Event Driven Programming

Introduction

Before moving on to more OOP concepts, we will look at event driven programming. This will enable you to become familiar with the concepts before moving on to the more challenging OOP concepts. Although OOP can be used in a console or event driven environment (e.g. Windows), you are used to console mode now so let's move on to programming in Windows.

Consider the normal procedural approach to writing an application that requests students' names and grades then outputs results.

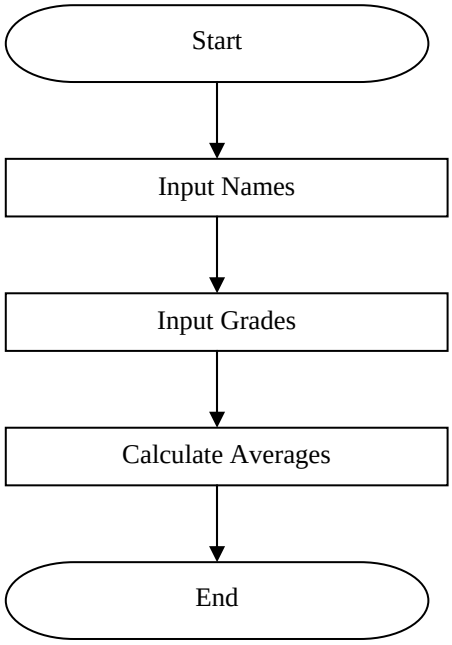


Figure 2 Procedural Programing Paradigm

All names are input then the grades. To go back to modify a name or a grade once that section is complete is not straight forward (possible of course but messy).

Contrast this with an event driven application.

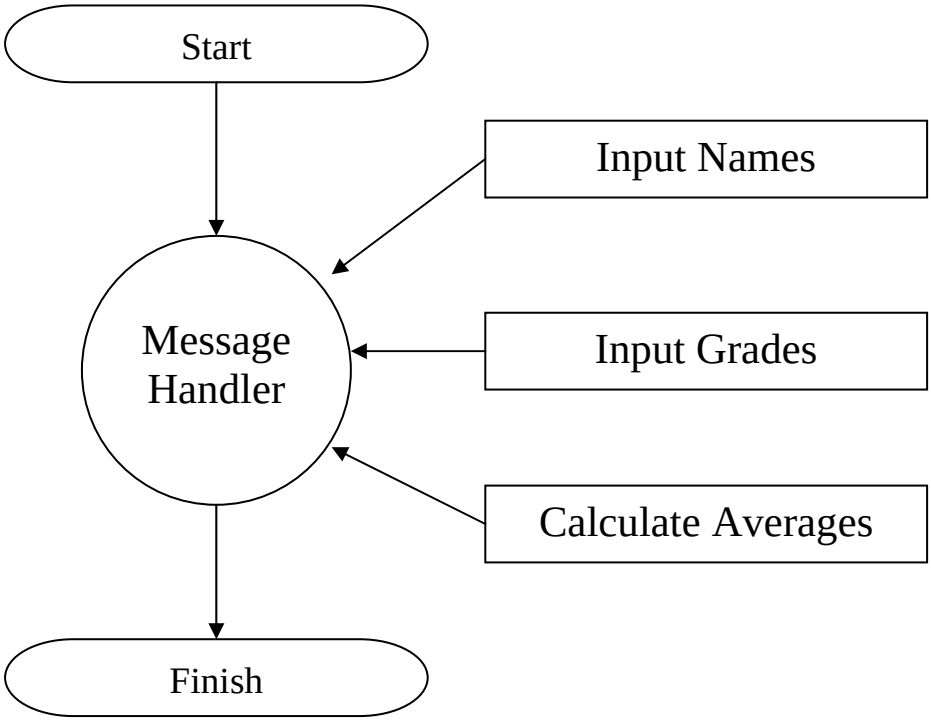


Figure 3 Event Driven Programing Paradigm

The application starts and sits in a loop waiting for an event. If the event is to input a test grade then this is dealt with then the application returns to the loop and waits for the next event. This is similar to a main program being the message processing loop and each of the inputs are interrupts. The main program is interrupted, the request is dealt with and return to main to continue looping until the next interrupt. Events and interrupts are often used synonymously. Eventually a 'quit' event is sent to the main program to terminate it.

Event driven programs are efficient and easier to write than procedural programs if the application has many possible asynchronous (can occur at any time) inputs. For example, embedded applications such as a video recorder would work in this way. The main program performs basic chores but can be interrupted by any of several events such as the play, record, rewind etc., buttons being pressed – or the detection of end of tape.

Windows programs tend to be event oriented as windows applications take input from the keyboard, mouse, and various system devices.

Windows applications

The windows programming environment discussed during this module will be heavily biased towards Microsoft Windows. This is chosen only for its popularity and availability and not necessarily on technical merit.

The functionality of windows based systems whether they are on an Apple Macintosh or UNIX X-Windows or Microsoft Windows are all basically the same. Windows relies on a system of messages. The main program is a continuous loop waiting for events in the form of messages. When the loop receives a message, the message is processed and the main program returns to waiting for the next one.

There are many types of message such as **mouse moved**, **left button pressed**, **key pressed** etc. The windows system is of course a bit more complex than that but that is basically what happens. The following diagram illustrates the message processing mechanism.

Windows can support several applications at the same time (multitasking) so needs to be able to handle messages sent to each of the individual programs.

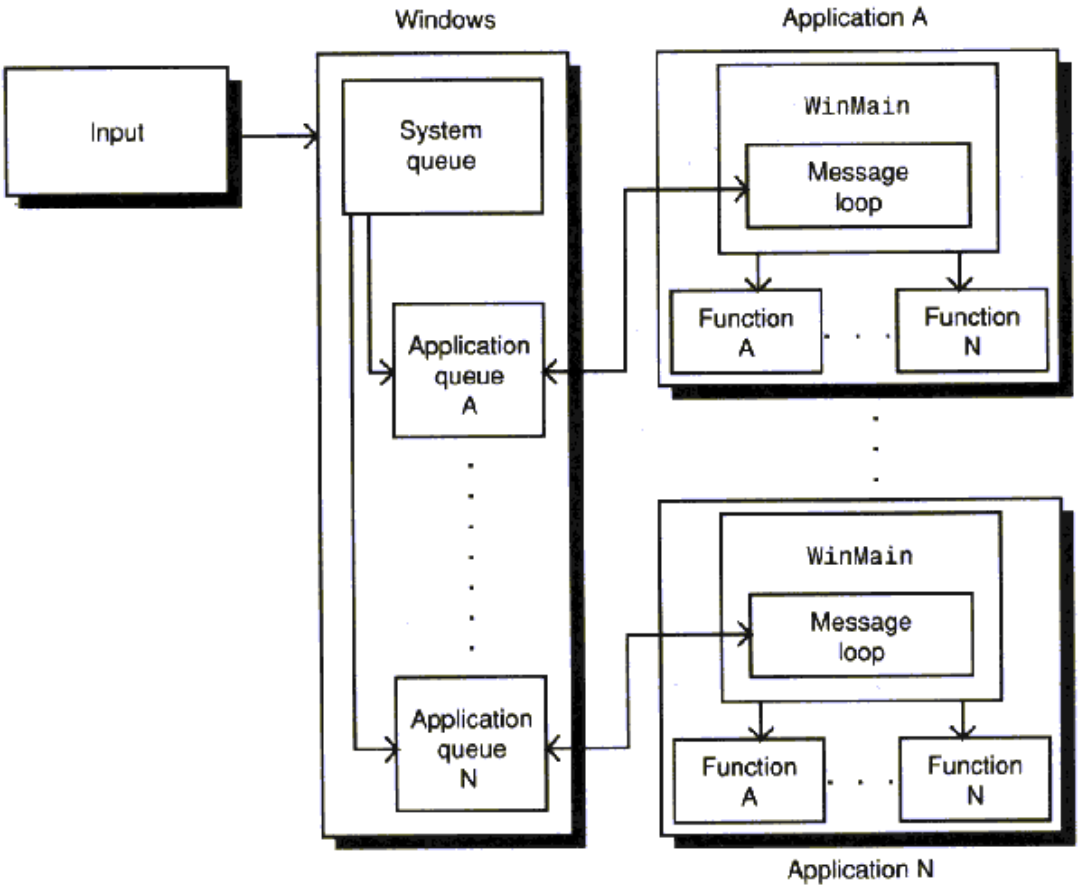


Figure 4 Windows messaging system

A message is generated from within or outside the system. For example, the user presses the left mouse button. This message is the **input**. The message is added to the system queue. Messages are queued as they may be generated faster than the system can process them. Consider what happens when moving the mouse cursor quickly from one side of the screen to the other - each increment of the mouse position generates a **mouse move** message which is added to the queue – so in the shift of the mouse, hundreds of mouse messages are added to the queue and wait to be processed.

The system identifies which message is aimed at which application and passes the messages to the message queue for that application. The messages are picked up from the application queue by the main program function **`WinMain()`**. **`WinMain()`** is responsible for either processing the message or passing it on to the window or

control (e.g. button) to which the event is aiming to interact with. On completion of that message, the next message is read from the queue and processed or passed on. The code which is executed for every message is the main message queue processing loop. This loop runs continuously for the duration of the program – i.e. it is the circle called "message handler" in Figure 3 on page 37.

```
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
  TranslateMessage(&Msg);
  DispatchMessage(&Msg);
}
```

The GetMessage() function gets the next message off the queue (e.g. mouse move, mouse button down or whatever) and as long as it returns greater then 0, it will execute the while block. Translate message does some conversion to key codes if the message is a key press and then the DispatchMessage function is called. This function passed the message to the appropriate function which will deal with the event (e.g. the mouse down event may be passed to a button which will cause the button to look like it has been pressed and the code associated with the button press is executed). On completion of the event, the GetMessage() function is called again to get the next one until the message it retrieves is the application quit message; in this case, GetMessage() returns with a negative value which terminates the loop and the application closes.

Windows Message

Messages are generated by Windows and by applications. Windows generates a message at each input event for example, when the user types, moves the mouse, or clicks a control such as a scroll bar. Windows also generates messages in response to changes in the system brought about by an application, such as when an application resizes one of its windows. An application can generate messages to direct its own windows to perform tasks or to communicate with windows in other applications.

A message is basically a structure which is filled by the device driver (mouse driver, keyboard driver) in response to an event such as a

mouse click. The structure contains information relevant to the event such as x,y co-ordinates of the mouse and which button was pressed. Windows receives this message from the device driver (or from itself in the case of resizing or moving a window) and passes the information to a Windows procedure. The window handle (hwnd) identifies the window for which the message is intended (e.g. the one which was active when the mouse was clicked on it). Windows uses the handle to determine which window procedure to send the message to.

Windows

There is no way that this subject can be given the full treatment as it is a huge area. Windows programmers tend to work in certain areas and know that area very well – it is extremely difficult to be familiar with the whole of the Windows development environment.

This module will provide an introduction into basic Windows programming to a level where you should be able to read up on the application areas of personal interest to you.

There are a lot of general questions to be answered during this introduction such as:

What language is Windows written in?

Do Windows applications use C or C++?

What is the Windows API?

What's the best development environment?

What's the difference between Borland C, C++, Builder, Visual C++, Visual Basic, Delphi etc.?

What is MFC, VCL and .NET?

These tend to be some of the main questions asked when venturing into the programming world of windows.

What language is Windows written in?

Kernel is probably largely in assembly. Most of the DLLs and such are in C and the final layer (IE, etc) is in C++.

Do Windows applications use C or C++?

Windows applications can be written in many languages. Popular languages are C, C++, Pascal (Delphi), Visual Basic, Java and C#. However, the API consists of 'C' functions so the other languages have to provide "wrappers" around the actual call such that the functions can be called in that particular language.

What is the Windows API?

Think of this as a set of C functions written by Microsoft that make up the Windows environment in the same way that DOS provides low level functionality to programmers in the form of interrupt functions such as `int 21`, `int 10` etc.

When Windows was written, an enormous number of functions were written in order to implement the environment. These functions are called continuously by the system but they are also available for the applications programmer to use. They are written in C so to use them you need the header files with the function prototypes and structure declarations. These are provided in various forms of Software Development Kits (SDK). For example, when C++ builder is installed, all the SDK files are also installed for development use.

Windows API functions tend to be complex to use and require extensive knowledge before they can be used effectively. For example, it is not uncommon to have a function with 10 parameters and some of these parameters are a result of calling other functions with equally complex parameter set. Programming at this level is complex, time consuming and tedious – but fast execution.

What's the best development environment?

C Compilers

If using a Borland or Microsoft 'C' compiler, you would likely be programming at the lowest level and have to call API functions for the whole application.

Borland and Microsoft C++ Development

Environments such as Borland C++ Builder and Microsoft Visual C++ offer more choice and support in the development of Windows applications. Both will allow direct calls to the API using C code. Both will allow direct API calls using C++ (as 'C' is a sub-set).

The Microsoft Foundation Class libraries (MFC) provide C++ classes which enable an object oriented approach to Windows programming. The class implementations call the system API functions so the classes acts as wrappers around API function calls. Although programming using the MFC is simpler so leads to faster code development, it is still non-trivial. The MFC became very popular so Borland had to included it in their Borland C++ builder package.

Rapid application Development (RAD)

To enable non-guru programmers to write Windows applications, Microsoft produced Visual Basic. This used a language that non-professional programmers could learn and provided a very high level programming interface which is very far removed from the API. It uses a language which has evolved from an unstructured inefficient amateur language – BASIC (Beginners' All-purpose Symbolic Instruction Code). Visual Basic, until recently was an interpreted language, i.e., it was not compiled into an executable file but partially compiled into a pseudo language which is executed at run time using support libraries (DLLs) to interpret the code.

Visual Basic was EXTREMELY slow and non-object oriented, but as it is a Microsoft product and it provided a rapid development environment using drag-and-drop component-based architecture, it became very popular. A window can be created with a single command and a button could be added to the window by dragging it off a tool bar. It is definitely a RAD environment and is very useful for producing prototype user interfaces quickly. When the customer was happy with the interface, the application tended to be re-written in C or C++. If any new components need to be added to the tool bar in VB, they need to be written in a compiled language – basically C, so it was not possible to extend the component library using the environment's own language.

Borland are Pascal experts and produced Delphi. This is their object oriented enhanced version of the structured programming language Pascal. The Delphi environment looks just like Visual Basic and is a

RAD environment. Where it scored massively over VB was in the fact that Delphi uses a Visual Component Library (VCL) which is based on object oriented technology so, new components can easily be created by deriving the functionality of the parent components using the environment's own programming language. Another massive advantage is that Delphi uses a compiler. Delphi uses object Pascal and Windows programs are compiled in incredibly short times. However, Pascal is not an industry standard programming language. Delphi seemed to be the answer but it used Pascal. Microsoft were not shaken by Delphi even though applications ran around 30 times faster then VB because they did not consider Pascal as a serious language and neither did most others. However, Delphi was so good that it was worth learning Pascal or in my case revising the language and getting used to it again – but it was worth it. But surely it can't be as fast as 'C'? Borland had already thought of this and the front end was Pascal but the code generator was the same one that they use in their C++ compilers so it was as fast as the equivalent code written in C++.

It still did not take off as large organisations could not take the risk of using a language that was supported by only one company. What was needed, and what would really frighten Microsoft, was a C++ version of Delphi. As C++ is now an industry standard it was worth Borland porting Delphi in the form of C++ builder. This really put the wind up Microsoft and has pushed them into speeding up their VB environment which now produces run time executable files in an object oriented environment.

C++ Builder takes longer to compile applications than Delphi as C++ is a very complex language (many feel too complex) and the run-time code is no faster. However, the source code supports an industry standard language so the risk to companies is drastically reduced.

The higher the level the development environment, the further away from the API it becomes. As all the environments will eventually call API code, these extra levels of code make the development process

faster but the run time speed slower. The beauty of the C++ builder environment is that you can write C applications for DOS or Windows using the API, or use C++, or use MFC, or use the VCL or Pascal or assembly language. Use a combination of any of them so the code can be as fast as possible. They have added the CLX library for development of windows applications which run on the Linux operating system.

You can even write Pascal code or directly compile Delphi source code within the same project as C++ source code! With these options it is possible to use RAD for the front end and the API for more speed critical areas or even assembly language if required as this can be inserted in-line or linked, as Turbo Assembler is also part of the environment.

The most popular development environment in industry is probably Visual studio and this is probably because it is a Microsoft product. Visual studio combined visual basic and visual C++ into the same development environment. This integration of languages has been adopted recently by Borland as they have integrated C++ builder into the Delphi environment.

.NET

Microsoft have gone one step further and integrated all their languages into the Visual studio .net development environment. They have developed a new language (C#) which aims to incorporate the best features of C++ and Java in a RAD environment. This enables Visual Basic, C++ and C# to be developed within the same development environment. The .net framework is essentially a set of library routines that interact with the windows operating system but can be called by any of the languages. This saves having to learn three sets of libraries if using all three languages. Visual Basic has virtually been rewritten to work within this environment so has a fairly steep learning curve if moving from the previous VB. Some "clunky" mods have been made to VC++ to shoehorn it into the environment but it does mean that the visual components available to VB and C# are now available for VC++ making it a RAD environment like C++ builder. The languages are compiled into an

intermediate byte code (not machine code) so the code is not platform specific.

A Just in time (JIT) compiler needs to reside on the target platform to compile and run the byte code. This does lead to some loss of efficiency, but does enable RAD in C++ which has the potential to run on various platforms.

The only language specifically written for this platform is C# which may become popular in the same way VB did even though it too is single sourced. It is possible to write C++ applications within Visual Studio .net which do not rely on the .net freamework and use the MFC libraries if required.

At the end of the day – you pay your money and take your choice. Once familiar with Builder or Visual C++ it should be relatively easy to move from one to the other or in fact any other related language such as C#, Java and PHP.

Windows programming

There is a substantial amount of work involved in even the most basic application. VC++ provides wizards which produce chunks of code in response to a few questions but design, drawing and locating controls is still tiresome. In C++ Builder, the application can be created totally visually and in an application such as this can be created without writing any code other than specifying what the message box should say when the button is pressed – this is a RAD approach.

Rather than typing repetitive code for each control, builder works in the same way as VB (and now C++ in VS .net) and each control has a set of properties that can be edited visually and the results of that can be seen immediately without running the application. Rather than using call-back functions with switch statements to filter out required events, each VCL control has events associated with it. For example, if a button is clicked, the click event is triggered and the function definition for the event is provided automatically – all you have to do is fill in the code in response to the event. For example, in C++ builder, a function signature is created automatically which will respond to a button click (the message is read from the queue and if it was a button click message destined for Button1, the message will be dispatched to this function. On completion of this function, execution returns to the message loop):

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  … put code in here to execute when button is clicked
}
```

These notes do not cover code development in any particular environment as it is a case of reading the user guide for any particular environment – the C++ coding is the same although the built-in libraries will be different. You will be given an introduction into an appropriate development environment in the tutorials where we will develop Windows applications. You will also be given some example

programs to work on and lab exercises; that way you can learn the development environment in your own time then bring questions for to the lab tutor to help you with.

Creating a simple Windows application

Run C++ Builder. It opens in Windows development mode. If you are not in that mode then choose File | new | Application. You will be presented with a blank form. Form is the name given to the control that will become your window at run time. The application will run at this point by pressing the run button; it produces a blank but resizable and movable window. This is because a Form is a class which already has all the functionality to behave as a window. We will add other controls and code to it to make it do what we want.

Save the project somewhere sensible and call say "Hello Windows". To do this use: File | Save Project as. By default the Form file is called Unit1.cpp so change this to MainForm.cpp and save it. You are then asked to save the project file. Whatever you call your project file will become the name of the .exe file so call this HelloWindows. If you run or build the application now, you will see five key files (there could be others but always these five):

1. HelloWindows.bpr: this is the project file
2. HelloWindows.cpp: this is main() which is written for you
3. MainForm.cpp: this is your code file for the main window
4. MainForm.dfm: this is definition of all the form's controls
5. MainForm.h: this is the header file for the MainForm's .cpp file

Do not delete individual files from this set or try to rename them in explorer because the project file keeps track of these and if you change them outside the environment it will get very confused and you will get very frustrated with weird errors. If you want to rename them, use File | SaveAs in the development environment. Note that files, 3, 4 and 5 must all have the same filename as they are a set.

Looking briefly at the contents of these files shows that the environment has created a main() for you (called WinMain in Windows) – don't edit this file. Don't edit the project file or the .dfm file. Your work is done in the MainForm.cpp and MainForm.h files.

Looking at MainForm.h for a moment, you can see that the header guard has been written for you, appropriate includes added and the class declared by default as TForm1. There is a __published section – this is reserved for the development environment to declare controls such as buttons etc., so don't edit anything you find in this section. There is a private and protected section for you to put in your private data and functions or public interface functions. Do not delete or modify any code that you have not written yourself.

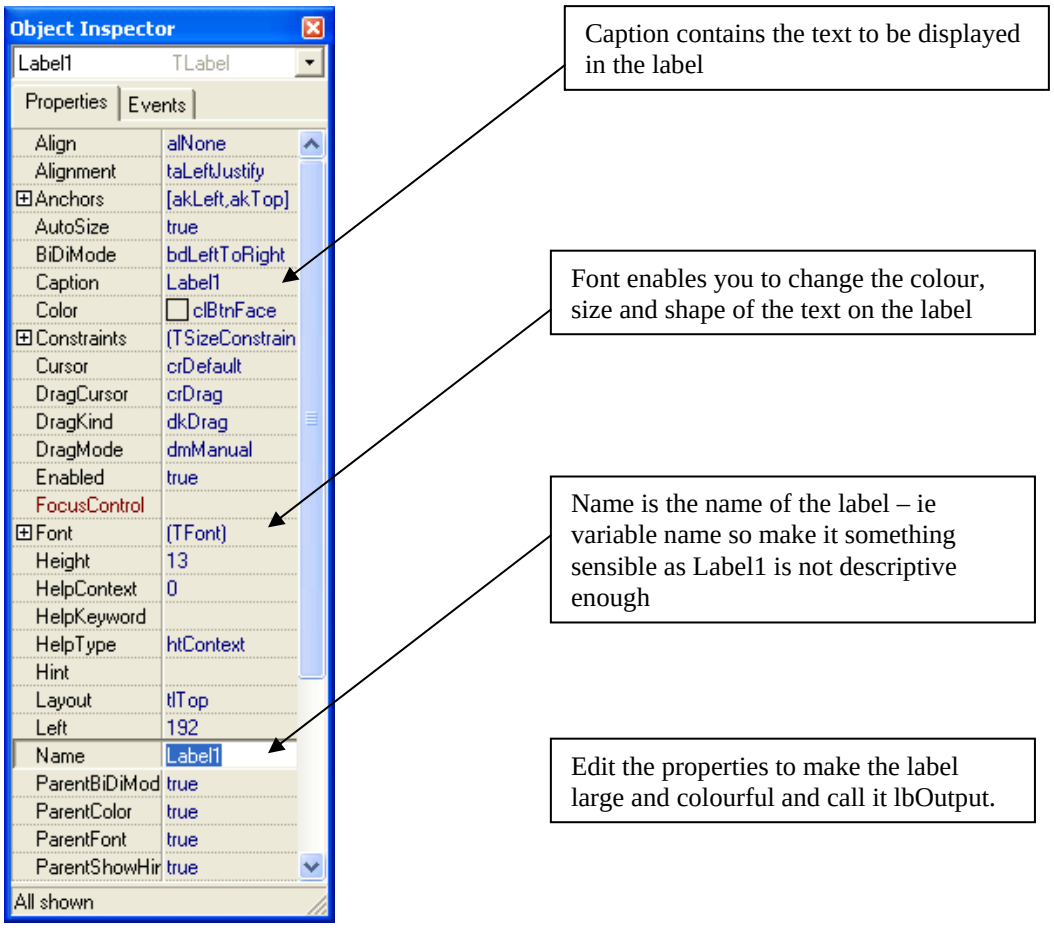
```
//---------------------------------------------------------------------------

#ifndef MainFormH
#define MainFormH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
class TForm1 : public TForm
{
__published:      // IDE-managed Components
private:    // User declarations
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

The MainForm.cpp file shows the declaration of the class constructor. If you have any initialisation to do before the program runs then it can be done in here.

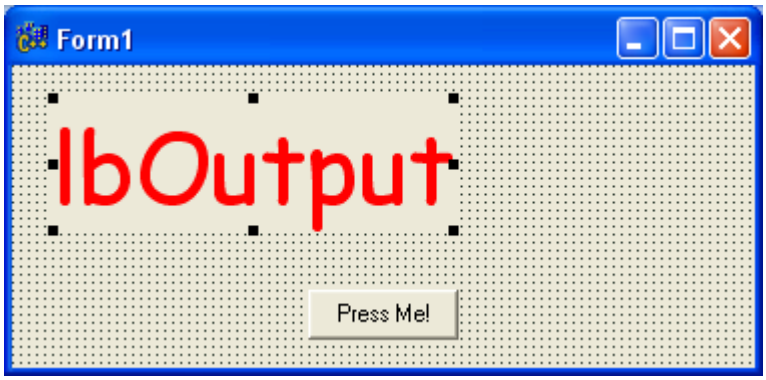
```
#include <vcl.h>
#pragma hdrstop

#include "MainForm.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
```

```
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner) : TForm(Owner)
{
 … Your initialisation code goes in here
}
```

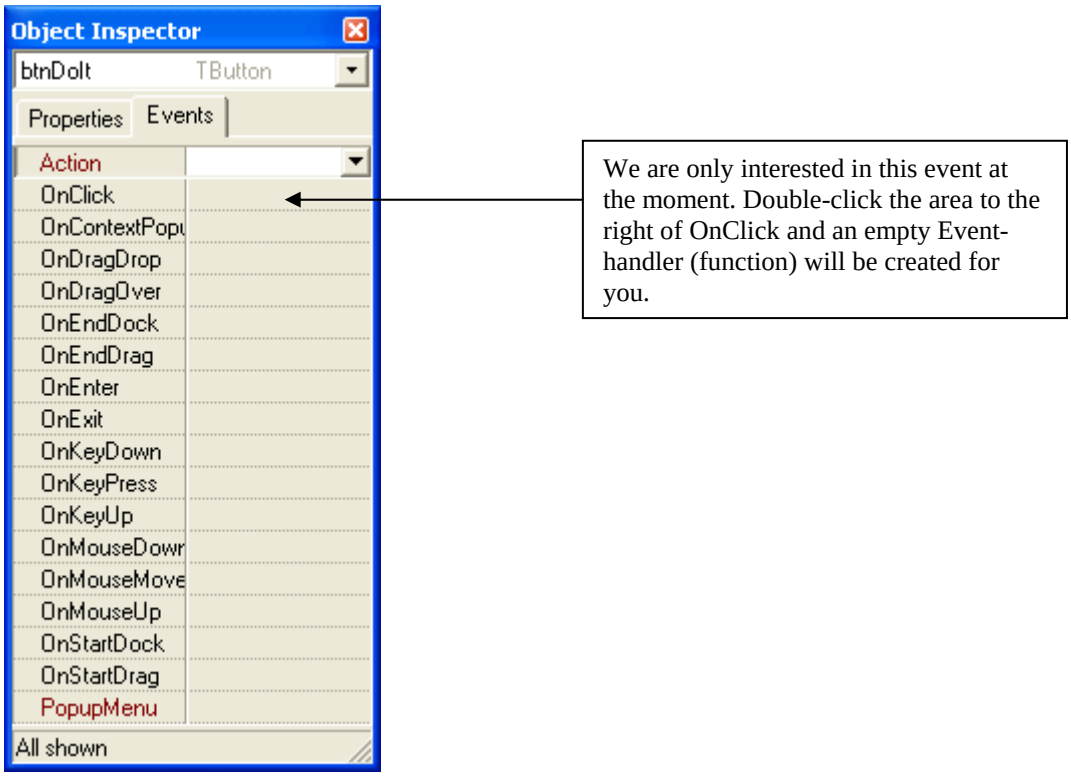
So let's create a HelloWindows program. Click on the standard component called label (has a capital A on it) then click on the form. A label is a convenient way of displaying simple text. Each component has properties – this one has things like Caption, font etc. To change a control's properties, press function key 11 to view the Object inspector.



Now click on a button on the standard component's toolbar then click on the form to drop it. Using the Object Inspector, name the button btnDoIt and change its caption to "Press Me". Resize the form so it is not too large and you should have something like this:



To make something happen when the button is clicked, you need to add some code in its Click event. To access events (functions that are called as a result of you interacting with the control) use the Events tab on the Property Inspector.



Enter the code in the event handler as shown in bold:

```
void __fastcall TForm1::btnDoItClick(TObject *Sender)
{
  lbOutput->Caption = "Hello Windows";
}
```

The constant string is copied to the Caption property of the label when the button is pressed. Press run and try it out.

Looking at the MainForm.h file again, you will see that the development environment has added declarations for the controls for you in the published section. Do not delete these from here, click on the control on the form and press delete – the dev env will remove the declaration. Also, do not delete the event handler code that was created for you. If you no longer want that event handler, remove all your code and the next time you save the file, the event handler will be removed automatically. In a nutshell: if you wrote it, you can modify it or delete it, if you didn't then leave it alone or you will end up in all sorts of trouble.

Although OOP can be done in console mode procedural programming, it can also be used in the event driven mode. Event driven applications are particularly powerful when used to interface to a user. As Windows is a graphics based environment as opposed to a text based environment, text based functions such as **cin** and **cout** for collecting user input and displaying output are no longer of any use to us. Input will be taken from graphical controls (which may enable text to be input through them) and output will be to graphical controls, some of which have been designed to display text.