

Apunte de clase C++

Diferencias entre C y C++

Comentario de una sola línea

C++ introduce la posibilidad de realizar comentarios de una sola línea a partir de una doble barra de división "//"

```
int a, b;
// Asignamos el valor de a
a = 10;
b = 5; // A partir de aquí es comentario hasta el final de la línea
//float c = 2.33f;
```

Sobrecarga de funciones

En C++ es posible crear varias funciones con el mismo nombre siempre y cuando el conjunto de argumentos que recibe cada una sea diferente de las demás. De esta manera podemos pensar que una función tiene diferentes implementaciones según los parámetros que se le pasan. Aunque lo que en realidad ocurre es que la lista de parámetros forma parte del nombre de la función; el compilador y *linker* ven, en el código que mostramos a continuación, la declaración de dos funciones con distinto nombre:

```
int comparar(int a, int b);
int comparar(char * cadena1, char * cadena2);
```

Cada implementación de `comparar()` evaluará la información según corresponda y retornará un resultado adecuado. Esta característica tiene sentido para el programador siempre que todos los "sabores" de la función `comparar()` cumplan con un propósito similar y sólo varíen en su código debido a que reciben entradas distintas. En el ejemplo anterior esto significa que las dos implementaciones deberán comparar las entradas. Las diferencias serán debidas a que una recibe enteros y la otra recibe cadenas de caracteres. Pero las dos harán una comparación y retornarán el resultado de la misma.

Declaración de variables en el código

Con C++ es posible mezclar declaraciones de variables con código. El siguiente ejemplo es válido en C++ pero no compilaría en C:

```
int main(int argc, char ** argv){
```

75.42 – Taller de Programación I

```

int a, b = 0;

a = b + 5; // Escribo código
char c = 'X'; // Declaro y defino c

return 0;
}

```

Después del código que asigna $b + 5$ en a , estamos declarando la existencia de una variable de tipo `char` llamada `c`.

Se recomienda hacer uso de esta característica con cuidado ya que puede provocar desorden en el código.

Pasaje de variables por referencia

Cada vez que llamamos una función en C, los argumentos que pasamos llegan a ella como copias de los valores que tenemos en el código que realiza la llamada. Por ejemplo, en el código siguiente `x` es una copia del valor de `n`. Cualquier cambio a `x` no produce ningún efecto en `n`.

```

int valor(int x) {

    x+=10;
    return x+5;
}

int main(.....) {
...
    int n = 1;

    valor(n);
    printf("n: %d\n", n);
...
}

```

Muestra "n: 1" porque dentro de `valor` se le sumó diez a `x` que es una variable diferente con que se inicializó al mismo valor de `n` en la llamada dentro de `main()`

Luego, si cambiamos la función `valor()` de la siguiente manera el valor de `n` se verá modificado después de la llamada. Esto sólo es posible en C++. Lo que hacemos es indicarle al compilador que `valor()` recibe una referencia a un entero que existe en otra parte del código. Una sola variable está en juego y es `n`. Dentro de la función `x` es otra forma de llamar a la variable `n`.

```

int valor(int &x) {

    x+=10;
    return x+5;
}

```

```
int main(.....) {
...
    int n = 1;

    valor(n);
    cout << "n: " << n << endl; // Utilizo iostream en lugar de stdio
...
}
```

Muestra "n: 11" porque valor recibió la referencia a n le sumó diez. A diferencia del caso anterior, x no es una variable diferente de n sino una referencia a ésta.

Entrada y salida estándar

En el ejemplo anterior agregamos una línea de código que puede resultar extraña a un programador de C. Para imprimir el mensaje de salida del programa se utilizó **iostream**.

En C++ disponemos de un *header* llamado **iostream** que contiene las declaraciones de las clases y objetos para manejar la entrada y salida estándar.

A continuación un ejemplo de su utilización:

```
#include <iostream>

using namespace std;

int main(int argc, char ** argv) {

    int n = 0;

    cout << "Ingrese un número entero: ";
    cin >> n;
    cout << "#: " << n << endl;

    return 0;
}
```

El operador << de **cout** permite enviarle objetos (integers, floats, strings, etcétera) a la salida estándar. **cout** se encarga de darles formato adecuado. Por otra parte **cin** (que representa la entrada) permite leer datos con el operador >>.

En caso de necesitar manipular el formato de salida, cout se puede configurar mediante llamadas a funciones como muestra el ejemplo:

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char ** argv) {

    float n = 0.0f;
```

75.42 – Taller de Programación I

```

cout << "Ingrese un número decimal: ";
cin >> n;

cout << "# n.n: " << n << endl;
cout << "# n.nn: " << setiosflags(ios::fixed) << setprecision(2) << n <<
endl;

return 0;
}

```

Primero se imprime el valor de `n` con el formato por defecto. Luego se especifica que se necesitan sólo dos dígitos decimales.

Clases

Por último en esta lista de diferencias, pero de ninguna manera menos importante, tenemos las clases. Para llamarlo de una forma más abarcativa podemos decir que C++ introduce las características que le permiten soportar la filosofía de Programación Orientada a Objetos ([más información en Wikipedia](#)).

C++ nos da la posibilidad de definir clases y luego crear objetos de las clases que hemos definido. Vamos a ver un ejemplo en el que se declara una clase `Persona` y se definen sus métodos (o funciones miembro).

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. class Persona {
6. private:
7.     char nombre[20];
8.     char sexo;
9.     int edad;
10.    static unsigned int pc;
11. public:
12.    Persona(const char * nombre, char sexo, int edad);
13.    virtual ~Persona();
14.    const char * getNombre() { return nombre; }
15.    char getSexo() { return sexo; }
16.    int getEdad() { return edad; }
17. };
18.
19.
20. int main(int argc, char ** argv)
21. {
22.     Persona * p = new Persona("Genius", 'm', 24);
23.
24.     const char * n = p->getNombre();
25.
26.     cout << n << ", " << p->getSexo() << ", " << p->getEdad() <<
endl;
27.

```

```

28.         delete p;
29.
30.         return 0;
31.     }
32.
33.     Persona::Persona(const char * nombre, char sexo, int edad){
34.         strncpy(this->nombre, nombre, sizeof(this->nombre)-1);
35.         this->sexo = sexo;
36.         this->edad = edad;
37.
38.         cout << "+Persona" << endl;
39.         cout << ".Hay " << ++pc << " personas." << endl;
40.     }
41.
42.     Persona::~~Persona() {
43.         cout << "-Persona" << endl;
44.         cout << ".Hay " << --pc << " personas." << endl;
45.     }
46.
47.     unsigned int Persona::pc;
48.

```

Las clases, a diferencia de las estructuras, permiten que definamos su comportamiento a través de métodos o funciones miembro. Una vez declarados y definidos los métodos todos los objetos existentes de una clase particular tendrán la capacidad de responder los mensajes relacionados con ellos. En el ejemplo anterior: todos los objetos de la clase Persona tienen un método `getNombre()` que devuelve una cadena de caracteres con el nombre de la persona.

Lo que las clases sí comparten con las estructuras es la posibilidad de tener **estado**. Y resaltamos la palabra estado porque es un concepto fundamental de la filosofía de programación orientada a objetos. Éstos deben tener estado ya que la identidad de un objeto —aquello que lo diferencia de todas las demás instancias— es el estado. Refiriéndonos nuevamente al ejemplo dado: el nombre, la edad y el sexo serán los atributos que diferenciarán a una persona de otra.

El código expuesto anteriormente contiene mucha más información que la mencionada hasta el momento. En la próxima sección veremos en detalle cómo se declara y define una clase.

Clases

Atributos

Las clases deben permitir construir objetos con estado. Esto se logra en C++ a través de la declaración de atributos (variables miembro). En el ejemplo de la sección anterior definimos la clase persona con tres atributos que representan el estado o identidad de cada objeto. Revisamos el ejemplo en este extracto de código simplificado:

```

1. class Persona {
2.     char nombre[20];
3.     char sexo;
4.     int edad;
5.     static unsigned int pc;
6. };

```



Los atributos son nombre, sexo y edad —vamos a ignorar la declaración del entero pc por el momento— y los tipos de cada uno son cadena de 20 caracteres, un caracter y un entero respectivamente.

De ahora en adelante cada objeto construido del tipo Persona tendrá espacio disponible en memoria para cada uno de sus atributos. El nombre de un hipotético objeto **pers1** será un espacio de memoria distinto e independiente de otro **pers2**.

Métodos

Los objetos tiene comportamiento. Es decir que se puede interactuar con ellos solicitándoles que realicen actividades determinadas. Para que sea posible es necesario que la clase a la que pertenecen tenga métodos (funciones miembro) definidos. C++ nos permite declarar y definir métodos en una clase. Luego, cada vez que se ejecute un método en un objeto de la clase, se correrá el código definido para el método en cuestión. Dado que el tramo de código de un método es único para la clase, existe una variable implícita llamada **this** que apunta al objeto específico.

Veamos la declaración de persona restringida a las líneas de código pertinentes:

```

1. class Persona {
2.     char nombre[20];
3.     char sexo;
4.     int edad;
5.     ...
6.     const char * getNombre() { return nombre; }
7.     char getSexo() { return sexo; }
8.     int getEdad() { return edad; }
9. };

```

Si tomamos como referencia el método `getSexo()` observamos que se declara y define en una sola línea. Lo único que hace es devolver el valor del atributo `sexo`. Obviamente el valor devuelto dependerá del valor que el atributo tenga para el objeto cuyo método se haya llamado. La misma línea de código podría escribirse como: `"return this->sexo;"` dejando explícitamente marcado que se refiere al atributo texto del objeto actual. Pero esto es redundante porque se asume que las llamadas a métodos y referencias a atributos dentro de una función miembro se refieren al objeto actual. El uso de **this** queda restringido a problemas de ambigüedad de nombres y preferencias de los programadores a la hora de hacer semánticas ciertas líneas de código.

Declarar y definir el método en un solo paso es una opción y tiene una consecuencia que se verá más adelante. El mismo método `getSexo()` podría haberse declarado en un archivo *header* (.h) y definido o implementado en un archivo de código fuente (.cpp)

```

1. class Persona {
2.     char nombre[20];
3.     char sexo;
4.     int edad;
5.     ...
6.     const char * getNombre() { return nombre; }
7.     char getSexo();
8.     int getEdad() { return edad; }
9. };

1. char Persona::getSexo() {

```

```

2.     return sexo;
3. }

```

Constructor y Destructor

Existen dos métodos especiales en las clases de C++. El constructor, cuyo código se ejecuta cada vez que un objeto de la clase es creado; y el destructor que se ejecuta cuando el objeto se destruye. Ambos comparten la característica de que su nombre es idéntico al nombre de la clase a la que pertenecen y no llevan tipo de retorno. El destructor lleva el carácter "~" delante de su nombre.

```

1. class Persona {
2. private:
3.     char nombre[20];
4.     char sexo;
5.     int edad;
6.     static unsigned int pc;
7. public:
8.     Persona(const char * nombre, char sexo, int edad);
9.     virtual ~Persona();
10.    const char * getNombre() { return nombre; }
11.    char getSexo() { return sexo; }
12.    int getEdad() { return edad; }
13. };
14.

```

Persona tiene un constructor que recibe valores para inicializar sus tres miembros. El destructor es siempre único y no recibe ni retorna valores, sólo es llamado para que el objeto pueda liberar recursos que haya solicitado durante su ejecución o en la construcción. Típicamente el recurso es memoria dinámica, pero no está necesariamente limitado a esto.

Si en nuestra clase hubiéramos omitido declarar un constructor aún existiría uno por defecto —creado por el compilador— que simplemente construiría los objetos miembros utilizando constructores por defecto. El constructor creado por el compilador no recibe parámetros.

Es posible que el programador decida implementar el constructor por defecto. Por ejemplo:

```

1. class Persona {
2. private:
3.     char nombre[20];
4.     char sexo;
5.     int edad;
6.     static unsigned int pc;
7. public:
8.     Persona() { nombre[0] = 0; sexo = 'M'; edad = 0; };
9.     virtual ~Persona();
10.    const char * getNombre() { return nombre; }
11.    char getSexo() { return sexo; }
12.    int getEdad() { return edad; }
13. };
14.

```

Siempre que exista un constructor declarado por el programador, sea o no el constructor por defecto, el compilador no agregará un constructor. Por lo tanto la clase Persona tal y como está definida en el ejemplo original sólo tiene un constructor que recibe tres parámetros. No tiene un constructor por defecto.

Operadores new y delete

Hasta ahora vimos cómo declarar una clase junto con su estado y comportamiento. Pero la finalidad de esto es poder crear objetos de dicha clase, darles valores y enviarles los mensajes a través de sus métodos.

En C++ tenemos dos formas de crear un objeto: podemos declararlo dentro de una porción de código (en una función o método): se comportará como cualquier variable automática; existirá dentro del segmento de código (scope) y se destruirá cuando finalice el mismo. O podemos crearlo utilizando memoria dinámica (heap) para lo cual se incluyeron los nuevos operadores new y delete que tomarán y liberarán, respectivamente, memoria dinámica.

En este ejemplo se crea un objeto del tipo Persona en la línea tres que existirá hasta la línea seis. En ese punto el objeto será destruido y se ejecutará el destructor que se haya declarado para la clase.

```

1. ...
2. {
3.     Persona empleado;
4.
5.     printf("Empleado sexo: %c\n", empleado.getSexo());
6. }
```

Por otra parte, en el siguiente ejemplo la variable empleado es un puntero a un objeto del tipo Persona. En la línea cuatro se crea el objeto utilizando su constructor por defecto y solicitando memoria dinámica con el operador new. En la línea ocho se destruye el objeto en forma explícita y esto es obligatorio cuando se solicita memoria dinámica con new porque de no hacerse se pierde la memoria alocada. En la línea nueve el puntero empleado ya no es válido.

```

1. ...
2. {
3.     int a = 0;
4.     Persona * empleado = new Persona();
5.
6.     printf("Empleado sexo: %c\n", empleado->getSexo());
7.
8.     delete empleado;
9.
10.    return a;
11. }
```

Palabras reservadas public, protected y private

El acceso a los miembros de una clase está restringido según la filosofía orientada a objetos para que exista un encapsulamiento de la implementación de la clase y que sólo se exponga al usuario de la misma la interfaz. En C++ esto se indica agrupando los miembros a continuación de una de las palabras reservadas de acceso.

Los miembros declarados después de la palabra public serán accesibles por todas las

clases y funciones de la aplicación. En cambio aquellos declarados después de la palabra `private` sólo podrán ser accedidos por la misma clase. En los ejemplos anteriores los tres miembros `nombre`, `edad` y `sexo` sólo pueden accederse desde la propia clase y por ello existen tres métodos públicos que retornan el valor de cada atributo.

Finalmente los miembros declarados como `protected` serán accesibles por la propia clase y por aquellas que hereden de la misma. Pero será privados para el resto de las clases.

```

1. class Object {
2. public:
3.     Object();
4.     Object(const char * name);
5.     const char * getName() const { return name; };
6.     void setName(const char * n);
7.     virtual ~Object();
8.
9. protected:
10.    int getInternalID() { return id; };
11.
12. private:
13.    int id;
14.    char * name;
15.    void nameToUpper();
16. };

```

En este ejemplo sólo las clases que hereden de `Object` podrán llamar al método `getInternalID()`. Sin embargo cualquier función o método de otra clase podrá crear objetos, obtener y definir el atributo `nombre` a través de los métodos `getName()` y `setName()` respectivamente. Dejamos la explicación de la herencia para más adelante.

Métodos de clase o static

Un método de una clase puede ser declarado `static`. Eso implica que el método no es miembro de un objeto sino que pertenece a la clase y por lo tanto puede ejecutarse incluso cuando no exista ningún objeto del tipo en cuestión. Los métodos miembro, por otra parte, necesitan que exista un objeto antes de poder ser ejecutados porque el contexto en el que se espera que tengan lugar es el objeto en cuestión. Cuando se llama `getSexo()` en un objeto de la clase `Persona` esperamos obtener el sexo de la persona representada por un objeto particular.

Los métodos `static` no están asociados a ningún objeto, por lo tanto no pueden acceder a ninguno de los atributos o métodos miembro. Podemos pensar en estos métodos como funciones de C que en lugar de estar sueltas en la aplicación se encuentran asociadas a una clase particular. Se utilizan en algunos patrones de diseño como `Singleton` y `Factory`.

```

1. #include <iostream>
2.
3. using namespace std;
4.
5. class Hijo {
6. public:
7.     Hijo(int n) : numero(n) {};
8.     inline int getNumero() const { return numero; };
9. private:
10.    int numero;
11. };
12.
13. class Madre {

```

75.42 – Taller de Programación I

```

14. private:
15.     Madre() : cc(0) {};
16.     int cc;
17.     static Madre * laMadre;
18. public:
19.     static Madre * madreHayUnaSola();
20.     Hijo * nuevoHijo();
21. };
22.
23. Madre * Madre::laMadre = NULL;
24.
25. Madre * Madre::madreHayUnaSola() {
26.     if (laMadre == NULL)
27.         laMadre = new Madre();
28.
29.     return laMadre;
30. }
31.
32. Hijo * Madre::nuevoHijo() {
33.     return new Hijo(++cc);
34. }
35.
36. int main(int argc, char ** argv) {
37.
38.     Hijo * h = Madre::madreHayUnaSola()->nuevoHijo();
39.
40.     cout << "Hijo: " << h->getNumero() << endl;
41.
42.     delete h;
43.
44.     return 0;
45. }

```

Implementación simple del patrón Singleton. Se utiliza un método miembro llamado `madreHayUnaSola()` que es llamado en la línea treinta y ocho para obtener el único objeto `Madre` de la aplicación. Como podemos ver en esa línea el método es llamado sobre la clase, no sobre un objeto. Adicionalmente el atributo `laMadre` es también declarado `static` lo que hace que exista sin necesidad de un objeto de la clase. La memoria de las variables estáticas se provista en el inicio de la ejecución en forma transparente para el programador. Pero debe indicarse su existencia en un archivo compilable (.cpp) como se muestra en la línea veintitrés.

Métodos const

Los métodos declarados como constantes aseguran que dentro del código de los mismos no existirá modificación del estado del objeto. Esto es, no se cambiarán los valores de sus atributos.

En la línea ocho del ejemplo de Singleton podemos ver que el método `getNumero()` de la clase `Hijo` está declarado como `const`. Con esa restricción, un código que modifique el estado del objeto no podría compilar:

```

1. class Hijo {
2. public:
3.     Hijo(int n) : numero(n) {};
4.     int getNumero() const { return numero++; };
5. private:

```



```
6.         int numero;  
7.    };
```

No está permitido incrementar el atributo `numero` dentro del método `getNumero()` porque se indicó que es un método constante y puede ser llamado asumiendo que no variará el estado del objeto tras su ejecución.

Cuando se pasa un objeto a un método y éste lo recibe como `const` sólo podrá llamar métodos declarados `const` ya que en este caso el compromiso es del método receptor en no modificar el estado del objeto recibido.

Estructuras vs. clases

- Las clases tienen métodos.
- En C++ una estructura es una clase con todos sus miembros públicos (y puedo agregarle métodos).
- Se puede modelar objetos en C con funciones que reciban siempre una estructura que represente al objeto.
- Las estructuras se pueden guardar fácilmente en un archivo a diferencia de las clases que deben implementar en cada caso particular la manera de guardar su estado.

Métodos inline

- Cuando el compilador encuentra una llamada a un método inline evalúa la posibilidad y decide si en lugar de realizar una llamada a función, escribe el código del método en el lugar desde dónde se llamaría.
- Los métodos definidos en la declaración de la clase son inline aunque no se escriba la palabra reservada.
- En todos los casos se trata de una sugerencia. El compilador tiene la última palabra y decide si hace un método inline independientemente de que el programador lo proponga o no.