Functions

Returns Value: Function
No Return Value: Procedure

A function has a **name**, **a sequence of parameters of particular types**, **a return type** (void if the function doesn't return a value), and a **body**

Every function should have a function comment that describes why the caller would want to call it, in terms of what it would do for them

The heading of a typical function in C looks like:
`returntype functionname( type1 param1, type2 param2... )`

A single value can be returned from a function using the `return` statement

Procedures use the `void` return type

An Example:

```
#include <stdio.h>
int squared( int x ) {
        return x*x;
}

int main( void ) {
        for( int i=0; i<20; i++ ) {
                printf( "%3d %d\n", i, squared(i) );
        }
        return 0;
}
```

If we compile and run this, we will see a table of the first 20 squares

C functions cannot define (or nest) other functions within them

C functions are written sequentially, one after the other

C is strictly defined as a one pass language: this means that every function should have been declared or defined before they are called. If we swap `squared()` and `main()` around and then recompile, we get a warning.

However, there are times when this is inconvenient or even impossible

Parameters are usually passed by value - the function receives copies, so it can't change the passed variables

So if `squared(x)` alters its parameter `x`, that can't change the corresponding actual parameter in the call (`main()`'s `i`)

However, when you pass an array to a function, all that is passed is a basal pointer to the start of the array

This means that the original array can be modified inside a function

You can ignore this subtlety, and write classic fixed size array code:

```
#define ARRSIZE 6
int sumarr( int arr[ARRSIZE] ) {
        int total = 0;
        for( int i = 0; i < ARRSIZE; i++ ) {
                total += arr[i];
        }
        return total;
}
int main( void ) {
        int arr[ARRSIZE] = {10,20,30,40,50,60};
        printf( "total = %d\n", sumarr( arr ) );
        return 0;
}
```

Some might prefer to omit the array size in the parameter declaration:

```
int sumarr( int arr[] ) {
        int total = 0;
        for( int i = 0; i < ARRSIZE; i++ ) {
                total += arr[i];
        }
        return total;
}
```

This is dangerous, as this can now take any sized array of integers - but still only adds up the first ARRSIZE elements.

A better, considerably more generic solution is to pass the number of elements separately:

```
int sumarr( int arr[], int nel ) {
        int total = 0;
        for( int i = 0; i < nel; i++ ) {
                total += arr[i];
        }
        return total;
}
```

The call must also pass the number of elements: `printf( "total = %d\n", sumarr( arr, ARRSIZE ) );`

Best Version: what's passed is a basal pointer, so make that explicit, in which I've also eliminated the ARRSIZE constant and reused our NELEMENTS macro:

```
int sumarr( int *arr, int nel ) {
        int total = 0;
        for( int i = 0; i < nel; i++ ) {
                total += arr[i];
        }
        return total;
}

#define NELEMENTS(arr) (sizeof(arr)/sizeof((arr)[0]))

int main( void ) {
        int arr[] = {10,20,30,40,50,60};
        int nel = NELEMENTS(arr);
        printf( "total = %d\n", sumarr( arr, nel ) );
        return 0;
}
```

Another consequence of arrays being passed as basal pointers: `sizeof()` no longer knows how big the array is `sizeof()` treats the “array” parameter as a basal pointer, whether you use array-like syntax or pointer-like syntax.

Every array parameter is a basal pointer

Because an array parameter is a basal pointer, we can modify the array elements inside a function

Multiplying every element by scale (a third parameter):

```
void scalearr( int *arr, int nel, int scale ) {
        for( int i = 0; i < nel; i++ ) {
                arr[i] *= scale;
        }
}
```

One can also use pointers in functions to simulate **var parameters** or **output parameters**

Suppose you want to write the Haskell:
`(d, m) = divmod( a, b );`

In C:

```
void divmod( int a, int b, int *dp, int *mp ) {
        *dp = a/b;
        *mp = a%b;
}

int main( void ) {
        int d, m;
        divmod( 33, 13, &d, &m );
        printf( "33/13 = div %d, mod %d\n", d, m );
        return 0;
}
```

Another use of this technique is to implement a swap function:

```
void swap( int *a, int *b ) {
        int temp = *b; *b = *a; *a = temp;
}

int main( void ) {
        int x = 42;
        int y = 77;
        printf( "x: %d, y: %d\n", x, y );
        swap( &x, &y );
        printf( "x: %d, y: %d\n", x, y );
        return 0;
}
```

Note that this function is perfectly happy to swap array elements:

```
int arr[] = {10,20,30,40,50,60};
swap( &arr[1], &arr[3] ); // or swap( arr+1, arr+3 );
```

We can declare that a function returns a pointer:

```
char *wibble( int n )
...
```

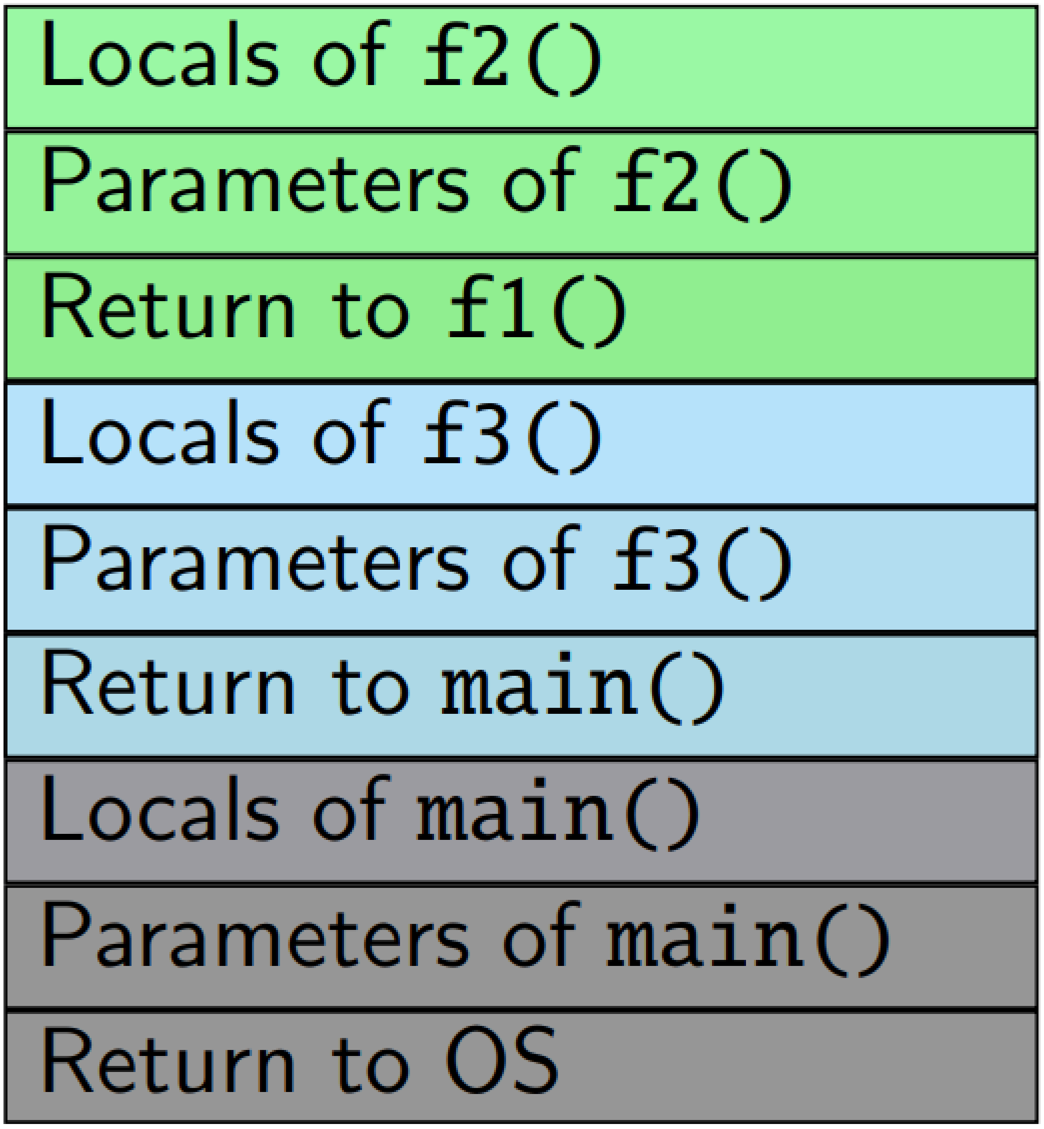
`wibble()` will return a pointer to a char

That could be a Pto1 (a pointer to a single char), or a PtoM (a pointer to the first of many adjacent chars)

The function comment should clarify this

In C, local arrays are allocated on the stack, not the heap. Values allocated on the stack cease to exist as soon as you leave the enclosing scope. Values allocated on the heap exist until they are explicitly deallocated, or the program terminates

Consider the case where we have `main`, which calls function `f1`, which in turn calls function `f2`



We add items to the stack every time a function is called (in a structure called a stack frame)

Then, we remove an entire stack frame every time a function call finishes

You can see how this supports recursion: multiple stack frames (with different parameters, locals, and return address) for each call of the recursive function

The stack can’t grow forever - there will be an upper limit (pretty large, implementation defined). If the stack overflows, our program crashes with a segmentation fault

Because of stack frames, within a function, the parameters and the local variables only exist for the duration of the current call to that function. Then they are destroyed

One consequence of this is that **you mustn’t return the address of a local variable**. For Example:

```
char *oops( char c ) {
        return &c;
}
```

Or:

```
char *oops( void ) {
        char arr[] = "hello";
        return arr;
}
```

Once the current call to `oops()` has returned, the parameters and locals `char c` and `char arr[]` no longer exist

If you return a pointer to any of them, it's a pointer to space your program no longer controls - a prime source of bugs and buffer overruns

Fortunately, `gcc` gives you a warning about this

You can add the keyword `extern` to the front of a **function definition**, omit the function's body, replacing it with a '`;`', and optionally remove the parameter names (leaving just the types) - and then you have a **function declaration**, for example:

```
extern int wibble( double, double, double );
```

A function declaration tells the compiler that the given function exists, with a certain number of parameters, of a particular sequence of types, and has a particular return type

This allows calls to be made to this function safely, with full type checking of it's parameters and return type

Generate a prototype for every function in your C file (except `main()`), place the block of prototypes near the top of your C file. Now the function definitions may be given in any order you like. For Example:

```
#include <stdio.h>
extern int squared( int ); // PROTOTYPES
```

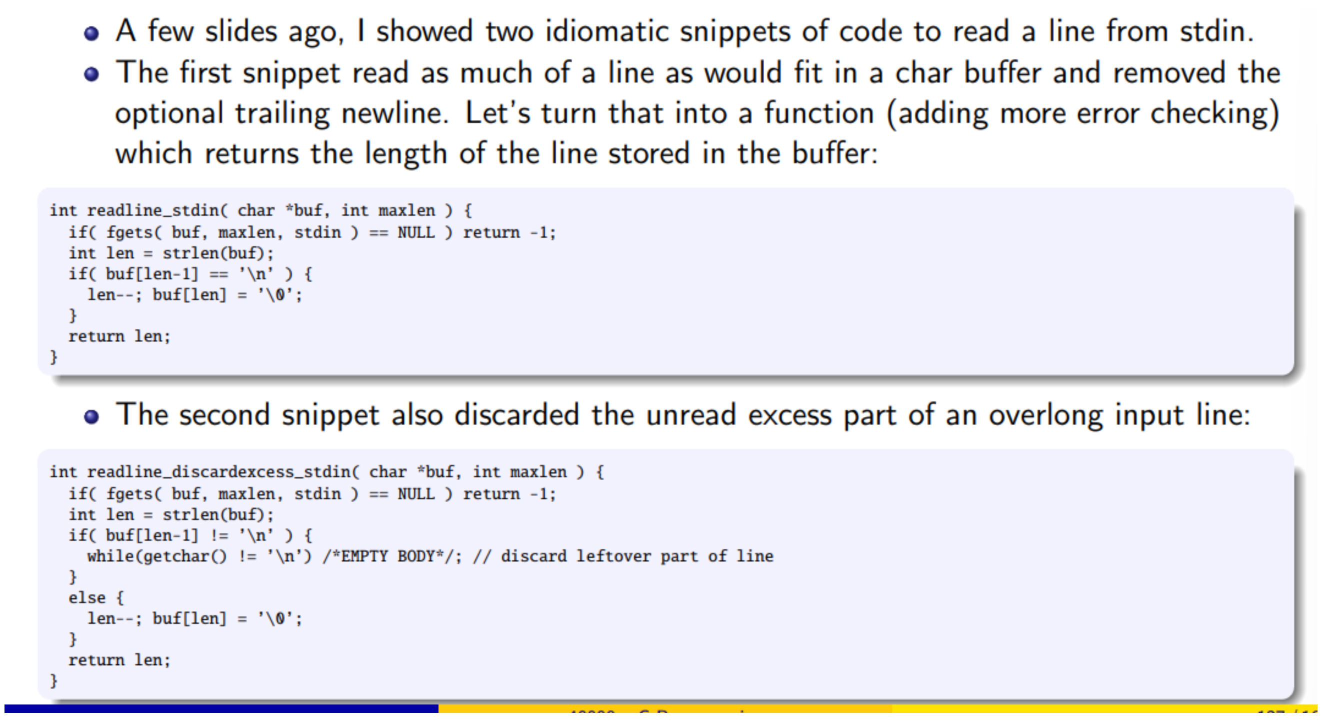
```
// Now the functions can be in any order you like
int main( void ) {
        for( int i=0; i<20; i++ ) {
                printf( "%3d %d\n", i, squared(i) );
        }
        return 0;
}

int squared( int x ) {
        return x*x;
}
```

Keeping the prototypes in sync with the true function definitions is an extra chore for us. My tool proto from C Tools lecture 3 offers a way of automatically generating the prototypes - with some limitations

A function declaration - or definition - with an empty parameter list doesn't declare a function that takes no parameters

Instead, it declares a function whose parameter list is **unknown**, allowing you to pass any parameters into the function

To properly declare such a function, place `void` in the parameter list: `extern int wibble( void );`



- A few slides ago, I showed two idiomatic snippets of code to read a line from stdin.
- The first snippet read as much of a line as would fit in a char buffer and removed the optional trailing newline. Let's turn that into a function (adding more error checking) which returns the length of the line stored in the buffer:

```
int readline_stdin( char *buf, int maxlen ) {
  if( fgets( buf, maxlen, stdin ) == NULL ) return -1;
  int len = strlen(buf);
  if( buf[len-1] == '\n' ) {
    len--; buf[len] = '\0';
  }
  return len;
}
```

- The second snippet also discarded the unread excess part of an overlong input line:

```
int readline_discardexcess_stdin( char *buf, int maxlen ) {
  if( fgets( buf, maxlen, stdin ) == NULL ) return -1;
  int len = strlen(buf);
  if( buf[len-1] != '\n' ) {
    while(getchar() != '\n') /*EMPTY BODY*/; // discard leftover part of line
  }
  else {
    len--; buf[len] = '\0';
  }
  return len;
}
```

There must only be one function definition for a function called `wibble` anywhere in a C codebase (collection of .c files that are linked together). If there's more than one, you will get an error during linking

To alleviate this problem, you can mark helper functions as `static`, making them private to the current C source file

In this case, it will not clash with any identically named public function elsewhere in your code base

To make a static prototype for a static function, replace the keyword `extern` from a normal prototype with the keyword `static`

If there is a prototype for a function, and one or more calls to that function, but if no definition of that function is found anywhere in a C codebase, then you will get an error