# 3. Clases I. Generalidades

Julio Vega

julio.vega@urjc.es


# Contenidos

```
#ifndef TIME_H
#define TIME_H

class Time {
public:
   Time(); // constructor
   void setTime (int, int, int); // set hour, minute and second
   void printUniversal () const; // print time in univ. format
   void printStandard () const; // print time in stand. format
private:
   unsigned int hour; // 0 - 23 (24-hour clock format)
   unsigned int minute; // 0 - 59
   unsigned int second; // 0 - 59
}; // end class Time

#endif
```

- El nombre de una clase (= fichero) siempre con mayúscula inicial.
- Uso de las directivas del preprocesador: `#ifndef`, `#define` y `#endif`.
  - Es la denominada envoltura del preprocesador.
  - Evita incluir los archivos de encabezado más de una vez por programa.
- Uso del archivo de encabezado en mayúsculas.
  - Sustituyendo el punto por un guión bajo.
- Por claridad, usa el especificador de acceso solo una vez.
  - Por legibilidad, los miembros `public` primero, fáciles de localizar.
  - Por seguridad, los atributos de la clase deben ser `private`.
    - A menos que se demuestre la necesidad de que sea `public`.

```
Time::Time() {} // constructor (special method)

void Time::setTime (int h, int m, int s) { // set new Time value
   if ((h >= 0 && h < 24) && (m >= 0 && m < 60) &&
      (s >= 0 && s < 60)) { // validate hour, minute and second
      hour = h; minute = m; second = s;
   } else throw invalid_argument ("H, M or S was out of range" );
} // end function setTime

void Time::printUniversal() const { // print Time in (HH:MM:SS)
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

void Time::printStandard() const { // print Time in AM or PM
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12) << ":"
      << setfill('0') << setw(2) << minute << ":" << setw(2)
      << second << (hour < 12 ? " AM" : " PM");
} // end function printStandard
```

- `setfill`: manipulador de flujo parametrizado.
  - Carácter de relleno al imprimir un entero en un campo más ancho.
    - Ejemplo: si el valor de minuto es 2, se mostrará como 02.
  - *Función pegajosa*: tal relleno se aplicará ya en todos los valores.
- `setw`: indica el número de digitos que se mostrarán. No es pegajosa.
- Cuando no se use una función pegajosa hay que restaurarla.
  - Ya lo veremos más adelante, en el tema de entrada/salida de flujos.
- `?:` es un operador condicional (*si* $A \Longrightarrow B$, *sino* $C$).

- Las definiciones de clase y sus métodos pueden estar en un fichero.
- Pero lo ideal es separar ambos códigos en dos archivos.
  - La definición de la clase en el archivo de encabezado (.h, *header*).
  - La definición de sus miembros en el archivo de código fuente (.cpp).
- Esta separación facilita la modificación de los programas.
  - Modificar la implementación de una clase no afecta a los clientes.
  - Siempre que se respete el interfaz definido en el *header*.
- El .h incluye comentarios para cliente; .cpp, para desarrollador.
- Los *Independent Sofware Vendor* o ISVs (software privativo).
  - Proporcionan solo las bibliotecas de clase para su venta.
    - Por contra está el movimiento *open source*.
  - Pero un cliente necesita poder enlazarse al código objeto de la clase.
  - Por ello se entregan los .h y los .o (pero no los .cpp).
  - Ya veremos (Cap. 3) que un .h debe incluir las funciones `inline`.
  - Pero hasta los atributos `private` pueden ocultarse (clase *proxy*).

- Los métodos y atributos pertenecen al alcance de la clase.
- Cualquier miembro de clase puede usar otro miembro de clase.
- Solo los miembros `public` pueden ser accedidos desde fuera.
  - Mediante el nombre, la referencia o el apuntador a un objeto.
- Los métodos se pueden sobrecargar por otros métodos.
  - Con el mismo identificador pero distintos parámetros.
    - `int getName (int index);` $\neq$ `int getName (float index);`
- Las variables declaradas en un método solo son visibles en este.
  - ¿Y si se declara una variable con el mismo nombre que un atributo?
    - El atributo se *oculta*, y para acceder a él hay que usar `::`

```
int main() {
   Time t; // instantiate object t of class Time

   cout << "The initial universal time is ";
   t.printUniversal(); // 00:00:00
   cout << "\nThe initial standard time is ";
   t.printStandard(); // 12:00:00 AM
   t.setTime( 13, 27, 6 ); // change time
   cout << "\n\nUniversal time after setTime is ";
   t.printUniversal(); // 13:27:06
   cout << "\nStandard time after setTime is ";
   t.printStandard(); // 1:27:06 PM
   // attempt to set the time with invalid values
   try {
      t.setTime( 99, 99, 99 ); // all values out of range
   } catch ( invalid_argument &e ) {
      cout << "\n\nException: " << e.what() << endl;
   } // end catch
}
```

- Una vez definida la clase se puede usar como cualquier tipo de datos.
  - Es como un `typedef struct` pero enriquecido con métodos.
- Lo primero es *instanciar* un objeto de la clase: `Time t;`
  - La clase es el *molde* del que salen copias/instancias.
- Lo siguiente ya es usar esa instancia `t`.
- Podremos invocar a todos sus miembros `public` mediante el `.`
  - Ya veremos más detenidamente esto de la visibilidad...
  - El operador `.` se usa tras una referencia a objeto.
    - Mientras que el operador `->` se usa tras un puntero a objeto.

## Creación de la clase Count

```
class Count {
public: // public data is dangerous
  void setX (int value) {
    x = value;
  } // end function setX

  void print() {
    cout << x << endl;
  } // end function print

private:
   int x;
}; // end class Count
```

## Usos de la clase Count con diferentes manejadores

```
int main() {
  Count counter; // create counter object
  Count *counterPtr = &counter; // create pointer to counter
  Count &counterRef = counter; // create reference to counter

  cout << "Set x to 1 and print using the object's name: ";
  counter.setX (1); // set data member x to 1
  counter.print (); // call member function print

  cout << "Set x to 2 and print using a reference to an object:";
  counterRef.setX (2); // set data member x to 2
  counterRef.print (); // call member function print

  cout << "Set x to 3 and print using a pointer to an object: ";
  counterPtr->setX (3); // set data member x to 3
  counterPtr->print (); // call member function print
} // end main
```

- Constructor y destructor son métodos especiales de la clase.
- Su función es la de crear/destruir el objeto cuando sea oportuno.
  - Invocados implícitamente por el compilador: instanciar/fin de ámbito.
    - Cuando se crea la instancia del objeto se está invocando al constructor.
- El nombre de ambos métodos ha de ser igual al de la clase.
  - Así, el compilador sabe a qué método buscar cuando ha de invocarlos.
  - En el caso del destructor, precedido por la virgulilla ($\sim$).
    - No es casualidad: $\sim$ es el operador de complemento a nivel de bits.
    - Y el destructor es el complemento del constructor...
- Ya veremos el gran protagonismo de constructor y destructor en C++.

```
#ifndef TIME_H
#define TIME_H

class Time {
public:
  Time(); // (a) basic constructor
  Time (int = 0, int = 0, int = 0); // (b) with defaults
  // [...]
private:
  // [...]
}; // end class Time

#endif
```

```
Time::Time() {} // (a) with empty implementation, not recommended

Time::Time() : hour(0), minute(0), second(0) {} // (a) init to 0

Time::Time (int h, int m, int s) { // (b) with defaults
  setTime (h, m, s);
}
```

- Lo ideal es que el constructor inicialice los miembros de datos con 0.
  - Recuerda: al crear un objeto se hace una llamada al constructor.
  - Así pues, esto asegura que el objeto empieza en un estado consistente.
- Podemos definir varios constructores (sobrecargados) para una clase.
  - Ya veremos sobrecarga $\iff$ polisemia: misma palabra $\neq$ significados.
- Un constructor puede llamar a otras funciones miembro de la clase.
  - Pero cuidado; es el encargado de inicializar el objeto.
  - Si usa un atributo antes de inicializarlo $\implies$ error lógico.

```
int main() {
  Time t; // default values are set in all members (h=0,m=0,s=0)
  Time t2 (2); // h = 2, m = s = 0
  Time t3 (21, 34); // h = 21, m = 34, s = 0
  Time t4 (12, 25, 42); // values for h, m, s
  Time t5 (27, 74, 99); // wrong values (checked by setTime)
}
```

- Corolario: un constructor con *defaults* garantiza consistencia.
  - Tanto en caso de no poner valores como de ponerlos incorrectos.

- Un destructor es `public`, no recibe parámetros ni devuelve nada.
  - Si no se especifica explicitamente, el compilador crea uno vacío.
- $\nexists$ sobrecarga de destructores $\iff$ solo puede haber uno por clase.
- El destructor se invoca implícitamente cuando se destruye un objeto.
  - E.g. cuando se sale de un ámbito en el que se creó un objeto.
- El destructor no libera la memoria del objeto.
  - Prepara esa zona de memoria para ser reusada cuando se reclame.

```
#include <string>
using namespace std;

#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy {
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

```
#include <iostream>
#include "CreateAndDestroy.h" // include class definition
using namespace std;

CreateAndDestroy::CreateAndDestroy (int ID, string msg) {
   objectID = ID; // set object's ID number
   message = msg; // set object's descriptive message

   cout << "Object " << objectID << " constructor runs "
      << message << endl;
} // end CreateAndDestroy constructor

CreateAndDestroy::~CreateAndDestroy () { // destructor
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << " destructor runs "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

- El compilador llama implícitamente a constructores y destructores.
- El constructor de un obj. global se llama 1.º, antes incluso que `main`.
- Los destructores se invocan en el orden inverso a los constructores.
  - El último invocado será el de un obj. global (tras finalizar `main`).
- La ejecución *normal* del programa se puede interrumpir.
  - No es una buena práctica, porque no se ejecutarán los destructores.

```
#include <iostream>
#include "CreateAndDestroy.h" // include class definition
using namespace std;

void create (void); // prototype

CreateAndDestroy first (1, "(global before main)"); // global ob.

int main() {
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second (2, "(local automatic in main)");
   static CreateAndDestroy third (3, "(local static in main)");

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth (4, "(local automatic in main)");
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
} // end main
```

```
// function to create objects
void create (void) {
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth (5, "(local automatic in create)");
   static CreateAndDestroy sixth (6, "(local static in create)");
   CreateAndDestroy seventh (7, "(local automatic in create)");
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

- Funciones: `exit(int code)`, `atexit()`, `abort()`, `return code`
- La función `exit(code)` obliga al programa a terminar de inmediato.
  - Es la forma menos brusca de terminar.
    - Ejecuta el destructor de los objetos `static` y los globales.
    - Cierra los ficheros que pudieran estar abiertos en operación I/O.
    - Fuerza la escritura de datos que quedasen en los buffers de I/O (*flush*).
    - Elimina los ficheros temporales que se estuviesen usando.
    - `code = 0` (o `EXIT_SUCCESS`) o `code = 1` (o `EXIT_FAILURE` o $\neq 0$).
- `atexit()` permite customizar `exit()` para realizar acciones extra.
- `abort()` es muy brusco: no hace nada de lo que hace `exit()`.
  - Levanta una excepción `SIGABRT`, pudiéndose —eso sí— manejar esta.
    - En tal manejador nosotros podemos cerrar las *cosas*.
- `return` o `return code` termina la ejecución del ámbito actual.
  - Devolviendo el control al ámbito desde el que fue llamado.

- Una referencia a objeto es un alias para el nombre del objeto.
  - Por tanto, puede usarse desde el lado izquierdo de una asignación.
- Se puede hacer método `public` que devuelva ref. a atributo `private`.

```
public:
  // [...]
  int &badSetHour (int); // (.h) DANGEROUS reference return
private:
   int hour;
  // [...]
```

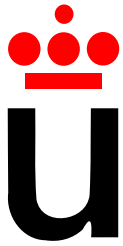
```
// [...]
// (.cpp) POOR PRACTICE: returning reference to a private member!
int &Time::badSetHour (int hh) {
  hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
  return hour; // DANGEROUS reference return
} // end function badSetHour
```

```
int main() {
  Time t; // create Time object

  // initialize hourRef with the reference returned by badSetHour
  int &hourRef = t.badSetHour (20); // 20 is a valid hour

  cout << "Valid hour before modification: " << hourRef;
  hourRef = 30; // use hourRef to set invalid value in object t
  cout << "\nInvalid hour after modification: " << t.getHour();

  // Dangerous: Function call that returns a reference
  //  can be used as an lvalue (left value modificable)!
  t.badSetHour(12) = 74; // assign another invalid value to hour

  cout << "\n\n*****************************\n"
    << "POOR PROGRAMMING PRACTICE!!!!!!!!\n"
    << "t.badSetHour(12) as an lvalue, invalid hour: "
    << t.getHour() << "\n*****************************" << endl;
} // end main
```

# 3. Clases I. Generalidades

Julio Vega

julio.vega@urjc.es

Universidad
Rey Juan Carlos