Making Decisions

C# Programming

Writing Software

- It is important when you write software that you ensure that you do it well
- A “good” program is not just one that works – although this does of course help
- For a program to be properly useful it is also important to ensure that it is well written

Well Written Code

- Easy to read
  - All the names in the text should add meaning
- Clean and consistent layout
  - The same format for common constructions
- Well managed
  - It should be clear who wrote the code and the reasons for any changes

Comments

- One way to add a lot of value to a program is to add comments
  - We already do this with sensible variable names, but comments allow even more detail
- A comment is something that the compiler completely ignores
  - It is only for use by the programmer

Creating a Comment

```
/* This program works out the result by adding
two numbers together  */
```

- The character sequence `/*` means the start of a comment
- The sequence `*/` means the end of a comment

Line Comments

```
x= 0; // put the cursor at the left edge
```

- The character sequence // starts a comment that extends to the end of the line
- You can use these to quickly explain what a statement is doing

Stupid Comments

```
count = count + 1; // add 1 to count
```

- Comments should add value
- They should not just replicate information that a programmer should know already

Program Flow

- At the moment every program we have written has just run through its statements in sequence
- This form of linear program flow is not always what you want
- The power of computer programs is that they can make decisions

The Three Types of Flow

1. Straight line:

   Perform one statement after another

2. Decision:

   Choose a statement based on a given condition

3. Loop

   Repeat statements based on a given condition

Conditional Execution - if

- The if statement lets a program react in a particular way to data it receives
- This allows us to use metadata in our programs to make them more effective
  - The double glazing program could reject widths and heights that are incorrect
  - This will protect us from lawsuits..

Double Glazing Program

- We are going to consider a program we are writing for a customer
  - Read in height and width of window
  - Print out area and length of glass to buy
- This is in the C# Yellow Book
- Before we can write the program we need to go find some *metadata*

What is Metadata?

- Metadata is data about data
  - Limits (maximum and minimum values)
  - Units (measured in metres, gallons, years)
- It gives a proper context for what the program is doing
- There is always a question about metadata in the 08101 examination

Where does Metadata come from?

- It must come from the customer
  - They are the only people who can tell you about their business
- Only the double glazing salesman knows that he measure his windows in meters
- If you assume that he uses feet and inches you will supply a useless program

Getting Metadata

- You need to go out and ask the customer for this information
- They will not necessarily think to tell you
- Two assumptions that lead to disaster
  - Customer assumes you know the units
  - You assume the customer measures his windows in feet
- Result = **FAIL**

University of Hull

Double Glazing Metadata

```
/* Window sizes measured in meters
   Invalid values:
   width less than 0.5 metres
   width greater than 5.0 metres
   height less than 0.75 metres
   height greater than 3.0 metres */
```

- This is the metadata that drives our value inputs for the double glazing program
- I have written it as a comment
  - This is not accidental

UNIVERSITY OF Hull

Conditional Statement

```
if (condition)
   statement we do if condition is true
else
   statement we do if condition is false
```

- This is the general form of the C# conditional statement
- The condition is an expression that returns a boolean result

Relational Operators

```
2 * ( width + height ) * 3.25
```

- We have seen how a operators can be used in arithmetic expressions to produce numeric results

```
height > 3.0
```

- We can use relational operators in expressions to produce boolean results which are true or false

Testing the height upper limit

```
if (height > 3.0)
   Console.WriteLine ( "too high");
else
   Console.WriteLine ( "not too high");
```

- This C# test validates the upper bound of the height value
- Note that it doesn’t check for heights which are to small or negative

Missing off the else part

```
if (height > 3.0)
   Console.WriteLine ( "too high");
```

- If you don’t need the else part you can leave it out
- Whether you have an else part depends on what you are trying to achieve with the code
  - Don’t feel obliged to add one

Relational Operators

- You use relational operators to perform comparisons
- A relational operator works between two numeric operands
- It returns a boolean result which is either true or false

== operator

```
if ( age == 21 )
   Console.WriteLine ("Happy 21st");
```

- The == operator returns true if the two operands are equal
- Note that this is not the same as the = operator, which performs assignment

== operator and Floating Point

```
if ( average == 1.0f )
   Console.WriteLine ("Average of 1");
```

- Because floating point values can’t be held exactly it is very dangerous to compare them for equality
- The condition may be unreliable because of errors in calculation

== operator and strings

```
if ( name == "Rob" )
    Console.WriteLine ("Hello Rob");
```

- We can compare strings for equality
- The comparison is case sensitive
  - The string "`rob`" would not be recognised by the above code

The != operator

```
if ( name != "Rob" )
    Console.WriteLine ("Your are not Rob");
```

- The != (not equals) operator returns true if the operands are not equal to each other
- This can be used in the same way as the == operator

UNIVERSITY OF Hull

The < and > operators

```
if ( width < 0.5 )
   Console.WriteLine ("width too low");
```

- The < and > operators test for less-than and greater-than respectively
- Note that if the operands are equal the result is not true

The <= and >= operators

```
if ( width >= 0.5 )
   Console.WriteLine ("not too low");
```

- These work like < and >, but also include the case where the two are equal
- To invert a < you have to use a >=
- The code above inverts the previous test

The ! operator

```
if ( !false )
    Console.WriteLine ("not false is true");
```

- The ! operator (not) can be used to invert a boolean value
- It works on one operand

Combining Logical Operators

- Sometimes a program needs to combine a number of logical expressions
  - If the height is too wide or the height is too high
- C# provides operators that can be used in this way:
  - `&&` for logical **and**
  - `||` for logical **or**

Testing both height limits

```
if ( (height > 3.0) || (height < 0.5) )
   Console.WriteLine ( "Invalid Height" );
else
   Console.WriteLine ( "Height OK" );
```

- The Logical Operator OR `||` can be used to combine two conditions
- If one **or** other of the conditions is true the operator will return true

Inverting the Condition

```
if ( (height <= 3.0) && (height >= 0.5) )
   Console.WriteLine ( "Height OK" );
else
   Console.WriteLine ( "Invalid Height" );
```

- This test inverts the condition to return true if the height is valid
- Note we have to invert the conditions **and** change the logical operator

Creating Blocks

```
if ( width > 5.0 )
{
  Console.WriteLine ("Width restricted") ;
  width = 5.0 ;
}
```

- If we want to perform more than one statement after a condition we can put the statements into a block

Code Blocks

```
{
   /* any number of statements
      here */
}
```

- We have seen blocks before
  - The body of a method is a block
- The `{` and `}` define the limits (delimit) a block of statements

Blocks and Layout

```
if ( width > 5.0 )
{
  Console.WriteLine ("Width restricted") ;
  width = 5.0 ;
}
```

- I indent code which is inside a new block
- This makes the program much easier to understand
- I often use blocks when I just have one statement

Magic Numbers

```
if ( width > 5.0 )
{
  Console.WriteLine ("Width restricted") ;
  width = 5.0 ;
}
```

- The value 5.0 is a *magic number*
- It actually means “the largest width you are allowed to have”
- But this is not very clear to a reader

Magic Numbers

```
if ( width > MAX_WIDTH )
{
   Console.WriteLine ("Width restricted") ;
   width = MAX_WIDTH ;
}
```

- We can create a variable which contains the maximum width value
- If we use this it makes the code much clearer

UNIVERSITY OF Hull

Declaring Magic Numbers

```
const double MAX_WIDTH = 5.0 ;
```

- By adding const in front of the declaration we can make a variable that is constant
- This stops other programmers from changing the value and making the program misbehave

Magic Number Double Bonus

- Not only do magic number variables make the program clearer, but they also make it simpler to maintain
- If the customer wants us to change the maximum window width it is now very easy to do this, just by changing the magic number declaration

Summary

- Well written code contains comments
- Successful programs are based on Metadata
- Programs can make decisions using conditional statements
- Programs can use relational operators to compare values
- Logical expressions can be combined to create more complex decisions

Labs this Week

- The labs this week are very similar to the ones you did last week
- Except that we will be writing programs that make decisions
- The starting points are very similar to the programs we have already written
- But when the code runs it makes a choice