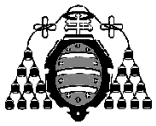


UNIVERSIDAD DE OVIEDO
Departamento de Ingeniería Eléctrica,
Electrónica, de Computadores y Sistemas

Introducción a la Programación Orientada a Objetos con C++

Ignacio Alvarez García

Ing. de Sistemas y Automática

Universidad de Oviedo

Septiembre 2014

Contenido

1. Introducción a la programación orientada a objetos con C++

La programación orientada a objetos permite trasladar a los programas la forma de pensar que tenemos en la solución o descripción de problemas en general. Normalmente nos referimos a entidades (objetos), que solemos clasificar en grupos con características y funcionamiento comunes (clases), y de las cuales nos interesan ciertas características (propiedades) y su funcionamiento (métodos).

Además, al relacionar unos objetos con otros para diseñar sistemas complejos, solemos utilizar la encapsulación, esto es, permitimos que los objetos interactúen entre sí sólo a través de las características y funcionalidades expuestas al exterior, mientras que las características y funcionalidades internas de cada uno son ignoradas por el resto.

El lenguaje C++ es una extensión a C para soportar la programación orientada a objetos (POO). Lo que sigue no es un manual de C++, sino una muy breve introducción para ser capaz de manejar los objetos, propiedades y métodos de las clases de MFC. Para una documentación más exhaustiva se sugiere la lectura de "Aprenda C++ como si estuviera en 1º", de la Universidad de Navarra, y descargable en Internet.

Para escribir código en C++ se pondrá la extensión '.cpp' a los archivos de código fuente.

1.1 Clases y objetos

El elemento fundamental de la POO es la clase. Una clase es una definición de propiedades y métodos para un conjunto de entidades que comparten características comunes.

En C++, una clase se define de manera similar a una estructura, pero utilizando la palabra clave 'class' en lugar de 'struct', y pudiendo declarar como integrantes de la misma tanto variables (propiedades) como métodos (funciones que manipulan dichas variables; sólo se indica su prototipo en la declaración de la clase).

Por ejemplo, para la manipulación de números complejos podemos crear una clase Complejo, que incluiría como variables la parte real e imaginaria, y como métodos el cálculo del módulo, argumento, la suma , multiplicación y división de otro complejo, la exponenciación, etc. Tanto las variables como los métodos pueden ser públicos (accesibles desde dentro y fuera del objeto), privados (accesibles sólo desde las funciones de la clase) o protegidos (accesibles desde las funciones de la clase y de sus derivadas).

EjemploPOO.cpp

```
...
class Complejo
{
// Attributes
public:
        float re,im;

// Operations
public:
        float Modulo();
        float Arg();
        void AnadirReal(float valor);
        ...
} ;
...
```

Las sentencias anteriores contienen la declaración de una clase. Para que tenga funcionalidad, se necesita:

- Escribir el cuerpo de las funciones de la clase Complejo. Para ello, se escribe el prototipo de la función, indicando que pertenece a la clase, y a continuación su cuerpo:

```
tipodevuelto clase::funcion(params)
{
}
```

- Crear variables de tipo Complejo, y utilizar las variables y métodos deseados para cada una de esas variables. Cuando se accede a un método, éste utiliza las variables de la instancia (variable, objeto) sobre la que se aplica.

EjemploPOO.cpp

```
...
class Complejo
{
...
} ;

float Complejo::Modulo()
{
        float result;
        result=sqrt(re*re+im*im);
}
...
void Complejo::AnadirReal(float valor)
{
        re+=valor;
}
...

main()
{
        Complejo a,b;
        float z;

        a.re=3;          a.im=4;
        b.re= a.Modulo();        b.im=a.im+10;
        a.AnadirReal(8);
}
```

Tras la ejecución del código anterior, 'a' será el complejo 11+4j, y 'b' será el complejo 5+14j.

Mediante el uso de variables y funciones miembro públicas y privadas se puede conseguir que desde el exterior a los objetos de la clase Complejo sólo se pueda acceder a las funcionalidades deseadas, resultando por tanto el comportamiento de dichos objetos una caja negra que sólo puede interactuar con otros elementos de la forma que haya previsto su programador.

Para facilitar la realización de programas orientados a objetos, es habitual que para cada clase diferente se utilicen 2 archivos, uno de cabecera (.h) y otro de implementación (.cpp). En el archivo de cabecera se coloca la definición de la clase, y en el archivo de implementación el cuerpo de las funciones. Para el ejemplo anterior, lo usual sería dividir el código en 3 archivos:

Complejo.h

```
class Complejo
{
...
} ;
```

Complejo.cpp

```
#include "complejo.h"

float Complejo::Modulo()
{
...
}

void Complejo::AnadirReal(float v)
{
...
}
```

EjemploPOO.cpp

```
#include "complejo.h"

main()
{
        Complejo a,b;
        float z;

        a.re=3;
        a.im=4;
        b.re= a.Modulo();
        a.AnadirReal(8);
}
```

1.1.1 Funciones constructor y destructor

Existen dos funciones especiales que se pueden añadir a las clases: constructor y destructor. El constructor es una función que se llama exactamente igual que la clase y que no devuelve nada (ni siquiera void). El constructor es llamado automáticamente por C++ cuando se crea un nuevo objeto de la clase. Además, puede haber varios constructores con diferentes parámetros. Para el caso de los complejos, podríamos hacer un constructor que dé el valor por defecto 0+0j, y otro que dé un valor inicial dado desde el exterior:

Complejo.h

```
...

class Complejo
{
...

// Operations
public:
        Complejo();
        Complejo(float i_re,float i_im);
        ...
} ;
```

Complejo.cpp

```
#include "complejo.h"
...
Complejo::Complejo()
{
   re=0; im=0;
}

Complejo::Complejo(float i_re,float i_im)
{
   re=i_re; im=i_im;
}
...
```

EjemploPOO.cpp

```
...
#include "complejo.h"

main()
{
         Complejo a;
         Complejo b(5,8);
`
```

Tras la ejecución del código anterior, 'a' será el complejo 0+0j (se ha utilizado el constructor sin parámetros), y 'b' será el complejo 5+8j.

De la misma manera existe la función destructor, que es llamada de forma automática por C++ cuando deja de existir el objeto. El destructor se declara como el constructor con una tilde (~) delante.

1.2 Clases derivadas

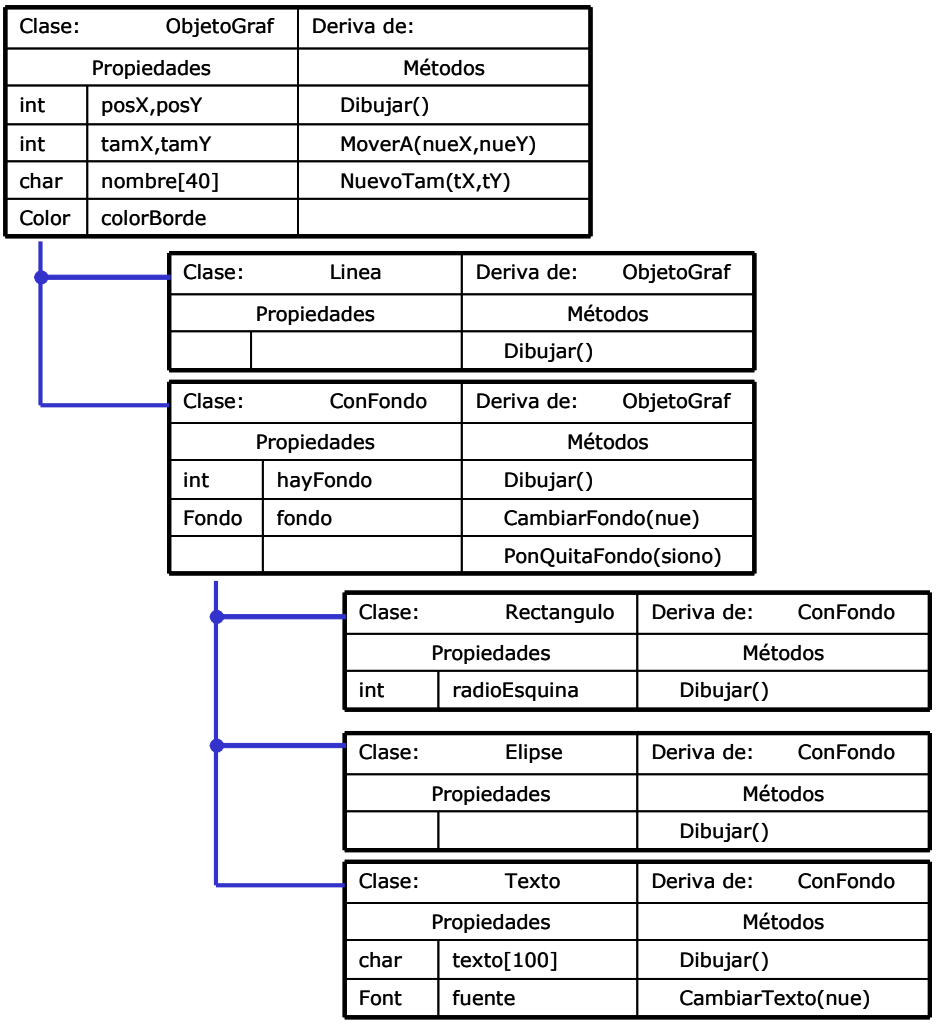
Dado que múltiples clases pueden compartir elementos comunes, la POO se suele realizar de forma jerarquizada, escribiendo clases 'base' con las partes comunes y derivando a partir de ellas clases más especializadas con otros elementos específicos. Una clase 'derivada' de otra hereda todas sus características, esto es, variables y funciones miembro, además de poder añadir las suyas propias. Un objeto de la clase derivada podrá acceder a todas las variables y funciones declaradas en ella, así como a las variables y funciones públicas o protegidas de la clase base. Asimismo, de la nueva clase puede derivarse otra, etc., formándose una jerarquía de clases.

Para derivar una clase de otra se indica en su declaración:

```
class clasederivada : public clasebase
{
      // Variables y funciones de la clase derivada
} ;
```

Puesto que en la misma declaración de la clase derivada se necesita conocer quién es la clase base, se debe realizar en el archivo de cabecera (.h) de la clase derivada la inclusión del archivo de cabecera de la clase base.

Por ejemplo, para realizar una serie de objetos gráficos que se puedan dibujar (línea, cuadrado, cuadrado con texto, elipse, etc.), se podría realizar una jerarquía de clases como la que sigue:



En el ejemplo anterior, la clase Texto tiene las variables miembro *posX,posY, tamX, tamY, nombre, colorBorde* por ser ObjetoGraf*; hayFondo* y *fondo* por ser ConFondo; *radioEsquina* por ser Rectangulo; y *texto* y *fuente* por ser Texto. Desde las funciones miembro de Texto se podrá acceder a aquellas variables miembro de las clases inferiores que hayan sido declaradas públicas o protegidas.

Lo anterior es asimismo aplicable a las funciones miembro.

1.3 Sobrecarga de funciones

Si una clase derivada tiene definida una función idéntica (con el mismo nombre, parámetros y tipo devuelto) que en la clase base, se dice que ha sobrecargado dicha función. Cuando se llama a esta función se utiliza la de la clase derivada y no la de la base. Esto permite que las clases derivadas tengan funcionalidades diferentes a la clase base con el mismo nombre de función.

1.3.1 Sobrecarga de operadores

Dentro de las funciones que se pueden sobrecargar están los operadores. Esto quiere decir que se puede redefinir el comportamiento de un operador, por ejemplo la suma, para una clase determinada. Así, se podría hacer (ver apartado 1.5.6 para la explicación de & y apartado 1.5.2 para la explicación de this):

Complejo.h

```
...
class Complejo
{
...

// Operations
public:
         Complejo& operator +=(const Complejo& otro);
         ...
} ;
...
```

Complejo.cpp

```
...
#include "complejo.h"
...
Complejo& Complejo::operator +=(const Complejo& otro)
{
   re+=otro.re;
   im+=otro.im;
   return *this;
}

...
```

EjemploPOO.cpp

```
...

#include "Complejo.h"

main()
{
    Complejo a,b;

    a.re=3;     a.im=4;
    b.re=2;     b.im=8;

    b+=a;
}
```

Tras la ejecución del código anterior, 'a' será el complejo 3+4j, y 'b' será el complejo 5+12j.

1.4 Funciones virtuales

Cuando se utilizan funciones sobrecargadas, en muchos casos es necesario llamar a la función de la clase derivada desde la clase base. En el ejemplo anterior, si se utiliza una tabla de objetos gráficos de forma genérica, para cada uno de ellos se utilizará la función de dibujo apropiada de la clase derivada. Cuando se desea utilizar una función sobrecargada por una clase derivada desde una referencia a la clase base, debe indicarse que esa función es virtual.

El siguiente ejemplo muestra este comportamiento para los objetos gráficos (ver significado de new y delete en 1.5.5):

EjemploGraf.cpp

```
#include "Texto.h"
#include "Elipse.h"

main()
{
   ObjetoGraf *uno;
   int n;

   // Pide valor de n
   ...
   if (n==1)
      uno=new Texto;
   else
      uno=new Elipse;

   uno->Dibuja();

   delete uno;
}
```

En el ejemplo anterior, el tipo de 'uno' es ObjetoGraf, pero se puede crear el objeto de cualquier clase que derive de ObjetoGraf. Si el operador selecciona n==1, 'uno' apuntará a un

objeto de tipo Texto (que también es ObjetoGraf), y si selecciona n==2 'uno' apuntará a un objeto de tipo Elipse.

Si la función Dibuja() no es virtual, se llamaría a la función Dibuja() de ObjetoGraf, ya que se resuelve a qué función se llama en tiempo de compilación y 'uno' es de tipo ObjetoGraf.

Si la función Dibuja() es virtual, en tiempo de ejecución se comprueba el tipo actual del objeto a que apunta 'uno' (Texto o Elipse), y se llama a la función adecuada: Texto::Dibuja() si se había escogido n==1, o Elipse::Dibuja() si se había escogido n==2.

1.5 Otras características interesantes de C++

A continuación se describen otras características del lenguaje C++ que pueden ser necesarias o útiles.

1.5.1 Declaración de variables locales

En C++ las variables locales pueden declararse en cualquier punto del programa, y no sólo al principio de un bloque de código como en C.

1.5.2 El puntero this

Dentro de las funciones de una clase existe la variable implícita 'this', que es un puntero al objeto de la clase que se está utilizando en ese momento.

1.5.3 Polimorfismo

En C++ se pueden declarar varias funciones con el mismo nombre pero con parámetros o tipo devuelto distintos. El compilador escogerá para cada llamada la función que se corresponda con los parámetros y el tipo devuelto de cada llamada.

Ejemplo:

Complejo.h

```
...
class Complejo
{
...

// Operations
public:
   void Sumar(float real,float imag);
   void Sumar(float real);
   void Sumar(const Complejo& otro);
   ...
} ;
```

Complejo.cpp

```
...
#include "Complejo.h"
void Complejo::Sumar(float real,float imag)
{
   re+=real;
   im+=imag;
}

void Complejo::Sumar(float real)
{
   re+=real;
}

void Complejo::Sumar(const Complejo& otro)
{
   re+=otro.re;
   im+=otro.im;
}
...
```

EjemploPOO.cpp

```
#include "Complejo.h"

main()
{
   Complejo a,b;

   a.re=3;        a.im=4;
   b.re=2;        b.im=8;

   a.Sumar(3,2);          // Usa 1ª versión de Sumar
   a.Sumar(2);            // Usa 2ª versión de Sumar
   a.Sumar(b);            // Usa 3ª versión de Sumar
}
```

1.5.4 Parámetros por defecto

Se pueden declarar parámetros por defecto en una función, poniendo en su declaración (en el .h) el valor por defecto. Si esos parámetros no se pasan, se utiliza el valor por defecto. Por ejemplo, en el caso anterior se podían haber unificado las versiones 1 y 2 de Sumar:

Complejo.h

```
...
class Complejo
{
...

// Operations
public:
         void Sumar(float real,float imag=0);
         void Sumar(const Complejo& otro);
         ...
} ;
```

Complejo.cpp

```
...
#include "Complejo.h"

void Complejo::Sumar(float real,float imag)
{
   re+=real;
   im+=imag;
}

void Complejo::Sumar(const Complejo& otro)
{
   re+=otro.re;
   im+=otro.im;
}
```

EjemploPOO.cpp

```
#include "Complejo.h"

main()
{
   Complejo a,b;

   a.re=3;        a.im=4;
   b.re=2;        b.im=8;

   a.Sumar(3,2);          // Usa 1ª versión de Sumar
   a.Sumar(2);            // Usa 1ª versión de Sumar con imag=0
   a.Sumar(b);            // Usa 3ª versión de Sumar
}
```

1.5.5 Creación y eliminación dinámica de objetos

Si se desea crear de forma dinámica un objeto, se utiliza el operador new seguido de la clase que se desea crear. Este operador realiza la asignación dinámica de memoria para el objeto, llama al constructor del mismo, y por último devuelve un puntero al objeto creado para poderlo utilizar. O sea, equivale a un malloc(sizeof(clase)) seguido de un clase::clase(). Además, se puede hacer que en la creación llame a otro constructor pasando los parámetros del mismo.

Cuando se desea eliminar un objeto creado dinámicamente se utiliza delete, que llama al destructor y luego elimina la memoria asignada.

EjemploPOO.cpp

```
...
#include "Complejo.h"

main()
{
        Complejo *ptA,*ptB;

        ptA=new Complejo;                  // *ptA vale 0+j0 (constructor
por defecto)
        ptB=new Complejo(5,5);    // *ptB vale 5+j5 (constructor con 2
parámetros)

        ptA->Sumar(*ptB);

        delete ptA;
        delete ptB;
}
```

1.5.6 Operador referencia (&)

En C++ existe un nuevo operador & que se aplica a la declaración de una variable o del tipo devuelto por una función, en cuyo caso esta variable o valor devuelto se convierten en una referencia, esto es, algo así como un puntero no modificable. Así, si se hace:

```
int x;

int& y=x;
```

'y' es una referencia a 'x', o sea, es la misma variable con otro nombre: cualquier cosa que se haga con 'y' se está haciendo con 'x' y viceversa.

Como parámetros de funciones, las variables de tipo referencia sirven para indicar el mismo objeto con otro nombre (algo similar a haber pasado un puntero a la variable, con la diferencia de que no hay que acceder con *puntero y que además esa referencia es fija, no se puede hacer que apunte a otra cosa).

Como valores de salida de funciones, sirven para dar una referencia al valor devuelto. En POO es muy habitual que una función pueda devolver una referencia al propio objeto (*this); ver ejemplo en 1.3.1. Ojo: no devolver una referencia a una variable local de la función, porque dicha variable desaparecerá al salir de la función.

2. Programación orientada a entornos gráficos

La programación para entornos gráficos difiere sustancialmente de la programación orientada a interfaz de tipo texto (modo consola). Mientras que en modo consola el programador organiza de forma secuencial tanto las instrucciones de cálculo como las de interacción con el usuario (printf, scanf, getchar, etc.), en un entorno gráfico no está definido el orden exacto en que el usuario interactuará con el programa (ej: pulsar una u otra opción de menú, maximizar la ventana, pulsar un botón, cambiar un texto, etc.).

Por ello, la forma de organizar un programa para ambos entornos es bastante distinta (figura 1).

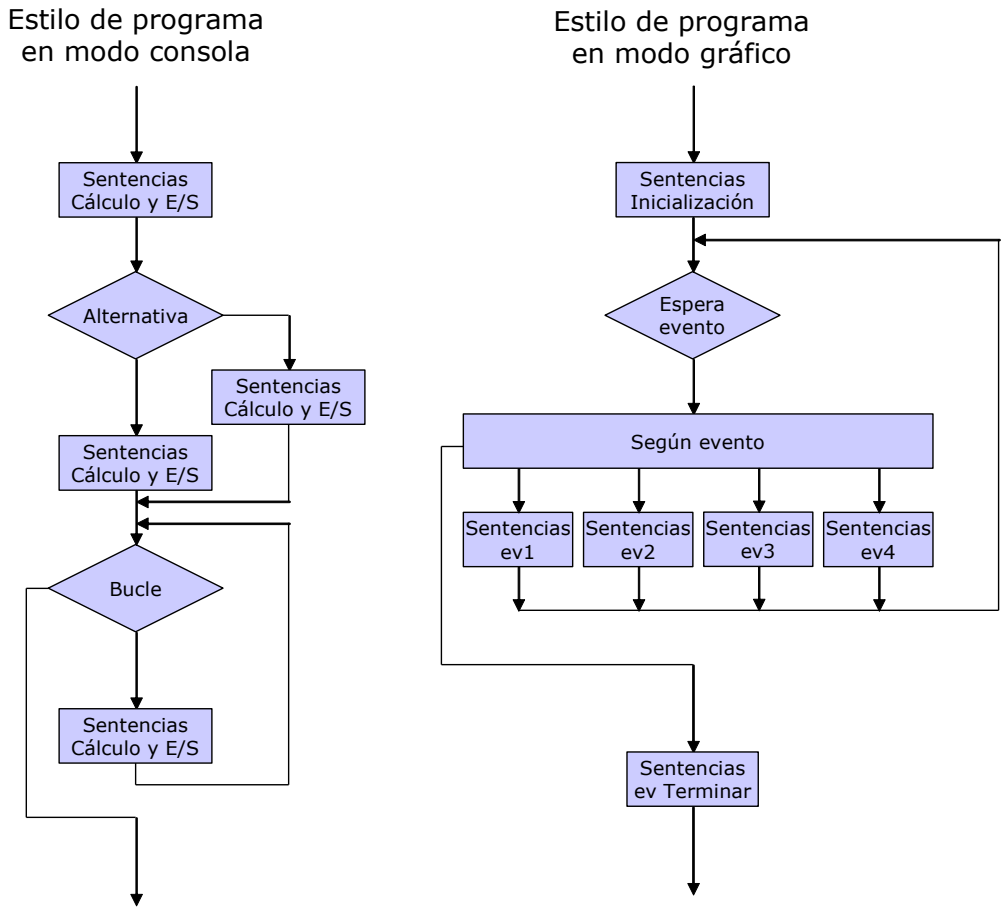


Figura 1

En **modo consola** se van intercalando las sentencias de cálculo y las de interacción con el usuario **en la secuencia que desea el programador**.

Sin embargo, **en modo gráfico** se ejecuta un bucle permanentemente: se espera por un evento (del usuario o del sistema), se ejecuta el código asociado a ese evento, y se vuelve a esperar el siguiente. Los eventos pueden ser variados: se ha pulsado un botón del ratón, se ha pulsado

una tecla, se ha seleccionado una opción de menú, se ha creado una ventana, se ha cambiado el tamaño de la ventana, etc. Además, **no se puede definir a priori el orden de los eventos, sino que depende del usuario**.

Dentro de los diferentes eventos en modo gráfico, es importante comprender los asociados al dibujo de la ventana. A diferencia del modo consola, en que se manda escribir algo en la consola (printf) y esa escritura es 'permanente' (esto es, no es necesario refrescarla), en modo gráfico será necesario redibujar la ventana completa o parte de ella cuando sea requerido, y este requerimiento puede provenir del programa o del sistema (p. ej., la ventana estaba siendo tapada por otra y lo ha dejado de estar), lo cual es indicado mediante un evento. Por ello, en modo gráfico es necesario que el programa tenga almacenado todos los datos necesarios para poder redibujar todo el contenido de la ventana en cualquier momento en que sea recibido al evento 'redibujar'.

Para realizar la programación de las ventanas y de sus eventos, el Sistema Operativo con entorno gráfico (en nuestro caso Windows) proporciona una serie de funciones en una librería. Ese conjunto de funciones se suele denominar API (Application Programming Interface), que en Windows se llama SDK (Software Development Kit). Las funciones ahí contenidas servirán para gestionar ventanas (crear, redimensionar, cerrar, etc.) de diferentes tipos (normal, menú, botón, cuadro de diálogo, cuadro de texto, lista de selección, etc.), obtener eventos, realizar acciones de dibujo, etc.

Un API es un conjunto de funciones muy extenso, ya que son necesarias muchas y muy variadas funciones para gestionar el entorno de ventanas. Además, muchas de las funcionalidades son muy repetitivas a lo largo de los programas (ej. crear ventana principal con opciones de menú), requiriendo un conjunto de llamadas al API relativamente complejo y pesado incluso para el programa más sencillo.

El siguiente ejemplo muestra, simplemente para ilustrar su complejidad, el código para la programación de una ventana sin ninguna funcionalidad adicional.

Ejemplo.c

```
#include <windows.h>

const char g_szClassName[] = "myWindowClass";

// Step 4: the Window Procedure (se llama cada vez que se despacha un mensaje en el
bucle principal de programa)
LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch(msg)
    {
        case WM_CLOSE:
            DestroyWindow(hwnd);
        break;
        case WM_DESTROY:
            PostQuitMessage(0);
        break;
        default:
            return DefWindowProc(hwnd, msg, wParam, lParam);
    }
    return 0;
}

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wc;
    HWND hwnd;
    MSG Msg;

    //Step 1: Registering the Window Class
    wc.cbSize        = sizeof(WNDCLASSEX);
    wc.style         = 0;
    wc.lpfnWndProc   = WndProc;
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName  = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm       = LoadIcon(NULL, IDI_APPLICATION);

    if(!RegisterClassEx(&wc))
        return 0;

    // Step 2: Creating the Window
    hwnd = CreateWindowEx(WS_EX_CLIENTEDGE,g_szClassName,"The title of my window",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
        NULL, NULL, hInstance, NULL);

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```

Para facilitar este trabajo, muchos entornos de desarrollo (en nuestro caso Qt 5) proporcionan una forma más sencilla de acceder al API, mediante una librería de clases que encapsula la mayor parte de la complejidad, y sólo deja como tarea al programador la realización de las partes específicas de su programa. Así, por ejemplo, la creación de la ventana principal estará encapsulada en una serie de objetos que hacen que se pueda crear el marco del programa ¡sin escribir una sola línea de código adicional a las clases ya disponibles!

Para ayudar aún más, los entornos de desarrollo suelen disponer de utilidades que permiten situar de forma gráfica los elementos de interfaz (menús, botones, cuadros de texto, etc.), e incluso enlazarlos con las funciones de servicio de sus eventos de una forma gráfica e intuitiva. De esta manera, el programador se ahorra escribir gran cantidad de código (se ahorra no sólo tiempo, sino también errores) y, además, la parte 'a la vista' de su programa es clara y concisa.

Dado que los elementos de interfaz gráfico se pueden describir y utilizar mucho más fácilmente como objetos, lo habitual es que se utilice la programación orientada a objetos para la programación de estos interfaces.

Un ejemplo de aplicación de ventana realizada en Qt contiene el código básico siguiente. A destacar que este código es creado de forma automática por el entorno de desarrollo, y que toda la funcionalidad requerida está encapsulada en las clases base (QMainWindow, QApplication, etc.):

MainWindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

El detalle de la programación orientada a entorno gráfico en Qt se encuentra en documentos siguientes.