Chapter 13: Debugging

Debugging is something that as a programmer you will do every time you sit down at a keyboard. It is not a task that is left until a program is written, it is a process that you must go through many hundreds of times before a program can be completed.

So what does debugging mean? It is the process of searching out bugs in your code and fixing them. There are a variety of tools available within Visual Studio and we will be looking at these in this chapter. Before we get to the tools, we will first take a look at the different types of bugs you will encounter.

|  |  |
| --- | --- |
| **Bug Type** | **Description** |
| Syntax Error | Code that will not compile. |
| Functional Error | Code that compiles but gives unexpected results. |
| Memory Leak | Program will run for a period of time and then implode. |

Table 10.1: Types of bugs

10.1 Example: Syntax Errors

Examples of syntax errors are incorrectly spelled keywords or variables and missing semi colons. There are far more reasons for a syntax error, but regardless of the reason Visual Studio will not compile the code and should in most instances give a detailed reason for the issue in the output window. At times the bug report will not be very helpful and will mean you have got to search for the source of the error yourself. Double clicking on a bug in the output window will take you directly to the area in code where Visual Studio thinks the issue is to be found.

int num1 = 5

if ( num > 5 )

cout << num1 << “ is greater than 5” << endl;

10.2 Example: Functional Errors

Examples of functional errors would be the use of a single = when you meant to use ==. This code would compile but would not give the results you expect. These types of bugs can be somewhat more difficult to find as the error might not become immediately apparent. That is to say, everything may run as expected until much later in the program execution.

int num1 = 10;

if( num1 = 5 )

cout << num1 << “ is equal to 5” << endl;

else

cout << num1 << “ is not equal to 5” << endl;

10.3 Example: Memory Leaks

When assigning memory it is vital that you free the memory once you are finished with it. Imagine a function that reserves memory for the task it is about to complete, but does not free the memory at the end of the function. This function is then called several hundred times. Eventually your computer will run out of memory and the program WILL crash.

void someFunction()

{

int\* pNum = new int;

\*pNum = 5;

cout << “pNum = “ << \*pNum << endl;

}

int main()

{

for ( int I = 0; i < 10000; i++ )

someFunction();

return 0;

}

**Breakpoints**

A breakpoint is a tool that allows you stop your code at run time. This allows you to look at the current values stored in variables and even step through the code and see where it goes.

There are two types of breakpoints that can be used in Visual Studio to help you track down functional errors.

The first and most commonly used is the bog standard breakpoint. To place a breakpoint simply click in the grey vertical bar that runs down the left hand side of your code. A red dot should appear. With this in place run your program and you will see that execution is paused when the breakpoint is hit. To remove this breakpoint simply click on the red dot and it will be removed. You can also disable a breakpoint from either the breakpoint window or from the debug menu. To disable a breakpoint means to leave it in place, but to turn it off. It will grey out slightly if disabled. A disabled breakpoint will not stop code execution. If you right click on the red dot you will see that options are presented. From this menu you can add conditions to a breakpoint. For example if you had a breakpoint within a loop, but you don’t want it to stop until it is on its 20th iteration, you can set this up from this menu.

The second type of breakpoint is a data breakpoint. This type of breakpoint is not placed on a line of code, but rather placed on a memory address. Whenever the contents of that memory address is altered the data breakpoint will stop execution of the code at the line of code that changed the memory. This is useful when allocating your own memory using pointers and you have a particularly difficult bug to track down. To assign a data breakpoint select breakpoints>data breakpoint from the debug menu. You must assign the memory address you are interested in monitoring.

When you have the program paused you can step through your code. The most commonly used functions are to step over a line of code, step into a line of code, or step out of a function. These are all available through the debug menu, the shortcut menu or shortcut keys. Familiarise yourself with these shortcut keys as you will be using them.

**Debug Windows**

As mentioned above there is a **Breakpoint Window**. This is where you can manage all breakpoints currently in your codebase. You can even remove them from you code from the breakpoint window.

The **Immediate Window** is where all local variables are shown. The values stored within these variables are shown to the right of the variable name. Any variable that changes during stepping through code will change red to show you it has changed.

There is a **Watch Window** available that allows you to add any variable names that you wish to watch. Once added you will be able to see the values of these variables in the same way as the immediate window detailed above.

**Program 30: Broken Code**

Enter the following program and fix all the syntax errors to get it working. Copy your fixed code in the relevant slot below along with a screenshot of the working program.

Tip: There are 6 errors that will stop this program running. Your output may say more, but once these 6 errors are fixed all compounded errors will disappear.

#include <iostream>

int mian()

{

int num1 = 5;

for( int i = 0; i < num1; i++ )

{

cout << “i = “ << i << endl;

}

int num1 = 15;

for( int i = 0; i < num2; i++ ){

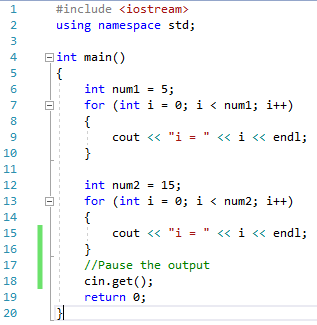
cout << “i = “ << i << endl;

/\*Pause the output/

cin.get()

return 0;

}





**Program 31: Unexpected Code**

Enter the following program and fix all the functional errors. This program will run, but it will not give the expected results. Copy your fixed code in the relevant slot below along with a screenshot of the working program.

The expected output is:

1 is an odd number

2 is an even number

3 is an odd number

#include <iostream>

using namespace std;

int main()

{

int count = 3;

for( int i = 1; i > count; i++ )

{

cout << i;

if( count = 2 )

cout << “ is an even number” << endl;

else

cout << “ is an odd number” << endl;

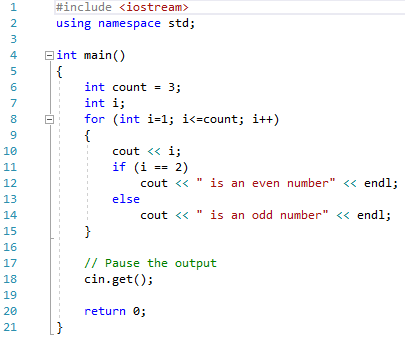
}

// Pause the output

cin.get();

return 0;

}





Chapter 14: Structures

At times as a programmer you need to put various different types together into a unified single type. An obvious example would be a 2D position on the screen. This position would be made up of two float **data types**, one called *x*, the other called *y*. How then do we do this then? Well a **struct** is the answer.

The **struct** statement will define a new **data type** that can be used throughout your program. The definition of a **struct** is as follows:

struct structName

{

variable definition;

variable definition;

…

};

11.1 Example: Defining a struct

So using the position example given at the beginning, we would set this up as:

struct Position

{

float x;

float y;

};

11.2 Example: Declaring a variable of struct data type.

Now this creates a new **data type**, but how do we now use this? We initialise it as we would any other data type. So to create a variable called *characterPosition* of **data type** *Position*, we could do one of the following:

Set the variable name as part of the **struct** statement.

struct Position

{

float x;

float y;

} characterPosition;

Alternatively, we can split this up and do it in two separate lines of code.

struct Position

{

float x;

float y;

};

Position characterPosition;

11.3 Example: Accessing hidden variables.

So, we now have our new **data type** *Position*, but how do we actually get access to the float variables hidden within. Well to do this we use the ‘.’ [dot] operator.

// Create a variable of data type Position.

Position characterPosition;

// Set the hidden float values for x and y.

characterPosition.x = 10.0f;

characterPosition.y = 15.0f;

11.4 Example: Structs and functions.

It is also possible for a **struct** to contain functions, although this is not so common since the introduction of **classes**. We will be including functions here for completion sake. The most obvious function to include in any **struct** is a **constructor**. A c**constructor** will be called on initialisation and can take parameters to set the internal variables. A **constructor** must have the same name as the structName and NOT return any values.

This code snippet shows two constructors for the *Position* data type. The first takes no parameters and sets the internal variables to zero, the second takes in the starting values with which to set up the internal variables. You can have as many **constructors** as needed, but they must have different parameters data types for the compiler to know which one to call.

struct Position

{

float x;

float y;

// Constructor 1: Set variables to zero.

Position()

{

x = 0.0f;

y = 0.0f;

}

// Constructor 2: Parameter values are used to set the starting values.

Position( float startX, float startY )

{

x = startX;

y = startY;

}

};

11.5 Example: Using constructors with structs.

There is a difference when declaring a variable of a struct data type using a **constructor** and it uses the following format for the first constructor detailed above:

Position characterPosition = Position();

And the following for the second constructor detailed in example 11.4:

Position characterPosition = Position( 10.0f, 15.0f );

**Program 32: Structs**

Write a program to prompt the user to enter their name [*string*], age [*int*] and telephone number [*string*]. This is to be stored in a **struct data type** that you have defined.

Output the **struct** details to the console through the use of a function defined within the **struct** definition. Something along these lines:

personDetails.PrintDetails();

