Topic 2 – Introduction to Programming and Sequence

HISTORY AND ORIGINS

- In this unit, we will be studying the C language.
- C was one of the first high-level languages that made it easy for programmers to write portable programs.

Advantages of C:

- It is easy to learn the basics.
- It is syntactically similar to many other languages that you may go on to study or learn later.
- It is relatively portable: works on many platforms.
- Programs written in C are very fast.
- It is a very powerful, flexible and still widely used.

Disadvantages of C:

- C imposes very few restrictions on what you can do: it doesn't stop you making your own mistakes!
    - Because of this, writing good and reliable programs in C requires discipline.
- It isn't as portable some languages, such as Java.
- To use it well, particularly its more advanced features, requires a good understanding of the underlying technical aspects of the computer's operation.
    - We will be sticking with the basics in this unit.

A “Hello World” Program

- It is quite common when learning a first language to write a so-called “Hello World” program.
- This is just a trivial program which prints “Hello World” to the screen and then exits.
- Although pointless from a functional point of view, it is useful because it demonstrates the basic syntax of C.

```
#include <stdio.h>

int main()
{
   /* Print the message and then a
      newline */
   printf(“Hello World!\n”);

   return(0);
}
```

Let's go through this program line by line...

```
#include <stdio.h>
```

This line is included in many C programs and is there primarily to allow us to use the input/output (IO) routines that C provides. In this case, `printf()` which prints things to the screen.

```
int main()
{ ... }
```

The code in a C program is split into sections called *functions*. All C programs have a so-called “main function”. This is where the running of the program begins and everything between the curly brackets is part of it.

```
 /* Print the message and then a new line */
```
This is called a “comment” and is there to tell people who read the code what various parts of the program do. Comments are very important to include in your code to allow other people to understand it and be able to potentially modify it to improve it or correct bugs. Everything between the `/*` and `*/` is treated as a comment and ignored by the compiler. Note that comments can stretch over many lines.

```
 printf(“Hello World!\n”);
```
Finally we have a program statement that actually does something! This prints the message “Hello World!” (without the quotation marks) to the screen. However it also includes a special `\n` character which means to print a new line. Note the semi-colon character `;` at the end of the line: this indicates that the program statement ends.

```
 return(0);
```
This tells the `main()` function to finish (the program exits). The zero simply means that the program exited normally without any errors.

Observations on C Syntax

- The rules that dictate how we write the code in a given language are referred to as the language's *syntax*.
- Even from the simple program above, we can learn a number of important syntactical rules relating to C.

- Firstly, apart from the `#include` declaration, all of the lines that make up the program are included within the `main()` function.
- That is, the program statements are all between the curly brackets: `{ ....  }`

- The second thing we can observe is that each program statement must end with a semicolon or `;` character.
  - Note that this doesn't apply to every physical line in the program.
  - It just applies to lines that actually do something (represent instructions).
  - Lines such as `int main()` don't actually do something so do not need a semicolon.
  - Also lines which begin with `#` like `#include <stdio.h>` are called “pre-processor directives” and aren't technically considered C statements.
  - Finally comments do not require a semicolon.

Code Layout

- Another thing you may notice is the careful indented structure of the code.
- Although it doesn't affect the execution of the code, indenting is *critically* important when writing programs because it assists the reading of that code later on.
- The pattern of the indenting identifies the structure of the program.
- Indentation and code style make up a significant part of your marks in this unit.

- The general rule you should apply is to always indent after an opening curly bracket `{`
- You then "unindent" when the matching closing curly bracket `}` occurs.

- Example:

```
int main()
{
   printf("Welcome to the program!");

   return(0);
}
```

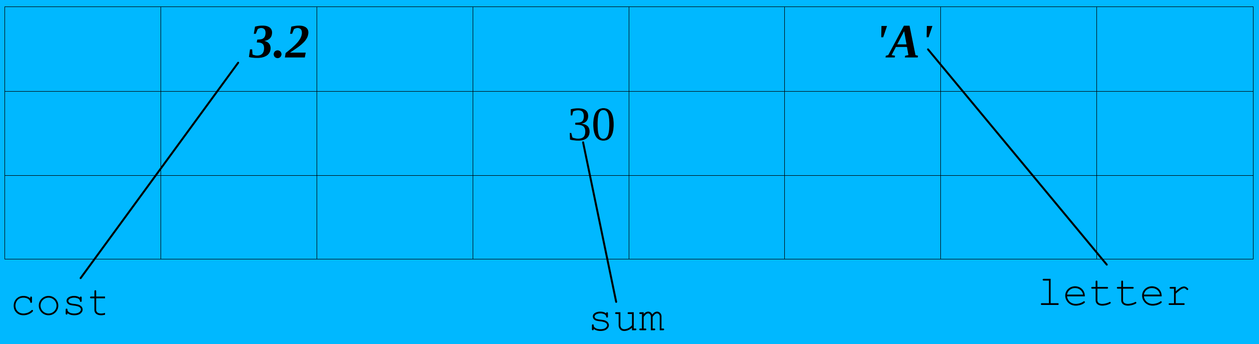
- Also note the line spacing: between the `printf()` and the `return` there is a blank line.
- Although perhaps less critical than indenting, this shows that the two statements are involved with doing separate things.

- So a collection of perhaps 3-5 statements that are all involved with one small step in your program should be together with a blank line between them and the next "mini-block" of code.

VARIABLES

- For a program to be useful, it must be able to store and manipulate data.
- Modern programming languages let us allocate small areas of memory in which to store our data.
- Each of these bits of memory are then given a name by which we can refer to it in the program and are generally called *variables*.

The following is a diagram illustrating this concept:



The variables are all given names (`cost, sum, letter`) and they refer to an area or cell of memory where the data is stored.

Also note that many cells aren't being used and so aren't given names. The contents of these cells are unknown.

- Variables are an important concept in programming.
- However, they are also important when writing algorithms in order to consistently identify the piece of data the algorithm is referring to.
    - When writing algorithms, always give suitable names to your pieces of data (variables) so you can refer to them later on.

Variable Names

- The choice of the right name for a variable is important.
- For *some* variables that have very limited use, you might just choose a short name:

```
int i;
```

- However, most of the time you should choose a name that helps to describe the purpose of the variable.
- The name shouldn't be too long though.

- Programming languages impose a set of restrictions on variable names and, although these restrictions do differ, they are mostly pretty similar.

In C you **must** choose variable names that:

- Do not begin with a number.
- Do not contain any punctuation characters (except for an underscore) or whitespace (space, tab, new line etc.)
- Are not so-called “reserved words” which are part of the C language and therefore already have meanings.

- You also cannot have two variables with the same name in the same part of the program.

DATA TYPES

- Of course, not all data is the same (see above diagram).
- Data comes in many different types and C provides a set of built-in ("primitive") data types for this purpose.

Numeric Types

- Here is a program which demonstrates the use of the two main numeric data types, `float` and `int` plus some other important C concepts.

```
#include <stdio.h>

int main()
{
   int x = 10;
   int y = 5;
   int total;
   float avg;

   /* Calculate total and then average */
   total = x + y;
   avg = total / 2.0;

   printf("Total is %d\n", total);
   printf("Average is %f\n", avg);

   return(0);
}
```

Again let's go through this program.

```
 int x = 10;
 int y = 5;
```

These lines create two new variables (data storage areas) in memory called `x` and `y` that both store whole or integer numbers, which C calls `int`. The allocation of memory storage for these and the assigning of the names to them is called *declaring the variables*. **Variables must be declared before being used and this must be done at the top of the function in which they appear**. However, these variables also have initial values assigned to them and this is called *initialisation*.

```
 int total;
 float avg;
```

These are two other variables being declared. The first called `total` is an integer, however, it is not initialised. This means it will have *an unknown (random) initial value*. The second variable `avg` is a so-called "floating-point number", which is a number with a fractional part. It too is not initialised.

```
 total = x + y;
 avg = total / 2.0;
```

The first line is an assignment statement similar to the initialisation of the other variables. It adds `x` and `y` to one another and then assigns the result to `total`. The second line divides `total` by two and assigns the result to `avg`.

- Note that this is so-called "floating point" division rather than integer division which is why "2.0" is used rather than "2".
- If you are dividing two integers but want the result to be a float then you need to force one of the numbers to be a float.

```
printf("Total is %d\n", total);
printf("Average is %f\n", avg);
```

These are just like the `printf()` statements before but this time they output the value of two variables. The `%d` in the first statement tells `printf` to expect a decimal (integer) value, in this case `total`. The `%f` tells the second `printf` to expect a floating point number, namely `avg`.

The `char` data type

- Apart from storing numbers (both integral and fractional), you can also store character data.
- This includes alphabetics, numbers and symbols.

- Characters are stored using the ASCII system, whereby each symbol is assigned a number and this number is stored by the computer.
- This is required because computers can only store and process numbers.

- The following is a subset of the ASCII table:

32 = (space)	33 = !	34 = "	35 = #	36 = $	37 = %
38 = &	39 = '	40 = (	41 =)	42 = *	43 = +
44 = ,	45 = -	46 = .	47 = /	48 = 0	49 = 1
50 = 2	51 = 3	52 = 4	53 = 5	54 = 6	55 = 7
56 = 8	57 = 9	58 = :	59 = ;	60 = <	61 = =
62 = >	63 = ?	64 = @	65 = A	66 = B	67 = C
68 = D	69 = E	70 = F	71 = G	72 = H	73 = I
74 = J	75 = K	76 = L	77 = M	78 = N	79 = O
80 = P	81 = Q	82 = R	83 = S	84 = T	85 = U
86 = V	87 = W	88 = X	89 = Y	90 = Z	91 = [
92 = \	93 =]	94 = ^	95 = _	96 = `	97 = a
98 = b	99 = c	100 = d	101 = e	102 = f	103 = g
104 = h	105 = i	106 = j	107 = k	108 = l	109 = m
110 = n	111 = o	112 = p	113 = q	114 = r	115 = s
116 = t	117 = u	118 = v	119 = w	120 = x	121 = y
122 = z	123 = {	124 = |	125 = }	126 = ~	

- In C, characters are stored using the `char` data type:

```
char letter = 'a';
...
letter = '5';
```

Take careful note of what is happening here!

You can use the table above to see what value is actually being stored.

The character variable `letter` stores the lower-case letter `'a'` (97) .
However, later the letter `'5'` (53) is assigned to it.

At all times it remains a `char` and not an `int` because, to the computer, character and number data types are completely different things.
This is indicated by the use of the `char` data type and also the single quotation marks, e.g. `'5'`.

So you cannot perform mathematical calculations on this `char` variable and expect to get correct results!

If you want to do any calculations then you should use a numeric type rather than a `char`.

Summary of C Data Types

These are some of the common basic C data types:

Type	*Example*	*Description*
`int`	`int n = 5;`	Integer (whole) number type.
`float`	`float length = 3.5;`	For “floating-point” or fractional numbers.
`char`	`char last = 'Z';`	For single alphanumeric and symbolic characters (ASCII).

Constants

- Quite often you will find that there is a value you use in your program that isn't going to change at all.
- Most programming languages, including C, allow you to define a *constant* which gives the unchanging value a label by which you can refer to it throughout the program.

- An example of this in C could be:

```
 const float PI = 3.14;
```

- Note that it is important to specify the type of a constant, just as for variables.
- Note that constant names are always written in ALL CAPITALS to allow them to be easily distinguished from variables when reading the code.

- There are two reasons for using constants.

- The first is that assigning a name to a value helps identify what the value means.
- So someone reading the program doesn't just see some value and then have to work out what it refers to.

- The second is that if you define a constant and then use it consistently whenever you need that particular value in your program, it makes modifying your program much easier.

- For example, you write a program to calculate the circumference of a circle using the value of pi above.
- But then you find out that the value you used for pi is not precise enough – you need more decimal places.

- You can simply edit the value of pi where it is defined as a constant at the start of your code and your program is now ready to go.

- If you hadn't used a constant you would need to find and correct every single usage of the value throughout your program.

- In a complex program this can be a big and error-prone task.

Type Casting

- It is possible to convert data from one type to another and this is called *type casting*.

- The general syntax of type casting is to put the type that you want to cast the variable to in brackets in front of the variable that you want to cast.

- Here is an example of casting an `int` to a `float` and then back again:

Type to cast the data to.

Variable who's value yo want to cast to another type.

```
#include <stdio.h>

int main()
{
   int x = 5;
   float y;

   y = (float) x;

   printf("The value of y is %f\n", y);
   y = y*y;

   x = (int) y;
   printf("The value of x is %d\n", x);


   return(0);
}
```

Type Casting `char` Data

- Remember that character data is stored as a number since this is all that computers can store.
- Which number represents which character is determined by the ASCII system.

- Therefore an important use of type casting is to determine the ASCII value of a given character.
- If you cast a `char` variable to an `int` then this will give you the ASCII value for that character.

```
char letter = 'A';
...
printf(“ASCII value for letter %c is %d\n”,
   letter, (int) letter);
```

Here we print out the variable `letter` which has the value `'A'` both as a character and then cast to an integer.

Note that `%c` tells `printf()` to expect a character variable.

ARITHMETIC OPERATORS AND ASSIGNMENT

- C has a set of so-called “operators” built into it that are very similar to most other modern programming languages.
- Operators “do something” on one or more pieces of data.

Arithmetic Operators

- These are the basic arithmetic operations of addition, subtraction, multiplication and division.
- However, there is another operator called modulus that calculates the remainder left over when doing integer division.

Operator	***Example***	***Description***
+	`var1 + var2`	Adds `var1` and `var2`
-	`var1 - var2`	Subtracts `var2` from `var1`
*	`var1 * var2`	Multiplies `var1` by `var2`
/	`var1 / var2`	Divides `var1` by `var2`
%	`var1 % var2`	Computes the remainder of dividing `var1` by `var2` (Modulus)

- The division operator requires special attention.
- As indicated before, a different division occurs depending on the types of the variables concerned.
- If both are integer then integer division occurs and the result will be a whole number – any fractional part will be thrown away and lost!
- If either operand is a float then the result will also be a float.

Assignment

- There's no point performing a calculation if you can't do something with the result.
- (However, many people do this by accident when first learning to program!)

- Nearly always you will want to assign the result of the calculation to another variable.
- We've already seen this in the example programs.
- Note that initialisation of a variable when it's declared and assigning of a value are very similar things, they just happen at different times.

The example from before:

```
avg = total / 2.0;
```

Note that this is floating point division, even though `total` is an integer.

Incremental Assignment Operators

- Incrementing (that is, increasing by a fixed amount) is a common operation.

- For example, if we need to count how many times a certain event happens in a program then it is common to create an integer variable to do this.
- This variable will be initialised to zero:

```
int count = 0;
```

- Then, as the program progresses, each time the event happens we can increment this counter:

```
count = count + 1;
```

- By the time the program finishes, the value of `count` will be the number of times the event has occurred.

- Because this is such a common operation, many programming languages, including C, include special operators to do this.
- So, instead of the above we could write:

```
count++;
```

- The two are identical in their behaviour.

- Note there is also a decrement operator too:

```
num_left--;
```

Other Assignment Operators

- Because programmers don't like to type anymore than they absolutely have to, there are a set of combined assignment and arithmetic operators!

These are:

Operator	*Example*	*Shortcut For*
`+=`	`var1 += var2`	`var1 = var1 + var2`
`-=`	`var1 -= var2`	`var1 = var1 - var2`
`*=`	`var1 *= var2`	`var1 = var1 * var2`
`/=`	`var1 /= var2`	`var1 = var1 / var2`
`%=`	`var1 %= var2`	`var1 = var1 % var2`

- They look confusing at first but are actually pretty simple!

Operator Precedence

- Because different operators can be combined, there are rules to decide in which order these operations apply.
- Get the order wrong and you will nearly always get the wrong result.

- In C there are 18 different precedence rules.
- But you only need to learn three!

1. Multiplication and division come first (as they always do...)
2. Put parentheses (brackets) around everything else.
3. If you have any doubt, put brackets around the stuff you want to happen first.

INPUT/OUTPUT

Simple Output: printf()

- A program is no use if it cannot somehow make available or “output” its results to the user.
- This is most commonly done by printing the results to the screen.
- C has a simple built in function called `printf()` that allows this.
- We have already seen some examples of this in the programs above but now we will look at it in more detail.

- Firstly the `printf()` function takes one or more of what are called *parameters*.
- These are variables or data supplied to the function when it is used or “called” and are separated by commas.

- The first parameter tells `printf()` how to format the output it produces (called a “format specifier”).
- The zero or more remaining parameters are the pieces of data that `printf()` needs to construct the properly formatted output.

- A simple example of this with just one parameter comes from our “Hello World” program:

```
printf(“Hello World!\n”);
```

- The parameter here is just to be treated literally and printed to the screen without any additional formatting.

- However, as we've already seen `printf()` can process different data types and output these too:

```
printf("Total is %d\n", total);
printf("Average is %f\n", avg);
```

- Both of these `printf()` statements have two parameters, separated by the comma.

- The first parameter for the first statement tells `printf()` to output the string "Total is " and then to output an integer decimal number (`%d`) and a newline (`\n`).
- The value from the second parameter, `total`, is then used as that decimal number to be output.
- So if `total` was 15 then `printf()` would output "Total is 15" then a newline.

- The second statement is the same except it tells `printf()` to expect a floating point number (`%f`) which will be the value of the variable `avg`.

- Although the two statements above are written separately, we could easily combine them into one with three parameters:

```
printf("Total is %d\nAverage is %f\n",
   total, avg);
```

- This demonstrates how `printf()` works when many different values are being printed at once.

Input using scanf()

- While output is important, a program that cannot obtain any data from the user is almost as useless as one which produces no results.
- By being able to obtain data while it is running (“at runtime”), a program can be written to solve a general type of problem and then be applied to specific instances of that problem simply by varying the input data each time it runs.

- There is a similar function to `printf()` called `scanf()` but this function is used for inputting or reading in data.
    - One key difference between `printf()` and `scanf()` is that the format specifier used determines the type of the data that is being read in.
    - Also, **the format specifier must have `%*c` after it** for reasons explained later on.
    - The variable into which the data being read in will be placed must also be specified prefixed by an ampersand.
    - An example will help to make this clear.

- Below is a modification of our previous program for calculating the average of two numbers.
- But this time the numbers aren't “hard-coded” into the program.

- Instead the user inputs these numbers to the program each time it is run.
- Thus the same program can calculate the average of any two given numbers without being specially modified and recompiled each time.

```
#include <stdio.h>

int main()
{
   int x, y;
   int total;
   float avg;

   /* Get data from user */
   printf(”Input first number: “);
   scanf(“%d%*c”, &x);

   printf(“Input second number: “);
   scanf(“%d%*c”, &y);

   /* Calculate total and then average */
   total = x + y;
   avg = total / 2.0;

   printf(“Total is %d\n”, total);
   printf(“Average is %f\n”, avg);

   return(0);
}
```

- One important thing to notice about the use of `scanf()` is that the format specifier (the first parameter) contains the correct format character (`%d` in this case) for the data type being read in plus the `%*c` string.
- The other thing is the presence of the ampersand (`&`) in front of the variable.
- This tells the C compiler that `scanf()` is going to put a new value into this variable, in this case the value that it reads in from the user.

- Because `scanf()` is so similar to `printf()` it is generally simple to use.
- However, it is important to be very careful about its syntax.
- In particular, many beginning C programmers forget the ampersand and end up with programs which crash or behave strangely!

Issues with scanf()

- You will have noticed that `scanf()` works a little bit differently to `printf()`.
- Specifically the need for the ampersand (`&`) in front of the variable name and the inclusion of `%*c` with every format specifier, regardless of what type it represents.

- The first issue relating to the & is a little difficult to explain without referring to some quite advanced concepts which we do not cover in this unit.
  - Therefore, it is best if you simply accept this on faith!

- However, the requirement to put `%*c` after every format specifier is slightly easier to explain.
  - It is not essential to know the explanation though and this won't be examined.

- The problem stems from `scanf()`'s need to know exactly what data it is dealing with.
  - If the user inputs the wrong data or there is extra data that `scanf()` is not expecting then things go wrong.
  - Unfortunately the way `scanf()` handles the newline character when the user presses the Enter key means there often is extra problematic data.

- Input to the computer via `scanf()` doesn't involve data being sent immediately to the program as you might expect.
- Instead, each key press is put into a queue where it waits until the program is ready to process it.
- However, when you enter a piece of data (of any type) you always press Enter after and this goes into the queue as well.

- Since that Enter key is counted as an input character then this must be processed by the program too.
- If it is not, then this key press remains in the queue until the program next goes to read some data from the keyboard via `scanf()`.

- When this happens, instead of the program getting the data it expects, it gets the Enter key press instead and the intended input data simply remains in the queue.
  - Therefore, `scanf()` essentially gets out of sync with the input data coming from the keyboard.

- The obvious solution to this is to simply process the Enter key every time a piece of data is read from the keyboard with `scanf()`.
  - This is what `%*c` does and why it needs to be included after the format specifier for the data you want to read.

- If you don't do this then some of your input lines may be skipped or you will read in apparently “empty” lines of data.
- However, even if you do this, if the user then inputs the wrong type of data etc. then `scanf()` will still get confused so this is not a perfect solution.

FROM ALGORITHM TO C PROGRAM

Simple Algorithm

Here is a simplified algorithm to calculate the total cost of a product including GST:

- Print “Enter price of product ex-GST: ”
- Read user input into *price*
- *total* = *price* * 1.1
- print “Total price is: ”, *total*

Note a big limitation of this algorithm is that it doesn't do any error checking (i.e., checking for negative results) or any rounding of the results to two decimal places.

Problem Definition

Algorithm Outputs:

- Total price

Algorithm Inputs:

- Price of the product (ex-GST).

Assumptions:

- GST is 10%.
- Only positive values will be supplied.
- No rounding of result is required.

Transition to Code

- Although this problem is quite simple, the problem definition still clearly defines an important piece of information we need to write a program.

- And that is the data that the program needs to deal with, in this case the total price and the original price (the outputs and inputs respectively).
- This tells us what variables we will need to declare as part of our program.
- There may be other variables that you need to create as you write the program but these will be minor, temporary variables which can be easily added as required.

- Another important piece of information the problem definition gives us is that we assume the GST is 10%.
- This is an important assumption because, if it is not true, our algorithm (and subsequent program) will become useless.
- Since the rate of the GST will not actually change *while* the program is running, this tells us that we should define a *constant* using `const` to specify the GST rate.

- This also means that if the GST rate does change, we can easily update our program simply by changing the value of the constant and without requiring any other changes.
- This is only a minor advantage for a program this simple but it becomes very important when dealing with larger programs.

Code Foundation

- Before we can actually write the code to implement our algorithm, we have to deal with some minor requirements of the language we're using, in this case C.

- All code in C must be part of a function so for this simple program we will put it all in the `main()` function.

```
#include <stdio.h>

int main()
{
   /* The code can go in here */

   return(0);
}
```

Data Declarations

- Now we can declare the data (variables and constants) that we will use.
- Remember we can take this information directly from the problem definition.

```
int main()
{
   /* GST is 10% */
   const float GST = 0.1;

   /* Float variables for price and total
   price including GST */
   float price, total;


   return(0);
}
```

Filling in the Code

- For this very simple case we can just write the code directly from the algorithm.
- For more complex problems we will later have to adopt alternative strategies.

```
int main()
{
   /* GST is 10% */
   const float GST = 0.1;

   /* Float variables for price and total
   price including GST */
   float price, total;


   printf("Enter price of product ex-GST: ");
   scanf("%f%*c", &price);

   /*   Calculate total price by multiplying
        ex-GST price by GST percentage + 100%
   */
   total = price * (1+GST);

   /* Result is not rounded to 2 decimal
         places */
   printf("Total price is: $%f\n", total);

   return(0);
}
```

ALGORITHM AND PROGRAM TESTING

OVERVIEW

- When you first start writing programs you might find that it takes a little while before you are able to get a program that will compile.
- However, when the program does compile this often does not mean that it is finished!
- Instead there will probably still be bugs in the program meaning that it will not always work as it is supposed to.

- Therefore it is important to test your program carefully so you can find any bugs that are there and remove them.

- The overall process for accomplishing this works more or less as follows:
    1. Identify a list of test data that covers the boundaries of the likely inputs that your program will have to deal with.
    2. Identify how you expect your program to respond to these inputs.
    3. Work through your algorithm using a pen and paper putting this test data into the algorithm and seeing how it behaves. This process is called *desk checking*. If the results from your algorithm are as you expected (from Step 2) then you algorithm is correct.
    4. Apply the same test data to your program once you've written it: if you get the same results as your algorithm, then your program is correct.

Test Tables

- A *test table* is a common way of showing test output from your program. Below is an example:

Test Description	Inputs	Expected Outputs	Algorithm Outputs	Program Success/Failure
…..				

- *Test Description*: this is a very brief outline of what this particular test aims to demonstrate.
- *Inputs*: a list of the inputs relevant to this particular test. **More information on how to select appropriate inputs will be found in the notes in Lab 2**.
- *Expected Outputs*: what you believe the correct results should be for this particular test.
- *Algorithm Outputs*: the results you get after desk checking the specified inputs *against the algorithm.* Sometimes this is labelled "Actual Outputs", however, it *still* refers to the result of the desk check of the algorithm and not the code.
- *Program Success/Failure*: does the output from the program match the output from the algorithm? Note that this assumes the algorithm performed correctly.

- In addition to filling in the test table for your program, for major programming projects (e.g., your assignments) you should also provide some copies of sample output from your program.
- Ideally these should cover all of the relevant test data from your test table.

CODING STYLE

Introduction

- Coding style is one of the most important issues in programming.
  - The way that the code is laid out has a massive impact on how easily it can be read and understood, both by the original programmer and by others.

- However, it is also one of the most poorly done things when it comes to beginner programmers.
  - This is partly due to the fact that there is a lot to learn all at once and so some things get forgotten.
  - But it is also because beginner programmers often fail to appreciate its importance.

- In this unit coding style has a significant impact on the marks you will receive for all your submitted programs.
- In general you can expect approximately 25% of these marks to relate to coding style.

- This is because the most useful code is not necessarily code that works correctly but that which can be easily understood.

- Code that works but is totally incomprehensible will probably have a short life because sooner or later (usually sooner) it will have to be edited and changed.
  - If it can't be understood then it will probably be thrown out and the programmer will start again from scratch.

- On the other hand code that isn't quite correct but is easily understood can be quickly fixed and improved.

Commenting

- Commenting is one of the biggest aspects of coding style as it tells the reader what the code does.

- As indicated comments in C begin with `/*` and end in `*/`
  - Anything between these will be ignored by the compiler.

- However, the way you use comments is very important.
- Comments should tell the reader what the code does.
- However, you *can* assume that the reader understands the language the code is written in!

- So comments should not spell out what is obvious to someone who understands the language.
- For example the following are **pointless** comments:

```
/* Initialise the integer to zero */
int i = 0 ;

/* Add up the numbers */
i = a + b;
```

- Do not waste your time typing these sorts of comments!
  - In particular, don't try and comment every line.

- **Think about what parts of the code you might want help understanding if the code belonged to someone else and you were reading it.**

An example of a more useful comment might be:

```
/*   Calculate total price by multiplying
     ex-GST price by GST percentage + 100%
*/
total = price * (1+GST);
```

- This may mean that, contrary to popular opinion, your programs *may* require relatively few comments.
    - Carefully thought-out comments in the required areas are far better than useless verbose comments.

- Your comments should also not be too “low-level”.
- Describing each individual step in a tricky algorithm is probably not going to help the reader because they will already understand the individual steps, just not how they fit together to accomplish the task.

- It's usually best to aim your comments at a level above this.
- Say both what this part of your code does and, in general terms, how it does it.

- It is also common practice to write “header comments” where at the start of a program (or program module – Topic 6) you write a few lines briefly describing what the program does.
    - Most people often include a date, the filename of the program and their name.

- These sorts of comments are a good idea and are expected as part of this unit.
- However, you will not see these sorts of comments in the programs in these lecture notes in order to save space.
    - But you should include them in your own programs!

Variable Naming

- Variable naming is a subtle aspect of coding style that is often neglected.

- Your variable names should tell the reader of the program about the data being stored.
  - A good variable name means you shouldn't need to add a comment explaining what the variable is for.

- The principal for choosing a variable name is that the name should be as descriptive as necessary and a short as possible.
- Generally, names should be no more than 10 characters.
- Feel free to use abbreviations but try to keep them self-explanatory.
  - If they aren't then you should add a comment at the point where you declare the variable.

- There are usually variable naming conventions that apply to each language you use so you should adhere to these also.

- A convention common to most languages is to start the name with a lower case.
- Since spaces cannot be used sometimes mixed-case or underscores are used to “separate” words, e.g.:

```
int myNum;
int my_num;
```

- So called “throw-away” variables that have limited uses as temporary variables or “counters” (Topic 5) are often given single letter names and this is perfectly acceptable:

```
int i;
```

Indenting and Whitespace

- Another issue relating to coding style is the use of so-called whitespace, including indentation.

- Indentation plays a vital role in allowing the program's structure to be easily identified when its code is being read.
- This has only limited impact on the very simple programs we've seen so far, however, in the next topic you will begin to see how important it is.

- The general rule to remember is to always **indent after** an opening curly bracket and to **unindent before** the *corresponding* closing curly bracket.
  - This marks out “blocks” of code that all relate to one particular part of the program.

For example:

```
int main()
{
	printf(“Notice the indentation?\n”);

	return(0);
}
```

- The indentation should always be by a fixed amount.
  - The easiest way to do this is using the TAB character as this is quick and ensures that the spacing is even.
  - This is the technique you should use in this unit.

- Finally use whitespace in the form of blank lines in your program to indicate “sub-blocks” of code, that is a small number of lines which all relate to the one simple task.
- Examine the GST program for the previous section for examples of this.