CIS*2520, LAB1

by Tao Xu (xut@uoguelph.ca)

A Quick Review of C Programming

Standards

- ANSI C 1989
- ISO 1990 (C89)
- C99
- C++
  - An extension of C for Object Oriented Programming

Algorithm = Logic + Control + Data

- Data structures and algorithms
  - Data structures = Ways of systematically arranging information, both abstractly and concretely
  - Algorithms = Methods for constructing, searching, and operating on data structures
- Characterizing Costs (as a function of input size)
  - Space
  - Time
- Applications—What's a good data structure/algorithm for a particular problem?

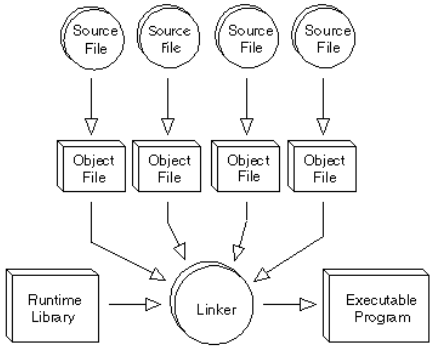
Program Structure

- A main entrance function
  - int main(void){…};
  - int main(int argc, char *argv[])
- Additional local functions
- External functions

C Programming Environment

- C source files and headers
- Provided supporting libraries, headers (stdio.h, stdlib.h)
- Compiler system
  - Compiler converts source code to platform-specific
  - Linker merges objects and libraries into executable modules
  - Such as gcc, integrated development environment

Compiler



Exercise. 1

- *Create* example file: `lab1.c`
- *Compile* using gcc:
  `gcc -o lab1 lab1.c`
- The standard C library *libc* is included automatically
- *Execute* program
  `./lab1`
- Note, I always specify an absolute path
- Normal termination:
  `void exit(int status);`
  - calls functions registered with `atexit()`
  - flush output streams
  - close all open streams
  - return status value and control to host environment

```
/* you generally want to
 * include stdio.h and
 * stdlib.h
 * */

//preprocessor
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   printf("Hello World\n");
   return 0;
}
```


Storage class and Scope:

- int v = 0; /* declared at some place in your program */
- What can you see from the declaration?

storage class	:	auto, register, static, extern
type	:	value domain
value	:	current value ∈ value domain
name	:	symbolic identifier
location	:	memory address
size	:	how many bytes it occupies
scope	:	where it can be accessed

Basic Types and Operators

- Basic data types
  - Types: *char, int, float and double*
  - Qualifiers: *short, long, unsigned, signed, const*
- Constant: 0x1234, 12, “Some string”
- Enumeration:
  - Names in different enumerations must be distinct
  - ```
    enum WeekDay_t {Mon, Tue, Wed, Thur, Fri};
    enum WeekendDay_t {Sat = 0, Sun = 4};
    ```
- Arithmetic: +, -, *, /, %
  - prefix ++i or --i ; increment/decrement before value is used
  - postfix i++, i--; increment/decrement after value is used
- Relational and logical: <, >, <=, >=, ==, !=, &&, ||
- Bitwise: &, |, ^ (xor), <<, >>, ~(ones complement)

Scope

- **Definition**
  - Region over which you can access a variable by name.
- There are 4 types of scope:
  - Program scope… widest
  - File scope
  - Function scope
  - Block scope … narrowest
- Always choose the narrowest scope that works.
  - Why?
    ```

Automatic Variables

- Lifetime: from the time when the program enters the block till it leaves the block.
- Scope: in the block where they are declared (function, block)

```
void f()
{
   int x, y; …
}
```

```
void f()
{
   …
   {
     int x, y; …
   }
   …
}
```


External Variables

- Lifetime: the entire program cycle
- Scope: source files in which they are declared (program)
- Note: Initialized only once at the compile time

```
/* f1.c */

int x;

void f(){
        x++;
}

void g(){
        x++;
}
```

```
/* f2.c */

extern int x;

void h(){
        x++;
}
```

Static Variables

Lifetime: the entire program cycle

Scope: within the block where they are defined (file, function, block)

Note: initialized only once at compile time

```
void f(){
        int x = 0;
        printf("%d\n", x++);
}

int main(){
        f(); // 0
        f(); // 0
        f(); // 0
}
```

```
void f(){
        static int x = 0;
        printf("%d\n", x++);
}

int main(){
        f(); // 0
        f(); // 1
        f(); // 2
}
```


Register Variables

- Similar to automatic variables except that they are located in CPU's registers

  (1) can not take the address of a register variable,

  (2) can not declare global register variables,

  (3) a register variable must fit into a single machine word,

  (4) the compiler may ignore register declaration.

```
/* search the given table to find the given key;
   return the index if found or –1 otherwise          */

int table_search(int a[], register int n, register int key){
        register int j;
        for (j = 0; j < n && a[j] != key; j++);
        return (j != n) ? j : -1;
}
```

Enumerated Types:

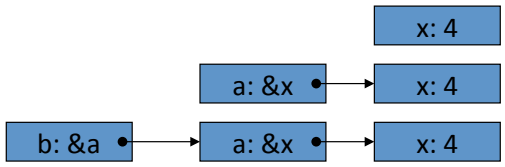
- Enumerated type is used to specify a small range of possible values.
- A enumerated type is defined by giving the keyword enum followed by an optional type designator and a brace-enclosed list of identifiers.

```
enum color {BLUE, RED, WHITE, BLACK};
enum myType { STRING = 2, INTEGER = 0, REAL};
```

- The list of ids represent a list of constants equal to their position in the list; or user may assign special values to ids in the list. The default value for item is 1 more than the item preceding it.

Review of pointer:

```
int x = 4;
int *a = &x;
int **b = &a;
```



x	4	a	addr(x)	b	addr(a)
&x	addr(x)	&a	addr(a)	&b	addr(b)
*x	illegal	*a	4	*b	addr(x)
*(&x)	4	**a	illegal	**b	4
				***b	illegal

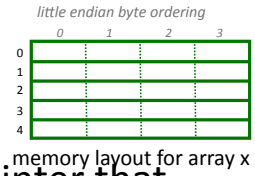
*b == a == &x
**b == *a == x

Arrays and Pointers

- A variable declared as an array represents a contiguous region of memory in which the array elements are stored.

```
int x[5]; // an array of 5 4-byte ints.
```

- All arrays begin with an index of 0



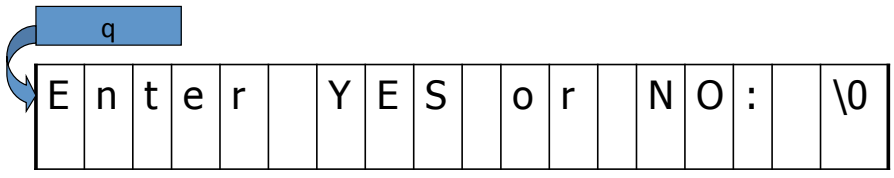
memory layout for array x

- An array identifier is equivalent to a pointer that references the first element of the array
  - `int x[5], *ptr;`
    `ptr = &x[0]` is equivalent to `ptr = x;`
- Pointer arithmetic and arrays:
  - `int x[5];`
    `x[2]` is the same as `*(x + 2)`, the compiler will assume you mean 2 objects beyond element x.

Strings:

- A string is an array of characters, terminated with a trailing null character, '\0'.

```
char *s = "CIS 2520";
char *q;
        q = "Enter YES or NO: ";
```



Ragged Array:

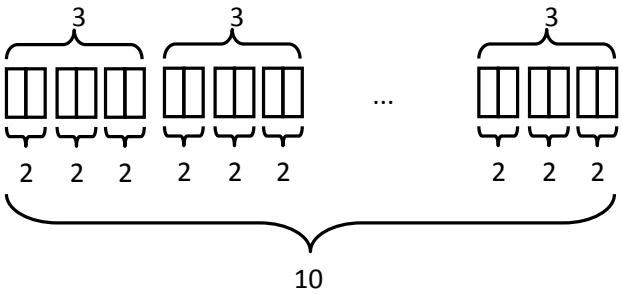
- A ragged array is an array of pointers where each entry in the array is a pointer to a string (arbitrary length). For example:

  char *days[] = { “monday”, “tuesday”, “wednesday”, “thursday”, “friday”, “saturday”, “sunday” };

  The compiler allocates an array containing 7 elements and assigns each element a pointer to the corresponding string.

Multidimensional Arrays

- Array declarations read right-to-left
- int a[10][3][2];
- “an array of ten arrays of three arrays of two ints”
- In memory



Arrays of Pointers:

- 2-D arrays contain the same number of elements in each row. For example:

```
char days[][10] = {
        {'m','o','n','d','a','y','\0'},
        {'t','u','e','s','d','a','y','\0'},
        …
};
```

Space is wasted in the rows containing shorter strings. Can we build an array whose rows can vary in length?

Coding Convention

- Constants:

```
#define     BUFSIZE          20
void somefun()
{…
   char  buf[BUFSIZE];
…}
```

Coding Convention

- Name:

  Use descriptive names for global vars, short name for local vars.

  Be consistent when naming functions, types, variables, and constants.

?
```
for (theElementIndex = 0;
        theElementIndex < numberOfElements;
        theElementIndex++)
        elementArray[theElementIndex] = 0;
```

```
for (i = 0; i < n; i++) a[i] = 0;
```


Coding Convention

- Do not use id's that contains two or more underscores in a row.
- Do not use id's that begin with an underscore.

```
int i_ _j = 11; // illegal
int _ _m = 20; // illegal
int _k = 10;      // not recommended
```

Coding Convention

- Do not change a loop variable inside a for loop block.
- Update loop variables close to where the loop condition is specified.
- All flow control primitives (if, else, while, for, do, switch, and case) should be followed by a block, even if it is empty.
- Statements following a case label should be terminated by a statement that exits the switch statement.
- All switch statements should have a default case.
- Use break and continue instead of goto.
- Do not have overly complex functions.
- Indent to show program structure (better readability).
- Parenthesize to resolve ambiguity in precedence.

Operator Precedence (from "C a Reference Manual", 5th Edition)

Tokens	Operator	Class	Precedence	Associates
names, literals	simple tokens	primary	16	n/a
a[k]	subscripting	postfix		left-to-right
f(...)	function call	postfix		left-to-right
.	direct selection	postfix		left-to-right
->	indirect selection	postfix		left to right
++ --	increment, decrement	postfix		left-to-right
(*type*){*init*}	compound literal	postfix		left-to-right
++ --	increment, decrement	prefix	15	right-to-left
sizeof	size	unary		right-to-left
~	bitwise not	unary		right-to-left
!	logical not	unary		right-to-left
- +	negation, plus	unary		right-to-left
&	address of	unary		right-to-left
*	indirection (*dereference*)	unary		right-to-left

Tokens	Operator	Class	Precedence	Associates
(*type*)	casts	unary	14	right-to-left
* / %	multiplicative	binary	13	left-to-right
+ -	additive	binary	12	left-to-right
<< >>	left, right shift	binary	11	left-to-right
< <= > >=	relational	binary	10	left-to-right
== !=	equality/ineq.	binary	9	left-to-right
&	bitwise and	binary	8	left-to-right
^	bitwise xor	binary	7	left-to-right
|	bitwise or	binary	6	left-to-right
&&	logical and	binary	5	left-to-right
||	logical or	binary	4	left-to-right
?:	conditional	ternary	3	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	assignment	binary	2	right-to-left
,	sequential eval.	binary	1	left-to-right

Problems with Precedence

- c = getchar() != EOF
  - **Expectation**: (c = getchar()) != EOF
  - **Actually**: c = (getchar() != EOF)
    - c is set equal to the true/false value
  - **Why**: == and != have higher precedence than assignment

Coding Convention

- Indent

```
for (j = 0; j < n; j++){
a[j] = j;
for (k = j ; k < n; k++){
if (a[j] < 5)
a[k] = a[j];
else
a[k] = k;
}
}
```

```
for (j = 0; j < n; j++){
    a[j] = j;
    for (k = j ; k < n; k++){
        if (a[j] < 5)
            a[k] = a[j];
        else
            a[k] = k;
    }
}
```

Coding Convention

- Be careful with side effects. Operators like ++ have side effects: besides returning a value, they also modify an underlying variable. Side effects can be extremely convenient, but they can also cause trouble because the actions of retrieving the value and updating the variable might not happen at the same time. In C the order of execution of side effects is undefined.

? str[j++] = str[j++] = 'a';

str[j++] = 'a';

str[j++] = 'a';

? a[j++] = j;

? scanf("%d%d", &x, &a[x]);

Coding Convention

- Break up complex expressions:

? ***x += (*xp = (2 * k < (n – m) ? c[k+1] : d[k--]));**

```
if  (2 * k < n – m)
        *xp = c[k+1];
else
        *xp = d[k--];
*x += *xp;
```

? **child = (!LC && !RC) ? 0 : (!LC ? RC : LC);**

```
if (LC == 0 && RC == 0)
        child = 0;
else if (LC == 0)
        child = RC;
else
        child = LC;
```

Coding Convention

- Number 0:

? char *str = 0;
? name[n-1] = 0;
? double x = 0;

/* reserve 0 for a literal integer zero */

char *str = NULL;
name[n-1] = '\0';
double x = 0.0;

Coding Convention

- Macros increase readability

```
/* often best to define these types of macro right where they are used */
#define CASE(str) if (strncasecmp(arg, str, strlen(str)) == 0)

void parse_command(char *arg)
{
  CASE("help") {
    /* print help */
  }
  CASE("quit") {
    exit(0);
  }
}

/* and un-define them after use */
#undef CASE
```

```
void parse_command(char *arg)
{
  if (strncasecmp(arg, "help", strlen("help")) {
    /* print help */
  }
  if (strncasecmp(arg, "quit", strlen("quit")) {
    exit(0);
  }
}
```

Macro Preprocessor pitfalls

- Example: the "min" function

```
int min(int a, int b)
 { if (a < b) return a; else return b; }
#define min(a,b) ((a) < (b) ? (a) : (b))
```

- Identical for min(5,x)
- Different when evaluating expression has side-effect:

  min(a++,b)
  - min function increments a once
  - min macro may increment a twice if a < b

Macro Preprocessor Pitfalls

- Text substitution can expose unexpected groupings

```
#define mult(a,b) a*b
mult(5+3,2+4)
```

- Expands to `5 + 3 * 2 + 4`
- Operator precedence evaluates this as

5 + (3*2) + 4 = 15 not (5+3) * (2+4) = 48 as intended

- Moral: By convention, enclose each macro argument in parenthesis:

```
#define mult(a,b) (a)*(b)
```

Coding Convention

- goto
  - More efficient runtime performance
  - Less readable
  - debatable

make and Makefiles, Overview

- Why use make?
  - convenience of only entering compile directives once
  - make is smart enough (with your help) to only compile and link modules that have changed or which depend on files that have changed
  - allows you to hide platform dependencies
  - promotes uniformity
  - simplifies my (and hopefully your) life when testing and verifying your code
- A makefile contains a set of rules for building a program

  target ... : prerequisites ...

  [tab] command

  ...
- Static pattern rules.
  - each target is matched against target-pattern to derive stem which is used to determine prereqs (see example)

    targets ... : target-pattern : prereq-patterns ...

    command

    ...

Exercise. 2

- *Create* file *makefile* in the same folder
- *Build the target*:
  - *make*
- *Remove target and intermediate files*
  - *make clean*

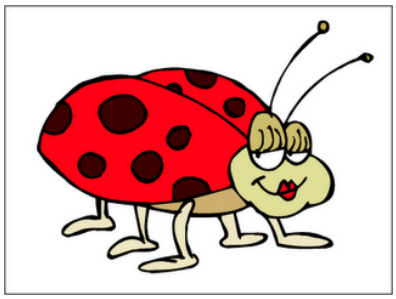
```
CC= gcc
CFLAGS = -g
LDFLAGS=
all: lab1

lab1: lab1.o
    # Commands start with TAB not spaces
     $(CC) $(LDFLAGS) -o $@ $^

lab1.o: lab1.c
     $(CC) $(CFLAGS) -c -o $@ $<

clean:
      rm -rf  *.o lab1
```


Bugs



- Bug: A defect or fault in a machine, plan, or the like.

  - Oxford English Dictionary

  (1) do not blame your computer

  (2) do not blame your compiler

  (3) do not blame the standard library

  ➔ experienced programmers know that, realistically, most problems are their own fault.

Debug

- Debugger
- Write self-checking code:

```
void check(char* s){
        printf("%s\n", s);
        fflush(stdout);
        abort();
}
Usage:
        check("…");             // before suspect
        // suspect code
        check("…");             // after suspect
```


Self-test:

For the following multiple choice questions circle all the answers for each question that are correct.

(1) Lifetime of a variable is defined by its

a) data type b) storage class

c) name d) memory address

(2) If a local variable has the same name as a global variable, what will happen?

a) A compiler error gets generated.

b) Both variables will share the same memory location.

c) The local variable will supersede the global variable, *i.e.* hide it.

d) Both are visible to the entire program.

(3) Given the declaration, which scanf would successfully read in a float value into x?

```
float  x, *y = &x, **z = &y;
```

a) scanf("%f", &x); b) scanf("%f", *y);

c) scanf("%f", *z); d) scanf("%f", y);

(4) Address of an array declared as int x[10] is indicated by

a) *(&x) b) &x[0]

c) &(*x) d) none of the above

(5) What would be the value of t after the following code executes?

```
int t = 0;
char *s = "Hello world!";
char *u = s + 2;
for ( ; *u; ++u, ++t);
```

a) 10 b) 12

c) 13 d) none of the above

(6) What's the difference between char and unsigned char?

(7) Give the difference between functions and Macros

(8) When do you need to define a function?

(9) 0, 0.0, '\0' and NULL, are they the same?

(10)
```
int x[10]
int *pi = x;
char *pc = (char*) x;
//is pi+5 == pc+5?
```


Trace the program and give the output

```
#include <stdio.h>
int main(void){
    char s[] = "bskbmshd";
    int i = 0;
    while(s[i]){
        if (s[i] < 'f')
            s[i++] -= 1;
        else
            s[i++] += 1;
    }
    printf("The resulting string is %s.\n", s);
    return 0;
}
```