



NWEN241 Writing larger programs

Qiang Fu

School of Engineering and Computer Science
Victoria University of Wellington



This Lecture

- Storage Classes
- Header Files
- Type Definition
- C Preprocessor
- Libraries
- Makefile
- Making Multiple Files

27/05/2015

2

Storage Classes

- Automatic variables
 - Created when declared within a block / function body and no storage specification is mentioned (`int i; float f;`)
 - Local to the block / function
 - Storage is allocated when declared and deallocated upon exit from the block / function

```
int main(void)
{ int i = 0, x = 0;
  ...
} /* a and b not available in this block */

int f(void)/*i and x not available in this block */
{ int a = 0, b = 0;
  ... /* a and b will be destroyed */
} /* after the function is executed */
```

27/05/2015

3

Storage Classes

- Register variables
 - A register is a small amount of storage in CPU
 - Contents in registers can be accessed more quickly than storage available elsewhere
 - We can request to store a variable in a register, but there is no guarantee we will get it

```
register int i; /* if the request is failed, we */
               /* will get an automatic variable instead */
```
- The movement of data out of and in registers is transparent to programmers
- We cannot manipulate the address of registers and of course the address of register variables (cannot do `&i`)
- Other than all the stuff above, a register variable is like an automatic variable

27/05/2015

4

Storage Classes

- External variables
 - Created when declared outside all blocks / functions
 - External to all blocks / functions
 - Storage and the value in the storage are retained upon exit from the block / function
 - A way to transmit information across blocks and functions

```
int i = 3, x =2;          /* external variables */
int n = f(), m = i*x;     /* can we do this??? */
int main(void)
{ ...
}

int f(void)               /* i and x are available */
{ ...                     /* in this block */
}
```

27/05/2015

5

Storage Classes

- External variables
 - Created when declared outside all blocks / functions
 - External to all blocks / functions
 - Storage and the value in the storage are retained upon exit from the block / function
 - A way to transmit information across blocks and functions

```
int i = 3, x =2;          /* external variables */
int n = f(), m = i*x;     /* no func call or expression*/
int main(void)
{ ...
}

int f(void)               /* i and x are available */
{ ...                     /* in this block */
}
```

27/05/2015

6

Storage Classes

- External variables (multiple files)

In file main.c

```
int i = 0, x =0;  /* external variables */
int a[10];
int main(void)
{ ...
}
```

In file f.c

```
extern int i, x, a[]; /* look for i, x, a[] elsewhere */
                        /* can we do extern int i=0, x=0; ?*/

int f(void)
{ ...                  /* i and x are available */
}                      /* in this block */
```

27/05/2015

7

Storage Classes

- External variables (multiple files)

In file main.c

```
int i = 0, x =0;  /* external variables */
int a[10];
int main(void)
{ ...
}
```

In file f.c

```
extern int i, x, a[]; /* look for i, x, a[] elsewhere */
                        /* no - redefine global variables */

int f(void)
{ ...                  /* i and x are available */
}                      /* in this block */
```

27/05/2015

8

Storage Classes

- Static variables
 - Created when declared with the keyword **static**
 - If declared outside all the blocks / functions, the variable is global **within this file**
 - If declared within a block / function, the variable is only available within the block / function. But upon exit from the block / function, its value and storage are retained.

```
static int i, x; /* static external variable */
                /* what does this mean? */

int main(void)
{ ...
}

int f(void) /* i and x are available in this block */
{ static int a, b;
  ...      /* a, b only available in this block */
}          /* upon exit a and b's value is retained */
```

27/05/2015

9

Storage Classes

- Static variables
 - Created when declared with the keyword **static**
 - If declared outside all the blocks / functions, the variable is global **within this file**
 - If declared within a block / function, the variable is only available within the block / function. But upon exit from the block / function, its value and storage are retained.

```
static int i, x; /* static external variable */
                /* only available in this file */

int main(void)
{ ...
}

int f(void) /* i and x are available in this block */
{ static int a, b;
  ...      /* a, b only available in this block */
}          /* upon exit a and b's value is retained */
```

27/05/2015

10

Storage Classes

- Static variables
 - Created when declared with the keyword **static**
 - If declared outside all the blocks / functions, the variable is global **within this file**
 - If declared within a block / function, the variable is only available within the block / function. But upon exit from the block / function, its value and storage are retained.

```
static int i, x; /* static external variable */
                /* only available in this file */

int main(void)
{ ...
}

int f(void)
{ static int i, x; /* redeclaration: is this legal? */
  ...
}
```

27/05/2015

11

Storage Classes

- Static variables
 - Created when declared with the keyword **static**
 - If declared outside all the blocks / functions, the variable is global **within this file**
 - If declared within a block / function, the variable is only available within the block / function. But upon exit from the block / function, its value and storage are retained.

```
static int i, x; /* static external variable */
                /* only available in this file */

int main(void)
{ ...
}

int f(void)
{ static int i, x; /* legal in different scopes */
  ...          /* the 1st set of i/x has no effect here */
}
```

27/05/2015

12

Storage Classes

- Functions
 - by default, all functions are external and thus global
 - We can make them only available within their own files by using keyword ...???

In file f.c

```
int g(void);
```

```
int a(void)
```

```
{ ...      /* g is available here and in f.c, */  
}          /* and in other files */
```

```
...
```

```
int g(void)
```

```
{ ...  
}
```

27/05/2015

13

Storage Classes

- Functions
 - by default, all functions are external and thus global
 - We can make them only available within their own files by using keyword ...???

In file f.c

```
static int g(void);
```

```
int a(void)
```

```
{ ...      /* g is available here and in f.c, */  
}          /* but not in other files */
```

```
...
```

```
static int g(void)
```

```
{ ...  
}
```

27/05/2015

14

Storage Classes

- Functions in multiple files

In file main.c

```
int a(void);
```

```
int b(void);
```

```
int c(void);
```

```
int main(void)
```

```
{ ...  
}
```

In file a.c

```
int c(void);
```

```
int a(void)
```

```
{ ...  
}
```

In file b.c

```
int a(void);
```

```
int b(void)
```

```
{ ...  
}
```

In file c.c

```
int b(void);
```

```
int c(void)
```

```
{ ...  
}
```

27/05/2015

15

Storage Classes

- Functions as arguments

– What is the point?

```
int l_minus_s(int l(int, int), int s(int,  
int), int, int);
```

```
int l_minus_s(int (*l)(int, int), int  
(*s)(int, int), int, int);
```

```
int l_minus_s(int (*)(int, int), int  
*)(int, int), int, int);
```

27/05/2015

16

Header Files

- Header files usually contain
 - definitions of data types
 - function prototypes
 - Declaration of global variables (not definition)
 - C preprocessor commands

27/05/2015

17

Header Files

- Functions in multiple files

In file main.c
#include "abc.h"

```
int main(void)
{ ...
}
```

In file a.c
#include "abc.h"

```
int a(void)
{ ...
}
```

In file b.c
#include "abc.h"

```
int b(void)
{ ...
}
```

In file c.c
#include "abc.h"

```
int c(void)
{ ...
}
```

27/05/2015

18

Type Definitions

```
typedef int (*ptrf)(void);
ptrf pf;    /* what is pf??? */
/* is it equivalent to int (*pf)(void); ? */
- A pointer to a function that returns a pointer to a function
  that returns a pointer to a char
/* char *(*(*)())() */
typedef char *pc;
typedef pc fpc();
typedef fpc *pfpc;
typedef pfpc fpcfpc();
typedef fpcfpc *pfpcfpc;

pfpcfpc a;
```

27/05/2015

19

C Preprocessor

- File inclusion (#include <filename>)

```
#include <stdio.h>
/* replace the line with */
/* the source code in stdio.h */
/* search stdio.h in /usr/include */

#include "afile.h"
/* search afile.h in current directory */
/* first and then try /usr/include */
```

27/05/2015

20

C Preprocessor

- Macros (#define)

- Symbolic constants

```
#define PI 3.14 /* what is the type here */
```

```
#define Node_Size sizeof(Node)
```

- Macros with arguments

```
#define SQ(x) ((x)*(x))
```

```
/* what is the type here */
```

```
/* like a inline function */
```

```
SQ(++r); /* problematic use of SQ(x) */
```

```
SQ(f()); /* problematic use of SQ(x) */
```

```
#define larger(x,y) (((x)>(y)))?(x):(y)
```

27/05/2015

21

C Preprocessor

- Conditional compilation

```
#define DEBUG 1
```

```
#if DEBUG
```

```
... /* code in between will be compiled */
```

```
#endif
```

```
-----
```

```
#define DEBUG
```

```
#ifdef DEBUG
```

```
... /* code in between will be compiled */
```

```
#endif
```

```
-----
```

```
#ifndef ... #endif
```

```
#if ... #elif ... #else ... #endif
```

```
#undef PI
```

```
#define PI 41.3
```

27/05/2015

22

C Preprocessor

- #define vs. typedef

```
#define mptrchar char *
```

```
typedef char *tptrchar;
```

```
mptrchar a, b;
```

```
tptrchar c, d;
```

```
const mptrchar aa, bb;
```

```
const tptrchar cc, dd;
```

```
/* can you typedef a pointer to const? */
```

27/05/2015

23

Libraries

- There are many libraries including the standard C library

```
%gcc usemath.c -lm
```

- If you want to create and use your own libraries

```
%ar ruv mylib.a myfunc1.o myfunc2.o ...
```

```
%ranlib mylib.a /* index of contents */
```

```
%gcc usemylib.c mylib.a
```

```
%gcc usemylib.c -L/home/myname/mylibs mylib.a
```

```
%gcc usemylib.c /home/myname/mylibs/mylib.a
```

- If you have your header files (e.g., mylib.h) in a separate directory

```
%gcc usemylib.c -L/... -I/home/myname/myheaderfiles
```

27/05/2015

24

Makefile

- Compile a program with multiple files

```
%gcc main.c file1.c file2.c file3.c ...
```

```
-----
```

```
%gcc -c main.c file1.c file2.c file3.c ...
```

```
%gcc main.o file1.o file2.o file3.o ...
```

```
-----
```

```
%gcc -c file1.c /*file1.c was modified */
```

```
%gcc main.o file1.o file2.o file3.o ...
```

```
%gcc main.o file1.c file2.o file3.o ...
```

```
%gcc -c file3.c /*file3.c was modified */
```

```
%gcc main.o file1.o file2.o file3.o ...
```

- Is this a good way to construct/maintain programs?

27/05/2015

25

Makefile

- The *make* utility (makefile)

```
myprogram: main.o file1.o file2.o file3.o ...
```

```
gcc -o myprogram main.o file1.o file2.o file3.o ...
```

```
main.o: main.c myheaderfile.h #dependency rule
```

```
gcc -c main.c #action rule with tap
```

```
file1.o: myheaderfile.h #ok, without .c file included
```

```
gcc -c file1.c
```

```
file2.o: myheaderfile.h #: left depends on right
```

```
gcc -c file2.c
```

```
file3.o: myheaderfile.h #if m...e.h is modified, ...
```

```
gcc -c file3.c
```

```
...
```

```
%make
```

```
%./myprogram
```

27/05/2015

26

Makefile

- The *make* utility (using macros)

```
CC = gcc
```

```
CFLAGS =
```

```
PROGRAM = myprogram
```

```
OBJECTS = main.o file1.o file2.o file3.o ...
```

```
MYLIBS = $(DIRT)/mylibs/mylib.a
```

```
MYHEADERFILE = myheaderfile.h
```

```
MYHEADERS = -I$(DIRT)/include
```

```
DIRT = /home/myname
```

```
$(PROGRAM): $(OBJECTS)
```

```
$(CC) $(CFLAGS) -o $(PROGRAM) $(OBJECTS) $(MYLIBS)
```

```
$(OBJECTS): $(MYHEADERFILE)
```

```
$(CC) $(CFLAGS) -c *.c $(MYHEADERS)
```

27/05/2015

27

Making Multiple files

- Modularise a big problem into multiple smaller problems
- Each of the small problems can turn into a function
- The functions from a related group, or the functions sharing the same major objects such as data structures, can be put in the same file
- If you make changes to an object and the changes only require further changes within the same file, it would be a good design
- The functions used to implement a major function can be put in the same file. The major function itself can occupy a separate file.
- One file contains main
- Several files may share the same header file

27/05/2015

28