = &Examen;

    /* La forma de acceder a los campos de la estructura
     * desde un puntero es por medio de -> en lugar de . */
    strcpy(P2Examen->Carnet, "199516766");
    P2Examen->Nota = 61;
    P2Examen->Aprobado = true;

    cout<< "Carnet: "   << P2Examen->Carnet    << endl
        << "Nota: "     << P2Examen->Nota      << endl;

    if( P2Examen->Aprobado )
        cout << "APROBADO" << endl;
    else
        cout << "REPROBADO" << endl;

    return 0;
}
```

codes/struct03.cpp

```
1 Carnet: 199516766
2 Nota: 61
3 APROBADO
```

codes/struct03.out

En el ejemplo `struct03.cpp` se muestra como usar punteros con estructuras definidas por `struct`. Notese la sustitución del uso de `.` (punto) por `->` para las llamadas y asignaciones de los campos.

new y delete

Las instrucciones `new` y `delete`, junto con los punteros, son la base para la asignación dinámica de memoria en C++. `new` reserva un espacio de memoria del tipo y cantidad de variable indicado. `delete` *libera* el espacio de memoria reservado por `new`. A diferencia de otros lenguajes de programación, C++ no implementa un *recolector de basura* generalizado, esto quiere decir que el espacio de memoria reservado por la instrucción `new` no se libera automáticamente al finalizar la ejecución del segmento donde fue definido. Sin embargo esto si sucede con los arreglos.

Una de las formas de utilizar `new` es la siguiente:

```
1 <var_type>* var_pointer = new <var_type>[longitud];
```

Donde `<var_type>` es cualquier tipo de variable predefino o definido por medio de `struct`. `longitud` es cuantos elementos de `<var_type>` se van a agrupar en el bloque de memoria. Notese que `new` devuelve un puntero del tipo `<var_type>`.

La sintaxis de `delete` es la siguiente:

```
1 delete var_pointer;
```

Donde `var_pointer` es un puntero a un bloque de memoria reservado por `new`.

```cpp
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
    /* Se reserva un bloque de memoria de 5 enteros */
    int* a = new int[5];

    /* Se puede verificar que la asignacion de mamoria
     * haya sido exitosa antes de continuar */
    if( !a )
    {
        cout << "Asignacion de memoria fallida";
        return 1;
    }

    /* Se puede utilizar a como si fuera un arreglo */
    for( int i = 0; i<5; i++)
        a[i] = i;

    /* Tambien se puede usar aritmetica de punteros */
    for( int i = 0; i<5; i++)
        cout<< (*a)++ << endl;

    /* Se libera el espacio de memoria reservado */
    delete a;

    return 0;
}
```

codes/adm01.cpp

```
1 0
2 1
3 2
4 3
5 4
```

codes/adm01.out

En el ejemplo `adm01.cpp` se muestra como reservar un bloque de memoria de 5 enteros. Se muestra como la asignación de valores puede ser realizada como si fuera un arreglo y como la *aritmética* de punteros puede ser aplicada a estos. Notese el uso de `delete` al final de código.

```cpp
#include <cstdlib>
#include <iostream>
#include <cstring>

using namespace std;

int main()
{
    struct Examen_t {
        char        Carnet[10];
        unsigned    Nota;
        bool        Aprobado;
    };

    /* Ahora se utiliza new para reservar el espacio
     * de memoria de una variable tipo Exament_t. */
    Examen_t* Examen = new Examen_t;

    /* Como Examen ahora es un puntero, se utiliza ->
     * en lugar de . para acceder a los campos. */
    strcpy(Examen->Carnet, "199516766");
    Examen->Nota = 61;
    Examen->Aprobado = true;

    cout<< "Carnet: "   << Examen->Carnet    << endl
        << "Nota: "     << Examen->Nota      << endl;

    if( Examen->Aprobado )
        cout << "APROBADO" << endl;
    else
        cout << "REPROBADO" << endl;

    return 0;
}
```

codes/adm02.cpp

```
1 Carnet: 199516766
2 Nota: 61
3 APROBADO
```

codes/adm02.out

El ejemplo `adm02.cpp` es el mismo `struct03.cpp` pero utilizando `new` para reservar el espacio de memoria y obtener el puntero.

De los ejemplos mostrados, el uso de `new` y `delete` no parece ofrecer ninguna ventaja sobre la declaración normal de variables y el uso de arreglos. De hecho, parece complicar más las cosas al tener que recordar liberar la memoria con `delete`. Entonces ¿para qué sirven estas instrucciones?

- `new` como se mencionó antes, reserva espacio de memoria y es independiente del segmento donde fue hecha la llamada. Esto tiene la *ventaja* de que es posible usar ese espacio de memoria desde varias subrutinas, teniendo que comunicar únicamente donde está por medio de un puntero. La desventaja es tener que recordar usar `delete` al ya no necesitar esa reserva.
- `new` tiene otras formas de llamada que se utilizan con `clases`, que es el elemento fundamental de C++.

Programación orientada a objetos

La programación de computadoras ha evolucionado a lo largo de los años, conforme se ha ido incrementando la velocidad y capacidad de procesamiento de estas.
Las primeras computadoras eran programadas cableando directamente las entradas para indicar los datos sobre los cuales debían operar.
Luego, por medio de tarjetas perforadas, ya no solo era posible indicar valores de entrada, sino también las operaciones que debían realizar sobre estas. En estas primeras máquinas la programación era en *lenguaje de máquina*, es decir *binario puro*, y el programador debía conocer todas las secuencias que la máquina podía operar.
Con el desarrollo de la electrónica, las computadoras redujeron considerablemente su tamaño e incrementaron la capacidad de computo, pero aún siguieron requiriendo ser programadas en binario.

El primer gran avance en lo referente a programación se hizo con el desarrollo de *assembler*. Assembler es un conjunto de recursos nemotécnicos, donde lo que se hace es sustituir una instrucción binaria por una frase y se utilizan *nombres de variable* para referirse a datos almacenados en memoria, así como también los datos se ingresan en decimal, caracteres, etc. Luego, otro programa se encarga de traducir este código sustituyendo las frases nemotécnicas y nombres de variables por sus equivalentes binarios. Aún con este avance, programar en assembler depende totalmente del conjunto de instrucciones soportadas por la computadora, ya que solo es una equivalencia del binario.

Uno de los primeros lenguajes de programación de *alto nivel* fue C, cuyo primer compilador fue escrito totalmente en assembler.
Por alto nivel se quiere decir que este lenguaje no debía ser dependiente de la computadora donde iba a ejecutarse, siendo el compilador el encargado de traducir al binario adecuado. Además el lenguaje es de tipo *procedural*, esto es, se programan subrutinas independientes que interactuan entre ellas por medio de llamadas. Nuevamente el compilador es el encargado colocar estas subrutinas en el orden necesario para que el computador las pueda interpretar y escribir el binario respectivo.

La programación orientada a objetos, *OOP* por sus siglas en inglés, es otro *paradigma* de programación en el cual el elemento conceptual base es el *objeto*, no el procedimiento o la subrutina.
Un objeto, es un ente conceptual, que posee *atributos* y que puede realizar acciones (*métodos*) basado en/o sobre sus atributos.
La motivación de este nuevo enfoque es simplificar la comprensión y programación de proyectos grandes donde la programación procedural se vuelve muy engorrosa. También se fomenta la reutilización de código en múltiples tareas.

Ejemplo 1 - Elevador

Atributos .

- Capacidad máxima
- Ocupación
- Posición (nivel)
- Estado dinámico (movimiento o estacionario)
- Dirección de movimiento (arriba o abajo)
- Estado de la puerta (cerrada o abierta)
- ...

Ejemplo 1 - Elevador

Métodos (acciones) .

- Abrir puerta
- Cerrar puerta
- Subir pasajeros
- Cambiar de nivel
- ...

Ejemplo 2 - Persona

Atributos .

- Nombre
- Fecha de nacimiento
- Lugar de nacimiento
- DPI
- ...

Ejemplo 2 - Persona

Métodos (acciones) .

- Decir su nombre
- Decir su edad
- Decir de donde es
- Presentarse (Soy ... tengo ... soy de ...)
- ...

Ejemplo 3 - Vector

Atributos .

- Dimensión
- Elementos (arreglo)
- ...

Ejemplo 3 - Vector

Métodos (acciones) .

- Calcular su magnitud
- Expresar en coordenadas polares (2 dimensiones)
- Expresar en coordenadas esféricas (3 dimensiones)
- Asignación (operador $=$)
- Igualdad (operador $==$)
- Suma (operador $+$)
- Producto interno (operador *)
- ...

En el paradigma procedural, la forma en que los procedimientos se comunican entre ellos es por medio de *llamadas*. En OOP, los objetos *se relacionan* entre ellos.

Existen dos formas básicas de relacionar objetos:

Composición Esta se dá entre dos objetos cuando el segundo objeto contiene entre sus atributos al primer objeto. Por ejemplo: Una reunión (2do. objeto) *tiene* personas (1re. objeto); Una base ortogonal (2do. objeto) *tiene* vectores (1re. objeto). Esta forma de relación es del tipo **tiene un (has a)**.

Herencia Esta se dá entre dos objetos cuando el segundo objeto *es un* objeto mas general o limitado del primero. Por ejemplo: Un estudiante (2do. objeto) *es una* persona (1re. objeto); Una vector unitario (2do. objeto) *es un* vector (1re. objeto). Esta forma de relación es del tipo **es un (is a)**.

En realidad C++ es un lenguaje *multi-paradigma* ya que se puede utilizar en forma procedural u OOP, esto es, se puede trabajar definiendo funciones al estilo C, definiendo objetos que se relacionan entre ellos, o una combinación de funciones y objetos. Esta última forma termina siendo la mas usada, gracias a su versatilidad para el desarrollo proyectos de mediana complejidad.

Clases

Las clases en C++ es donde se definen lo que serán los objetos de programación. La clase en sí se concibe como un ente abstracto donde se definen los atributos y los métodos. Los *objetos* serán las *instancias* de estas clases.
En general, la definición de clases se coloca en archivos con extención `.h`, `.hh` o `.hpp`. La implementación de los métodos se coloca en archivos `.cc` o `.cpp`. La función `main` desde donde se hace la instanciación de estas clases va en otro archivo, también con extensión `.cc` o `.cpp`.
A continuación se muestra un ejemplo.

```hpp
#ifndef __SIMPLE_HPP__
#define __SIMPLE_HPP__

#include <string>

/* Clase simple: Clase de ejemplo. */
class simple
{
    /* Tanto los atributos como los metodos pueden ser definidos
     * privados o publicos. Generalmete los atributos son privados,
     * esto quiere decir que solo es posible acceder a ellos a
     * traves de los metodos puplicos de la clase. */
private:
    /* Estos son los atributos del objeto (data members)
     * Son diferentes para cada instancia de la clase */

    /* Atributo de cada instancia. */
    std::string Frase;

    /* Atributo de clase: es comun a todas las instancias. */
    static int Numero;

public:
    /* Estos son los metodos del objeto (member functions) */

    /* Constructor sin argumentos */
    simple();
    /* Constructor con argumentos */
    simple( std::string La_Frase );
    /* Destructor */
```

codes/simple.hpp

```
31      ~simple();
32
33      /* Metodo para solicitar que imprima la frase.
34       * Se llaman a travez de las instancias de la clase */
35      void Decir_Frase();
36
37      /* Metodo de clase: Indica cuantas instancias estisteb
38       * de la clase. No puede llamarse desde una instancia */
39      static void Cuantos( );
40
41  };
42
43  #endif /* __SIMPLE_HPP__ */
```

codes/simple.hpp

```cpp
#include <iostream>

#include "simple.hpp"

/* El atributo de clase debe ser inicializado como si fuera
 * una variable global. */

int simple::Numero = 0;

/* Constructor sin argumentos (defecto) */
simple::simple( )
{
    /* Cada vez que se llame este constructor se mostrara este mensaje. */
    std::cout<< "Soy el constructor SIN argumentos" << std::endl;
    /* Se asigna un valor por defecto al atributo Frase */
    Frase = "No hay";
    /* Se incrementa el valor del atributo de clase. */
    Numero++;
}

/* Constructor con argumentos */
simple::simple( std::string La_Frase )
{
    /* Cada vez que se llame este constructor se mostrara este mensaje. */
    std::cout<< "Soy el constructor CON argumento" << std::endl;
    /* Se asigna al atributo Frase el valor del argumento */
    Frase = La_Frase;
    /* Se incrementa el valor del atributo de clase. */
    Numero++;
}
```

codes/simple.cpp

```
31
32 /* Destructor */
33 simple::~simple()
34 {
35     /* Cada vez que se llame este destructor se mostrara este mensaje. */
36     std::cout<< "Soy el destructor" << std::endl;
37     /* Se decrementa el valor del atributo de clase. */
38     Numero--;
39 }
40
41 /* Implementacion del metodo Decir_Frase: Se llama a traves de las
42  * instancias de la clase */
43 void simple::Decir_Frase()
44 {
45     std::cout << "La frase es: " << Frase << std::endl;
46 }
47
48 /* Metodo de clase: indica cuantas instancias van de la clase */
49 void simple::Cuantos()
50 {
51     std::cout << "Van " << Numero << " instancias de simple" << std::endl;
52 }
```

codes/simple.cpp

```cpp
#include <cstdlib>
#include <iostream>

#include "simple.hpp"

using namespace std;

/* Inicio del programa */
int main()
{
    /* Se crean dos instancias de la clase simple. La
     * primera sin argumentos, el constructor por defecto
     * es llamando. */
    simple primero;
    /* Para la segunda instancia se utilizan argumentos, el
     * constructor con argumentos es llamado. */
    simple segundo("Hoy no es viernes");

    /* Se llama el metodo de clase simple::Cuantos() que
     * nos indica cuantas instancias van de la clase.
     * Notese que su llamada es independiente de las instancias. */
    simple::Cuantos();

    /* Se crea un puntero a un objeto de tipo simple. Esto
     * no llama al constructor ya que lo que se esta creando
     * es un puntero, no una instancia de clase. */
    simple* tercero = 0;

    /* Se verifica cuantas instancias van. */
    simple::Cuantos();
```

codes/class01.cpp

```
31
32      /* Se crea una instancia en memoria de la clase simple
33       * utilizando new. La ubicacion de esta clase se almacena
34       * en el puntero creado anteriormente. Notese que se
35       * llama la forma con argumentos. */
36      tercero = new simple("Voy de ultimo");
37
38      /* Se verifica cuantas instancias van. */
39      simple::Cuantos();
40
41      /* Se llaman los metodos de los objetos. Estos deben
42       * ser llamados desde una instancia e interactuan
43       * con los atributos de cada instancia. */
44      primero.Decir_Frase();
45      segundo.Decir_Frase();
46      tercero->Decir_Frase();
47
48      /* Se libera el espacio de memoria creado por new.
49       * El destructor es llamado. */
50      delete tercero;
51
52      /* Se verifica cuantas instancias van. */
53      simple::Cuantos();
54
55      /* Al finalizar la funcion con el comando return
56       * las instancias que aun queden deben ser eliminadas.
57       * Los destructores deben ser llamados para cada
58       * instancia. */
59      return 0;
60  }
```

codes/class01.cpp

```
1  Soy el constructor SIN argumentos
2  Soy el constructor CON argumento
3  Van 2 instancias de simple
4  Van 2 instancias de simple
5  Soy el constructor CON argumento
6  Van 3 instancias de simple
7  La frase es: No hay
8  La frase es: Hoy no es viernes
9  La frase es: Voy de ultimo
10 Soy el destructor
11 Van 2 instancias de simple
12 Soy el destructor
13 Soy el destructor
```

codes/class01.out

Para compilar este ejemlo en linux con gcc el comando es:

```
1 g++ simple.cpp class01.cpp
```

y luego correr el `a.out`

Tarea 1.II.1

1. Modificar la clase `simple` del ejemplo añadiendo un atributo entero llamado ID.
2. Hacer que este nuevo atributo ID sea igual al valor de Numero al momento de crear una nueva instancia de la clase.
3. Agregar un método que muestre el ID de cada instancia de la clase.
4. Modificar el código en class01.cpp para utilizar el nuevo método agregado.

Clase persona

- Los archivos `persona.hpp`, `persona.cpp` y `class02.cpp` contienen otro ejemplo de definición, implementación y uso de una clase.
- Esta clase es un ejemplo de composición, ya que los atributos son instancias de otras clases (`string` y `vector`). También se utiliza la estructura (`struct`) `tm` de C para almacenar una fecha.
- En esta clase se muestra como usar punteros para los atributos del objeto y como esto puede ser usado para saber si estos campos ya fueron definidos.

Clase persona (*continuación*)

- En el caso de la clase `vector` esta es un *template* de un contenedor. Esta se utiliza para almacenar punteros a la misma clase `persona`.
- El atributo de clase `instancias` almacena las direcciones de memoria de cada instancia que se crea (ver constructores). Estas direcciones de memoria se obtiene por medio del puntero `this` que apunta a la instancia misma.
- Luego, el atributo de clase `instancias` es usado para llamar a todas las instancias y que estas digan su nombre (ver método de clase `LLamar_Todos()`).

Tarea 1.II.2

1. Modificar la clase `persona` añadiendo los siguiente métodos de asignación (*set*).

```
int Asignar_Nombre(std::sting elNombre);
int Asignar_Fecha_Nacimiento(int Anio, int Mes, int Dia);
int Asignar_Lugar_Origen(std::string Lugar);
```

Utilizar el método `Asignar_DPI` como ejemplo.

Tarea 1.II.2 (*continuación*)

2. Modificar la clase `persona` añadiendo los siguiente métodos de obtención (*get*).

```
std::string Obtener_Nombre();
int Obtener_Anio_Nacimiento();
int Obtener_Mes_Nacimiento();
int Obtener_Dia_Nacimiento();
std::string Obtener_Lugar_Origen();
```

Utilizar el método `Obtener_DPI` como ejemplo.

Clase VecR2

- Los archivos `VecR2.hpp`, `VecR2.cpp` y `class03.cpp` contienen otro ejemplo de definición, implementación y uso de una clase: Vectores en R^2.
- El propósito principal de este ejemplo es mostrar la *sobrecarga de operadores*, es decir, como definir la forma en que actúan algunas funciones especiales llamadas operadores sobre las clases que se están implementado.
- Por ejemplo: El operador + internamente en C++ es una función, de forma que al escribir `a+b` esto se transforma en `a.operator+(b)`, que debe retornar lo que se defina como la suma de `a` y `b`.

Clase VecIntR2 (*continuación*)

- También se muestra como sobrecargar el operador de salida <<. Este es un caso especial, ya que lo que retorna es la forma en que se desea desplegar la clase. Este operador no es parte de la clase, sino que debe ser declarado como *amigo* para poder acceder a los atributos privados de esta.
- Este ejemplo muestra como utilizar un atributo de clase como un *flag*. Es este caso, este flag se utiliza para indicar si la forma de desplegar el vector se hará en forma polar o cartesiana.

Tarea 1.II.3

Utilizando la clase `VecR2` como ejemplo, definir e implementar una nueva clase llamada `VecR3`, que son vectores en R^3. Esta clase debe sobrecargar los siguientes operadores:

1. `operator+` para suma de dos vectores.
2. `operator-` para resta de dos vectores.
3. `operator-` para *negación* de un vector.
4. `operator*` para producto punto de dos vectores.
5. `operator*` para multiplicar un vector por un flotante.
6. `operator/` para dividir un vector por un flotante.
7. `operator%` para producto cruz de dos vectores.

Tarea 1.II.3 (*continuación*)

7. `operator=` para asignación.
8. `operator==` para compraración de igualdad entre vectores.

También se debe sobrecargar las siguientes funciones como *amigas*

9. `operator*` para multiplicar un flotante por un vector.
10. `operator<<` para desplegar el vector con `cout`.

De la misma forma que en `VecR2`, se debe implementar un atributo estático que indique si al desplegar el vector, lo haga en forma cartesiana (x, y, z) o esférica (r, θ, ϕ).

STL y Boost

C++ Templates

En C++ un *Template* es una forma especial de clase que puede adaptarse al tipo de dato que se desee. Este tipo de dato se indica al momento de llamar el constructor de la clase.

Template `tsimple`

- En los archivos `tsimple.hpp` y `template.cpp` se muestra un ejemplo de definición, implementación y uso del template `tsimple`.
- Este template es un ejemplo de un *contenedor*, una clase que se utiliza para almacenar en forma de atributos, algún tipo de variable *que se define* al momento de crear la instancia.
- `tsimple` es un contenedor elemental ya que solo contiene un atributo.

Template `tsimple` (*continuación*)

- La forma de llamar un template es la siguiente:

```
tsimple< T > var_name;
```

- Esto crea una instancia de `tsiple` para el tipo de variable/clase `T`, por ejemplo:

```
tsimple<int> itype;
```

crea una instancia *del tipo entero*, y todos los métodos del objeto se adecuan para este tipo de dato.

Template `tsimple` (*continuación*)

- El método `Asignar_Valor` su forma general es:

```
int tsimple< T >::Asignar_Valor( T valor);
```

- Cuando se crea la instancia indicando que el tipo `T` es `int` esto se convierte en:

```
int tsimple<int >::Asignar_Valor( int valor);
```

es decir, en método `Asignar_Valor` espera un argumento del tipo entero. Sucede de la misma forma para los demás métodos.

STL - Standard Template Library

El STL o *Standard Template Library* es un conjunto de librerías de templates para uso general en C++. Se compone en forma general de los siguientes elementos:

- Contenedores.
- Iteradores.
- Algorítmos.
- Funciones.

Contenedores

Los contenedores son estructuras de datos que implementan, según sea el caso, diferentes enfoques de organización del almacenamiento en memoria y de funcionalidad de inserción y extracción de estos.

En el ejemplo de implementación de la clase `persona`, se utilizó el contenedor `vector` el cual es un template que funciona como una extensión de los *arreglos* tradicionales de C. Utiliza espacio continuo de memora para almacenar los datos pero implementa métodos para insertar y remover elementos de forma dinámica de este.

Iteradores

Los iteradores son elementos que actúan sobre los contenedores para acceder de forma más eficiente a estos. Se podrían ver como la evolución de la aritmética de punteros. Existen de varios tipos, según la forma en que se desee recorrer o acceder a los elementos de contenedor.

Algorítmos

Los algoritmos son procedimientos implementados para actuar sobre contenedores y arreglos. El objetivo es proveer mecanismos altamente eficientes para labores de uso general. Entre los algoritmos se tienen:

- Ordenamiento.
- Búsqueda.
- División.

Funciones

Las funciones, o mas exactamente, *funciones-objeto* (*functors* en inglés) son un tipo especial de clases que definen una función. Estas funciones-objeto son parametrizadas al momento instanciar la clase de manera que su comportamiento es diferente según la instancia. El objetivo es que la instancia de estas funciones-objeto solo tenga que recibir un parámetro como argumento.

Boost

Boost (`bost.org`) es un conjunto de librerías que no pertenecen a la implementación estándar de C++, sin embargo, han ganado una gran popularidad y son ampliamente utilizadas. Han sido contribuidas, implementadas y verificadas de forma comunitaria. Se podría decir que Boost es la materialización de uno de los objetivos de OOP, *reutilización de codigo*.
Las librerías que se pueden encontrar en Boost son trememdamente variadas, desde contenedores circulares hasta manejo de archivos de configuración.

Herencia

La *herencia* (*inheritance*) es la otra forma básica de relacionar clases. Como se mencionó, se utiliza cuando una nueva clase *es* una extensión o restricción de una clase previa.
La clase *derivada* hereda los atributos y los métodos de la clase *base*. En esta se pueden ya sea extender la clase base añadiendo nuevos atributos y/o métodos, sobrecargar los métodos de la clase base para modificar su comportamiento, restringir la clase base llamando por defecto el constructor de esta con parámetros específicos, etc.
Sin embargo, a pesar de heredar los atributos y los métodos, las clases derivadas no pueden acceder a las partes etiquetadas como privadas de la clase base, solo puede acceder a las publicas y las protegidas.

Ejemplo: clase estudiante

- En los archivos `estudiante.hpp`, `estudiante.cpp` e `inheritance.cpp` se muestra como definir una nueva clase llamada estudiante, la cual es derivada de la clase base persona.
- La relación de herencia se determina a partir de que un *estudiante* **es una** *persona*.
- La clase hace una herencia con acceso *público* de la clase base. Esto significa que lo que haya sido etiquetado como público en la clase persona, será publico en la clase estudiante.

Ejemplo: clase estudiante (*continuación*)

- En este ejemplo se añaden los siguientes atributos a la clase base:

```
std::string* Registro_Academico;
std::tm*     Fecha_inscripcion;
std::string* Carrera;
```

- También se añaden los métodos:

```
int Asignar_Registro_Academico( std::string Num_RegAcad );
std::string Obtener_Registro_Academico( );
void Decir_Registro_Academico();
```

Ejemplo: clase estudiante (*continuación*)

- Se sobrecarga el método `Decir_Nombre()` de la clase base.
- Notar como la sobre carga del método `Decir_Nombre()` es llamada desde instancias de la clase `estudante`, pero al ser llamada desde el método de clase `LLamar_Todos()` se utiliza la versión no sobrecargada, es decir, la de la clase base `persona`.

Es posible derivar clases desde mas de una clase base, esto es una *herencia múltiple*, así como varias formas de heredar los métodos aparte de la herencia pública.