


Introduction to Computers and Programming

Lecture 5 –
Program scope
and basic pointer
Chap 10 & 11

Tien-Fu Chen

Dept. of Computer Science and
Information Engineering

National Yang Ming Chiao Tung Univ.



Program organization

Local Variables

- ❑ A variable declared in the body of a function is said to be ***local*** to the function:

```
int sum_digits(int n)
{
    int sum = 0;    /* local variable */

    while (n > 0) {
        sum += n % 10;
        n /= 10;
    }

    return sum;
}
```

Local Variables

- ❑ Default properties of local variables:
 - **Automatic storage duration.** Storage is “automatically” allocated when the enclosing function is called and deallocated when the function returns.
 - **Block scope.** A local variable is visible from its point of declaration to the end of the enclosing function body.
- ❑ In C99, it’s possible for a local variable to have a very small scope:

```
void f(void)
{
    ...
    int i;
    ...
}
```

} scope of i

Parameters

- ❑ Parameters have the same properties as local variables
 - automatic** storage duration and block scope
- ❑ Each parameter is initialized automatically when a function is called (by being assigned the value of the corresponding argument).

```
void store_zeros(int a[], int n)
{
    int i;

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

```
store_zeros(b, 100);
```

External / Global Variables

- ❑ Passing arguments is one way to transmit information to a function.
- ❑ Functions can also communicate through **external variables** (also known as **global variables**)—variables that are declared outside the body of any function.

```
/* external variables */
int contents[100];
int top = 0;

void make_empty(void)
{
    top = 0;
}

bool is_empty(void)
{
    return top == 0;
}
```

```
void push(int i)
{
    if (is_full())
        stack_overflow();
    else
        contents[top++] = i;
}

int pop(void)
{
    if (is_empty())
        stack_underflow();
    else
        return contents[--top];
}
```

External Variables

- ❑ Properties of external variables:
 - Static storage duration
 - File scope
- ❑ Having **file scope** means that an external variable is visible from its point of declaration to the end of the enclosing file.

File1.c

```
int x;
```

File2.c

```
int y;  
extern int x;
```

Static Local Variables

- ❑ Including `static` causes a local variable to have ***static storage duration***.
- ❑ A variable with static storage duration has a permanent storage location, so it retains its value throughout the execution of the program.

- ❑ Example:

```
void f(void)
{
    static int i;    /* static local variable */
    ...
}
```

- ❑ A static local variable still has block scope, so it's not visible to other functions.

Pros and Cons of External Variables

- ❑ External variables are convenient:
 - when many functions must share a variable
 - when a few functions share a large number of variables.
- ❑ it's better for functions to communicate through parameters rather than by sharing variables:
 - If we change an external variable, we'll need to check every function to see how the change affects it.
 - If an external variable is assigned an incorrect value, it may be difficult to identify the guilty function.
 - Functions that rely on external variables are hard to reuse in other programs.
- ❑ Don't use the same external variable for different purposes in different functions.

Pros and Cons of External Variables

- ❑ Making variables external when they should be local can lead to some rather frustrating bugs.
- ❑ Code that is supposed to display a 10 × 10 arrangement of asterisks:

```
int i;

void print_one_row(void)
{
    for (i = 1; i <= 10; i++)
        printf("*");
}

void print_all_rows(void)
{
    for (i = 1; i <= 10; i++) {
        print_one_row();
        printf("\n");
    }
}
```

- ❑ Instead of printing 10 rows, `print_all_rows` prints only one.

Blocks

- ❑ compound statements of the form
`{ statements }`
- ❑ C allows compound statements to contain declarations as well as statements:
`{ declarations statements }`
- ❑ This kind of compound statement is called a ***block***.

```
if (i > j) {  
    /* swap values of i and j */  
    int temp = i;  
    i = j;  
    j = temp;  
}
```

Blocks

- ❑ The storage of a variable declared in a block is automatic:
 - storage for the variable is allocated when the block is entered and deallocated when the block is exited.
- ❑ The variable has block scope; it can't be referenced outside the block.
- ❑ A variable that belongs to a block can be declared `static` to give it static storage duration.

Blocks

- ❑ The body of a function is a block.
- ❑ Blocks are also useful inside a function body when we need variables for temporary use.
- ❑ Advantages of declaring temporary variables in blocks:
 - Avoids cluttering declarations at the beginning of the function body with variables that are used only briefly.
 - Reduces name conflicts.
- ❑ C99 allows variables to be declared anywhere within a block.

Scope

- ❑ Scope rules:

- to determine a variable where is relevant at a given point in the program.

- ❑ The most important scope rule:

- When a declaration inside a block names an identifier that's already visible
- a new declaration temporarily “hides” the old one, and the identifier takes on a new meaning.

- ❑ At the end of the block, the identifier regains its old meaning.

i is a variable with static storage duration and file scope

```
int i; /* Declaration 1 */
```

```
void f(int i) /* Declaration 2 */
```

```
{  
    i = 1;  
}
```

i is a parameter with block scope.

```
void g(void)  
{
```

```
    int i = 2; /* Declaration 3 */
```

i is an automatic variable with block scope.

```
    if (i > 0) {
```

```
        int i; /* Declaration 4 */
```

```
        i = 3;
```

```
    }
```

i is also automatic and has block scope.

```
    i = 4;
```

```
}
```

```
void h(void)
```

```
{
```

```
    i = 5;
```

```
}
```

Organizing a C Program

- ❑ There are several ways to organize a program so that these rules are obeyed.
- ❑ One possible ordering:
 - `#include` directives
 - `#define` directives
 - Type definitions
 - Declarations of external variables
 - Prototypes for functions other than `main`
 - Definition of `main`
 - Definitions of other functions

Write comments for functions

- ❑ It's a good idea to have a boxed comment preceding each function definition.
- ❑ Information to include in the comment:
 - Name of the function
 - Purpose of the function
 - Meaning of each parameter
 - Description of return value (if any)
 - Description of side effects (such as modifying external variables)

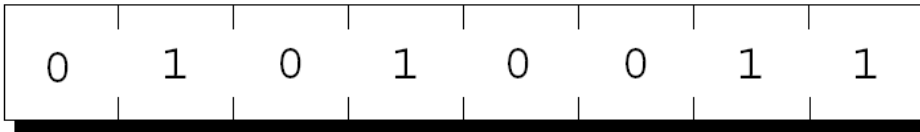
```
//  
// this is to comment a function  
//  
void func(int A, int B) {  
    //C = A*B;        /* this is another  
    comment line after C statement */  
}
```



Basic pointer

Pointer Variables

- ❑ In most modern computers, main memory is divided into **bytes**, with each byte capable of storing eight bits of information:

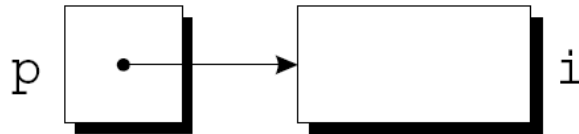
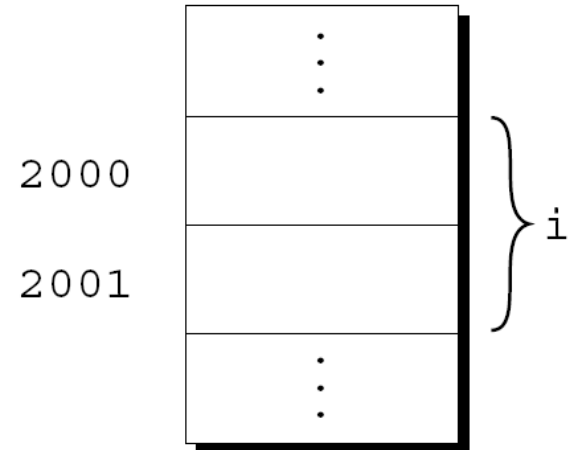


- ❑ Each byte has a unique **address**.
- ❑ If there are n bytes in memory, we can think of addresses as numbers that range from 0 to $n - 1$:

Address	Contents
0	01010011
1	01110101
2	01110011
3	01100001
4	01101110
	⋮
$n-1$	01000011

Pointer Variables

- ❑ The address of the first byte is said to be the address of the variable.
- ❑ The address of the variable `short int i` is 2000:
- ❑ Addresses can be stored in special ***pointer variables***.
- ❑ When we store the address of a variable `i` in the pointer variable `p`:
`p` “points to” `i`.



Declaring Pointer Variables

- ❑ When a pointer variable is declared, its name must be preceded by an asterisk:

```
int *p;
```

- ❑ `p` is a pointer variable capable of pointing to **objects** of type `int`.

- ❑ Pointer variables can appear in declarations along with other variables:

```
int i, j, a[10], b[20], *p, *q;
```

- ❑ C requires that every pointer variable point only to objects of a particular type (the **referenced type**):

```
int *p;      /* points only to integers    */
double *q;   /* points only to doubles      */
char *r;     /* points only to characters */
```

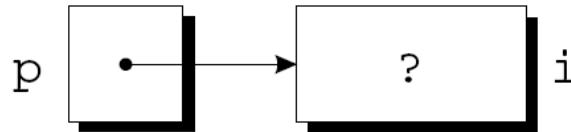
The Address Operator

- ❑ C provides a pair of operators for pointers.
 - Use the `&` (address) operator to get the address
 - Use the `*` (***indirection***) operator to gain access to the object.
- ❑ Initialize: Assign the address of a variable to a pointer variable

```
int i, *p;
```

```
...
```

```
p = &i;
```



The Address Operator

- ❑ It's also possible to initialize a pointer variable at the time it's declared:

```
int i;
```

```
int *p = &i;
```

- ❑ The declaration of `i` can even be combined with the declaration of `p`:

```
int i, *p = &i;
```

The Indirection Operator

- ❑ If `p` points to `i`, we can print the value of `i` as follows:

```
printf("%d\n", *p);
```

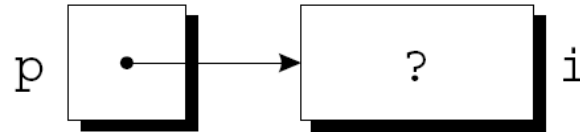
- ❑ Applying `&` to a variable produces a pointer to the variable. Applying `*` to the pointer takes us back to the original variable:

```
j = *&i;    /* same as j = i; */
```

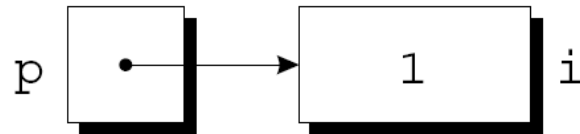
- ❑ When `p` points to `i`, `*p` is an ***alias*** for `i`.
 - `*p` has the same value as `i`.
 - Changing the value of `*p` changes the value of `i`.

The Indirection Operator

```
p = &i;
```



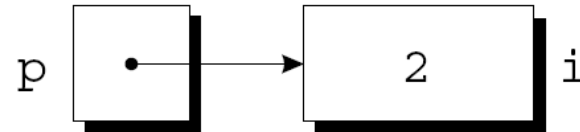
```
i = 1;
```



```
printf("%d\n", i);      /* prints 1 */
```

```
printf("%d\n", *p);     /* prints 1 */
```

```
*p = 2;
```



```
printf("%d\n", i);      /* prints 2 */
```

```
printf("%d\n", *p);     /* prints 2 */
```

The Indirection Operator

- ❑ Applying the indirection operator to an uninitialized pointer variable causes undefined behavior:

```
int *p;  
printf("%d", *p);    /** WRONG ***/
```

- ❑ Assigning a value to `*p` is particularly dangerous:

```
int *p;  
*p = 1;    /** WRONG ***/
```

Pointer Assignment

- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

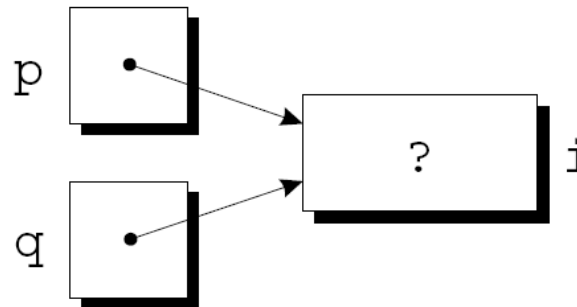
- Example of pointer assignment:

```
p = &i;
```

- Another example of pointer assignment:

```
q = p;
```

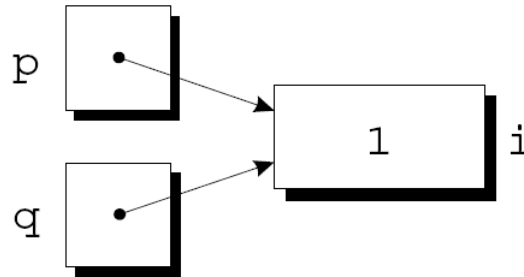
q now points to the same place as p:



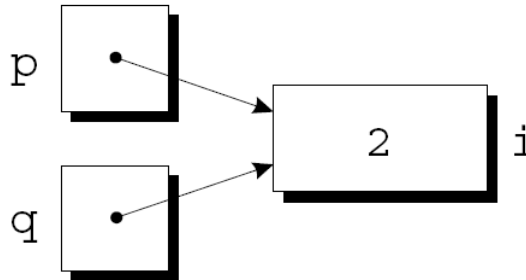
Pointer Assignment

- If p and q both point to i , we can change i by assigning a new value to either $*p$ or $*q$:

$*p = 1;$



$*q = 2;$



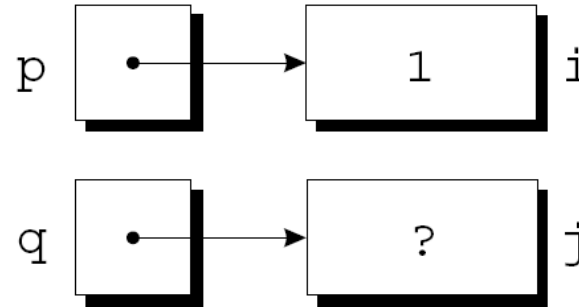
- Any number of pointer variables may point to the same object.

Pointer Assignment

```
p = &i;
```

```
q = &j;
```

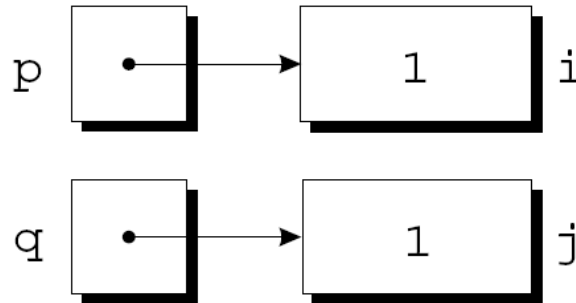
```
i = 1;
```



```
q = p;
```

pointer assignment

```
*q = *p;
```



Pointers as Arguments

- ❑ New definition of decompose:

```
void decompose(double x, long *int_part,  
               double *frac_part)  
{  
    *int_part = (long) x;  
    *frac_part = x - *int_part;  
}
```

- ❑ Possible prototypes for decompose:

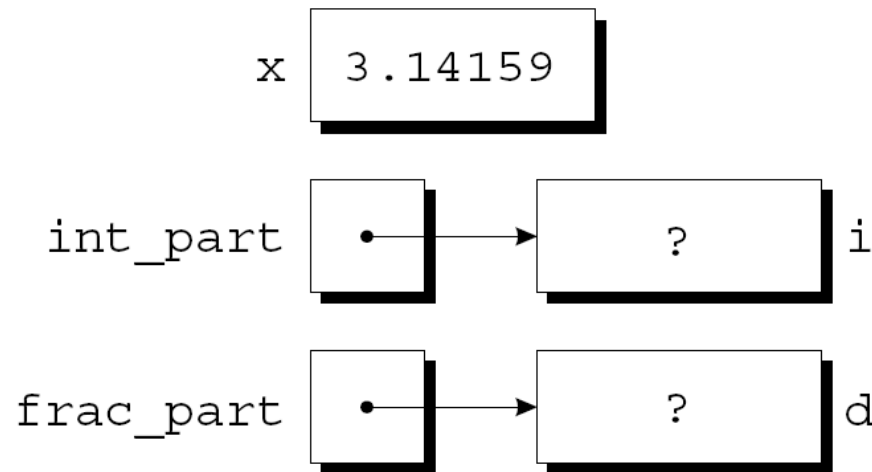
```
void decompose(double x, long *int_part,  
               double *frac_part);  
  
void decompose(double, long *, double *);
```

Pointers as Arguments

- ❑ A call of `decompose`:

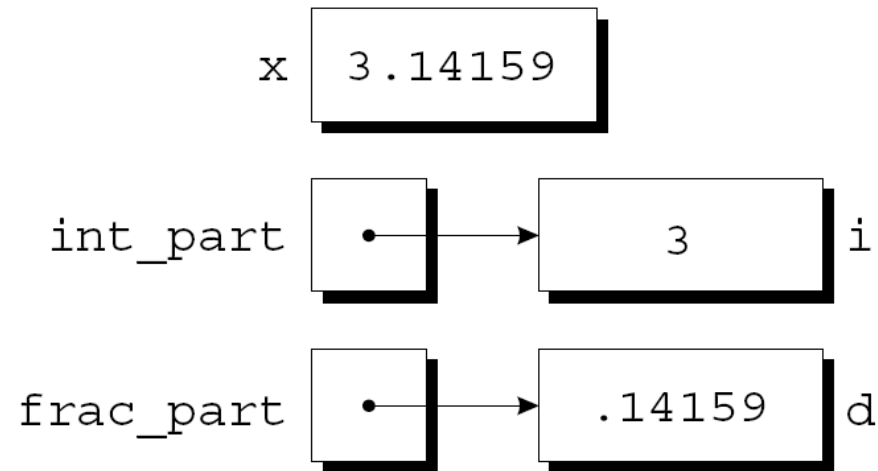
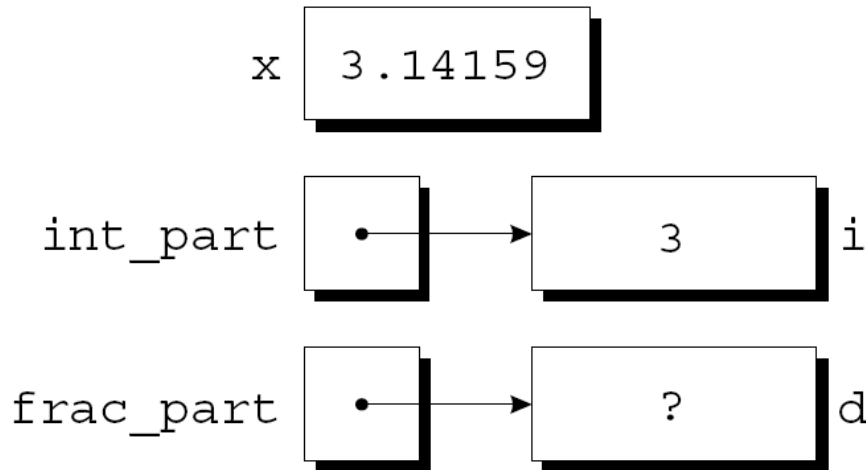
```
decompose(3.14159, &i, &d);
```

- ❑ As a result of the call, `int_part` points to `i` and `frac_part` points to `d`:



Pointers as Arguments

```
*int_part = (long) x;  
*frac_part = x - *int_part;
```



Pointers as Arguments

- ❑ Arguments in calls of `scanf` are pointers:

```
int i;
```

```
...
```

```
scanf("%d", &i);
```

Without the `&`, `scanf` would be supplied with the *value* of `i`.

Failing to pass pointers may go undetected.

Pointers as Arguments

- ❑ Although `scanf`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

```
int i, *p;
```

```
...
```

```
p = &i;
```

```
scanf("%d", p);
```

- ❑ Using the `&` operator in the call would be wrong:

```
scanf("%d", &p);    /* ** WRONG ** */
```

Pointers as Arguments

- ❑ Failing to pass a pointer to a function when one is expected can have disastrous results.
- ❑ A call of `decompose` in which the `&` operator is missing:

```
decompose(3.14159, i, d);
```

- ❑ When `decompose` stores values in `*int_part` and `*frac_part`, it will attempt to change unknown memory locations instead of modifying `i` and `d`.
- ❑ If we've provided a prototype for `decompose`, the compiler will detect the error.

Using `const` to Protect Arguments

- ❑ When an argument is a pointer to a variable `x`, we normally assume that `x` will be modified:

```
f (&x) ;
```

- ❑ We can use `const` to document that a function won't change an object.
- ❑ `const` goes in the parameter's declaration, just before the specification of its type:

```
void f(const int *p)
{
    *p = 0;    /** WRONG **/
}
```

Attempting to modify `*p` is an error that the compiler will detect.

Pointers as Return Values

- ❑ Functions are allowed to return pointers:

```
int *max(int *a, int *b)
{
    if (*a > *b)
        return a;
    else
        return b;
}
```

- ❑ A call of the `max` function:

```
int *p, i, j;
...
p = max(&i, &j);
```

After the call, `p` points to either `i` or `j`.

Pointers as Return Values

- ❑ A function could also return a pointer to an external variable or to a static local variable.
- ❑ Never return a pointer to an *automatic* local variable:

```
int *f(void)
{
    int i;
    ...
    return &i;
}
```

The variable `i` won't exist after `f` returns.

Pointers as Return Values

- ❑ Pointers can point to array elements.
- ❑ `a` is an array: `&a[i]` is a pointer to element `i` of `a`.
- ❑ A function that returns a pointer to the middle element of `a`, assuming that `a` has `n` elements:

```
int *find_middle(int a[], int n) {  
    return &a[n/2];  
}
```

Memory segment in Linux system

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack

