Introduction to Computers and Programming

Lecture 12 – Chap 20+chap18
Low level programming
Variable declaration

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Low level programming

- bit operations

Bitwise Shift Operators

- C provides six ***bitwise operators,*** which operate on integer data at the bit level.
- The bitwise shift operators shift the bits in an integer to the left or right:

  `<<` left shift

  `>>` right shift
- The operands for `<<` and `>>` may be of any integer type (including `char`).

Bitwise Shift Operators

- The value of `i << j` is the result when the bits in `i` are shifted left by `j` places.
  - For each bit that is "shifted off" the left end of `i`, a zero bit enters at the right.
- The value of `i >> j` is the result when `i` is shifted right by `j` places.
  - If `i` is of an unsigned type or if the value of `i` is nonnegative, zeros are added at the left as needed.
  - If `i` is negative, the result is implementation-defined.

Bitwise Shift Operators

```
unsigned short i, j;
  i = 13;
    /* i is now 13 (binary 0000000000001101) */
  j = i << 2;
    /* j is now 52 (binary 0000000000110100) */
  j = i >> 2;
    /* j is now  3 (binary 0000000000000011) */
```

- Compound assignment operators `<<=` and `>>=`:

```
  i <<= 2;
    /* i is now 52 (binary 0000000000110100) */
```

- The bitwise shift operators have lower precedence than the arithmetic operators, :

  `i << 2 + 1` means `i << (2 + 1)`, not `(i << 2) + 1`

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- four additional bitwise operators:

  `~` bitwise complement

  `&` bitwise *and*

  `^` bitwise exclusive *or*

  `|` bitwise inclusive *or*

- The `~`, `&`, `^`, and `|` operators perform Boolean operations on all bits in their operands.
- The `^` operator produces 0 whenever both operands have a 1 bit, whereas `|` produces 1.

Examples of ~, &, ^, and | operators

```
unsigned short i, j, k;

i = 21;
  /* i is now    21 (binary 0000000000010101) */

j = 56;
  /* j is now    56 (binary 0000000000111000) */

k = ~i;
  /* k is now 65514 (binary 1111111111101010) */

k = i & j;
  /* k is now    16 (binary 0000000000010000) */

k = i ^ j;
  /* k is now    45 (binary 0000000000101101) */

k = i | j;
  /* k is now    61 (binary 0000000000111101) */
```

Bitwise Complement, *And*, Exclusive *Or*, and Inclusive *Or*

- The `~` operator can be used to make low-level programs.
  - An integer whose bits are all 1: `~0`
  - An integer whose bits are all 1 except for the last five: `~0x1f`
- Each of the `~`, `&`, `^`, and `|` operators has a different precedence:

  Highest: `~`

  `&`

  `^`

  Lowest: `|`
- Examples:

  `i & ~j | k` means `(i & (~j)) | k`

  `i ^ j & ~k` means `i ^ (j & (~k))`

Compound assignment &=, ^=, and |=

```
i = 21;
  /* i is now 21 (binary 0000000000010101) */
j = 56;
  /* j is now 56 (binary 0000000000111000) */
i &= j;
  /* i is now 16 (binary 0000000000010000) */
i ^= j;
  /* i is now 40 (binary 0000000000101000) */
i |= j;
  /* i is now 56 (binary 0000000000111000) */
```

Bitwise Operators to Access Bits

- ***Setting a bit.*** The easiest way to set bit 4 of `i` is to *or* the value of `i` with the constant `0x0010`:

```
i = 0x0000;
   /* i is now 0000000000000000 */

i |= 0x0010;
   /* i is now 0000000000010000 */
```

- If the position of the bit is stored in the variable `j`, a shift operator can be used to create the mask:

```
i |= 1 << j;          /* sets bit j */
```

- Example: If `j` has the value 3, then `1 << j` is `0x0008`.

Bitwise Operators to Access Bits

- ***Clearing a bit.*** Clearing bit 4 of `i` requires a mask with a 0 bit in position 4 and 1 bits everywhere else:

```
i = 0x00ff;
   /* i is now 0000000011111111 */

i &= ~0x0010;
   /* i is now 0000000011101111 */
```

- A statement that clears a bit whose position is stored in a variable:

```
i &= ~(1 << j);      /* clears bit j */
```

Bitwise Operators to Access Bits

- ***Testing a bit.*** An `if` statement that tests whether bit 4 of `i` is set:

```
if (i & 0x0010) …    /* tests bit 4 */
```

- A statement that tests whether bit `j` is set:

```
if (i & 1 << j) …    /* tests bit j */
```

Bitwise Operators to Access Bits

- Suppose that bits 0, 1, and 2 of a number correspond to the colors blue, green, and red, respectively.
- Names that represent the three bit positions:

```
#define BLUE  1
#define GREEN 2
#define RED   4
```

- Examples of setting, clearing, and testing the `BLUE` bit:

```
i |= BLUE;          /* sets BLUE bit   */
i &= ~BLUE;         /* clears BLUE bit */
if (i & BLUE) …     /* tests BLUE bit  */
```

Bitwise Operators to Access Bits

- It's also easy to set, clear, or test several bits at time:

```
i |= BLUE | GREEN;
   /* sets BLUE and GREEN bits    */

i &= ~(BLUE | GREEN);
   /* clears BLUE and GREEN bits */

if (i & (BLUE | GREEN)) …
   /* tests BLUE and GREEN bits   */
```

- The `if` statement tests whether either the `BLUE` bit or the `GREEN` bit is set.

Modifying a bit-field

- Modifying a bit-field requires two operations:
  - A bitwise *and* (to clear the bit-field)
  - A bitwise *or* (to store new bits in the bit-field)

```
i = i & ~0x0070 | 0x0050;
  /* stores 101 in bits 4-6 */
```

- `j` contains the value to be stored in bits 4–6 of `i`.
- `j` will need to be shifted into position before bitwise :

```
i = (i & ~0x0070) | (j << 4);
  /* stores j in bits 4-6 */
```

Retrieving a bit-field

- Fetching a bit-field at the right end of a number (in the least significant bits) is easy:

```
j = i & 0x0007;
  /* retrieves bits 0-2 */
```

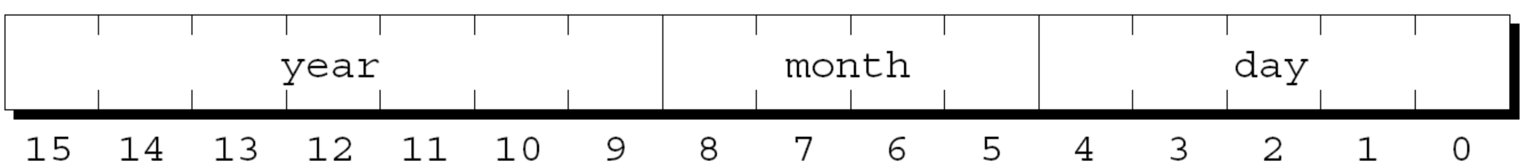
- If the bit-field isn't at the right end of `i`, shift first the bit-field and extract the field using the `&` operator:

```
j = (i >> 4) & 0x0007;
  /* retrieves bits 4-6 */
```

Bit-Fields

Bit-Fields in Structures

- DOS stores the date at which a file was created or last modified.
- Since days, months, and years are small numbers, storing them as normal integers would waste space.
- DOS allocates only 16 bits for a date, with 5 bits for the day, 4 bits for the month, and 7 bits for the year:



Bit-Fields Structures in C

- File date bit-fields for an identical layout:

```
struct file_date {
  unsigned int day: 5;
  unsigned int month: 4;
  unsigned int year: 7;
};
```

- A condensed version:

```
struct file_date {
  unsigned int day: 5, month: 4, year: 7;
};
```

Bit-Fields in Structures

- A bit-field can be used as any other member of a structure:

```
struct file_date fd;

fd.day = 28;
fd.month = 12;
fd.year = 8;      /* represents 1988 */
```

- Appearance of the `fd` variable after these assignments:

0	0	0	1	0	0	0	1	1	0	0	1	1	1	0	0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bit-Fields in Structures

- The address operator (`&`) can't be applied to a bit-field.
- Because of this rule, functions such as `scanf` can't store data directly in a bit-field:

```
scanf("%d", &fd.day);    /*** WRONG ***/
```

- We can still use `scanf` to read input into an ordinary variable and then assign it to `fd.day`.

How Bit-Fields Are Stored

- The same structure with the name of the `seconds` field omitted:

```
struct file_time {
  unsigned int : 5;        /* not used */
  unsigned int minutes: 6;
  unsigned int hours: 5;
};
```

- The remaining bit-fields will be aligned as if `seconds` were still present.

How Bit-Fields Are Stored

- The length of an unnamed bit-field can be 0:

```
struct s {
  unsigned int a: 4;
  unsigned int : 0;      /* 0-length bit-field */
  unsigned int b: 8;
};
```

- A 0-length bit-field tells the compiler to align the following bit-field at the beginning of a storage unit.
  - If storage units are 8 bits long, the compiler will allocate 4 bits for `a`, skip 4 bits to the next storage unit, and then allocate 8 bits for `b`.
  - If storage units are 16 bits long, the compiler will allocate 4 bits for `a`, skip 12 bits, and then allocate 8 bits for `b`.

Declarations

Declaration Syntax

- A declaration with a storage class and three declarators:

storage class　　　declarators

```
static float x, y, *p;
```

type specifier

- A declaration with a type qualifier and initializer but no storage class:

type qualifier　　　declarator

```
const char month[] = "January";
```

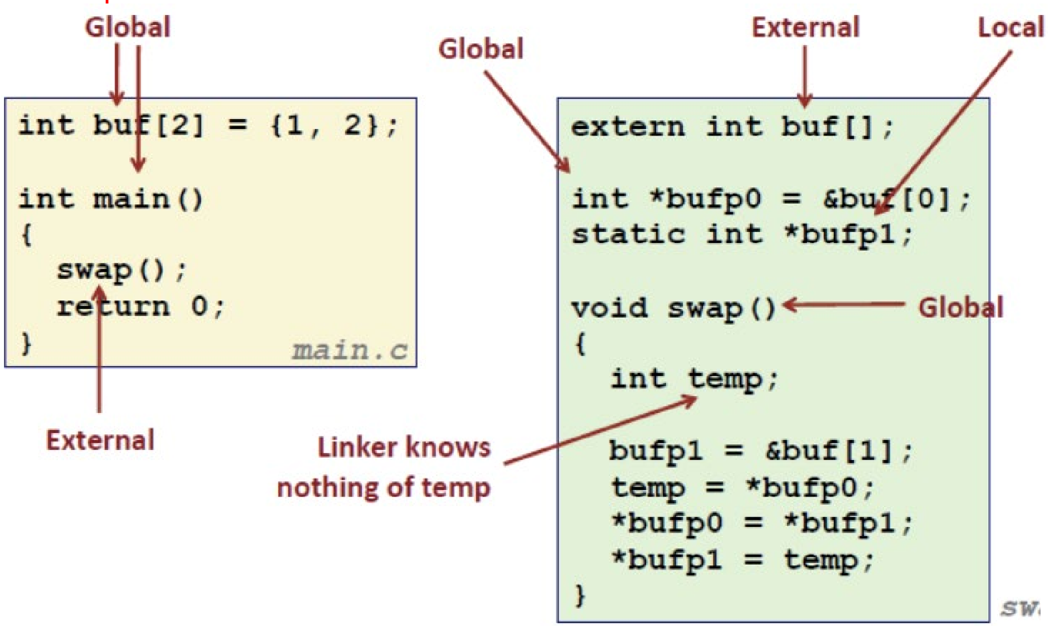
type specifier　　　initializer

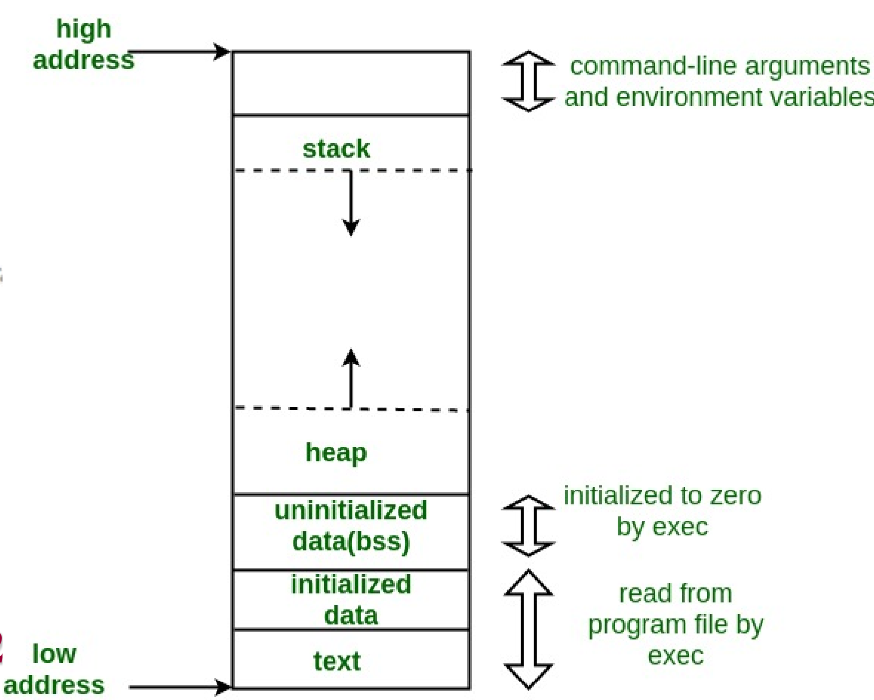
Declaration Syntax

- There are four ***storage classes:*** `auto`, `static`, `extern`, and `register`.
- In C89, two ***type qualifiers:*** `const` and `volatile`.
- ***type specifiers:*** `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, and `unsigned` are all.
- Type specifiers also include specifications of structures, unions, and enumerations.
  - Examples: `struct point { int x, y; }`, `struct { int x, y; }`, `struct point`.
- Declarators include:
  - Identifiers (names of simple variables)
  - Identifiers followed by `[]` (array names)
  - Identifiers preceded by `*` (pointer names)
  - Identifiers followed by `()` (function names)

Dynamically allocating Storage

- `malloc` and the other memory allocation functions obtain memory blocks from a storage pool known as ***heap.***






```
char *p = "this is a book";
```

Properties of Variables

- Example:

```
int i;
```

static storage duration
file scope
external linkage

```
void f(void)
{
    int j;
}
```

automatic storage duration
block scope
no linkage

- We can alter these properties by specifying an explicit storage class: `auto`, `static`, `extern`, or `register`.

The `static` Storage Class

- Example:

```
static int i;
```
static storage duration
file scope
internal linkage

```
void f(void)
{

   static int j;

}
```
static storage duration
block scope
no linkage

The `static` Storage Class

- Declaring a local variable to be `static` allows a function to retain information between calls.
- use `static` for reasons of efficiency:

```
char digit_to_hex_char(int digit)
{
  static const char hex_chars[16] =
    "0123456789ABCDEF";

  return hex_chars[digit];
}
```

- Declaring `hex_chars` to be `static` saves time, because `static` variables are initialized only once.

The `register` Storage Class

- `register` is best used for variables that are accessed and/or updated frequently.
- The loop control variable in a `for` statement is a good candidate for `register` treatment:

```
int sum_array(int a[], int n)
{
  register int i;
  int sum = 0;

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

Type Qualifiers

- There are two type qualifiers: `const` and `volatile`.
- `const` is used to declare "read-only" objects.
- Examples:

```
const int n = 10;
const int tax_brackets[] =
  {750, 2250, 3750, 5250, 7000};
```

- `volatile` declaration of a pointer variable that will point to a volatile memory location:

```
volatile BYTE *p;
  /* p will point to a volatile byte */
```

Summary example

```
int a;
extern int b;
static int c;

void f(int d,
    register int e)
{
  auto int g;
  int h;
  static int i;
  extern int j;
  register int k;
}
```

Name	*Storage Duration*	*Scope*	*Linkage*
a	static	file	external
b	static	file	†
c	static	file	internal
d	automatic	block	none
e	automatic	block	none
g	automatic	block	none
h	automatic	block	none
i	static	block	none
j	static	block	†
k	automatic	block	none