Topic 5 – Modular Programming

Top Down Design

Programming is Hard!

- Most programmers and computer scientists would probably agree that programming is hard.

- This is not so much that it is hard to learn (although you might think it is right now!).
- But rather that writing a program which solves any non-trivial problem and works reliably (without bugs) is very difficult.

- The main reason for this is that, for all but the simplest programs, a lot of complexity is required to provide a comprehensive piece of software.
- In fact, computer programs are among the most complex and sophisticated things that humankind has ever created.

- Computer scientists want to make this process easier and so a lot of work has gone into reducing the complexity.

- If computer code can be made more manageable then it will be easier to write programs and will have fewer bugs.

- Just as importantly, it will also be easier to fix any bugs that are found.

Taking it from the Top

- Top down design is one way of trying to make the complexity more manageable.

- TDD means that we approach the task of solving the problem (and writing the code) starting from the top (the high level details) and working our way down to the bottom (the low-level details).

- Back in Topic 1 we developed a high-level algorithm and gradually refined it to the individual steps needed to solve the problem.
    - This is exactly how TDD works.

- So TDD involves:
    - Developing a *high level algorithm* containing a number of steps which will solve the problem.
        - These should be general steps and will not usually involve the specific calculations needed.

    - Taking each of these steps and *decomposing* it.
        - In other words, working out what more detailed, lower-level steps need to be performed to complete the higher-level step.
        - This is often called *refining* your high level solution.

    - Sometimes you will have to have several layers of decomposition/refinement, depending on the complexity of the problem.

Modularity and Top-Down Design

- So how does this process of breaking down the solution into more detailed, lower-level steps actually work?

- In practice this is closely linked with the notion of *modularity*.

  - **This means that each of the refined steps is separated out into a distinct *module* of code.**

- Each of these modules performs each of the refined sub-steps from the high-level algorithm.

- However, for TDD to be successful in terms of providing a useful methodology for dealing with complexity, these modules must be constructed in a certain way.

- Specifically they should have a number of important properties.

- These are:
  - An appropriate level of *abstraction* between modules.
  - Ability to *re-use* the module is maximised.
  - *High cohesion* within each module.
  - *Low coupling* between modules.

Abstraction

- Abstraction is involved with hiding details that aren’t necessary for us to know while understanding in general terms what is going on.

- For example, most people drive a car but few are mechanics.
- We know that our cars have engines which consume fuel when we drive and we need to re-fuel regularly otherwise our cars will stop working.
  - But we don’t **need** to understand the details of what is actually going on inside our car’s engine when it burns the fuel.
  - Instead we just focus on the general things that we need to know to do what we need to do.

- Therefore we *abstract* the details of how our cars work.

- The same applies when we split up a problem into steps/modules and then refine the details of exactly how each of those modules performs its task.

- Although all of the modules must work together to solve the problem, no module should need to know exactly how another module works.
- Instead they just need to know in general terms what the module does (primarily what its outcomes are), plus any data that it requires to perform its task.

- In general as little information as possible about *how* a module works (as opposed to what it does or its outcomes) should be available outside of that module.

Code Re-Use

- The concept of code re-use is summed up in a common programmer's catch-cry:

Don't re-invent the wheel!

- If code has already been written to solve a particular problem, why develop and write your own version of that code?
  - This is particularly true if the existing code has been extensively debugged and tested.

- Code re-use applies in terms of modules.
- A single program will often use one of its modules a number of times for slightly different purposes.
  - For this reason it is best to make modules *as general as possible* in their behaviour providing it still solves the original problem and does not become much more complex.

  - For example, if you need a module to calculate the average test marks for a class of 10 students, it would be better to write the module so it will work for **any** number of students.
  - That way, if/when the class size changes, you can re-use the module.

- Re-use also applies to so-called "code libraries" which are collections of modules made available to all programmers working with a particular language/platform.
- For example, the `printf()` and `scanf()` functions we have been using are modules provided as part of C.

High Cohesion

- The code which makes up a module should exhibit *high cohesion*.

- What this means is that all of the code in that module should be concerned with the same thing.
- A module that does a number of things which aren't all related to the same sub-step is not cohesive and becomes less general and harder to maintain.
    - This is because the code associated with a specific task will no longer be isolated to a single module but possibly spread over many separate modules.

- So all the code in a module should be associated with the same logical task.

Low Coupling

- Low coupling between modules means that each module should have limited ability to interact with data belonging to other modules.
- If a module can alter the data belonging to another module then this could cause that module to fail and the bug would be very hard to find.

- Low coupling is closely related to abstraction since hiding the data associated with a module helps to abstract the behaviour of that module.

- We'll look at how data abstraction works in more detail a little later on.

MODULARISATION IN C

Overview

- In virtually all programming languages there is some way of splitting up the body of code that makes up the program into separate modules.

- However, there are different philosophies and paradigms for doing this.

- The C language uses a process of *proceduralisation* whereby the program is broken up into modules of code.
    - These modules are called *functions*.

- Each of these functions performs a specific task related to solving the overall problem.
    - This relates to the notion of top-down design.
    - The design of the functions that make up a C program should adhere to the principles of abstraction, re-use, high cohesion and low coupling.

Function Syntax

- We have actually already been using functions in our C programs!

- All programs must have a function called the "`main` function" and this is where execution begins.
  - Our programs so far have only had a single function, namely the `main()` function.

- Programs that are made up of other functions still have a `main()` function.

- However, the job of the `main()` function is no longer to contain all the code to solve the entire problem.
- Instead it now *calls* the other functions which perform the individual sub-steps that make up the algorithm.
  - This means the `main()` function effectively becomes the implementation of our high level algorithm with the other functions representing the refined steps.

Here is a simple program consisting of three functions showing how the calling process works:

```
#include <stdio.h>

void Step1()
{
   printf("Now performing Step 1.\n");
   /* Step 1 code in here */

   return;
}

void Step2()
{
   printf("Now performing Step 2.\n");
   /* Step 2 code in here */

   return;
}

int main()
{
   Step1();
   Step2();

   return(0);
}
```

- Note the use of curly brackets to show where a function begins and ends.
- Generally the `main()` function should be at the end of the program with the functions that it calls above it.
- Functions have names, just like variables, but this time they represent a piece of code rather than data.

- Unlike variable names, function names always have brackets after them, e.g., `main()`
    - In the program above these brackets are empty but they can contain things (later!).

Function Declarations and Function Calls

- Just like variables, functions are both declared (i.e., initially defined) and used.
    - When a function is used, this is referred to as *calling the function.*

- Calling a function involves executing the code it contains.
- Therefore, it is the calls to functions, rather than their declaration, that defines what the program actually does and in what order.
    - It's quite possible to define a function that you never call, although this is not particularly useful!

- Therefore, the basic hierarchical structure of a top-down, modular program is defined by the function calls.

- In the previous example program, the calls to the functions are in `main()`, whereas the definitions of the functions are above `main()`.
- As long as the declarations of the functions `Step1()` and `Step2()` remain, altering the calls to these functions will change the behaviour of the program.

- Here is an example…

```
void Step1()
{
  printf("Now performing Step 1.\n");
  /* Step 1 code in here */

  return;
}

void Step2()
{
  printf("Now performing Step 2.\n");
  /* Step 2 code in here */

  return;
}

int main()
{
  Step2();
  Step1();
  Step2();

  return(0);
}
```

- Note that changing the order of the function calls in `main()` completely changes what the program does, even though the order of the declarations remains the same.

Function Return Types

- Functions can produce a result, called a *return value*.
  - We've already seen this with the `sqrt()`, `tolower()` and `pow()` functions.
- When you define the function by writing out the code included in that function you need to specify the data type of the return value.
- This can be any primitive data type (e.g., `int`, `char`, `float`).

- For example, the `main()` function conventionally returns an `int`:

```
 int main()
```

- Our `Step1()` and `Step2()` functions, however, return `void`.
  - This means they produce no result.
  - In other words, they perform some operation but don't return any value.

- The `return` statement inside a function makes the function finish executing and return to the next line after where it was originally called.
- It also allows the return value of the function to be specified, e.g., `return(0)` causes the function to return the value zero.

FUNCTION PARAMETER PASSING

Introduction

- Being able to get a value out of a function as a result is very useful.
- We've already seen examples of this with the `tolower()` and `sqrt()` functions:

```
response = tolower(response);


geomean = sqrt(n1 * n2);
```

- But return values have two obvious limitations:
    - Only a single piece of data can be obtained from each function (big problem).
    - How do we get data into the function in order to process it and get the result? (Bigger problem!)

- So procedural modular programming languages like C provide a means for us to pass data both **in** and **out** of functions.
    - *This is in addition to being able to return values from them.*

- These data values passed in and out of a function are called *parameters*.

Simple Parameter Passing Example

```
#include <stdio.h>

/* Calculate the sum of two numbers */
int Sum(int x, int y)
{
   return(x+y);
}

int main()
{
   int a, b, total;

   printf(“Enter number 1: “);
   scanf(“%d%*c”, &a);

   printf(“Enter number 2: “);
   scanf(“%d%*c”, &b);

   total = Sum(a, b);
   printf(“The sum is: %d\n”, total);

   return(0);
}
```

Note:

- The variables `a` and `b` are passed into the function `Sum` as parameters.
- The variables `x` and `y` take on the values of `a` and `b` respectively as determined by the order the variables are passed into the function.
- The types of the parameter variables `x` and `y` must be specified and this is done inside the brackets where the function is defined.
- Also note the way the return value is assigned to `total`.
    - If you don’t do something with the result of a function which returns a value, then this is discarded and lost.

Variable Scope

- Remember the principles of abstraction and low coupling to which modules should adhere?
- This applies to data (i.e., the variables being used) in a special way.

- Variables cannot be “seen” (i.e., their values accessed and changed) everywhere throughout the program.
- In fact, normally *variables are only visible within the function in which they are declared*.
- This is called the variable’s *scope*.

- So the variables `a`, `b` and `total` from the `main()` function in the above program are **not** visible inside the `Sum()` function.
    - This is why we can’t just write `return(a+b)` inside the `Sum()` function.
    - (It would also be poor abstraction and high coupling.)

- Also the parameter variables (like `x` and `y`) are only visible within the function into which they are passed.

- By limiting the scope of variables to the function in which they are declared we can enforce abstraction (other functions don’t know what data is inside one another and they don’t need to know) and low coupling (other functions cannot accidentally change the data within a function.)
- Also because of scope the way parameters are passed into and out of a function (plus any return values) become very important.
    - These define the *interface* to that function.

Another Parameter Passing Example

This program gets two integer numbers and calculates the sum and average of them.

```
/* Returns the sum of two integers */
int Sum(int x, int y)
{
   return(x+y);
}

/* Returns the average of two integers */
float Average(int x, int y)
{
   int sum;

   sum = Sum(x, y);

   return(sum/2.0);
}

int main()
{
   int a, b;

   printf(“Enter first number: “);
   scanf(“%d%*c”, &a);

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

   printf(“The sum is: %d\n”, Sum(a, b));
   printf(“The average is: %f\n”,Average(a, b));

   return(0);
}
```

Things to Note:

- The different return types used by `Sum()` and `Average()`, `int` and `float` respectively.
- The names given to the parameter variables in `Sum()` and `Average()` are the same (`x` and `y`), but this doesn't matter because the scope of these variables is restricted to that function only.
  - In fact the variables `a` and `b` inside `main()` could also be called `x` and `y` and the program would work correctly.
- There is code re-use going on as the `Average()` function calls `Sum()`.
- The `main()` function which calls the other two functions doesn't "need to know" anything about how they work (abstraction).
  - For example, it doesn't know that `Average()` calls `Sum()`.
  - All it needs to know is what data to pass into the function and what data comes back (interface).
- There is a comment before each function briefly describing what it does, the data it takes and returns.
  - This is good practice, even for trivial functions.
- The return values from the `Sum()` and the `Average()` functions are not actually stored in a variable anywhere but are passed directly to the call to `printf()`.
  - Note that `printf()` doesn't care if it's parameter values are variables, some arithmetic expression or a function's return value.

- However, you might notice a problem with this program too.

- Even though the program works correctly, the `main()` function does not really adhere to the principles of good top-down design.
- Specifically the code inside `main()` is not just responsible for calling the other functions to solve the problem but also for solving parts of the problem.

- A high-level algorithm for this problem might look like:
  - Get input data from user.
  - Calculate Sum
  - Calculate Average
  - Print Results

- This being the case, the `main()` function probably should just contain four function calls for each of these steps.
  - In a trivial program like this, it doesn't **really** matter much if `main()` does contain some code to solve the problem so long as the function remains fairly short.
  - In fact, for simplicity's sake it's probably preferable!

- But it's still not good top-down design so it is a worthwhile exercise to separate these out.

- But how can we do this?
  - We could write a function to read in the two values but we can only return a single value at a time!

- What is needed is a way of passing multiple pieces of data back from functions the same way we can pass data into them in the first place.

Pass by Value

- So far the parameter passing we've done has been one-way.
- The data has been passed into the function but no data is passed out (except for the function's return value which is not considered parameter passing).

- This type of parameter passing is known as *pass by value*.
- Pass by value works by making a copy of the variable being passed.

- For example the following line from the `main()` function involves passing two variables into `Sum()`:
  ```
  Sum(a, b);
  ```

- The function definition for the `Sum()` function looks like:
  ```
  int Sum(int x, int y)
  ```

- And we know that the parameter variables `x` and `y` contain the values that were in `a` and `b` respectively.

- This is pass by value because `x` and `y` are the copies of `a` and `b`.
  - They might have the same values but they are different variables and are stored in separate memory locations.

- This means that if you make a change to the value of `x` inside `Sum()` it will **NOT** affect the value of `a` inside `main()`.

- So the change is being made to the copy, not the original and once the function call ends the parameter variables disappear and the changes are lost.

Pass by Reference

- The way pass by value works is the reason we cannot use this for passing data back from a function.

- However, there is another technique called *pass by reference* which allows us to do this.

- Pass by reference involves creating a special type of variable (the *reference*) which refers back to the **original** variable.

- This means if we change the value of something passed by reference, the changes are actually happening to the original variable and will still exist once the function finishes and returns.

- C gives the programmer a lot of control over data passed by reference, which makes it a very powerful feature.
- However, it also means it is possible to make mistakes and you need to be very careful.

- For the purposes of this unit we are going to use a simplified version of how pass by reference is implemented in the C language.

- This involves using a feature of the C++ language which is an evolution of C and is backwardly compatible.
- So we are still using C but borrowing a feature from C++.

- This is not important to you in this unit but is good to be aware of.

Simplified Pass by Reference

- To use the simplified pass by reference system of C++ in our C programs means there is only one small change that needs to be made.

- When making a call to a function and passing the parameters by reference, *no changes need to be made:*

```
GetData(a, b);
```

- However, the declaration of the function should indicate that the variables are passed by reference with the inclusion of *an ampersand in front of their name*:

```
void GetData(int &x, int &y)
{
```

- Once inside the function, you can use a variable that has been passed by reference *exactly the same as normal*.
    - The only difference will be that any changes made inside the function will still exist when the function returns.

- Also note that because these programs are using C++ features they have to be compiled as C++ programs.
- For the `bcc32` compiler that most people are using this involves saving the source code file as `.cpp` rather than `.c`
- For example, `myprog.c` becomes `myprog.cpp`

Here is a full sum and averaging program that uses pass by reference exclusively (no function return values) and represents good top-down design:

```
/* Get user input */
void GetData(int &x, int &y)
{
   printf("Enter first number: ");
   scanf("%d%*c", &x);

   printf("Enter second number: ");
   scanf("%d%*c", &y);

   return;
}

/* Returns the sum of two integers */
void Sum(int x, int y, int &sum)
{
   sum = x + y;

   return;
}

/* Returns the average of two integers */
void Average(int x, int y, float &avg)
{
   /* Temporary variable */
   int sum;

   Sum(x, y, sum);
   avg = sum/2.0;

   return;
}
```

```
/* Display results to the user */
void PrintResults(int sum, float avg)
{
   printf("The sum is: %d\n", sum);
   printf("The average is: %f\n", avg);

   return;
}

int main()
{
   int a, b;
   int sum;
   float avg;

   GetData(a, b);

   Sum(a, b, sum);
   Average(a, b, avg);

   PrintResults(sum, avg);

   return(0);
}
```

Things to Note:

- Ampersands are used to indicate which variables are passed by reference in the function's definition.
- A temporary variable is needed in the Average function to hold the value passed back (by reference) from the Sum function.
- Extra variables are also required inside the `main()` function to hold the data passed back.
- Since none of the functions return values they are all type `void`.

Here is a final example that uses both pass by reference and function return values.

It is consistent with good top-down design and the principles of modular programming.

```
#include <stdio.h>

/* Get user input */
void GetData(int &x, int &y)
{
   printf("Enter first number: ");
   scanf("%d%*c", &x);

   printf("Enter second number: ");
   scanf("%d%*c", &y);

   return;
}

/* Returns the sum of two integers */
int Sum(int x, int y)
{
   return(x+y);
}

/* Returns the average of two integers */
float Average(int x, int y)
{
   int sum;

   sum = Sum(x, y);

   return(sum/2.0);
}
```

```
/* Display results to the user */
void PrintResults(int sum, float avg)
{
   printf("The sum is: %d\n", sum);
   printf("The average is: %f\n", avg);

   return;
}

int main()
{
   int a, b;
   int sum;
   float avg;

   GetData(a, b);

   sum = Sum(a, b);
   avg = Average(a, b);

   PrintResults(sum, avg);

   return(0);
}
```

Pass by Reference vs Return Values

- So pass by reference and return values do similar things:
  - Both allow you to get data out of a function, which would otherwise be impossible due to the limitations on variable scope.
  - However, these are *two completely different techniques*.
  - Even though the result is the same, the way they work is quite different.
  - Also remember that a function can only return a single value whereas as many parameters as you like can be passed by reference.

- There are also significant differences in how you use each of these features.

Rules for Pass by Reference

- With pass by reference it is very easy.
  - You simply put an ampersand (&) in front of the name of the variable that you want to pass by reference in the function declaration:

```
void GetData(int &x, int &y)
```

- Note that the call to the function remains the same.

Rules for Return Values

- Use of return values is perhaps a bit more complicated than pass by reference.
- Specifically, there are **three things** you need to do whenever returning a value from a function.

These are:

1. Declare the function with the appropriate return type.
   - That is, if the function returns a value then you must indicate what its data type is.
   - For example:

     ```
     int Sum(int x, int y)

     float Average(int x, int y)
     ```

   - Here the `Sum()` function returns an integer and the `Average()` function returns a float.

2. Return a value of the appropriate type in the function.
   - This means you need a return statement somewhere inside that function which has the values/variable/expression that you want to return inside brackets, for example:

     ```
     return(sum/2.0);
     ```

   - This is the return statement from the Average() function and noticed that the data type of the value returned is a float.
   - Note usually the return statement is the last statement in the function.

3. “Catch” the return value back in the calling function.
   - For the Sum() function in the previous program this involved passing the return value straight to a call to printf():

```
printf(“The sum is: %d\n”, Sum(a, b));
```

   - However, more commonly the return value is assigned to a variable for future use:

```
sum = Sum(x, y);
```

   - Or alternatively, it could be used in an expression as if it were a variable or a constant.
   - For example, the last two lines of the Average() function are as follows:

```
sum = Sum(x, y);
return(sum/2.0);
```

   - But these could just as easily have been condensed into one line:

```
return(Sum(x, y)/2.0);
```

   - In other words, the call to a function which returns a value effectively takes on whatever value it returns.
   - Always remember to do something with this value, otherwise it is simply thrown away and your program continues executing.

- **This is a very common error by beginner programmers!**

So when should you pick return values over pass by reference?

- If a function produces more than one value then pass by reference must be used.
  - But this doesn't mean it has to be used for all of these values and it may be appropriate to also use a return value for one of them.
  - Similarly, just because just one piece of data comes out of the function does not mean that you **have** to use return values.

- If a function produces a *single* value then it may be appropriate to use return values instead of pass by reference.
  - You should **definitely** do this when this value is the result of some calculation.
  - This is probably the only clear rule that can be applied.

Function Order and Prototypes

- In order for a program to call a function, the compiler must know that the function exists.
- In the programs we've seen so far, this is accomplished by declaring the function before it is called.
    - For example, in our Step 1/2 program, if the main function was placed at the top of the code, the program would no longer work.
    - This is because the functions `Step1()` and `Step2()` would not have been defined yet.

- To get around this problem, so called *function prototypes* can be defined at the top of the program.
- These basically tell the program the function's name, return type, parameter types and parameter order.
    - Note that it is not necessary to put the actual names of variables in the function prototypes.
- Once function prototypes have been declared, it becomes possible to call the functions in any order.
- Note that use of function prototypes is not required in this unit but may be beneficial in certain programs.

```
#include <stdio.h>

void GetData(int&, int&);
int Sum(int, int);
float Average(int, int);
void PrintResults(int, float);


/* Get user input */
void GetData(int &x, int &y)
{
   printf("Enter first number: ");
   scanf("%d%*c", &x);
```

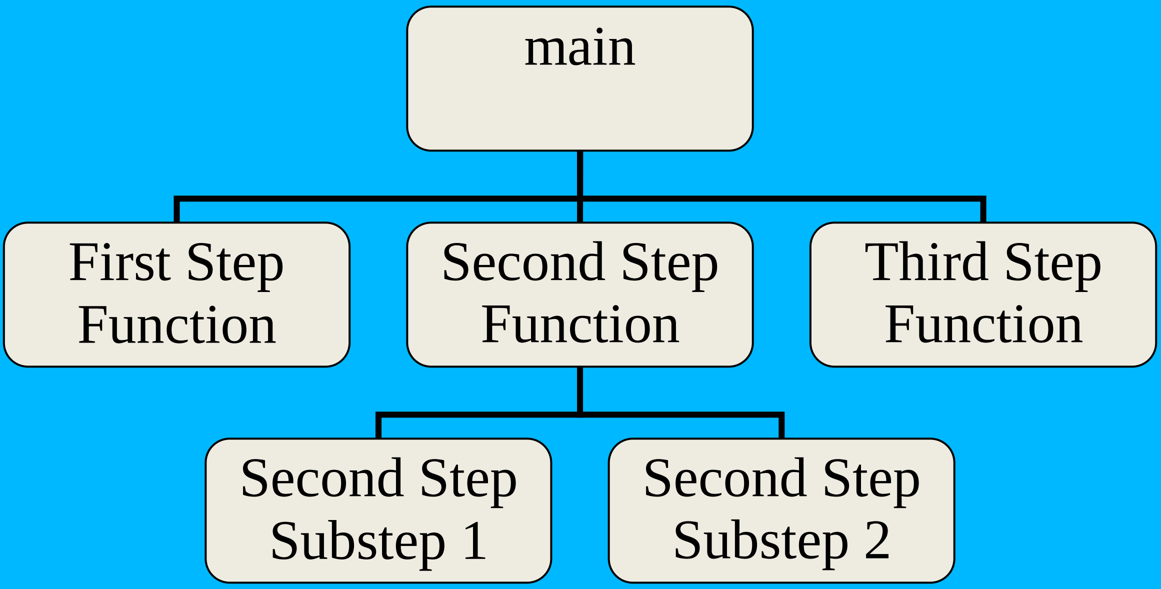
Summary of Modular Programming Concepts

So here's the modular programming concepts we've covered so far:

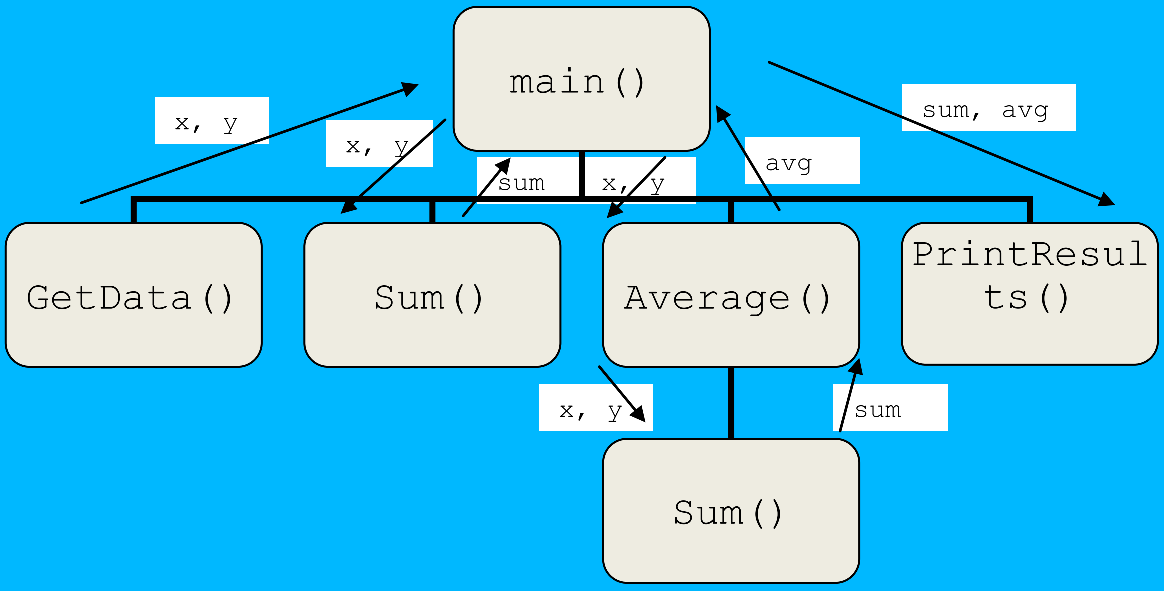
- We can use *top-down design* where we start with a high-level (general) view of the solution and "fill in the gaps", step by step.
- This process of starting with a general solution and refining each step to work out the details makes it much easier to develop correct solutions to complex problems.
- This approach relates closely to *modular programming* where we split up the code into separate parts, each of which solves a specific part of the problem.
- These modules (called *functions* in C) should all adhere to the principles of *abstraction*, *code re-use*, *high cohesion* and *low coupling*.
- Functions can either produce a result (returning a piece of data of a given type) or just perform an operation (return *void*).
- We can pass data into a function and also pass data back out as well (in addition to returning a single value).
- If the data only needs to go into the module then it is *passed by value* and this involves making a copy of the data: any changes are made to the copy and not the original.
- Alternatively data can be *passed by reference* where a reference to the original data is passed into the module: changes are therefore made to the original and so these persist after the function has returned.

STRUCTURE CHARTS

- Structure charts are an important tool for diagrammatically showing how the different modules in a program are related.
- Structure charts show each of the modules in the program and show which modules call which others.
- They are arranged hierarchically and always begin with the *main* module at the top.
- The main module calls the first layer of modules, each of which is responsible for performing the high-level steps of the algorithm.
- These may then call other modules in larger programs.
- Sometimes data flows are shown on structure charts: these show the data that is passed into the function and that comes out of it.



Here is a structure chart for the final version of the summing and averaging program above:



CODE RE-USE AND LIBRARIES

- Code re-use was mentioned as an important principle applying to modular programming.
- This means that when you write a module, ideally its interface should be general enough that it can be re-used.
  - High cohesion, low coupling and abstraction help to facilitate this.
  - However, while the module should solve the problem in a general way to facilitate re-use, it should also maintain high cohesion.

- However, module re-use is not just limited to within the same program.
- Often modules are general enough and solve a common enough problem that it is likely they could be used again in other programs.
- This leads to the collection of modules together into groups called *libraries* which can then be used in many programs.
  - This saves the programmer re-writing code that they or someone else has already written.

- Most programming languages come with extensive libraries for various purposes such as:
  - Character and "string" manipulation.
  - Input/output: both to files and interactively (i.e., keyboard and screen).
  - Graphics manipulation and user interfaces.

- For example, `printf()` and `scanf()` are part of the *Standard C Library*.

Advanced Modular Programming Concepts

Public Interface

- The modules that you write may become part of a library and be used by lots of different programmers in solving lots of different problems.
  - As a result, the details of *exactly* how they are designed matter a lot.

- One important aspect of this is the first principle of modular programming that we looked at: *abstraction*.

- That is, just because a programmer is going to use a module that someone else has written, it doesn't mean they necessarily need to know exactly how that module works.
- Instead they just need to know what it does and how to use it.

- This is known as the *interface* of the module.
  - In fact, libraries are often described as *application programming interfaces* or APIs.

- Because the interface to a module needs to be known in order to use it, this part of the module is described as being *public*.

Private Implementation

- So long as you know a module’s interface, you don’t need to know the details of exactly how it works (abstraction).
- So it is often said that the details of how a module works are *private*.
  - This implies that they are virtually hidden from the programmer who is using the module and to some extent this is true.
- The code that actually makes up the module is often described as its *implementation*.
- The *implementation* of a module is often kept separate from its *interface* in order to enforce the separation between *public* and *private*.
  - This helps to enforce the principle of abstraction.
- As long as the implementation is consistent with the described interface then this does not matter.

The C Approach

- The way C approaches this is to use two separate files when writing a library.
- One file contains the (private) implementation and this gets compiled in a special way and becomes the library object code that is linked in with other program code.

- The other file is the interface and this is called a *header file* and ends with the extension `.h`
  - These are the files that are referenced in the `#include` directives at the start of most C programs.

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

- The header file itself doesn't contain any real code (implementation), just a set of definitions stating what functions are part of library, what parameters they take and values they return.
- It also usually contains extensive comments describing what each function does and what its parameters are.
  - This effectively defines the interface for those library functions.

- Other languages do things slightly differently.

- Although we won't be creating any libraries in this unit, we will be using modules from such libraries so these concepts are important to understand.

USING BUILT-IN FUNCTIONS

Random Numbers Generation

- The generation of random numbers is a relatively common task in many programs.
  - However, note that most modules for this purpose actually generate *pseudorandom* numbers.
  - These numbers are not truly random and are potentially predictable, so should not be used in situations where unpredictability is important.

- Computers can't simply pick random numbers -- they can only “calculate” them.
- However, this means that if an algorithm for calculating random numbers is performed, the result will be the same every time.

- To avoid this, the random number generator must be “seeded”, basically by giving the computer a random value on which to base the sequence it calculates.

- In C, the random number generator can be seeded using the `srand()` function.
- However, this function must be given a value as the seed.
- The easiest way to do this is to use the time, which is constantly changing and therefore will normally give different results each time:

```
srand(time(0));
```

  - Note that seeding should only be done once at the start if generating a sequence of numbers in a program.
  - Otherwise the same number will be generated.

Generating the Numbers

- The `rand()` function generates a number (semi-)randomly chosen between the ranges of `0` and the built-in constant `RAND_MAX`.

- The following program demonstrates this behaviour:

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


int main()
{
   int r;

   srand(time(0));
   r = rand();

   printf("Random number chosen between zero
                 and %d: ", RAND_MAX);
   printf("%d\n", r);

   return(0);
}
```

- However, most of the time is more useful for the programmer to specify the maximum random number to be obtained.

Limiting the Range

- The modulus operator can be used to limit the maximum value obtained when generating a random number.
  - Modulus gives the remainder left over after division.
  - The remainder can be between 0 (if it goes in evenly) or, at most, one less than that number.

- Therefore, calculating the modulus of a random number by the largest possible value you would like to get back (max) will give you a number in the range: 0–(max-1).
  - Adding one to this result will give you a number between 1 and max.
- The following program illustrates this technique using a maximum value provided by the user:

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


int main()
{
   int max, r;


   printf("Enter largest possible number: ");
   scanf("%d%*c", &max);

   srand(time(0));
   r = (rand() % max) + 1;

   printf("Random number chosen between 1 and
                 %d: %d\n", max, r);

   return(0);
}
```