GCC: the GNU Compiler Collection

We will be primarily concerned with the C compiler, `gcc`.

The program `gcc` is actually a front-end for a suite of programming tools.

For the purposes of CS 2505, the underlying tools include:

`cpp`	the GNU C preprocessor
`cc`	the GNU C language compiler
`as`	the GNU assembler
`ld`	the GNU linker

We will begin by considering only the use of `gcc` itself.

Getting Started with gcc

Download the example `caesar.c` from the course website if you want to follow along with the following examples.

Execute the following command: `gcc caesar.c`

You should not get any messages; list the files in your directory and you'll find a new file named `a.out` – that's the executable file produced by `gcc`.

Execute the command `a.out`; you should see a message from the program showing how to invoke it correctly.

Execute the command `a.out` with valid parameters, say:

```
a.out 2 AMansAManForAThat.txt
```

and you should see the original file, unmodified, echoed to the console window.

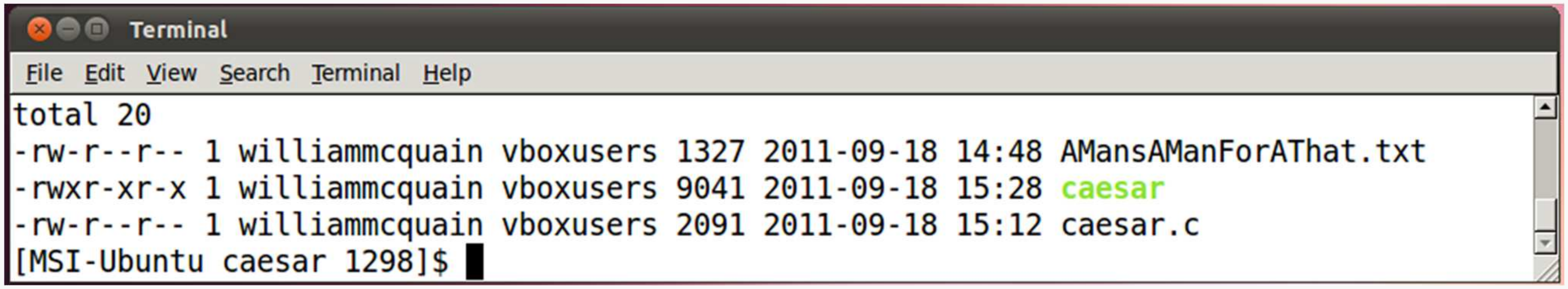
That may not seem surprising since a critical function has an empty implementation.

Fundamental gcc options: -o

First of all, the default name for the executable file is `a.out`, which is both strange and unsatisfying.

Use the `-o` option to specify the name you want to be given to the executable:

```
gcc -o caesar caesar.c
```



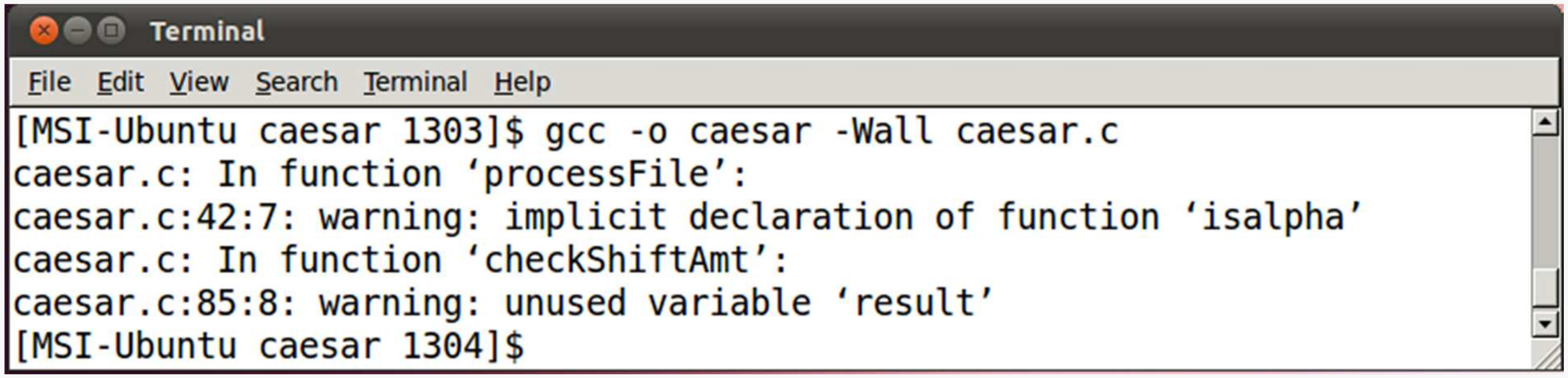
```
total 20
-rw-r--r-- 1 williammcquain vboxusers 1327 2011-09-18 14:48 AMansAManForAThat.txt
-rwxr-xr-x 1 williammcquain vboxusers 9041 2011-09-18 15:28 caesar
-rw-r--r-- 1 williammcquain vboxusers 2091 2011-09-18 15:12 caesar.c
[MSI-Ubuntu caesar 1298]$
```

Side note: as is often the case, the space after the `-o` option is optional.

Fundamental gcc options: -Wall

Use the `-Wall` option to direct `gcc` to display all relevant warning messages:

```
gcc -o caesar -Wall caesar.c
```



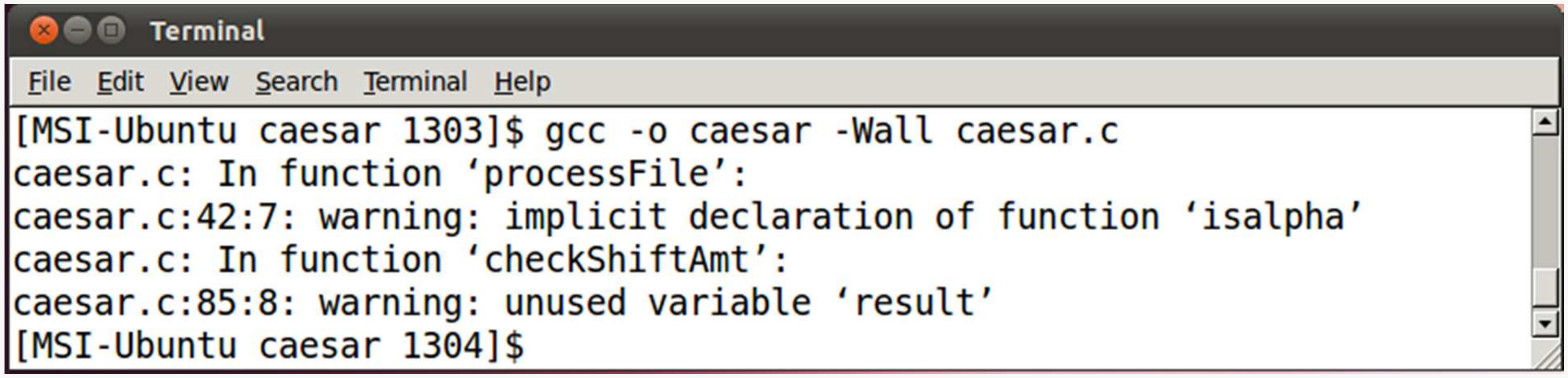
```
[MSI-Ubuntu caesar 1303]$ gcc -o caesar -Wall caesar.c
caesar.c: In function ‘processFile’:
caesar.c:42:7: warning: implicit declaration of function ‘isalpha’
caesar.c: In function ‘checkShiftAmt’:
caesar.c:85:8: warning: unused variable ‘result’
[MSI-Ubuntu caesar 1304]$
```

So, the supplied C code *compiles*, but does not *compile cleanly*.

The first two lines tell us that at line 42 of the file `caesar.c`, in the function `processFile`, we have called a function `isalpha` that has not been declared explicitly.

All too true. The Standard Library function `isalpha` is declared in the header file `ctype.h` and the supplied code doesn't have an `include` directive for that header; in this case, we got away with that, but the issue should be fixed.

Fundamental gcc options: -Wall



```
[MSI-Ubuntu caesar 1303]$ gcc -o caesar -Wall caesar.c
caesar.c: In function ‘processFile’:
caesar.c:42:7: warning: implicit declaration of function ‘isalpha’
caesar.c: In function ‘checkShiftAmt’:
caesar.c:85:8: warning: unused variable ‘result’
[MSI-Ubuntu caesar 1304]$
```

The next two lines tell us that at line 85 of the file `caesar.c`, in the function `checkShiftAmt`, we have declared a variable `result` whose value is never used.

Again, this is true.

However, in this case the variable was used in order to capture the return value from the library function `strtol`, which we do not make any further use of.

This is deliberate, and fits with the design of the function `checkShiftAmt`, and so we'll leave it unaltered.

More gcc warning options

You may also turn on certain specific categories of warnings if `-Wall` is too intrusive:

`-Wformat`

warnings regarding mismatches between format specifiers and arguments

`-Wunused`

warnings regarding unused varaibles

`-Wimplicit`

warnings regarding functions that are called without being declared; usually results from missing `include` directives or misspellings

`-Wconversion`

warnings regarding implicit conversions that could result in errors

`-Wshadow`

warnings regarding name hiding

`-W`

a variety of additional warnings not included in `-Wall`

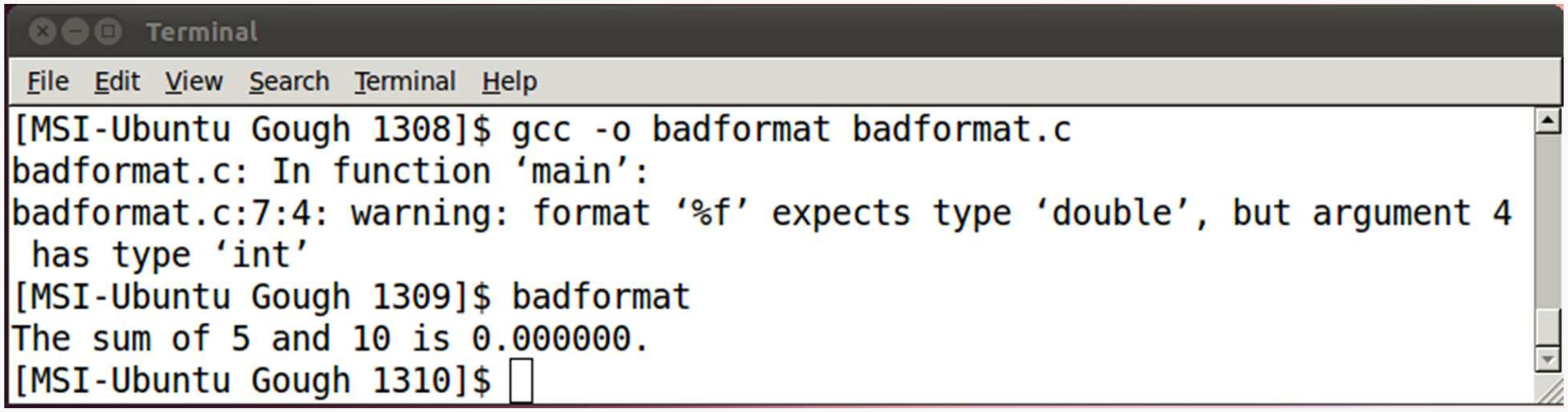
A Not So Benign Warning Message

```c
// Adapted from An Introduction to GCC, Gough, p. 8
#include <stdio.h>

int main() {

   int a = 5, b = 10;
   printf("The sum of %d and %d is %f.\n", a, b, a+b);

   return 0;
}
```



```
[MSI-Ubuntu Gough 1308]$ gcc -o badformat badformat.c
badformat.c: In function ‘main’:
badformat.c:7:4: warning: format ‘%f’ expects type ‘double’, but argument 4
 has type ‘int’
[MSI-Ubuntu Gough 1309]$ badformat
The sum of 5 and 10 is 0.000000.
[MSI-Ubuntu Gough 1310]$ 
```

Compliance-related gcc options: -std

Use the `-std` option to direct `gcc` to require code to comply with a particular C langauge stardard:

```
gcc -o caesar –Wall –std=c99 caesar.c
```

This is necessary in order to use some features added in C99 (such as declaring for-loop counters within the loop header).

It is also necessary with some legacy code in order to sidestep requirements imposed by newer standards.

Unless explicitly stated otherwise, in CS 2505 we will always specify compliance with the C99 standard (as shown above).

GCC Reference Manual

The official reference manual for GCC is available at:

http://gcc.gnu.org/onlinedocs/

This is a very useful, if somewhat verbose, resource.

Behind the Scenes

Executing `gcc` with the `-save-temps` option results in the preservation of some temporary intermediate files created by/for the underlying tools.

For example if you use the file `caesar.c`:

`caesar.i`	written by the preprocessor; input to the compiler
`caesar.s`	written by the compiler; input to the assembler
`caesar.o`	written by the assembler; input to the linker
`caesar`	written by the linker

By default, only the final, executable file is preserved once the process is complete.

We will gradually see that the intermediate files are occasionally of use, if for no reason than that they shed light on the actual process of program translation from a high-level language to the machine level.

The Preprocessor: cpp

Try executing the command

```
cpp caesar.c > caesar.i
```

`cpp` writes its output to standard output; this *redirects* it into a (new) file named `caesar.i`.

If you examine this (text) file, the first 2000 or so lines indicate the processing of the `include` directives in the source file; so declarations from those files are available to the compiler.

At the end of the file, you will find a modified copy of the original source:

- all the comments have been stripped out
- the values that were `define`d in the source file have been substituted into the source code

The Compiler: cc

Now, try executing the command

```
cc –S caesar.i
```

With the `-S` option (case-sensitive!), the compiler writes its output to a file; the name is generated from the name of the input file; in this case, the output is written to `caesar.s`.

This file contains the assembly code generated by the compiler from the pre-processed C source code.

The Assembler: as

Now, try executing the command

```
as -o caesar.o caesar.s
```

The assembler writes its output to `a.out` by default; the name can be specified using the option `-o`, as with `gcc`.

This file contains a partial translation of the assembly code into native machine language code; calls to library functions haven't been resolved completely since the code for those is not in the local source file.

The Linker: ld

Now, try executing the command

```
ld caesar.o
```

The linker will complain about a large number of undefined references because it doesn't know where to find the system files that contain the implementations of the relevant features in the Standard Library.

Fortunately, `gcc` takes care of these settings for us and invokes the linker as needed:

```
gcc -o caesar caesar.o
```