This document is part of the HTML publication "**An Introduction to the Imperative Part of C++**"

The original version was produced by **Rob Miller** at **Imperial College London**, September 1996.

Version 1.1 (modified by **David Clark** at **Imperial College London**, September 1997)

Version 1.2 (modified by **Bob White** at **Imperial College London**, September 1998)

Version 1.3, 1.4, 2.0, ..., 2.15 (modified by **William Knottenbelt** at **Imperial College London**, September 1999-September 2016)

Appendix 2: Debugging

A.2.1 General Tips on Debugging

The "`assert`" Function

A useful debugging procedure is to put the include directive

```
#include <cassert>
```

at the top of your program (old-style header `assert.h`), and add various statements of the form

```
assert(logical-expression);
```

at key points within it, to check that certain conditions which you think should be true (at those points) do indeed hold. If the program executes the above statement and the expression "*`logical-expression`*" is false, the program will terminate at that point with an appropriate error message. "assert" statements can be especially useful to check maximum/minimum value conditions, e.g.:

```
...
assert(my_var >= 0 && my_var <= MAXIMUM_VALUE);
...
```

or check for the success of file operations:

```
...
assert(!in_stream.fail());
...
```

You can "turn off" and "turn on" the assert checking by respectively adding and deleting (or commenting out) the line "`#define NDEBUG`", before the line "`#include <cassert>`". (See Savitch, Section 5.5, for further information about the "cassert" library.)

(BACK TO COURSE CONTENTS)

Stubs and Drivers

It is good practice to test each function you write with a separate short "driver" program before using it in other longer or more complex programs. The driver program might typically be a simple "while" loop that allows you to pass various parameters to the function from the keyboard, until you are satisfied that the function works properly.

If you are unsure about the overall control flow of your main program, it is often convenient to first test it with "stub" functions, which merely return an arbitrary value of the correct data type. This technique is especially useful when using a top down design method, and facilitates procedural or functional abstraction.

(BACK TO COURSE CONTENTS)

Some Common Errors to Look Out For

- Typing "=" instead of "==" (and vice-versa). When comparing a variable with a constant, it is possible to avoid some of these errors by getting into the habit of putting the constant on the left. For example, the compiler will usually spot the error

  ```
  if (0 = my_var) ...
  ```

  but won't spot

  ```
  if (my_var = 0) ...
  ```

- Omitting a ";" particularly before a "}". Note that C++ compilers often "read ahead" by 2 or more tokens in this situation before realising there is an error, so you may have to look back several lines in the program.

- Forgetting the "()"s when calling a function with no parameters. e.g.

  ```
  x = foo();
  ```

  not

  ```
  x = foo;
  ```

- Putting the wrong case in identifiers. "FOO", "foo" and "Foo" are all different identifiers. By convention upper case is usually used for constant identifiers (and macros).

- Omitting a "#include" at the beginning of a program. Suspect this if common library functions are reported as unidentified.

- Mistaking a value parameter for a reference parameter and not getting the desired side effect from a call to the function.

- Omitting brackets round a conditional test. E.g. should be

  ```
  if (a<7) a+=1;
  ```

- Using integer division when floating is required (and vice versa). E.g. you may need

  ```
  i/2.0
  ```

  rather than

  ```
  i/2
  ```

- Getting the test and increment the wrong way round in a "for" statement. E.g. don't write

  ```
  for(int i=0; i++; i<10)
  ```

- Using "<" in a for loop when "<=" is required (and vice versa). E.g.

  ```
  for (i=1; i<10; i++) {...}
  ```

only loops 9 times.

- Assuming an array "a" of length L has an element a[L]. It doesn't, it only has elements a[0] to a[L-1].
- Having a "#include" directive in a header file and source file. These can be in one place or the other, but not both.
- Doing a "#include" on another source file rather than a header. If your program has several files, each should be added in the project list (then compiled separately and linked together).

Debugging Very Bad Programs

If your program is very bad, the best way to debug it is to throw it away and start again.

(BACK TO COURSE CONTENTS)

A.2.2 The GNU debugger gdb

The gdb debugger is a powerful tool for tracking down errors in your programs. You can run gdb from the UNIX prompt by typing **gdb *program*** or you can run gdb from inside emacs by typing **ESC-x gdb** (this has the advantage that emacs will indicate the current program line by placing an arrow '=>' in the left hand margin of the corresponding source file). To include debugging information remember to compile your program with the **-g** option.

Below is a summary of the most useful commands. For a full description of all available commands type **info gdb** at the UNIX prompt; alternatively consult the official gdb manual page or try a good gdb tutorial.

```
r            run program until breakpoint, natural termination or fatal error
c            continue/resume program execution
s            step until next line (will step into function calls)
n            continue until next line (steps over function calls)
finish       continue until function in current stack frame returns
break        set breakpoint (argument can be function or line number)
watch        set watchpoint (argument is an expression; debugger will
               stop execution whenever expression changes)
info break   list all breakpoints and watchpoints
where        show stack trace
up           move to higher stack frame
down         move to lower stack frame
print        displays the values of a variable in the current stack frame
quit         quits the debugger
```

(BACK TO COURSE CONTENTS)
