CS2002 Computer Systems Lecture 2

Functions and Variables

Jon Lewis (JC 0.26)
School of Computer Science
University of St Andrews

Overview

- Control
- Functions
  - introduction
  - function prototypes
  - header files
  - recursion
- Type conversion
  - Implicit and explicit casting
- Visibility and Scope
  - declarations and definitions
  - static and extern

Control Statements

```
if ( x == 0 ) printf("x == 0\n");
else printf("x != 0\n");

for(int i = 0; i < 10; i++) { ... }

while(x > 0) { ... }

do { ... } while( bob );
```

Functions

- Functions in C are like static methods in Java.
- They have a return type (that may be void) and take a number of arguments.
- Functions can be defined only once in a whole program.

```
int add(int a, int b) {
    return a + b;
}

int equal(char a, char b){
    if(a == b) return 1;
    else return 0;
}
```

Function Prototypes

- A function should be declared before it is used.
- A function prototype lets you declare a function, without its body.

```
int add(int a, int b);
int equal(int a, int b);
float addf(float a, float b);
```

- declare' is independent of 'define'
- A function should be declared in every file it is used in.
- If you don't declare it, the compiler will guess return type int and parameter types based on how you call it (which could be wrong), so always declare first
- There is an efficient way to do this: **header files**

Function Prototypes

```
float addf(float a, float b);


int main() {
  float i, j;

  printf("Type in two floats: ");
  scanf("%f%f", &i, &j);
  printf("%f + %f = %f\n", i, j, addf(i,j));
}


float addf(float a, float b) {
  return a + b;
}
```

Header Files

- Header files can contain function prototypes.

```
// fmax.h
float fadd( float a, float b );

// fmax.c
#include "fmax.h"
float fadd(float a, float b) { return a + b; }

// test_fmax.c
#include <stdio.h>
#include "fmax.h"
int main() {
  float f = 1.0, g = 2.0;
  printf( "%f + %f = %f\n", f, g, fadd(f,g) );
}
```

Including fmax.h in fmax.c is optional, but useful to check your prototypes.

Function Examples

```
char to_upper(char c) {
  if( c >= 'a' && c <= 'z') {
    return c - 'a' + 'A';
  }
  else return c;
}
```

```
#include<math.h>

double myTan(double angle) {
  return sin(angle)/cos(angle);
}
```

A Recursive Function

```
/* Implements factorial */
// 0! = 1, n! = n * (n-1)!


// Prototype
int factorial(int n);


int factorial(int n) {
  if(n == 0) return 1;

  return n * factorial(n-1);
}
```

typedef

- Use `typedef` to give new names to types:

```
typedef oldname newname1, newname2;

typedef unsigned int day, month;
typedef char byte;
```

Casting

- Type casting converts a value from one type to another. Casting may be *implicit* or *explicit*.

- `float f1 = 1.6, f2 = 1.6;`

- `int i = f1;` `// Implicit. i = 1;`
- `int j = f1 + f2;` `// Implicit. j = 3;`
- `int k = (int)(f1 + f2);` `// Explicit. k = 3;`
- `int m = (int)f1 + (int)f2;` `// Explicit. m = 2;`

Declaration - Definition

- A *declaration* specifies the type of an identifier (variable or function), without *defining* it.
  - keyword ***extern*** (see later) indicates declaration
    - extern is not needed for function declarations

- You can *declare* an identifier many times, but only *define* it once.
- A function / variable can be *declared* in many files, but can be only be *defined* once if globally visible

Defining Variables

- We only have to worry about global variables, as variables in functions are always local to their function.
- The following two files will not 'link' together, as `i` is defined twice.

```
// file1.c

int i = 1;
void f();
int main(void) {
  f();
  printf("%d\n", i);
}
```

```
// file2.c

int i = 1;

void f() { i++; }
```

Defining Variables (2)

- There are two different ways of defining/declaring global variables so they work in multiple files, `static` and `extern`.

- `static`: this variable (or function) is unique ("private") to this file and not globally visible.
  - That means each file gets its own.

- `extern`: The variable (or function) declared for use here is defined in some other file, but I will use it here.
  - The variable / function has to be defined in some file or linker error will occur

Defining Variables (3)

- The copies of `i` are distinct.
- Outputs 1

```
// file1.c

static int i = 1;
void f();
int main(void) {
  f();
  printf("%d\n", i);
}
```

```
// file2.c

static int i = 1;

void f() { i++; }
```

Defining Variables (3)

- Will not compile and link, as `i` is not *defined* anywhere.
  - Both files will compile separately, but you will get linker error

```
// file1.c

extern int i;
void f();
int main(void) {
  f();
  printf("%d\n", i);
}
```

```
// file2.c

extern int i;

void f() { i++; }
```

Defining Variables (3)

- Compiles and runs!
- Outputs 2

Extern decl. would normally be in a header file, e.g. "file2.h"

```
// file1.c

extern int i;
void f();
int main(void) {
  f();
  printf("%d\n", i);
}
```

```
// file2.c

int i = 1;

void f() { i++; }
```

Variable & Function Decls/Defs

Defined	void f() { ... }	int i; int i = 1;
static defined (just in this file)	static void f() { ... }	static int i; static int i = 1;

Declared here but defined in another file (or elsewhere in this file)	void f();
	extern void f();
	(Identical)
		extern int i;
		~~extern int i = 1;~~

static variables in functions.

- A "static" variable in a function effectively makes the variable behave like a global – the value is kept over different calls – but it is “private” to the function

```
#include <stdio.h>
int i = 0; // This does not interact with i in get_number

int get_number() {
    static int i = 0;
    return i++;
}

int main(void) {
    printf("%d\n", get_number());
    printf("%d, %d\n", get_number(), i);
}
```

Separate Compilation

```
clang -c prog.c          // Compiles prog.c to
                         // object prog.o

clang prog.o -o prog     // Links prog.o into
                         // executable prog

clang -c prog.c lib.c    // Compiles source into
                         // prog.o and lib.o

clang prog.o lib.o -o myfile // Links prog.o and
                             // lib.o into myfile
```

Makefiles

- Makefile contain rules for making programs.
- Only new parts are recompiled!

- A Makefile rule has three parts:
  - The **target** name (often a file being built)
  - The **requirements** (a list of dependancies)
  - The **commands** to execute to build the target.

Makefiles

```
prog : source.c source.h
	clang -o prog source.c
```

- Building prog requires source.c and source.h. The command to execute is ‘clang -o prog source.c’.
- Make will run this rule if either source.c or source.h are newer than prog

Makefiles (2)

- You write rules which do things based on a name.
- This rule will build ‘stage1’ and ‘stage2’ when we ask for ‘all’.

```
all: stage1 stage2
```

- A rule to remove binaries:

```
clean:
    rm *.o stage1 stage2
```

Makefile Catch

- One annoying feature of Make: When giving the commands to execute a rule (clang here) you have to use the ‘tab’ key, **not spaces**!

```
prog : file.c
	clang -o exec source.c
```

Using make

- By default, make uses the file `Makefile` in the current directory.

```
> make                    # Makefile's default (first) target

> make test               # make 'test'

> make prog.o test2       # make prog.o and test2

> make -f Makefile.in # Use Makefile.in instead of default
```

MAKEFILE EXAMPLE ON STUDRES