C and C++

2. Functions — Preprocessor

Alastair R. Beresford

University of Cambridge

Lent Term 2007

Control flow and string example

```
#include <stdio.h>
#include <string.h>

char s[]="University of Cambridge Computer Laboratory";

int main(void) {

  char c;
  int i, j;
  for (i=0,j=strlen(s)-1;i<j;i++,j--) /* strlen(s)-1 ? */
    c=s[i], s[i]=s[j], s[j]=c;

  printf("%s\n",s);
  return 0;
}
```

Functions

- C does not have objects, but does have function support
- 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);`
  - The use of the `extern` keyword for function declarations is optional
- All arguments to a function are copied or *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
- Functions cannot be nested

Function type-system nasties

- A function definition with no values (e.g. `power()`) is not an empty parameter specification, rather it means that its arguments should not be type-checked! (this is not the case in C++)
- Instead, a function with no arguments is declared using `void`
- An ellipsis (`...`) can be used for partial 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`)
- In comparison, C++ uses operator overloading to provide better I/O type safety (more on this later)

Recursion

- Functions can call themselves recursively
- On each call, a new set of local variables are 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:

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

Compilation

- The compiler transforms each 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, together with
  - *undefined* or *imported* symbols
- For each object file, the compiler generates:
  - defined symbols for defined function names and global variables
  - undefined 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

Multi-source file example

Header File — `example4.h`

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

Source File — `example4a.c`

```
#include <string.h>
#include "example4.h"

/*reverse a string in place */
void reverse(char s[]) {
int c, i, j;
for (i=0,j=strlen(s)-1;i<j;i++,j--)
c=s[i], s[i]=s[j], s[j]=c;
}
```

Source File — `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
- 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

C Preprocessor

- The preprocessor is executed before any compilation takes place
- It manipulates the textual content 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
- 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 #included file is changed, all source files which depend on it should be recompiled

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* as well; for example: `#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
- This means that syntax analysis and type checking doesn't occur until the compilation stage
- This can, initially at least, generate some confusing compiler warnings on line numbers where the macro is used, rather than when it is defined; for example:
  `#define JOIN(A,B) (A ## B))`

Example

```
#include <stdio.h>

#define PI 3.141592654
#define MAX(A,B) ((A)>(B)?(A):(B))
#define PERCENT(D) (100*D)              /* Wrong? */
#define DPRINT(D) printf(#D " = %g\n",D)
#define JOIN(A,B) (A ## B)

int main(void) {
  const unsigned int a1=3;
  const unsigned int i = JOIN(a,1);
  printf("%u %g\n",i, MAX(PI,3.14));
  DPRINT(MAX(PERCENT(0.32+0.16),PERCENT(0.15+0.48)));

  return 0;
}
```

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 a (somewhat limited) integer *expression* as an argument and only retains the code between `#if` and `#endif` (or `#elif`) if the expression evaluates to a non-zero value; for example:
  `#if SOME_DEF > 8 && OTHER_DEF != THIRD_DEF`
- The built-in preprocessor function `defined` accepts a name as it's sole argument and returns `1L` if the name has been `#define`d; `0L` 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.

Example

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

```
#if SYSTEM_SYSV
#define HDR "sysv.h"
#elif SYSTEM_BSD
#define HDR "bsd.h"
#else
#define HDR "default.h"
#endif
#include HDR
```

```
#ifndef MYHEADER_H
#define MYHEADER_H 1
...
declarations & definitions
...
#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__`.

Exercises

1. Write a function definition which matches the declaration `int cntlower(char str[]);`. The implementation should return the number of lower-case letters in a string
2. Use function recursion to write an implementation of merge sort for a fixed array of integers; how much memory does your program use for a list of length *n*?
3. Define a macro `SWAP(t,x,y)` that exchanges two arguments of type `t` (K&R, Exercise 4-14)
4. Define a macro `SWAP(x,y)` that exchanges two arguments of the same type (`int` or `char`) *without using a temporary*