**261205 Data Structures and Algorithms**

**Week 1**

**C++ Basics**

**Topics:-**

**1.0 Introduction…………………………………………………………….. 2**

**1.1 Variables………………………………………………………………… 2**

**1.1.1 Variable Types………………………………………………... 4**

**1.1.2 Variable Declaration…………………………………………. 5**

**1.1.3 Variable Assignment…………………………………………. 5**

**1.1.4 cin & cout……………………………………………………... 6**

**1.2 Control Structures……………………………………………………... 6**

**1.2.1 Control Structure 1 – Branching…………………………… 6**

**1.2.2 Control Structure 2 – Looping……………………………… 8**

**1.3 Programming Style…………………………………………………….. 9**

**1.4 Functions……………………………………………………………….. 10**

**1.4.1 Programming Defined Functions…………………………… 11**

**1.4.2 Pre-defined Functions……………………………………….. 12**

**1.4.3 Variable Scope……………………………………………….. 13**

**1.4.4 Overloading Function Names……………………………….. 14**

**1.4.5 *void* Functions………………………………………………… 15**

**1.4.6 Call By Reference vs Call By Value………………………… 16**

**1.0 Introduction**

In this course we will be implementing our data structures and algorithms using C++. Why is C++ called C++? (I hear you ask…) There was a programming language called **BCPL** (Basic Combined Programming Language) developed in the 1960’s and it became referred to as ‘**B**‘. Although there was no predecessor called ‘**A**’, when a subsequent program was developed (in the 1970’s) it was called ‘**C**’. ‘**C**’ is a low level language, but once again was enhanced and then named ‘**C++**’ and that is what we’ll be using in this course. Why wasn’t it called ‘**D**’? Well, as you may know ‘++’ is an operation within the language which means to increment (i.e. add one), so it was a nice pun!

A C++ program contains lines of instructions, which can be written in any text editor, ***Notepad*** for example. Rather than using notepad, more sophisticated text editors, such as ***Ultra Edit*** have features which make your code easier to read, by highlightingknown commands in different colours. After writing your code in C++, a ***compiler*** is needed to translate your instructions into machine language for the computer to understand. You may use any compiler / environment you like for the lab associated with this course, but I will be using Visual Studio. You should already be familiar with programming in C++, so this basic guide is to make sure you are up to speed.

* 1. **Variables**

Remember computers have ***input devices***, ***output devices***, a ***processor*** and***memory***. Most programs are designed to manipulate data, to *take data inputs, process them and produce outputs*. Often while this is taking place, pieces of data have to be stored for later use – this is where the *memory* comes in. When the user inputs some data it needs to be stored in the memory, while the processor is manipulating this data it often needs to temporarily store values in the memory and when the output is being made ready, this too often needs to be stored in memory. To store data / information in memory, we use ***variables***, hence being able to use ***variables*** is a key fundamental of programming.

The memory is ‘*a set of locations where data can be stored’* and that data is stored, in binary, in a memory location of one or more bytes. These memory locations can be thought of like Pigeon Holes or mailboxes are in the physical world.

|  |  |
| --- | --- |
| [pigeon-holes](http://cis.payap.ac.th/wp-content/uploads/2009/11/pigeon-holes.gif) | Pigeon Holes are called ‘*pigeon holes*’ because they resemble the cages that racing pigeons are kept in.  They are a set of named (or numbered) locations where messages can be left, and collected.  ***Memory*** on a computer is similar, a set of named locations, where data can be left, and collected. |

Programmers use variables when placing data into these memory locations. A ‘***variable***’ is a named location where we can store data. Each variable is given a ‘***name***’, which is the name we use to refer to the memory location – in order to leave data, or collect data. You can choose to name your variable using letters, digits or an underscore ‘\_’, BUT, it is a good idea to name your variable something sensible, related to its purpose. For example, if you have a variable to store the interest rate, it is a good idea to name it ‘interest\_rate’.

|  |  |
| --- | --- |
| ***VARIABLES – NAMING CONVENTION*** | |
| [Yoda](http://cis.payap.ac.th/wp-content/uploads/2009/09/Yoda.jpg) | “Use a sensible naming convention” |

Here are some different variables, and why they may or may not be valid.

|  |  |  |
| --- | --- | --- |
| x1 | valid | BUT – not good, as no logical meaning |
| 1x | invalid | Variable names cannot start with a digit. |
| \_abc | valid | But, they can start with an underscore (this variable has no logical meaning either though) |
| %change | invalid | Only letters, numbers or underscores allowed. |
| data-1 | invalid | Only letters, numbers or underscores allowed. |
| Annual\_Bonus | valid | Good! |
| annual\_bonus | valid | Good – but this is a different variable to the previous one, as names are case sensitive. |

**1.1.1 Variable Types**

There is one key difference between variables and pigeon holes though – variables have a ‘***type***’ of information that they can contain. While you can put anything in a pigeon hole, there are limitations on what you can put in variables as there are different types data and we need to instruct the computer what type of information to expect. One reason for this is that arithmetic operations can’t be done on an input that has letters (you can’t do *“ken” \* 3)*, so to avoid problems like these, the type of data that the variable will contain needs to be declared explicitly, and then only one type of data may be stored in that variable (memory location).

Variables which contain letters are called ‘strings’ – we will deal with the string type of variable later, but for now we shall concentrate on digit variable types – there are several different options;

|  |  |  |  |
| --- | --- | --- | --- |
| Type Name | Syntax Memory | Syntax Range | Syntax Precision |
| short | 2 bytes | -32,767 – 32,767 | (N/A) |
| int (or long) | 4 bytes | -2,147,483,647 – 2,147,483,647 | (N/A) |
| float | 4 bytes | 10e-38 – 10e38 | 7 digits |
| double | 8 bytes | 10e-308 – 10e308 | 15 digits |
| long double | 10 bytes | 10e-4932 – 10e4932 | 19 digits |

Notice the first two types, ‘***short***’ and ‘***int***’ deal with whole numbers – with no decimal places / fractions. These are called ‘***integers***’ and ‘***int***’ is short for integer. The other three types deal with numbers with decimal places or fractions, these are called ‘***floating point***’ numbers and ‘***float***’ is short for floating point. There is sometimes a problem with accuracy when programming with Floating point numbers and many advise to avoid them whenever possible. ‘***double***’ and ‘***long double***’ offer further precision over the ‘***float***’ type. Notice ‘***double***’ uses double the memory of ‘***float***’ (which is sometimes called ‘single’) and provides over twice the accuracy. Avoid using the wrong type of variable, as this will often cause a ‘***Type Mismatch***’ error, for instance if you tried to store a number with decimal places in an integer variable.

It is worth introducing 2 further types at this stage;

‘***char***’ is a type that can store nonnumeric characters, such as letters or symbols. ‘***char***’ can hold any single character from the keyboard – a single character string.

‘***bool***’ is a type, short for Boolean. A boolean type can be either ‘***true***’ or ‘***false***’ and is named after George Boole, who create rules for mathematical logic.

**1.1.2 Variable Declaration**

Before we use a variable, we have to declare it – that is inform the computer that we intend to use a variable, therefore, very often variables are declared at the start of a function, for instance the main function.

You declare variables using the syntax;

*Type\_Name Variable\_Name\_1, Variable\_Name\_2, . . . ;*

For Example;

int count, number\_of\_fish;  
double distance;

These statements will create 3 variables, two of type ‘***int***’, and one of type ***‘double***’.

**1.1.3 Variable Assignment**

Once Variables have been declared, the next stage is to assign them a value to store. The syntax of an assignment statement uses the ‘***=***’ sign. This is not used as in standard math – rather than stating that both sides of an equation are equal, in programming, we use it to set the left hand part of the statement to equal the right hand side;

*Variable = Expression;*

For Example;

count = 10;  
number\_of\_fish = 5;  
distance = 5.3;

The ‘***expression***’ part can be an equation, for instance;

count = count + 1;  
distance = number\_of\_fish \* count;  
tax = income \* tax\_rate;

It is possible to initialise (or assign a value) to a variable during the declaration statement;

int count = 10, number\_of\_fish = 5;

Notice that when using the variables, they do not have “” around them. When displaying the contents of a variable as output, you can simply use the variable name, for example;

***cout << count;*** //displays ’10’

***cout << “count”;*** //displays the word ‘count’

**1.1.4 cin & cout**

From the above, cout can be used to output to the console window. cin can be used to get input from the keyboard. Both cin and cout are in the iostream library, which needs to be included if you want to use input and output.

#include <iostream>

using namespace std;

void main()

{

int age;

cout << “How old are you?”;

cin >> age;

cout << “You are “ << age << “ years old!”;

}

**1.2 Control Structures**

Control structures are used to make programs more interesting and respond in different ways at different times. They include “Branching” and “Looping”.

**1.2.1 Control Structure 1 – Branching.**

Often we need the program to react differently under different circumstances – for example, you may want to write some code which states if the cube is too big or too small – this is known as branching, one branch for each option. When programming, the simplest way of dealing with branching is using an ‘***If-Else***’ statement.

|  |  |  |  |
| --- | --- | --- | --- |
| ***IF*** condition is true  ***Do This***  ***Else***  ***Do That*** | ***If***(Boolean\_Expression)  Yes\_Statement  ***Else***  No\_Statement | ***If***(The box is too small)  Provide some output to say “The box is too small”  ***Else***  Provide some output to say “The box is not too small” | if(size < 30) cout << “Cube too small!”;  elsecout << “Cube ok!”; |

The above case shows 1 line of code for each scenario, if we need to perform several tasks in each case, we surround the tasks in ‘***{***‘ and ‘***}***’

if (size < 30)  
{  
 cout << “Cube too small!”;  
 Do something else…;  
}  
else  
{  
 cout << “Cube ok!”;  
}

When using an if statement the response should be ‘true’ or ‘false’, i.e. Boolean. There for we always need to include a comparison operator, such as;

|  |  |  |
| --- | --- | --- |
| Math Symbol | English | C++ Notation |
| = | equal to | = = |
|  | not equal to | ! = |
| < | less than | < |
|  | less than or equal to | <= |
| > | greater than | > |
|  | greater than or equal to | >= |

Notice that for equals ‘= =’ is the C++ Notation – remember we use a single ‘=’ symbol for variable assignment, so for comparison we need to use 2.

Within an If statement we may wish to check for more than one thing – (is the size too small and the object too large?) For this we can use;

|  |  |
| --- | --- |
| && | AND |
| || | OR |

Another option is to use ‘***Nested If statements***‘. For nested if statements, we can insert a second if

statement within the first, so after the computer checks one, it checks the next one, for instance;

if (size < 30)

{  
 cout << “size is less than 30″;  
 if (size < 20)  
 {  
 cout << “Size is far too small”;  
 }  
 else  
 {  
 cout << “But it’s more than 20, so be careful”;  
 }  
}  
else  
{  
 cout << “No Probs”;  
}

In this code, the computer checks if the size is less than 30. If it is less than 30, it then checks if it is less than 20, and provides a different message for each scenario.

**1.2.2 Control Structure 2 – Looping**

The second control structure we will cover involves situations where we need to run the same piece of code many times. This is called ‘***Looping***’. A basic Looping mechanism is a ‘***Do While’*** loop, where you instruct the computer to **Do** *something***While***a condition is true.*We will look at two versions of this loop.

|  |  |
| --- | --- |
| **While** (*Boolean\_Expression*)  {  *Statement\_1;*  *Statement\_2;* } | **While** (*count > 0*)  {  *cout << “Hello!”;  count = count – 1;* } |
| **Do**  {  Statement\_1;  Statement\_2; } **while** (*Boolean\_Expression*) | **Do**  {  *cout << “Hello!”;  count = count – 1;* } **while** (*count > 0*) |

The difference between the two is that in the second version, the test is conducted at the end. The test is also a comparison operation, with a true or false (boolean) outcome. If we know how many times our code should run, a good alternative is the ‘if’ statement;

for (int number = 100; number >= 0; number--)

{  
 cout << number << “ bottles of beer on the wall.” << endl;  
}

**1.3 Programming Style**

Listen when Yoda speaks!

|  |  |
| --- | --- |
| ***COMMENT YOUR CODE*** | |
| [Yoda](http://cis.payap.ac.th/wp-content/uploads/2009/09/Yoda.jpg) | “Put comments throughout your code!” |

It is important to comment your code. Comments are indicators about what the code is supposed to do – this can help you a lot when you come back to code you’ve written before, acting as a reminder for what you’ve done in the past. It also helps a lot when others come to look at your code.

Comments in C++ can be added in two ways. For a one line comment precede it with ‘***//***‘. For comments over more than one line enclose them between ‘/\*’ and ‘\*/’.

// This is a comment

/\* This is also a comment  
but this one goes on more than one line! \*/

When C++ compiles your code, it ignores the comments, they are purely for the programmer’s benefit!

|  |  |
| --- | --- |
| ***INDENT YOUR CODE*** | |
| [Yoda](http://cis.payap.ac.th/wp-content/uploads/2009/09/Yoda.jpg) | “Indent your code to make it easier to read” |

You may have noticed C++ do this for you already, but it is a good idea to indent your code, for instance parts of code that appear between braces ‘***{***‘ should be indented so you can easily see which opening brace is connected to which closing brace.

### 1.4 Functions

Remember Yoda’s advice from Introduction to ISNE? – to ‘***Divide and Conquer***’? When faced by a large problem, it is a good idea to break the problem down into more manageable sub problems. The programs written to solve these sub problems are called ‘***subprograms***’ or ‘***functions***’. So, in ‘Going to Bangkok’, we developed a function to deal with the problem of movement (such as stepping), and another one to deal with the problem of opening a door. We could then use these functions whenever we needed to perform one of those actions. We will start by using ‘***Arithmetic Functions***’. Arithmetic functions are functions we can use to perform a mathematical task, such as calculating a factorial, or a square root. Suppose we write some code to calculate the factorial of a number (4!), we could make it into a function so that the next time we want to calculate a factorial (not often granted!) rather than rewriting the code for a factorial, we can simply ‘**call**’ the existing program, send our number to it and get it to calculate it for us. When we do this, our factorial program becomes a function.

There are two different types of function – ‘**Pre-defined**’ and ‘**Programmer-defined**’. If we use the factorial program as a function, this would be a programmer defined function, as we have made it. Fortunately, we don’t need to write a program to perform square root, C++ already contains a function we can use, this is a predefined function.

### 

### 1.4.1 Programmer Defined Functions

Defining a function has 2 parts. A ‘**Function Declaration**’ and a ‘**Function Definition**’. Just as we need to declare a variable before we can use it, we also need to declare a function before we can use it. The declaration tells you everything needed to use the function.

FUNCTION DECLARATION

|  |
| --- |
| Type\_ReturnedFunction\_Name(Parameter\_List); |

There are 3 parts to the declaration, as can be seen above.

Type\_Returned – The function, for now, will have a return value (in the case of the factorial program, the return value is the factorial of the input number). The first part of the function declaration defines the type  
of the return value (e.g. int or double).

Function\_Name – If we are going to call our function, we need to give it a name (we could call it factorial!)

Parameter\_List – When calling a function we need to send some information to the function (for instance the number we want to calculate). The information sent to a function is called a parameter.

|  |  |
| --- | --- |
| double total\_cost(int number, double price); | Declares a function called ‘total\_cost’ which takes the number & price, and returns a double type result. |
| intfactorial(int value); | Declares a function called ‘factorial’ which takes an integer type value and returns an integer. |

The function declaration should appear prior to any calls to the function – conventionally near the top of your code. It is a good idea to have a comment next to your function declaration to explain what the function is for.

#### FUNCTION DEFINITION

The function definition is a small program – the actual function, where the computer is instructed how to compute the return value. It appears very much like the code we have written already. To convert the factorial program into a function, we only need to add a couple of elements, a ‘***function header***’ and a ‘***return statement***’. The ‘***function body***’ is the calculation.

Function Header – the function header looks the same as the function declaration, except is without the semi colon. It works like the ‘int main()’ statement we have already used, so just like the main function the function body should be enclosed within braces ‘{‘ and ‘}’.

Return Statement- The return statement part of the function (if it exists) should return the result of its calculation, given the parameters input. The return type should be of the same type as declared in the function declaration.

|  |
| --- |
| double total\_cost(int number, double price) {  some code;  …;  return total\_cost; } |

Once a function has been declared and defined, it can be called from anywhere within the program – by a ‘***function call***‘. The function call sends the required parameters and indicates where the return result should be stored.

|  |  |
| --- | --- |
| ken = total\_cost(number, price) | sets the variable ‘ken‘ to the result of the total\_cost function with parameters number and price. |

**1.4.2 Pre-Defined Functions**

With a function such as the factorial function, we may want to use it accross a variety of programs, so rather than including the body of the function within our program, we may wish to create a file containing a variety of useful mathematical functions. Then each time we need to calculate a factorial, we can make a call to the mathematical function file. Well, C++ already has a mathematical function file called ‘***cmath***’, which contains a variety of mathematical functions, such as *‘****sqrt****’*(square root), *‘****pow***’ (powers) or *‘****ceil***’ (rounding up). We are able to use these, and many other, ‘***pre-defined functions***’, by simply including the file they are in within our project. We have already used an ‘***include***’ statement, to include the ‘***iostream***’ to assist with input and output (by defining ‘cin’ and ‘cout’!) To include the ‘***cmath***’ functions, extend the include statement to;

|  |
| --- |
| #include <iostream>  #include <cmath>  using namespace std; |

Once the cmath library is included in a program, calls can be made to any of the functions in it, in the same way as programmer defined functions are called. So, to calculate the square root of a number we call the ‘***sqrt***’ function. This function takes one parameter – the number you want to root. (Assuming variables have been declared).

side = sqrt (area);

Alternatively the ‘***power***‘ function takes 2 parameters, a number and the power, so 5 to the power 4 needs ‘***5***‘ and ‘***4***‘ to be included in the function call.

cout << “5 to the power 4 is ” << pow(5,4);

Note with the above, you don’t always need to store the return value of a function in a variable, you can display directly as output. Also note the order parameters are sent to a function, ‘4 to the power 5′ will return a different value to ‘5 to the power 4′.

|  |
| --- |
| **Note on Variable types**  Previously we introduced different types of variables – *integer*to contain whole numbers or *double* to contain floating point numbers. Occasionally code can produce unexpected results if the variable type is confused. For example, ‘***9/2***’ will produce the answer ***‘4***’ if the numbers involved are integers. EVEN if you store the result of 9/2 as a double. This code will produce ‘**4**‘  *double* answer;  answer = 9/2; cout << answer;  If you are using variables, the chance of using the wrong type increases. So the value returned from division, or a function such as sqrt might be unexpected. A useful function here, is a function to convert a number from one type to a double type. The static\_cast function allows us to change a number or variables type.  **static\_cast<double>(9)** *returns* 9.0  **static\_cast<double>(number)** *returns* 9.0 (*if number is currently an integer value 9*) |

Note that functions can be considered as ‘***black boxes***’ – that is, we don’t need to know how they work, just what they do. All we need to know is what to provide the function, and what it will return us.

### 1.4.3 Variable Scope

So far when we have used variables, we have used ‘***local variables***’, which means they are local to the function in which they are declared – you are not able to use a variable if it is declared in a different function. Given that functions are self-contained black boxes, this is helpful, as a variable in one function will not interfere with a variable in a different function (should they have the same name ‘max’ for example).

When we pass parameters to a function, once again local variables are created to be used within the function. Sometimes we do want to have variables which are usable throughout the program. We may have several functions which need to access or manipulate a common value – for instance we may want to allow our input value to be manipulated by both the factorial and the cube functions. In this case we want to create a ‘**global variable**‘. To create a global variable, simply declare the variable outside of a function, a sensible place is along with the ‘#include’ statements, and other function declarations. However, we rarely need to use a global variable, and they can make your code harder to understand and maintain. It is more likely that you will use a global constant – similar to a variable, whose value doesn’t change (is constant!)

Constants are declared using the ‘**const**’ keyword, for example;

const double PI = 3.14159;  
const double tax\_rate = 0.05;

In the same way as variables – if you declare the constant in a function, it is local to that function, if you declare it outside the function it becomes global to the program. It isn’t only constants and variables which can be local or global – ‘**namespaces**’ are also local or global. Often we just use the ‘std’, or standard, namespace, but in the future you may want to use different namespaces particularly when working on big projects with a group of programmers. Moreover, you may wish to use different namespaces in different functions. Declaring it outside the function makes the namespace choice global, so a better place to put it is inside the function braces ‘{‘.

**1.4.4 Overloading Function Names**

A further function concept for this week – ‘overloading’. Sometimes, we may want to use the same function name for more than one function. For example, one function called ‘ave’ might be used to calculate the average of 3 parameters, while a second function also called ‘ave’ might be used to calculate the average of 4 parameters. It is possible to do this in C++, but greater care needs to be taken when doing this! This is called ‘***overloading***’ and it can be very helpful – allowing the most sensible name to be given to a function.

But how does the computer know which function to use? Well, this depends on the arguments supplied in the function call and the parameters expected in the function declaration. When overloading, each function must have a different number or type of parameter.

|  |
| --- |
| *double* ave(*double* n1, *double* n2, *double* n3);  *double* ave(*double* n1, *double* n2, *double* n3, *double* n4);  Here one function has 3 parameters and one has 4;  result = ave(3, 4, 5); result = ave(3, 4, 5, 6); So, these two function calls will call different functions. |

This is perfectly acceptable, and sometimes very useful – consider the divide function ‘/’ – the computer uses a different function if both values are integers, to produce an integer result (**9/2=4**, while **9.0/2=4.5**)!

NOTE : As the ‘ave’ function expects ‘*double*‘ type arguments, it will automatically convert the integer inputs (3, 4, 5) to double type before it does the calculation (assuming there are no conflicting overloaded functions!)

**1.4.5 *void* Functions**

The functions discussed so far have been designed to return one value – however sometimes no value is being returned to the program. Alternatively, several values could be generated by a function – consider if a subtask needed to get several values from the user? In C++ a function which generates more than one value is treated in a similar way to a function which generates no values – they are called ‘***void***’ functions.

First consider a function which has no return value – for instance a function which only displays output to a user.

|  |
| --- |
| void show\_results(int value, int factorial)  {  using namespace std;  cout << “The factorial of ” << value << “ is ” << factorial << “.”;  return; } |

If we made a call to this function as;  
*show\_results(4, 24);*

The response would be to display to the screen;  
*The factorial of 4 is 24.*

Alternatively variables could be sent as the parameters. But how does it work? Notice first that instead of naming the type of the return value (*int* for example), the key word ‘***void***‘ is used to indicate to the computer that no return value will be returned. Parameters are sent to the function in the same way as before, and these are displayed using the cout command. The only other difference is that the ‘***return***‘ statement has no value to return to the main program, it simply ends the function.

**Note 1** : The return statement need not be the final line of a function – in fact if the return statement is the final line of a void function it is optional (for many compilers). This structure can often be useful;

If (condition is true)

{  
 Return;  
}  
Else  
{  
 Do something;  
}

**Note 2**: It isn’t always necessary to use a parameter to send a value to a function. The brackets are necessary, but they can be empty – as in the main function;

void initialise\_screen()

{  
 using namespace std;  
 cout << endl;  
 return;  
}

Called by;  
initialise\_screen()

**1.4.6 Call-by-Reference *vs* Call-by-Value**

So far we have been using ‘***Call by Value***’, but now we will investigate using ‘***Call by Reference***’. These two terms refer to parameters, and when arguments are sent to a function. With ‘*Call by Value*’ parameters a value is sent to the function, while with ‘*Call by Reference*’, a reference to a value is sent – or the memory location of a value is sent. A Call by reference parameter allows us to directly manipulate the value stored within a memory location, or variable. Consider if we were to write a function to receive  
an input from the user, we would want the function to store the value directly into a variable, so we could use a Call by Reference parameter, with the name of the memory location.

A Call by Reference parameter is indicated by an ampersand sign ‘***&***’ attached to the end of the variable type e.g.- *double& value*.

This is how we could convert getting an input into a function.

|  |  |
| --- | --- |
| In a function | Called from a function by;  *get\_input(value)* |
| cout << “Input the value you want to calculate “;  cin >> value; | void get\_input(double& input)  {  using namespace std;  cout << “Input the value you want to calculate “;  cin >> input; } |

**Note :**A Call by value parameter creates a local variable, local to the function. A Call by Reference parameter effects the variable created.

Consider this;

|  |
| --- |
| [code3](http://cis.payap.ac.th/wp-content/uploads/2009/11/code3.JPG) |

What output would you expect? Experiment with the code here to make sure you understand the differences.

Remembering the advice of divide and conquer, we can expand on this, and divide a problem into subproblems, and then write a function which solves each subproblem. Take the problem of

“Take 2 numbers from the user, order them so the lowest number is first, and then returning them to the user.”

This can be broken into subtasks;

1) Get input from user.  
2) Find out which number is lower and if necessary;  
3) Swap the 2 numbers.  
4) Return the output to the user.

Each of these subtasks can be solved by a function – the main part of the program (or main function) is then responsible for declaring variables and calling functions when necessary. This means that each subproblem can be solved individually. It also tidies the code up and makes it easier to read, and if there is a problem with the program makes it easier to isolate where the problem is, by testing each function independently.

Consider the solution following;

|  |
| --- |
| [code2](http://cis.payap.ac.th/wp-content/uploads/2009/11/code2.JPG) |

Each of the functions above performs a small part of the program – the comment after the function declaration explains each functions purpose