Debugging

Your program will have bugs in it

Debugging

Your program will have bugs in it

Some are algorithmic bugs: those are your problem

Debugging

Your program will have bugs in it

Some are algorithmic bugs: those are your problem

We shall look at the bugs caused by your implementation in C

Debugging

Your program will have bugs in it

Some are algorithmic bugs: those are your problem

We shall look at the bugs caused by your implementation in C

As mentioned, `valgrind` is good at finding some (but not all!) memory errors

Debugging

Your program will have bugs in it

Some are algorithmic bugs: those are your problem

We shall look at the bugs caused by your implementation in C

As mentioned, `valgrind` is good at finding some (but not all!) memory errors

But it's not a mitigation for sloppy programming: you should at least try to get it right yourself!

Debugging

The first kind of error you will come across is errors in the compilation

Debugging

The first kind of error you will come across is errors in the compilation

By which we mean errors in your program the the compiler spots

Debugging

The first kind of error you will come across is errors in the compilation

By which we mean errors in your program the the compiler spots

You should always take note of both errors and warnings produced by the compiler

Debugging

The first kind of error you will come across is errors in the compilation

By which we mean errors in your program the the compiler spots

You should always take note of both errors and warnings produced by the compiler

We shall look at a few example error messages

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)
- the function: in question `add`

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)
- the function: in question `add`
- the variable in question: `b`

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)
- the function: in question `add`
- the variable in question: `b`
- the line number in the file: `9`

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)
- the function: in question `add`
- the variable in question: `b`
- the line number in the file: `9`
- which character in that line: `40`

Debugging

```
heap2.c: In function 'add':
heap2.c:9:40: error: 'b' undeclared
```

A function used a variable `b` without declaring it

In this example we get

- the filename: `heap2.c` (we might be compiling several files at once)
- the function: in question `add`
- the variable in question: `b`
- the line number in the file: `9`
- which character in that line: `40`

Though we don't always get such fine detail

Debugging

As a contrast, Clang reports:

```
heap2.c:9:40: error: use of undeclared identifier 'b'
  printf("b = %d\n", b);
                     ^
```

Debugging

```
heap2.c:13:3: error: too many arguments to function 'add'
heap2.c:4:5: note: declared here
```

A call to `add` on line 13 had too many arguments; as a reference to compare against, `add` was declared on line 4

Debugging

```
warn.c: In function 'bar':
warn.c:5:3: warning: 'return' with no value, in function
  returning non-void
```

A warning that a function didn't return a value when it was declared to return a value

Debugging

And so on

Debugging

And so on

There are very many more messages, of course

Debugging

And so on

There are very many more messages, of course

You will become experienced in reading these kinds of messages!

Debugging

If you manage to get your program to compile without errors and warnings there are still those errors where the program is not doing what you thought it was

Debugging

If you manage to get your program to compile without errors and warnings there are still those errors where the program is not doing what you thought it was

A basic form of debugging is to put `printfs` in your code to print out the values of interesting variables as the program runs

Debugging

If you manage to get your program to compile without errors and warnings there are still those errors where the program is not doing what you thought it was

A basic form of debugging is to put `printfs` in your code to print out the values of interesting variables as the program runs

```
printf("x is %d\n", x);
x = wibble(x);
printf("after wibble x is %d\n", x);
```

Debugging

Then you can narrow down where things are going awry

Debugging

Then you can narrow down where things are going awry

Don’t underestimate this as a way of debugging programs!

Debugging

Then you can narrow down where things are going awry

Don't underestimate this as a way of debugging programs!

Exercise. Most library functions set an error message when something goes wrong, e.g., "no permission to write to file". These are described in the "ERRORS" section of their man page. Investigate the error reporting mechanism `errno`, `strerror()` and `perror()`

Debugging

For more fine-grained inspection of the running program you can use a *debugger*

Debugging

For more fine-grained inspection of the running program you can use a *debugger*

Debuggers, like `gdb` and `ddd` are line oriented or graphical tools that allow detailed control of the execution of a program

Debugging

For more fine-grained inspection of the running program you can use a *debugger*

Debuggers, like `gdb` and `ddd` are line oriented or graphical tools that allow detailed control of the execution of a program

If you use an IDE, it will probably have an in-built debugger

Debugging

For example

```
#include <stdio.h>

int main(void)
{
  int *a = 0;

  // writing to unmapped memory
  a[0] = 42;
  return 0;
}
```

produces

```
% ./buggyprog
Segmentation fault
```

Debugging

If we compile with the `-g` option, the compiler puts in extra context information that helps the debugger

```
% cc -Wall -g -o buggyprog buggyprog.c
```

Debugging

If we compile with the `-g` option, the compiler puts in extra context information that helps the debugger

```
% cc -Wall -g -o buggyprog buggyprog.c
```

We now run the program under the debugger (some messages removed)

Debugging

```
% gdb ./buggyprog
GNU gdb (GDB) SUSE (7.1-3.12)
Copyright ...
Reading symbols from buggyprog
done.
(gdb)
```

This is the debugger prompt. We can now run the program

```
(gdb) run
```

Debugging

```
Starting program: buggyprog
...
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004f4 in main () at buggyprog.c:9
9         a[0] = 42;
(gdb)
```

It broke at line 9

Debugging

We can inspect the values of variables

```
(gdb) print a
$1 = (int *) 0x0
```

This is of course the problem; `a` does not point anywhere sensible

Debugging

Debuggers can do a lot more. We can

Debugging

Debuggers can do a lot more. We can

- look at general areas of memory

Debugging

Debuggers can do a lot more. We can

- look at general areas of memory
- inspect variables in other functions: e.g., a function `foo` calls `bar`, which breaks; the debugger drops us into `bar`; we can “move up” and look into `foo`

Debugging

Debuggers can do a lot more. We can

- look at general areas of memory
- inspect variables in other functions: e.g., a function `foo` calls `bar`, which breaks; the debugger drops us into `bar`; we can “move up” and look into `foo`
- insert values into memory

Debugging

Debuggers can do a lot more. We can

- look at general areas of memory
- inspect variables in other functions: e.g., a function `foo` calls `bar`, which breaks; the debugger drops us into `bar`; we can "move up" and look into `foo`
- insert values into memory
- continue the program from where it broke (not always a good idea), possibly after patching up the values of some variables or some memory

Debugging

- put in *breakpoints*: a marker in the program; the debugger will run the program until it hits a breakpoint and then stops to allow you to inspect things

Debugging

- put in *breakpoints*: a marker in the program; the debugger will run the program until it hits a breakpoint and then stops to allow you to inspect things
- single step: run single lines of code one by one

Debugging

- put in *breakpoints*: a marker in the program; the debugger will run the program until it hits a breakpoint and then stops to allow you to inspect things
- single step: run single lines of code one by one
- And so on

Debugging

Different debuggers have different ways of doing things, but it is worthwhile getting familiar with at least one

Debugging

Different debuggers have different ways of doing things, but it is worthwhile getting familiar with at least one

Exercise. Experiment with a debugger and explore its capabilities

Libraries

A lot of code exists for you to use in your C programs

Libraries

A lot of code exists for you to use in your C programs

For example, we have extensively used `printf`, `malloc` and `free`

Libraries

A lot of code exists for you to use in your C programs

For example, we have extensively used `printf`, `malloc` and `free`

We have mentioned `atol` and hinted at others, like `cos`

Libraries

A lot of code exists for you to use in your C programs

For example, we have extensively used `printf`, `malloc` and `free`

We have mentioned `atol` and hinted at others, like `cos`

To use one of these functions we must

Libraries

A lot of code exists for you to use in your C programs

For example, we have extensively used `printf`, `malloc` and `free`

We have mentioned `atol` and hinted at others, like `cos`

To use one of these functions we must

- declare the type of the function, so the compiler knows how to to use it correctly: both the argument types and the return type

Libraries

A lot of code exists for you to use in your C programs

For example, we have extensively used `printf`, `malloc` and `free`

We have mentioned `atol` and hinted at others, like `cos`

To use one of these functions we must

- declare the type of the function, so the compiler knows how to to use it correctly: both the argument types and the return type
- make the code (binary) available to our program so our program can actually run it!

Libraries

For the first, we use `#include`

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

While `malloc` and `free` live in `stdlib`

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

While `malloc` and `free` live in `stdlib`

And `sqrt`, `cos` etc. live in the `math` library

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

While `malloc` and `free` live in `stdlib`

And `sqrt`, `cos` etc. live in the `math` library

The directive `#include <stdio.h>` reads in a file from a standard place (e.g., `/usr/include/stdio.h`) that contains declarations of types of many things, particularly `printf`

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

While `malloc` and `free` live in `stdlib`

And `sqrt`, `cos` etc. live in the `math` library

The directive `#include <stdio.h>` reads in a file from a standard place (e.g., `/usr/include/stdio.h`) that contains declarations of types of many things, particularly `printf`

It is as if the line `#include` is textually replaced by the contents of the referenced file

Libraries

For the first, we use `#include`

The `printf` function lives in the standard I/O library `stdio`

While `malloc` and `free` live in `stdlib`

And `sqrt`, `cos` etc. live in the `math` library

The directive `#include <stdio.h>` reads in a file from a standard place (e.g., `/usr/include/stdio.h`) that contains declarations of types of many things, particularly `printf`

It is as if the line `#include` is textually replaced by the contents of the referenced file

`#include` will read in any file you like and can be placed anywhere you like in your source. It is overwhelmingly used for *header files* (`.h` files) at the start of C code

Libraries

`stdio.h` is one of many standard header files

Libraries

`stdio.h` is one of many standard header files

If you look in that file you find something like

```
extern int printf(char *format, ...);
```

(simplified)

Libraries

`stdio.h` is one of many standard header files

If you look in that file you find something like

```
extern int printf(char *format, ...);
```

(simplified)

This declares `printf` to be a function that takes a string (the format string) and a variable number of other arguments of unspecifed types

Libraries

`stdio.h` is one of many standard header files

If you look in that file you find something like
```
extern int printf(char *format, ...);
```
(simplified)

This declares `printf` to be a function that takes a string (the format string) and a variable number of other arguments of unspecifed types

The `extern` says the actual code of the function is somewhere else, not right here: this is just a declaration of the type of the function

Libraries

The `#include` is a convenience for the programmer: they could define all the types of all the standard functions they use themselves in every program, but why bother. They are all in this ready-written file. Just include them.

Libraries

The `#include` is a convenience for the programmer: they could define all the types of all the standard functions they use themselves in every program, but why bother. They are all in this ready-written file. Just include them.

In `<stdlib.h>` we might find

```
extern void *malloc(long int size);
```

(simplified)

Libraries

The `#include` is a convenience for the programmer: they could define all the types of all the standard functions they use themselves in every program, but why bother. They are all in this ready-written file. Just include them.

In `<stdlib.h>` we might find

```
extern void *malloc(long int size);
```

(simplified)

In `<math.h>` we might find

```
extern double cos(double x);
```

(simplified)

Libraries

How do we know while file to include for which function?

Libraries

How do we know while file to include for which function?

Use the manual pages: `man cos`

Libraries

```
COS(3P)  POSIX Programmer's Manual  COS(3P)

NAME
       cos, cosf, cosl - cosine function

SYNOPSIS
       #include <math.h>

       double cos(double x);
       float cosf(float x);
       long double cosl(long double x);

       Link with -lm.


...
```

Libraries

That takes care of the declarations. Now where are the actual implementations of these functions?

Libraries

That takes care of the declarations. Now where are the actual implementations of these functions?

Again, in standard files, e.g., `/usr/lib/libc.so` contains the binaries for many standard functions, including `malloc` and `printf`

Libraries

That takes care of the declarations. Now where are the actual implementations of these functions?

Again, in standard files, e.g., `/usr/lib/libc.so` contains the binaries for many standard functions, including `malloc` and `printf`

The compiler will, by default, always go to this standard library and pick up the code as necessary

Libraries

That takes care of the declarations. Now where are the actual implementations of these functions?

Again, in standard files, e.g., `/usr/lib/libc.so` contains the binaries for many standard functions, including `malloc` and `printf`

The compiler will, by default, always go to this standard library and pick up the code as necessary

However, other functions, like `cos` are not automatically picked up; it's not in `libc` for a start

Libraries

If your code uses cosine you should

Libraries

If your code uses cosine you should

- `#include <math.h>` to get the type declaration

Libraries

If your code uses cosine you should

- `#include <math.h>` to get the type declaration
- compile with the `-lm` option to get the code:
  `cc -Wall ...  -lm`

Libraries

If your code uses cosine you should

- `#include <math.h>` to get the type declaration
- compile with the `-lm` option to get the code:
  `cc -Wall ...  -lm`

The `-l`*`libname`* option tells the compiler to pick up code from the standard library named *libname*

Libraries

If your code uses cosine you should

- `#include <math.h>` to get the type declaration
- compile with the `-lm` option to get the code:
  `cc -Wall ... -lm`

The `-l`*`libname`* option tells the compiler to pick up code from the standard library named *libname*

The maths library has the very short name “`m`”

Libraries

Libraries live in standard places, e.g., `/usr/lib/libm.so`

Libraries

Libraries live in standard places, e.g., `/usr/lib/libm.so`

Again, see the manual pages for the right libraries to use

Libraries

Libraries live in standard places, e.g., `/usr/lib/libm.so`

Again, see the manual pages for the right libraries to use

Multiple libraries are used in the obvious way:

```
cc ...  -lm -lGL
```

Multi-file Programs

Libraries are code that you can use that were written and compiled separately from your program

Multi-file Programs

Libraries are code that you can use that were written and compiled separately from your program

C allows us to write a program in several pieces, compile them separately, then link them all together into the final running program

Multi-file Programs

Libraries are code that you can use that were written and compiled separately from your program

C allows us to write a program in several pieces, compile them separately, then link them all together into the final running program

For example, some programs sources are too big to fit sensibly into one file

Multi-file Programs

Libraries are code that you can use that were written and compiled separately from your program

C allows us to write a program in several pieces, compile them separately, then link them all together into the final running program

For example, some programs sources are too big to fit sensibly into one file

Or you need separate people to work on separate parts of the program

Multi-file Programs

File `prog1.c`

```
#include <stdio.h>

extern int foo(int n, int m);
int g;

int main(void)
{
  int m;

  g = 23;
  m = foo(7, 11);
  printf("m = %d\n", m);

  return 0;
}
```

Multi-file Programs

File `prog2.c`

```
// stdio not really necessary here
#include <stdio.h>

extern int g;
static int hidden = 99;

int foo(int p, int q)
{
  return p*q + g + hidden;
}
```

Multi-file Programs

In `prog1.c` `main` refers to a function `foo`, defined elsewhere

Multi-file Programs

In `prog1.c` `main` refers to a function `foo`, defined elsewhere

The compiler need to know the type of `foo` to compile a call to it correctly, so we must declare it

Multi-file Programs

In `prog1.c` `main` refers to a function `foo`, defined elsewhere

The compiler need to know the type of `foo` to compile a call to it correctly, so we must declare it

The declaration is just the start of the function without the body: it contains all the information we need

Multi-file Programs

In `prog1.c` `main` refers to a function `foo`, defined elsewhere

The compiler need to know the type of `foo` to compile a call to it correctly, so we must declare it

The declaration is just the start of the function without the body: it contains all the information we need

When `main` calls `foo`, it know it should take two `int`s and return an `int`

Multi-file Programs

In `prog1.c` `main` refers to a function `foo`, defined elsewhere

The compiler need to know the type of `foo` to compile a call to it correctly, so we must declare it

The declaration is just the start of the function without the body: it contains all the information we need

When `main` calls `foo`, it know it should take two `int`s and return an `int`

So it knows to compile code to set up the arguments correctly; and code to receive the result

Multi-file Programs

The names given as parameters in the declaration are purely for the benefit of the programmer and need not be the same as the actual parameters in the function definition

Multi-file Programs

The names given as parameters in the declaration are purely for the benefit of the programmer and need not be the same as the actual parameters in the function definition

C also allows declarations of functions without parameter names:

```
extern int foo(int, int);
```

means the same thing

Multi-file Programs

The names given as parameters in the declaration are purely for the benefit of the programmer and need not be the same as the actual parameters in the function definition

C also allows declarations of functions without parameter names:

```
extern int foo(int, int);
```

means the same thing

Further, on functions, the `extern` is optional:

```
int foo(int, int);
```

C can tell it is a declaration and not a definition by the lack of a body

Multi-file Programs

`prog1.c` also defines a globally visible variable named `g` (this is bad programming. . .)

Multi-file Programs

`prog1.c` also defines a globally visible variable named `g` (this is bad programming. . .)

`prog2.c` refers to `g`, and so needs to declare its type before it is used

Multi-file Programs

`prog1.c` also defines a globally visible variable named `g` (this is bad programming. . .)

`prog2.c` refers to `g`, and so needs to declare its type before it is used

The `extern` says “here is the type of the thing, but it is actually defined somewhere else”

Multi-file Programs

`prog2.c` also declares a *static* variable

Multi-file Programs

`prog2.c` also declares a *static* variable

This is visible only within the file `prog2.c`

Multi-file Programs

`prog2.c` also declares a *static* variable

This is visible only within the file `prog2.c`

It is not visible in `prog1.c`. In fact `prog1.c` could have its own separate variable also named `hidden`

Multi-file Programs

`prog2.c` also declares a *static* variable

This is visible only within the file `prog2.c`

It is not visible in `prog1.c`. In fact `prog1.c` could have its own separate variable also named `hidden`

The `static` says “within this file only”, and is useful for hiding things from other parts of the code in other files

Multi-file Programs

`prog2.c` also declares a *static* variable

This is visible only within the file `prog2.c`

It is not visible in `prog1.c`. In fact `prog1.c` could have its own separate variable also named `hidden`

The `static` says "within this file only", and is useful for hiding things from other parts of the code in other files

C is not strong on modules/namespaces: this is the only hiding mechanism it has

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

This produces binary object files `prog1.o` and `prog2.o`

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

This produces binary object files `prog1.o` and `prog2.o`

These are not runnable, but need to be combined to produce a runnable binary: this is called *linking*

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

This produces binary object files `prog1.o` and `prog2.o`

These are not runnable, but need to be combined to produce a runnable binary: this is called *linking*

```
% cc -Wall -o prog prog1.o prog2.o
```

produces a binary `prog`

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

This produces binary object files `prog1.o` and `prog2.o`

These are not runnable, but need to be combined to produce a runnable binary: this is called *linking*

```
% cc -Wall -o prog prog1.o prog2.o
```

produces a binary `prog`

Linking resolves all the cross-file references: e.g., it determines the `g` in `prog2.c` is the same as the `g` in `prog1.c`

Multi-file Programs

We compile the source code using `-c` to make *object* files

```
% cc -Wall -c prog1.c
% cc -Wall -c prog2.c
```

This produces binary object files `prog1.o` and `prog2.o`

These are not runnable, but need to be combined to produce a runnable binary: this is called *linking*

```
% cc -Wall -o prog prog1.o prog2.o
```

produces a binary `prog`

Linking resolves all the cross-file references: e.g., it determines the `g` in `prog2.c` is the same as the `g` in `prog1.c`

```
% ./prog
m = 199
```

Multi-file Programs

The final binary must contain exactly one `main`

Multi-file Programs

The final binary must contain exactly one `main`

And exactly one definition of each global name (functions and variables)

Multi-file Programs

The final binary must contain exactly one `main`

And exactly one definition of each global name (functions and variables)

In the final linking step you would also name the various libraries needed, e.g., `-lm`

Multi-file Programs

The final binary must contain exactly one `main`

And exactly one definition of each global name (functions and variables)

In the final linking step you would also name the various libraries needed, e.g., `-lm`

Thus: use `-c` to compile only; without `-c` the compiler will compile any `.c` files, and link in any `.o` files

Multi-file Programs

The final binary must contain exactly one `main`

And exactly one definition of each global name (functions and variables)

In the final linking step you would also name the various libraries needed, e.g., `-lm`

Thus: use `-c` to compile only; without `-c` the compiler will compile any `.c` files, and link in any `.o` files

Thus `% cc -Wall -o prog prog1.c prog2.o` will compile `prog1.c` (without leaving a `.o`), then it will link in `prog2.o` to produce a runnable binary (if all went well)

Multi-file Programs

The final binary must contain exactly one `main`

And exactly one definition of each global name (functions and variables)

In the final linking step you would also name the various libraries needed, e.g., `-lm`

Thus: use `-c` to compile only; without `-c` the compiler will compile any `.c` files, and link in any `.o` files

Thus `% cc -Wall -o prog prog1.c prog2.o` will compile `prog1.c` (without leaving a `.o`), then it will link in `prog2.o` to produce a runnable binary (if all went well)

Better is to separate the compile and link steps as it separates the errors each stage might report

Multi-file Programs

In larger programs, it is convenient to gather together your various type declarations into one or more of your own header files and use `#include` to read them

Multi-file Programs

In larger programs, it is convenient to gather together your various type declarations into one or more of your own header files and use `#include` to read them

This is easier to manage and enables you to keep the various declarations in sync with each other

Multi-file Programs

```
#include <stdio.h>
#include "prog.h"
int g;

int main(void)
{
  int m;

  g = 23;
  m = foo(7, 11);
  printf("m = %d\n", m);

  return 0;
}
```

Multi-file Programs

```
// stdio not really necessary here
#include <stdio.h>
#include "prog.h"

static int hidden = 99;

int foo(int p, int q)
{
  return p*q + g + hidden;
}
```

Multi-file Programs

In `prog.h`

```
// Useful declarations
extern int foo(int n, int m);
extern int g;
```

Multi-file Programs

Points:

Multi-file Programs

Points:

- We use quotes in the `#include`. The <> indicates a standard system header, while `""` indicates a specific file name

Multi-file Programs

Points:

- We use quotes in the `#include`. The <> indicates a standard system header, while `""` indicates a specific file name
- `extern` says “here is the type, the actual definition is somewhere else”

Multi-file Programs

Points:

- We use quotes in the `#include`. The <> indicates a standard system header, while `""` indicates a specific file name
- `extern` says "here is the type, the actual definition is somewhere else"
- `g` is actually defined in the first file: it is a matter of taste or code management where you actually define it, but it must be defined somewhere

Multi-file Programs

Points:

- We use quotes in the `#include`. The <> indicates a standard system header, while `""` indicates a specific file name
- `extern` says “here is the type, the actual definition is somewhere else”
- `g` is actually defined in the first file: it is a matter of taste or code management where you actually define it, but it must be defined somewhere

`#include` can be nested, i.e., an included file can include another: it works

Multi-file Programs

Points:

- We use quotes in the `#include`. The <> indicates a standard system header, while `""` indicates a specific file name
- `extern` says “here is the type, the actual definition is somewhere else”
- `g` is actually defined in the first file: it is a matter of taste or code management where you actually define it, but it must be defined somewhere

`#include` can be nested, i.e., an included file can include another: it works

Just be careful not to create a loop…

Multi-file Programs

Exercise. Find out what would happen if we put the `static` declaration into the header file or if we left out the definition of `g`

Exercise. Find out how to collect several `.o` files together in a single library `.a` file; then look up the compiler `-L` option

Exercise. Find out how to make and use *shared library* `.so` files

Exercise. Look at `make` and *Makefiles* as a simple way of managing multi-file programs

Declarations

Exercise. Sometimes functions need to refer to each other

```
int foo(int n)
{
  ... bar(n + 1) ...
}

int bar(int m)
{
  ... foo(m - 1) ...
}
```

The C compiler needs to know the type of `bar` before compiling `foo` and the type of `foo` before compiling `bar`. What declarations would keep the compiler happy?

Preprocessor

We finish with a brief look at C's preprocessing language

Preprocessor

We finish with a brief look at C's preprocessing language

Before the source code is compiled it is sent through the *C preprocessor*, CPP

Preprocessor

We finish with a brief look at C's preprocessing language

Before the source code is compiled it is sent through the *C preprocessor*, CPP

This is a program that *textually* manipulates the code

Preprocessor

We finish with a brief look at C's preprocessing language

Before the source code is compiled it is sent through the *C preprocessor*, CPP

This is a program that *textually* manipulates the code

It does not understand the syntax of C terribly well, it merely pushes parts of the text about

Preprocessor

We finish with a brief look at C's preprocessing language

Before the source code is compiled it is sent through the *C preprocessor*, CPP

This is a program that *textually* manipulates the code

It does not understand the syntax of C terribly well, it merely pushes parts of the text about

The result of this is then passed on to the C compiler proper

Preprocessor

One element of this we have seen many times already:
`#include`

Preprocessor

One element of this we have seen many times already:
`#include`

When CPP sees this it reads the named file and places its contents in the source at the indicated position

Preprocessor

One element of this we have seen many times already:
`#include`

When CPP sees this it reads the named file and places its contents in the source at the indicated position

It doesn't need to be a header file, and the directive doesn't need to be at the start of the C file

Preprocessor

One element of this we have seen many times already:
`#include`

When CPP sees this it reads the named file and places its contents in the source at the indicated position

It doesn't need to be a header file, and the directive doesn't need to be at the start of the C file

But, unless you are doing some devious tricks, this is your most likely use

Preprocessor

So `#include` is mostly used so include standard bits of program (usually declarations) that we don't want to write repeatedly in every program file

Preprocessor

So `#include` is mostly used so include standard bits of program (usually declarations) that we don't want to write repeatedly in every program file

The `#define` directive defines a symbol, or *macro*, in CPP

Preprocessor

So `#include` is mostly used so include standard bits of program (usually declarations) that we don't want to write repeatedly in every program file

The `#define` directive defines a symbol, or *macro*, in CPP

For example

```
#define PI 3.141
```

Preprocessor

So `#include` is mostly used so include standard bits of program (usually declarations) that we don't want to write repeatedly in every program file

The `#define` directive defines a symbol, or *macro*, in CPP

For example

```
#define PI 3.141
```

Thereafter whenever CPP sees `PI` it textually replaces it with 3.141

Preprocessor

So `#include` is mostly used so include standard bits of program (usually declarations) that we don't want to write repeatedly in every program file

The `#define` directive defines a symbol, or *macro*, in CPP

For example

```
#define PI 3.141
```

Thereafter whenever CPP sees `PI` it textually replaces it with 3.141

`x = 2.0*PI;` becomes `x = 2.0*3.141;`

Preprocessor

The replacement text can be anything we like, including other CPP symbols

If

```
#define P2 PI*PI
```

then

```
x = 1.0/P2;
```

becomes

```
x = 1.0/3.141*3.141;
```

Preprocessor

The replacement text can be anything we like, including other CPP symbols

If

```
#define P2 PI*PI
```

then

```
x = 1.0/P2;
```

becomes

```
x = 1.0/3.141*3.141;
```

Notice any problem?

Preprocessor

Because CPP does *textual* processing, it doesn't care about the semantics of C programs

Preprocessor

Because CPP does *textual* processing, it doesn't care about the semantics of C programs

It doesn't know that you probably meant
`x = 1.0/(3.141*3.141);`
and not
`x = (1.0/3.141)*3.141;`
which is C's interpretation of the expression without parentheses

Preprocessor

Because CPP does *textual* processing, it doesn't care about the semantics of C programs

It doesn't know that you probably meant
```
x = 1.0/(3.141*3.141);
```
and not
```
x = (1.0/3.141)*3.141;
```
which is C's interpretation of the expression without parentheses

So if you want the parentheses, you'll have to put them in yourself in the definition

Preprocessor

Because CPP does *textual* processing, it doesn't care about the semantics of C programs

It doesn't know that you probably meant
`x = 1.0/(3.141*3.141);`
and not
`x = (1.0/3.141)*3.141;`
which is C's interpretation of the expression without parentheses

So if you want the parentheses, you'll have to put them in yourself in the definition

```
#define P2 (PI*PI)
```

Preprocessor

Because CPP does *textual* processing, it doesn't care about the semantics of C programs

It doesn't know that you probably meant
`x = 1.0/(3.141*3.141);`
and not
`x = (1.0/3.141)*3.141;`
which is C's interpretation of the expression without parentheses

So if you want the parentheses, you'll have to put them in yourself in the definition

```
#define P2 (PI*PI)
```

We now get `x = 1.0/P2; -> x = 1.0/(3.141*3.141);`

Preprocessor

Macros can take arguments

```
#define mul(a, b) a*b
```

Preprocessor

Macros can take arguments
`#define mul(a, b) a*b`

Now `mul(1+2,3+4) -> 1+2*3+4`

Preprocessor

Macros can take arguments

```
#define mul(a, b) a*b
```

Now `mul(1+2,3+4) -> 1+2*3+4`

So put in the parentheses

```
#define mul(a, b) (a)*(b)
mul(1+2,3+4) -> (1+2)*(3+4)
```

Preprocessor

Macros can take arguments

```
#define mul(a, b) a*b
```

Now `mul(1+2,3+4) -> 1+2*3+4`

So put in the parentheses

```
#define mul(a, b) (a)*(b)
mul(1+2,3+4) -> (1+2)*(3+4)
```

What of `1/mul(1+2,3+4)`?

Preprocessor

Macros can take arguments

```
#define mul(a, b) a*b
```

Now `mul(1+2,3+4) -> 1+2*3+4`

So put in the parentheses

```
#define mul(a, b) (a)*(b)
mul(1+2,3+4) -> (1+2)*(3+4)
```

What of `1/mul(1+2,3+4)`?

This will be `1/(1+2)*(3+4)`

Preprocessor

So, to be safe, put parentheses everywhere

Preprocessor

So, to be safe, put parentheses everywhere

```
#define mul(a, b) ((a)*(b))
```

Preprocessor

So, to be safe, put parentheses everywhere

```
#define mul(a, b) ((a)*(b))
```

Now `1/mul(1+2,3+4) -> 1/((1+2)*(3+4))`

Preprocessor

These are just small examples to show the problems that can arise

Preprocessor

These are just small examples to show the problems that can arise

In real programs macros are useful tools for naming things and making code easier to read

Preprocessor

These are just small examples to show the problems that can arise

In real programs macros are useful tools for naming things and making code easier to read

Or harder to read...

Preprocessor

Many standard headers define useful symbols

Preprocessor

Many standard headers define useful symbols

For example, `math.h` defines several symbols starting `M_` (to avoid name clashes with user-defined symbols)

- `M_PI`
- `M_E`
- `M_PI_2` for $\pi/2$
- `M_SQRT2`
- and more, all often needed in programs

Preprocessor

Many standard headers define useful symbols

For example, `math.h` defines several symbols starting `M_` (to avoid name clashes with user-defined symbols)

- `M_PI`
- `M_E`
- `M_PI_2` for $\pi/2$
- `M_SQRT2`
- and more, all often needed in programs

Exercise. Find out what symbols are defined in `math.h` and other common header files

Preprocessor

The CPP also has *conditional compilation*

```
y = 2.0;
#ifdef FAST
x = y - y*y*y/6;
#else
x = sin(y);
#endif
z = x + x;
```

CPP symbols are conventionally all upper-case

Preprocessor

If the symbol FAST is `#defined` the `x = y - y*y*y/6;` part of the text is kept and the `x = sin(y);` is discarded

Preprocessor

If the symbol FAST is `#define`d the `x = y - y*y*y/6;` part of the text is kept and the `x = sin(y);` is discarded

The code effectively becomes

```
y = 2.0;
x = y - y*y*y/6;
z = x + x;
```

Preprocessor

If `FAST` is not defined, we get

```
y = 2.0;
x = sin(y);
z = x + x;
```

Preprocessor

The symbol `FAST` need not have any particular value, we can even do

```
#define FAST
```

to make it defined with an empty value

Preprocessor

The symbol `FAST` need not have any particular value, we can even do

```
#define FAST
```

to make it defined with an empty value

This kind of thing is very useful when you need minor variants on a basic program, but don't want to have to rewrite the whole program for each variant

Preprocessor

The symbol `FAST` need not have any particular value, we can even do

```
#define FAST
```

to make it defined with an empty value

This kind of thing is very useful when you need minor variants on a basic program, but don't want to have to rewrite the whole program for each variant

`#ifdef...#else...#endif` nests as you might expect

Preprocessor

The symbol `FAST` need not have any particular value, we can even do

```
#define FAST
```

to make it defined with an empty value

This kind of thing is very useful when you need minor variants on a basic program, but don't want to have to rewrite the whole program for each variant

`#ifdef...#else...#endif` nests as you might expect

Another good source of unreadable code when taken too far

Preprocessor

To find out what CPP is doing to your program

`cc -E myprog.c`

will run just the preprocessor and show the result

Preprocessor

To find out what CPP is doing to your program
`cc -E myprog.c`
will run just the preprocessor and show the result

You will be surprised how much there is in the `included` files!

Preprocessor

To find out what CPP is doing to your program
`cc -E myprog.c`
will run just the preprocessor and show the result

You will be surprised how much there is in the `included` files!

One final warning: watch out for strange semantics of macroexpansion. They are well-defined, but probably not what you think

Preprocessor

To find out what CPP is doing to your program
`cc -E myprog.c`
will run just the preprocessor and show the result

You will be surprised how much there is in the `included` files!

One final warning: watch out for strange semantics of macroexpansion. They are well-defined, but probably not what you think

It is only a problem if you start doing tricks with CPP

Preprocessor

To find out what CPP is doing to your program
`cc -E myprog.c`
will run just the preprocessor and show the result

You will be surprised how much there is in the `included` files!

One final warning: watch out for strange semantics of macroexpansion. They are well-defined, but probably not what you think

It is only a problem if you start doing tricks with CPP

See the Obfuscated C Competition

Preprocessor

Exercise. Look up `#if`

Exercise. Look up stringification and token pasting

Exercise. What does

```
#define x y
#define y x
x++;
```

do (or not do)?

Exercise. Browse the Obfuscated C Competition
`http://www.no.ioccc.org/years.html`

NULL

Exercise. Learn about `switch`, `break`, `enum`, `const`, `restrict` and other C keywords

Exercise. Learn about defining and using functions that take a variable number of arguments (e.g., `printf`): varargs

Exercise. Read up on C variants, e.g., C++, Objective C, CUDA, Unified Parallel C, etc.

NULL

Exercise. Write lots of C programs

Exercise. Learn C