

# Programming in C and C++

## Lecture 2: Functions and the Preprocessor

---

David J Greaves and Alan Mycroft  
(Materials by Neel Krishnaswami)

# Functions

- C does not have objects with methods, but does have functions
- A function definition has a return type, parameter specification, and a body or statement; for example:  
`int power(int base, int n) { stmt }`
- A function declaration has a return type and parameter specification followed by a semicolon; for example:  
`int power(int base, int n);`

## Functions, continued

- Functions can be declared or defined extern or static.
- All arguments to a function are copied, i.e. passed-by-value; modification of the local value does not affect the original
- Just as for variables, a function must have exactly one definition and can have multiple declarations
- A function which is used but only has a declaration, and no definition, results in a link error (more on this later)
- Functions cannot be nested

# Function Type Gotchas

- A function declaration with no values (e.g. `int power();`) is not an empty parameter specification, rather it means that its arguments should not be type-checked! (luckily, this is not the case in C++)
- Instead, a function with no arguments is declared using void (e.g., `int power(void);`)
- An ellipsis ( `...` ) can be used for optional (or varying) parameter specification, for example:  
`int printf(char* fmt,...) { stmt }`
- The ellipsis is useful for defining functions with variable length arguments, but leaves a hole in the type system ( `stdarg.h` )

# Recursion

- Functions can call themselves recursively
- On each call, a new set of local variables is created
- Therefore, a function recursion of depth  $n$  has  $n$  sets of variables
- Recursion can be useful when dealing with recursively defined data structures, like trees (more on such data structures later)
- Recursion can also be used as you would in ML:

```
1      unsigned int fact(unsigned int n) {  
2          return n ? n * fact(n-1) : 1;  
3      }
```

# Compilation

- A compiler transforms a C source file or execution unit into an object file
- An object file consists of machine code, and a list of:
  - defined or exported symbols representing defined function names and global variables
  - undefined or imported symbols for functions and global variables which are declared but not defined
- A linker combines several object files into an executable by:
  - combining all object code into a single file
  - adjusting the absolute addresses from each object file
  - resolving all undefined symbols

The Part 1b Compiler Course describes how to build a compiler and linker in more detail

# Handling Code in Multiple Files in C

- C separates declaration from definition for both variables and functions
- This allows portions of code to be split across multiple files
- Code in different files can then be compiled at different times
  - This allows libraries to be compiled once, but used many times
  - It also allows companies to sell binary-only libraries
- In order to use code written in another file we still need a declaration
- A header file can be used to:
  - supply the declarations of function and variable definitions in another file
  - provide preprocessor macros (more on this later)
  - avoid duplication (and `./.` errors) that would otherwise occur
- You might find the Unix tool `nm` useful for inspecting symbol tables

# Multiple Source File Example

## example4.h

```
/* reverse s in place */  
void reverse(char str[]);
```

## example4a.c

```
#include <string.h>  
#include "example4.h"  
  
void reverse(char s[]) {  
    for (int i=0, j=strlen(s)-1;  
         i < j; i++, j--) {  
        char c=s[i];  
        s[i]=s[j], s[j]=c;  
    }  
}
```

## example4b.c

```
#include <stdio.h>  
#include "example4.h"  
  
int main(void) {  
    char s[] = "Reverse me";  
    reverse(s);  
    printf("%s\n", s);  
    return 0;  
}
```



# Variable and Function Scope with static

- The `static` keyword limits the scope of a variable or function
- In the global scope, `static` does not export the function or variable symbol
  - This prevents the variable or function from being called externally
  - BEWARE: `extern` is the default, not `static` This is also the case for global variables.
- In the local scope, a `static` variable retains its value between function calls
  - A single `static` variable exists even if a function call is recursive
  - Note: `auto` is the default, not `static`

# Address Space Layout

A typical x86 32-bit address-space layout:

Description	Address
Top of address space	0xffff ffff
...	
Stack (downwards-growing)	typical start 0x7fff ffff
...	
Heap (upwards-growing)	typical start 0x0020 0000
...	
Static variables	typical start 0x0010 0000
C binary code	typical start 0x0000 8000
...	
Null – often trapped	0x000 0000

(64 bit is messier, but not fundamentally different: see layout.c)

# C Preprocessor

- The preprocessor executes before any compilation takes place
- It manipulates the text of the source file in a single pass
- Amongst other things, the preprocessor:
  - deletes each occurrence of a backslash followed by a newline;
  - replaces comments by a single space;
  - replaces definitions, obeys conditional preprocessing directives and expands macros; and
  - it replaces escaped sequences in character constants and string literals and concatenates adjacent string literals

# Controlling the Preprocessor Programmatically

- The preprocessor can be used by the programmer to rewrite source code
- This is a powerful (and, at times, useful) feature, but can be hard to debug (more on this later)
- The preprocessor interprets lines starting with `#` with a special meaning
- Two text substitution directives: `#include` and `#define`
- Conditional directives: `#if` , `#elif` , `#else` and `#endif`

# The #include Directive

- The #include directive performs text substitution
- It is written in one of two forms:  
    #include "filename"  
    #include <filename>
- Both forms replace the #include ... line in the source file with the contents of filename
- The quote ( " ) form searches for the file in the same location as the source file, then searches a predefined set of directories
- The angle ( < ) form searches a predefined set of directories
- When a #include-d file is changed, all source files which depend on it should be recompiled (easily managed via a 'Makefile')

# The #define Directive

- The #define directive has the form:  
`#define name replacement-text`
- The directive performs a direct text substitution of all future examples of *name* with the *replacement-text* for the remainder of the source file
- The *name* has the same constraints as a standard C variable name
- Replacement does not take place if *name* is found inside a quoted string
- By convention, *name* tends to be written in upper case to distinguish it from a normal variable name

# Defining Macros

- The `#define` directive can be used to define macros; e.g.:  
`#define MAX(A,B) ((A)>(B)?(A):(B))`
- In the body of the macro:
  - prefixing a parameter in the replacement text with `'#'` places the parameter value inside string quotes (`"`)
  - placing `'##'` between two parameters in the replacement text removes any whitespace between the variables in generated output
- Remember: the preprocessor only performs text substitution!
  - Syntax analysis and type checking don't occur until compilation
  - This can result in confusing compiler warnings on line numbers where the macro is used, rather than when it is defined; e.g.  
`#define JOIN(A,B) (A B)`
  - Beware:  
`#define TWO 1+1`  
`#define WHAT TWO*TWO`

## Example

```
1  #include <stdio.h>
2
3  #define PI 3.141592654
4  #define MAX(A,B) ((A)>(B)?(A):(B))
5  #define PERCENT(D) (100*D)           /* Wrong? */
6  #define DPRINT(D) printf(#D " = %g\n",D)
7  #define JOIN(A,B) (A ## B)
8
9  int main(void) {
10     const unsigned int a1=3;
11     const unsigned int i = JOIN(a,1);
12     printf("%u %g\n",i, MAX(PI,3.14));
13     DPRINT(MAX(PERCENT(0.32+0.16),PERCENT(0.15+0.48)));
14
15     return 0;
16 }
```



# Conditional Preprocessor Directives

Conditional directives: `#if` , `#ifdef` , `#ifndef` , `#elif` and `#endif`

- The preprocessor can use conditional statements to include or exclude code in later phases of compilation
- `#if` accepts an integer expression as an argument and retains the code between `#if` and `#endif` (or `#elif` ) if it evaluates to a non-zero value; for example:

```
#if SOME_DEF > 8 && OTHER_DEF != THIRD_DEF
```

- The preprocessor built-in `defined` takes a name as its argument and gives `1` if it is `#define`-d; `0` otherwise
- `#ifdef N` and `#ifndef N` are equivalent to `#if defined(N)` and `#if !defined(N)` respectively
- `#undef` can be used to remove a `#define`-d name from the preprocessor macro and variable namespace.

# Preprocessor Example

Conditional directives have several uses, including preventing double definitions in header files and enabling code to function on several different architectures; for example:

```
1  #if SYSTEM_SYSV
2  #define HDR "sysv.h"
3  #elif SYSTEM_BSD
4  #define HDR "bsd.h"
5  #else
6  #define HDR "default.h"
7  #endif
8  #include HDR
```

```
1  #ifndef MYHEADER_H
2  #define MYHEADER_H 1
3  ...
4  /* declarations & defns */
5  ...
6  #endif /* !MYHEADER_H */
```

## Error control

- To help other compilers which generate C code (rather than machine code) as output, compiler line and filename warnings can be overridden with:

*#line constant "filename"*

- The compiler then adjusts its internal value for the next line in the source file as *constant* and the current name of the file being processed as "filename" ("filename" may be omitted)
- The statement *#error some-text* causes the preprocessor to write a diagnostic message containing *some-text*
- There are several predefined identifiers that produce special information: `__LINE__` , `__FILE__` , `__DATE__` , and `__TIME__`