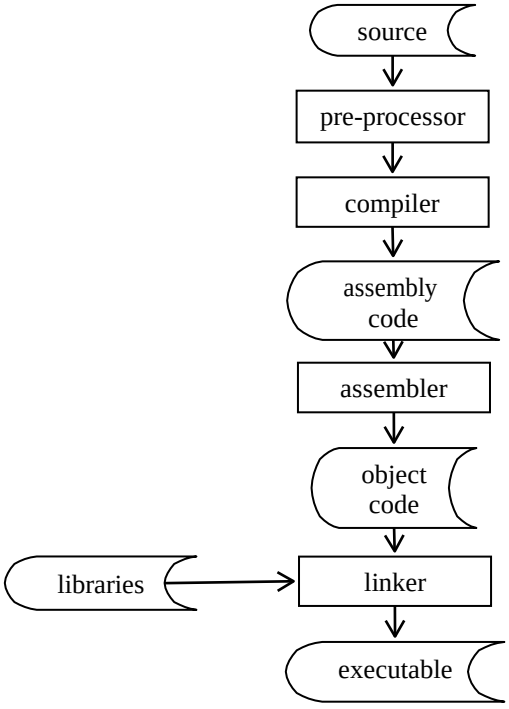
Building Complex Programs

1. The Compilation Process

When we decide to write programs for a computer we are required to make a decision about which language to use. Because each computer make and model has differing hardware characteristics, it would be silly to write our program in machine code – especially since this is so difficult. Higher level languages were created so that not only can we avoid machine language, but we can write in a language closer to the way we design our programs. That is, nearer to the algorithm.

This means that, once we have chosen the language, we need to be able to translate our program from the programming language source file to a machine executable form. This process is called **compilation**, although there are usually several steps in the translation. Let's restrict or source files to C++, although the concepts can be generalised. Let's start with what is contained in a single C++ source file.

If we wrote a program using just the syntax understandable to the C++ compiler, we'd have many difficulties. We rely on **libraries** to provide already written components of our programs, whether it be input and output, or mathematical functions, or operating system interfacing. Our C++ program uses compiler **directives** to indicate to the compilation process to incorporate, in our source file, material provided by the compiler writers to handle such tasks. In fact these directives (such as `#include`) are not really for the compiler, but the **pre-processor**. This is the first of the sub-steps in the compilation process.



The pre-processor replaces any such directives by the appropriate code and passes the resultant source on to the compiler. Even though there may be many lines of source code added in the process, the original line numbers of the source file are maintained, so that the compiler can refer any syntax errors to the original source lines. The pre-processor is a program called `cpp`. This program outputs to standard output. Thus, to see what the pre-processed form for a source file called `x.cpp`, just enter

```
$ cpp x.cpp
```

We can pipe the output through `less` or redirect it to a file for inspection. (The output is not very interesting.)

The next sub-step is the **compiler**. We are using the GNU compiler `g++` to compile our C++ programs. On many computers, there is a choice of compiler. On Solaris, for example, Sun provides a compiler called `CC`. When we invoked the compiler to process our source file as in

```
$ g++ x.cpp
```

We are asking for the entire compile process to be carried out, unless errors cause a termination at some point in the sequence of sub-tasks. If no errors are encountered, the result of the compilation process is the executable which, by default, is called `a.out`. If there are syntax errors, the compile process will be terminated, with error messages sent to standard error. This output can be redirected to a file but is a two-step redirection. Firstly the standard error output is directed to standard out, then the usual file redirection can be used, as in

```
$ g++ x.cpp 2>&1 > file
```

(`2>&1` means output 2 – stderr is input to 1 – stdout. Note the use of `&`, just as with output parameters.)

Without special request, the sub-tasks are hidden from us, and the 'files' displayed above as the result of the compiler and assembler stages are not retained. By using compiler options, we can force the sub-steps to become more apparent. So we can terminate the compile process at the pre-assembler stage by using

```
$ g++ -S x.cpp
```

which creates a file `x.s` which contains the assembly code. As the header files mostly contain prototypes and definitions, they do not contribute much to the size of this file.

Let's move on to the **assembler** step. This converts the assembly code into relocatable object code. This code is relocatable in that the addresses used in memory are all relative to a base, so that it does not matter where in memory such code will be placed. The code contains unresolved references to library routines, but also external references to any functions defined in the source code, so that such functions can be used by other blocks of object code. In fact, a compiler option allows us to stop the compile process at this point by using

```
$ g++ -c x.cpp
```

which outputs a file called `x.o`. We will see soon that, once a source file contains no syntax errors, this object code file can be used in its stead, avoiding the compile and assembler stages from construction of the executable code. These .o files are not human readable.

One of the results of the steps so far in the compile process is that much of the information about the original source file has been lost. Variable names have been replaced by memory locations. Comments disappeared at the pre-processor step. Line numbers may still be evident. To aid in the debugging process, we can ask that the compiler maintain the information regarding the relationship between variable names and memory locations, called a **symbol table**, into the executable. We do this be using another compile option.

```
$ g++ -g x.cpp
```

which can also be used with the `-c` option.

The final step in the compile process is the **linker**, a program called `ld` on Linux. This step endeavours to resolve any references to library functions or between source files. `g++` automatically calls `ld` with the appropriate information about where to find the libraries. It would be more difficult for a user to try to call `ld` directly. However, if we already have a `.o` file, we can use `g++` to perform the linking process as in

```
$ g++ x.o
```

Errors that occur at this step are reported as `ld` errors, usually involving undefined references or multiple references. When not interrupting the compile process by using the appropriate options, look for the `ld` indicator when errors are occurring at the link stage.

This covers how we can compile one source file. As our programs get more complicated, the design process dictates that we break our program up into more than one source file.

2. Organising Multi-file Programs

When we first learn to program, our programs are simple. Traditional procedural programming involves a structure of IPO: input followed by processing producing output. Our programs are designed by looking at the form of the input to the program, what is to be done to that input and how the results of that processing is to be reported.

In our earlier lectures, we followed a different design process, called **object-based design**. Here, we looked at the information contained within the program and then decided what was required to input such entities, what activities could be performed on the information and how we might output them. More on this topic was covered in other courses. At this point, we merely indicate that this is just another way of looking at what is required by our program. The result of either approach is a main function which coordinates input, processing and output, and functions which perform these tasks. The object-based approach leads to many functions that are directly related to internal data storage. These are easily recognised as a distinct set of functions which can be separated from the main function by placing them in a source file of their own. Procedural design usually requires the programmer to identify the similarities between functions and determine the best collecting of like functions into separate files.

When a program has more than one source file, it means that functions may be called in one source file, while the code for that function is in another. So that the file with the call knows about the function, C++ requires a **prototype** to appear in the file before the function is used. When we have only one file, we often just place the

prototypes at the top of the file, near where we include the header files for libraries. With multiple files we set up header files of our own, containing the prototypes of functions and definitions of data structures that files using these functions need to know. Our header files use the notation that C++ use to use for library headers, namely a file with the same name as the source of the functions but with extension `.h`. For example, suppose we have a set of functions to handle a video store, as in our revision example. We could place those functions in a source file called `Video.cpp`. The prototypes of any functions within that source file that we want to be able to call from another source file are placed in the file `Video.h`. If we have any definitions of data structures that also need to be known outside of `Video.cpp`, we place those in the header file as well. If functions are used internally in `Video.cpp`, or definitions are local to the source file, don't put them in the header. The header is an interface to the contents of `Video.cpp`, not a list of what `Video.cpp` needs. Now, wherever these prototypes and definitions are needed, even `Video.cpp` itself, we can include the header file as

```
#include "Video.h"
```

Note the different syntax to the use of library headers. Library headers are enclosed in < >. The C++ pre-processor searches a specific sequence of file locations for library headers. For user headers, enclosed in " ", the pre-processor looks in the current directory. You can specify where to look for headers, both library and personal, by using the option

`-I`*`pathname`*

on the compile command, where the *`pathname`* is either relative or absolute. Similarly you can tell the linker where a library is located by using the option

`-L`*`pathname`*

to show where the code for the library can be located, if it is not found in the usual set of directories. You may wonder why the latter is needed. It turns out that users can create their own libraries – we'll leave that to the experts.

Some reminders about include:

Don't use include to place source code contained in one file into another source file. Use only for including header files.

Never place variable declarations in a header file. Such variables will be considered global to more than one file and hence cause errors when the files are compiled.

Don't place includes of other headers in headers, unless they are needed by the contents of the header, not the content of the source file. We'll see later how pre-processor directives can avoid multiple inclusions and recursive inclusion.

2.1 Compiling multi-file programs

When such programs were encountered in CS111, the simplest form of compilation was introduced. If all the source files are contained in a directory, and are the only source files in that directory, then the UNIX wild card can be used as

```
$ g++ *.cpp
```

where the `*.cpp` is expanded into the list of all `.cpp` files in the current directory. For example, if x.cpp, y.cpp and z.cpp are the source files for the program and the header z.h is also to be used, the above command would be interpreted as

```
$ g++ x.cpp y.cpp z.cpp
```

The header file does not appear in the compile command – it is added to the code by the pre-processor. The problem with this 'shotgun' approach is that, if we've already got the errors out of `y.cpp` and `z.cpp` and are only working on `x.cpp`, this command repeatedly compiles the other two files. The following section describes how UNIX provides a better approach.

3. make

As our programs become more complex, with numerous interdependencies between files, a simple compile line involves unnecessary recompilation of program files that have been successfully compiled before. The UNIX command **make** provides a facility to describe these interdependencies and how to build a complex program from source files to executable. make can also determine which files have to be recompiled because of recent changes.

make operates using three sources of information:

- a user-supplied descriptor file usually called Makefile or makefile
- filenames and last-modified times from the file system, and
- built-in rules to bridge some gaps

Let's start with an example not using any of the innate knowledge of make. Suppose we have a program, whose executable is to be called prog, and is made up of three C++ program files x.cpp, y.cpp and z.cpp where the latter has a header file called z.h which the first of the files also uses. Here are the contents of the simplest Makefile:

```
prog: x.o y.o z.o
      g++ x.o y.o z.o -o prog
x.o: x.cpp z.h
      g++ -c x.cpp
y.o: y.cpp
      g++ -c y.cpp
z.o: z.cpp z.h
      g++ -c z.cpp
```

There are two different types of lines in this example. The first is a **dependency line** which starts in the first character position of the line with a target filename, is followed by a :, a tab, and then a list of files on which this target file depends. For example the line starting with x.o indicates that the file x.o depends on x.cpp and z.h. The make command checks to see if x.o has a last-modified date later than the files it depends on. If any of the files is newer, then make would know that x.o needs to be "remade".
So if we said

```
$ make x.o
```

then make would do nothing if x.o is up-to-date. Otherwise it would perform the **command strings** on the lines following the dependency line. The command strings appear on line(s) following a dependency line and have a **tab** as their first character.

If we typed

```
$ make prog
```

then make would look for the dependencies for prog, then check any of its dependencies for "up-to-datedness". Without an argument make tries to build the first target file in the makefile. If any command string fails, the make is terminated. If we use the -n option on the make command, we'll get a printout of what make would do, but it would not do it.

What else can appear in a makefile?

Comments, as always, are important for readability. Any line starting with a # is ignored as a comment.

Apart from the manufacture of program components, make can be used to clean up files no longer needed, such as .o files once a final (working) executable has been made. So, we can include requests such as

```
clean:
      rm *.o
```

Within a make program we can use simple substitutions in dependency lines and command strings. These are called macros and their values are recalled by a leading $ sign. However, if the macro is more than one character, it has to be encased in ().

So, in our example above, we could define

```
OBJECTS = x.o y.o z.o
```

and then say

```
prog: $(OBJECTS)
      g++ $(OBJECTS) -o prog
```

Note how spaces can appear in the definitions. This is because a macro is merely a text substitution, terminated by the end-of-line. We can even have macros with number names such as

```
2 = xyz
```

We can also define macros to mean nothing by placing nothing (other than a newline) after the =.

We said earlier that `make` knows how to do certain things without being told. These transformation rules are stored in an internal table that looks like a `makefile`. We'll restrict ourselves to C++, although `make` also knows about some other languages. Here's what `make` (on Ubuntu Linux) knows about C++.

```
CC = g++
CFLAGS =
.cpp.o:
      $(CC) $(CFLAGS) -c $<
```

The `CC` macro defines the name of the compiler. `CFLAGS` specifies any compiler flags (none here). And the `.cpp.o` specifies that the rule is for converting a `.cpp` file into a `.o` file. The `$<` substitutes as the file being converted. Thus, if a C++ file does not have any dependencies – other than itself of course – then we need not specify how to make it. If it has any others, such as for `x.cpp` and `z.cpp` above, we need only specify that dependency and no command string as in

```
x.o z.o: z.h
```

Note that on Solaris systems (Sun UNIX) `.cpp` is not a regular extension for C++ program files (whereas `.C` and `.cc` are) and hence there are no default rules for converting `.cpp` files to `.o` files. Thus, a complete `make` command string would be required. So that `make` files are more portable, it is advised to avoid the default rule by using the dependency rule and its command string explicitly as in

```
x.o:  x.cpp z.h
      $(CC) $(CFLAGS) -c x.cpp
```

4. Program Testing

4.1 The Properties of Good Software

When we write our own programs we should aim to produce software that has the properties that we expect of software we buy. These properties include:

Dependability
- The software should produce the correct results when provided valid input
- It should handle erroneous data in a sensible fashion – good error reporting
- The software is robust – it does not crash

Usability
- The user interface should be for novice and expert alike
- Inputs and outputs should be self explanatory

Efficiency
- Software should not waste resources – either memory or time

Documentation
- User documentation should be comprehensive and readable

Maintainability
- If problems occur with the software someone who may not be the original author should have enough information to correct the software – given access to the source

Some of these properties involve the ability to write English – either documentation that comes with the software or comments within the code. As a code developer, you should place enough comments in your source files so that you **and** other programmers can understand what's going on at all times. It is not necessary to explain the intricacies of C++ – another programmer will know that.

In industry, you are unlikely to be the sole writer of a complete program, so you need to be able to follow specifications provided for the component of the program you are to develop. This also means you need to follow guidelines provided regarding layout of the component, input order and form, output organisation, function prototypes, and error messages to be reported. This is also true of assignments and laboratory work in this subject. As someone else is going to run your components, perhaps as part of a larger program, and test whether your program runs by entering data in an expected form to see if the output is correct and arranged properly, you need to ensure you follow any instructions given.

So, how do we go about achieving these properties in our programs, where we are the sole programmer? Obviously, some are a little intangible, especially reliability and maintainability. Even purchased software products aren't error-free. Large programs are difficult to maintain so there has to be rigorous testing to ensure that the software behaves reasonably in most situations. Reliability is a measure of how successfully the program's observed behaviour conforms to the specifications.

Often the problem of writing good software comes down to an understanding of the kinds of problems a program can exhibit. Software **failure** is where there's any deviation of the observed behaviour from the specified behavior. A software **error** is a state where further processing by the program will lead to a failure. A software **fault** is the mechanical or algorithmic cause of an error.

So our final program may exhibit problems due to:

Faulty design – the planning stage missed the aim of the project
Algorithmic – process does not give the desired output
Data – user gives the program invalid or malformed data which the program processes
User Interaction – user gives incorrect data but expects the correct answer

So, how do we go about dealing with these problems?

4.2 Error Prevention and Detection

We try to not introduce errors during the construction of the program. We use a good programming methodology to reduce the complexity of the design process. For procedural design, we use a top-down approach to reduce each sub-task to a small function. With object-based design, we consider each object (dataset) to have a number of **methods** associated with it – input, output, processing tasks. Each of these smaller components are created separately and tested separately – before combining into the larger project. Version control can be used to ensure consistency from one development step to the next.

Error detection is performed once the program is running. Often faults described above will not be detected until then. This process is easier if three things are done: testing, debugging and monitoring.

4.3 Testing

Many programmers see testing as dirty work – their programs always work! To develop an effective test you must have a detailed understanding of the program, a knowledge of testing techniques and the skill to apply these techniques in a n effective and efficient manner.

Testing is best done by independent testers primarily because developers view their programs as perfect. Sometimes a different perspective is useful. Programmers often test a program with the same data set – not very creative. When planning testing, think as the end user and try many different scenarios.

Programmers should always be mindful that your program will never work perfectly for third parties the first time. Users bring out the worst in programs – a good thing.

There are many ways to test a program:
(i) Unit Testing. Test the individual components such as each function. Confirm that the subsystems correctly function as intended.
(ii) Integration Testing. Test how the components interact together.

(iii) Systems Testing. Test whether or not the system meets the specified requirements.
(iv) Acceptance Testing. Evaluate the system delivered by the developers. Prove the system meets the requirements and is ready for customer use.

It should be made clear that testing and coding go hand in hand – they are not isolated.

There are three basic levels of testing:
(i) Informal – brought on by incremental coding.
(ii) Static Analysis. Hand execution by reading the source code is one. Here you "play computer" – put values on paper and follow the program, performing each line of code to see what happens. Often a programmer misses problems by performing the code as they think it works not how it really works. Often an informal presentation to others by walking through the code can find problems – often the programmer sees it before the others. In a group, a formal presentation provides the rest of a group a more detailed inspection of the code.
(iii) Dynamic Analysis. This involves formal testing techniques called **Black Box** and **White Box Testing**.

Before we start with any form of testing we need **test cases**. These are formally developed sets of data, along with the expected output. We have to know what the program is supposed to do.

4.3.1 Black Box Testing

This is where the tester treats the program as a black box. There is no detailed knowledge of what is going on in the program. All that is known is what the data is supposed to be and what the output for the data is supposed to be. The goal is to reduce the number of test cases by equivalence partitioning. Divide the input conditions into equivalence classes, where inputs in a class expect the program to behave in a consistent or identical manner. Then choose one test case for each equivalence class. For example, if the project is supposed to accept a negative number, testing one negative number is enough.

The criteria for test cases are:
(i) Coverage – every possible input has to belong to one of the equivalence classes;
(ii) Disjointedness – no input belongs to more than one equivalence class;
(iii) Representation – if the execution demonstrates an error on a member of the class, others will demonstrate the same.

For example, if you are testing a program which accepts data, you would have test cases from three equivalence classes: within the range of expected values, below the range, and above the range.

Suppose you have a calculator. If you enter 3.14159 and use the square root key, you get 1.772453102341. Black Box testing isn't interested in how the result was created – just that it is the correct result.

4.3.2 White Box Testing

This is a more thorough form of testing – each statement in a component of the program needs to be executed at least once to ensure everything is working. So the source code is required – plus a knowledge of the language it is written in. The tester can see inside the box. With that knowledge the tester can see what forms of input can exercise different parts of the code.

There are four types of white box testing:
(i) Path Testing – all the logical flow paths of the program are executed;
(ii) Statement Testing – a single statement is tested;
(iii) Loop Testing – focus on loops: do they execute 0 times, one, many times?
(iv) Branch Testing – each possible outcome of a condition is tested at least once.

White Box testing is in part about determining paths. Consider the following function.

```
void FindMean(ifstream& ScoreFile)
{
      double SumOfScores = 0.;
      int NumberOfScores = 0;
   1  double mean, score;

      ScoreFile >> score;
   2  while (!ScoreFile.eof())
```

```
    {
3       if (score > 0.)
        {
            SumOfScores += score;
4           NumberOfScores++;
        }
5       ScoreFile >> score;
    }
6   if (NumberOfScores > 0)
    {
        mean = SumOfScores/NumberOfScores;
7       cout << "The mean score of the " << NumberOfScores <<
              " non-negative values is " << mean << endl;
    }
    else
8       cout << "No valid scores found in the file." << endl;
}
```

By identifying the different blocks as numbered above, we could draw a flow diagram of this function and work out test cases to check each of the eight parts. We could set up test files containing no negative values, all negative values, even an empty file.

4.3.3 So what should you do?

White Box testing often tests what has been done instead of what should be done – programmers are often unable to see their own errors.
Black Box testing is about testing behavior and ensuring that behavior meets requirements.
Develop test cases – at the same time as developing your code – so that your program can be tested at all steps of the design process.
Test branches, loops, records, fields, and arrays.
Check for divisions by zero and non-initialisations.

When writing your program, keep functions small. This will introduce less errors, keep the code readable, and be more easily tested.

In top-down design, you should ideally test the overall flow of control between your main program and functions before implementing them. Even object-based design can also be built this way – using a driver program as a temporary main to exercise the functions. Use **stubs** to test this – an empty function body that merely satisfies the prototype. Even a simple output statement indicating that the function has been called helps in the analysis of the flow.

Get into the habit of unit testing by testing each function individually. Once you are convinced that the functions work as specified, combine them.

Importantly, use any tools available on the programming platform to help you. The following section discusses such tools available on Unix/Linux systems. Many programmers also use IDEs – integrated development environments where the editing, compiling and debugging of programs are contained within one system. While these are useful, you may find yourself, in the future, on a computer without these tools or with one you are unfamiliar with.

5. Debugging

As a result of the testing procedures above, we may find that a program under development is faulty. We need procedures to assist in locating the faults, be they design-related or algorithmic. A knowledge of the program is, of course, required.

5.1 Debug statements

The simplest form of debugging that requires no tools at all is the addition of output statements in the code to determine both the flow and the validity of partial results. For this to work efficiently, you must understand the flow of the program. A simple

```
cout << "got here\n";
```

is often enough to know that the execution of a particular block has occurred. It is advisable to use `cerr` as the output destination, as `cout` is a buffered device, and the program may crash before the buffer is cleared.
If you know the expected value of a variable at a particular point in the code, try outputting the value to see if it is what is expected.

Sometimes, we would like to be able to turn debugging statements on and off. There are some pre-processor directives that allow for debugging code to be included or not included in the compiled program.

5.1.1 More pre-compiler directives

Prior to C++ and its ability to set identifiers to be constant, C programmers had to use symbol substitution by the pre-processor to incorporate constants. The command

```
#define ident value
```

specified that the pre-processor replace every occurrence of the identifier `ident` with the characters contained in the `value`. For example, a simple constant definition would be

```
#define PI 3.14159265
```

All occurrences of the identifier `PI` in the code following this statement would be replaced by the constant. This doesn't mean sequences of characters like `applePIe` would be altered, nor `PI` inside a comment or a string constant – only where `PI` was a valid identifier.
But the value is not restricted to a numerical value, or even any legitimate C++ value. It can be any expression at all. For example,

```
#define ABC a + b + c
```

would substitute the characters "`a + b + c`" (without the quotes) for occurrences of `ABC`. Such code substitutions are **not** advised – let's leave it in C in the past.

There are preprocessor directives that can be used like if tests to conditionally include code in the compilation. But we will restrict ourselves to one particular form. When a symbol appears in a `#define`, even if the value is missing, then the pre-processor considers the symbol to be defined. So

```
#define DEBUG
```

would indicate to the pre-processor that a symbol `DEBUG` can be considered defined. The reverse is the command

```
#undef DEBUG
```

which would have the pre-processor forget the existence of `DEBUG`.

With this concept we can then surround our debug code with

```
#ifdef DEBUG
      cerr << "a debug statement\n";
#endif
```

to only include the code if `DEBUG` is defined. In this way debug code can be turned on and off throughout a piece of code by judicious use of defining and undefining a symbol. We can even test

```
#ifndef DEBUG
```

to see if a symbol is undefined.
If our program has debug statements protected by `ifdef…endif` blocks, we can turn these blocks on and off from the compiler command by using

```
$ g++ -DDEBUG main.cpp
```

to define the symbol `DEBUG`, meaning no `#define` is needed.

5.1.2 Multiple header inclusions avoided

When we have a complex program where header files refer to other header files, we must ensure that any header file is not included more than once. We can use the pre-processor directives above to ensure this.

For any given header, say `header.h`, we can define a pre-processor symbol `HEADER_H`, changing the text to upper case and the period to underscore. We then encapsulate the contents of `header.h` by

```
#ifndef HEADER_H
#define HEADER_H
      Contents of the header
#endif
```

This also protects from recursive includes. The name of the symbol used is not a requirement, merely a way of avoiding already existing symbols.

5.2 Assert

The assert library provides what may be a preprocessor macro (yes, it is a C library, with header file `cassert`) called

```
assert(expression);
```

When this statement is executed, the (logical) `expression` is evaluated. If its value is true, the program continues. Otherwise, we get a message on standard error displaying the expression, the name of the source file, and the line it occurs on. The program then terminates.

This is a useful debugging tool. If, at a certain point in a program, a particular expression should be true, an `assert` containing that expression will cause the program to terminate if it is not true. For example,

```
assert(x < 10);
```

would be ignored if the variable `x` is less than 10, but would output the message and terminate if not.

This is **not** a replacement for an error message – only the author of the program will understand the meaning of the message. Asserts can be turned off by using the pre-processor symbol `NDEBUG`, either by `#define` or by compiler directive `-DNDEBUG`.

5.3 Using a debugger

As mentioned earlier, many programmers use an IDE, which provides many tools to assist in the debugging of programs. We'll discuss two Linux debuggers which provide similar tools, although not integrated into the editing process.

5.3.1 gdb

This is the standard text-based Linux debugger. (Unix has a similar tool called `dbx`.) To enable a debugger to have access to the source code of the program, we need to make the compiler maintain the symbol table by using the `-g` option on the compiler.

Once the program is compiled, the debugger is started by

```
$ gdb executable
```

where `executable` is the name of the program to be debugged, such as `a.out`. The program is loaded into a runtime environment which allows the user to control the execution via `gdb` commands. Once the gdb environment is loaded (after a number of lines of ignorable output), the program can be run by typing

```
(gdb) run
```

This will then run the program to completion without debugging.
If, however, there is some fault that causes the program to crash, then `gdb` will provide access to the program for debugging. You can also type ctl-C to stop the program.

The command

`(gdb) where`	will tell us the active functions on the stack – called a **walkback**.

That is, it will show us where we are, where the current function was called from (and what the arguments were), and continue to show this back to the main call. Then we can use the power of gdb to interrogate the program.

The command

`(gdb) start`	will take you to the `main` function ready to begin execution.

To see the code

`(gdb) list`	will list ten lines around the current line
or	
`(gdb) list n`	list 10 lines about line `n` of the current file
or	
`(gdb) list m, n`	list lines `m` to `n`

Once the code is ready to execute – at the main – we can run the code in many different ways.

`(gdb) step`	will execute one line and stop on a function call. If `step` is entered at a function call, `gdb` will go into the function and step through the lines of code.
`(gdb) step n`	will execute the next `n` lines.
`(gdb) next [n]`	executes the entire function or a specified number of lines.
`(gdb) continue`	will execute to termination (or to next break point).

Whenever the program is not running, we can define **breakpoints**. A breakpoint is like a stop sign. Whenever `gdb` gets to a breakpoint, it halts execution of the program.
To set a breakpoint, just enter

```
(gdb) break [filename:][linenumber]
```
as in
```
(gdb) break main.cpp:72
```

Breakpoints can also be defined on function names (as above, instead of `linenumber`). `gdb` will stop prior to entry into the function. When a program has many breakpoints, keeping track of them may get difficult.

`(gdb) info break`	lists the current breakpoints with a number associated.
`(gdb) delete breaknumber`	deletes a specified breakpoint. No number deletes all breakpoints.
`(gdb) disable breaknumber`	disables a breakpoint temporarily.
`(gdb) enable breaknumber`	enables a breakpoint.
`(gdb) ignore breaknumber [n]`	ignores a breakpoint for a number of times.

The purpose of breakpoints is to pause execution so that we can inspect what is going on in the program at that point. To inspect data, the following `gdb` commands are available.

`(gdb) print expr`	prints the value of the expression at the current paused location.

If a particular variable is required, but it is not in the current scope, prefix the variable name with either the function name, or the source file in single quotes, as in

```
(gdb) print main::x
```
or

`(gdb) print 'subs.cpp'::val`	for a global in a different file.
`(gdb) display expr`	prints the value of the expression each time execution stops. `undisplay` stops the printing.
`(gdb) whatis var`	prints the datatype of `var`.
`(gdb) info locals`	displays the value of all automatic variables in the current function

If a particular variable has a value it shouldn't, it can be changed using

```
(gdb) set var = expr
```

5.3.2 ddd

If you don't like a command line driven debugger, Linux provides a GUI frontend called `ddd`. Unix's version is `xdb`.) This provides a window-based display of source code, variable values and menu-operated commands to perform all the above. There's even a printout of the equivalent `gdb` commands. The subject website has an online (html-based) manual for using `ddd`. Practice makes perfect. Examples of `ddd`'s use will appear in the lectures across future topics to show the worth of such analysis.