Introduction to Computers and Programming

Lecture 6 –
All about pointer
Chap 11 & 12

**Tien-Fu Chen**

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Basic pointer

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`:
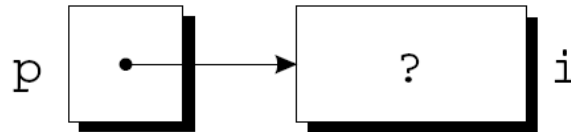
  `p` "points to" `i`.
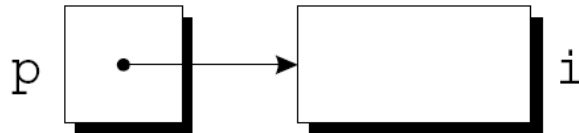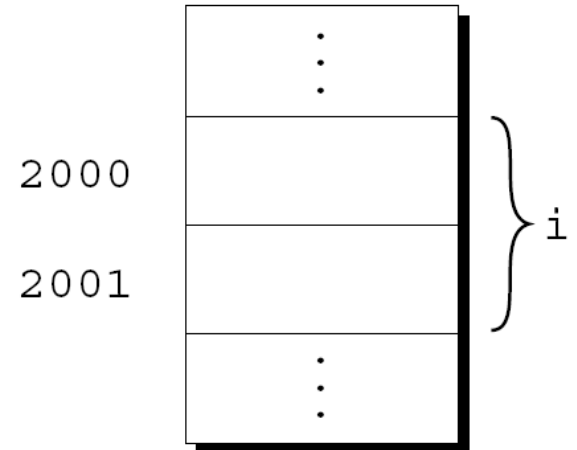
The Address Operator

❑ 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 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`.

Pointer Assignment

❑ Assume that the following declaration is in effect:
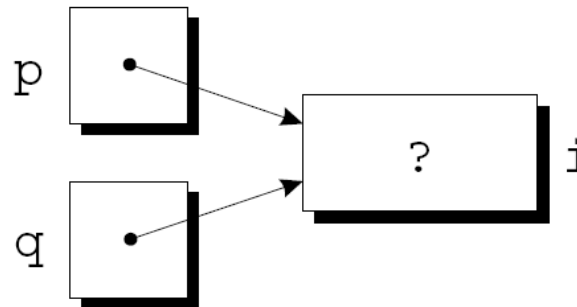
```
int i, j, *p, *q;
```

❑ Example of pointer assignment:

```
p = &i;
```

❑ Another example of pointer assignment:

```
q = p;
```

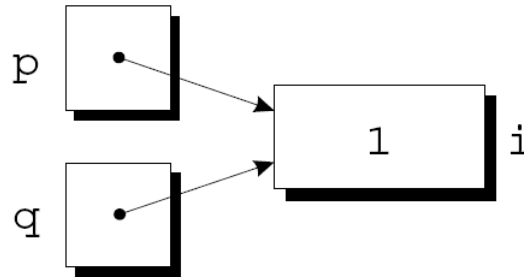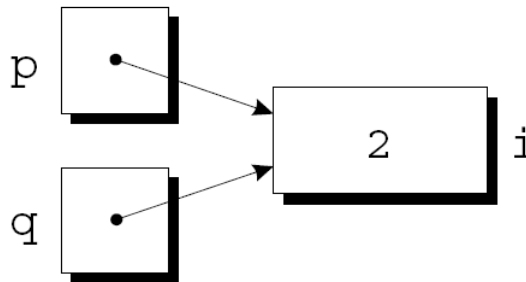q now points to the same place as p:

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;
```
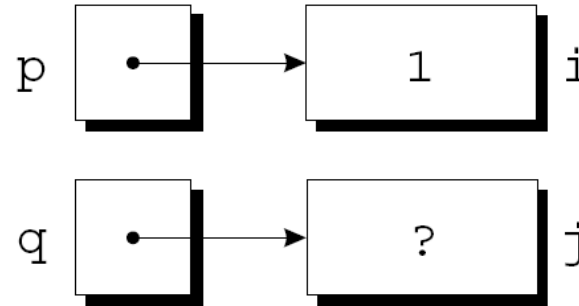

q = p;   **pointer assignment**

```
*q = *p;
```
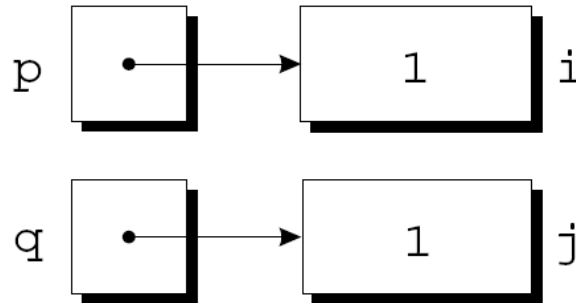
Pointers as Arguments

❑ 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`:
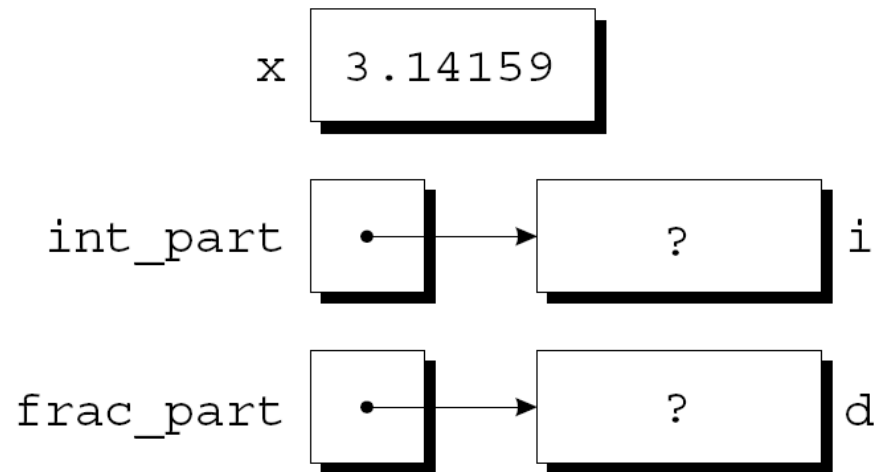
Pointers as Arguments

```
*int_part = (long) x;
*frac_part = x - *int_part;
```
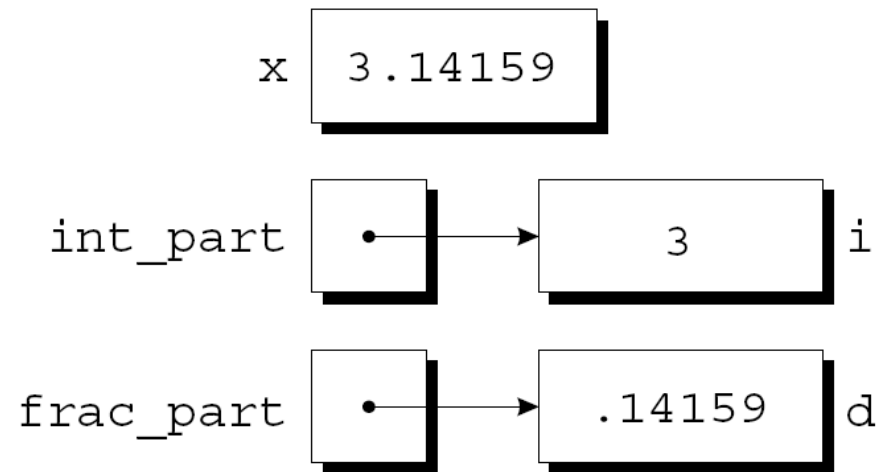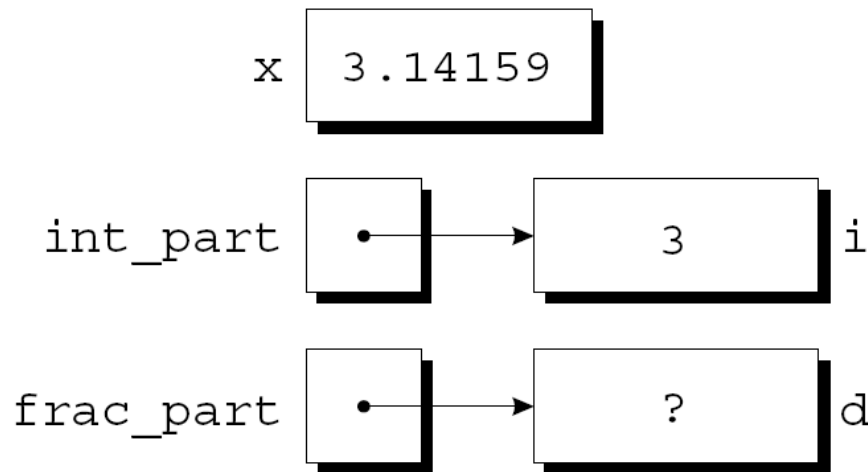
Pointers as Arguments

❑ 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 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`.

Quiz

A swap function is to exchange the values of two variables:

```
void swap (int *i, int *j)
{
    int temp = i;
    i = j;
    j = temp;
}

int A, B;
swap(A, B);
```

(a) Which variables are automatically allocated?
(b) Correct the above statements to make swapping A and B.

Pointer and array

Pointer Arithmetic

❑ A pointer variable can point to array elements:

```
int a[10], *p;
p = &a[0];
```

❑ A graphical representation:
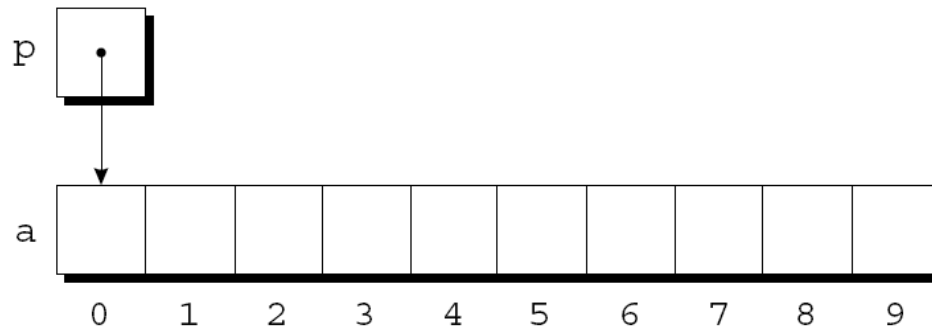
Pointer Arithmetic

❑ We can now access `a[0]` through `p`; for example, we can store the value 5 in `a[0]` by writing

```
*p = 5;
```

❑ An updated picture:
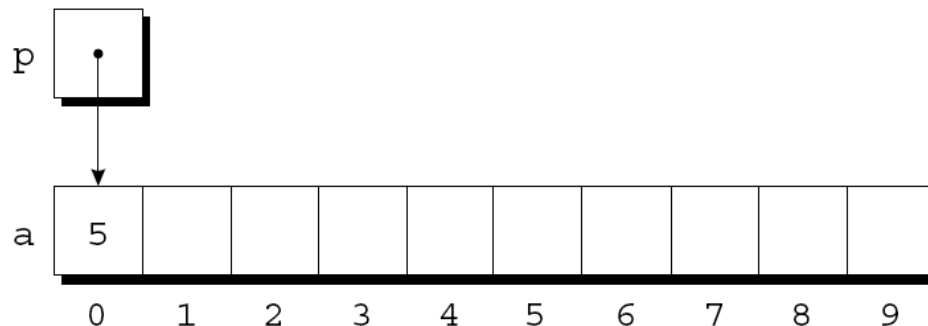
Pointer Arithmetic

❏ If `p` points to an element of an array `a`, the other elements of `a` can be accessed by performing *pointer arithmetic* (or *address arithmetic*) on `p`.

❏ C supports three forms of pointer arithmetic:

   – Adding an integer to a pointer

   – Subtracting an integer from a pointer

   – Subtracting one pointer from another

Adding an Integer to a Pointer

❑ Example of pointer addition:

`p = &a[2];`
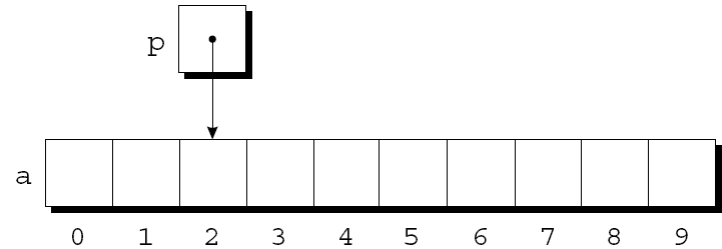
`q = p + 3;`
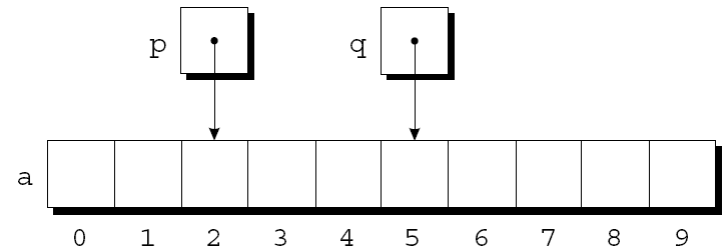
`p += 6;`
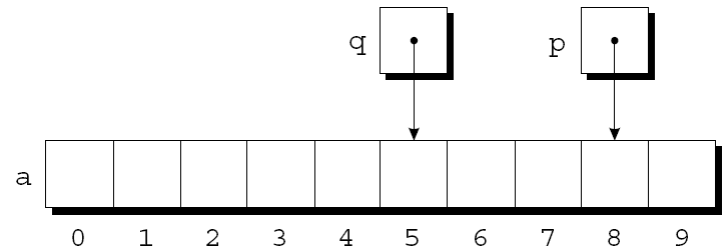
Adding an Integer to a Pointer

❑ Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.

❑ More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
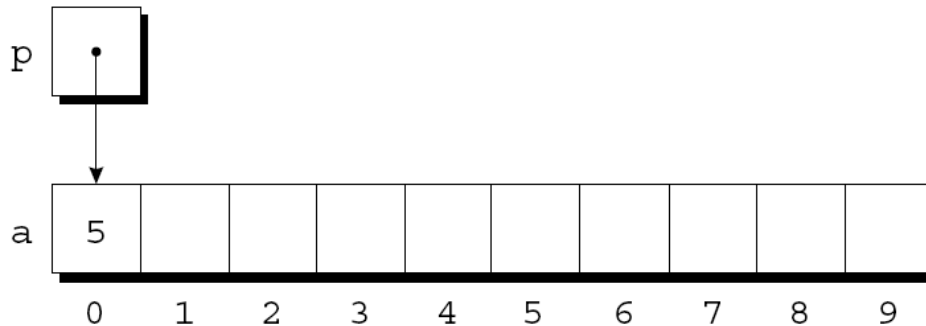
Subtracting an Integer from a Pointer

❑ If `p` points to `a[i]`, then `p - j` points to `a[i-j]`.

❑ Example:

```
p = &a[8];
```
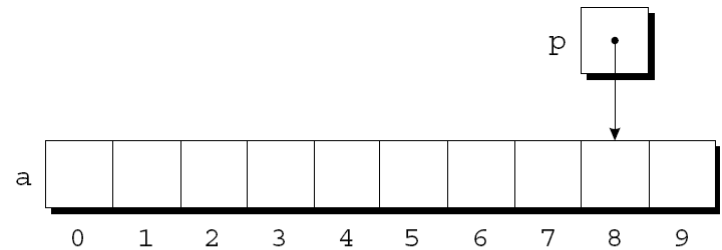
```
q = p - 3;
```
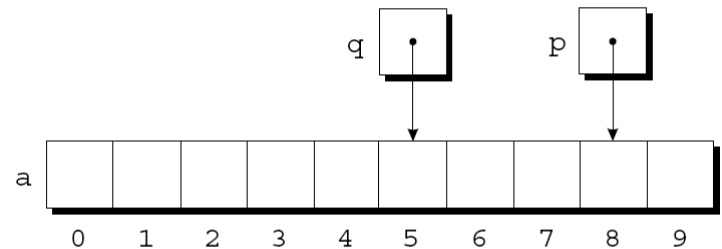
```
p -= 6;
```
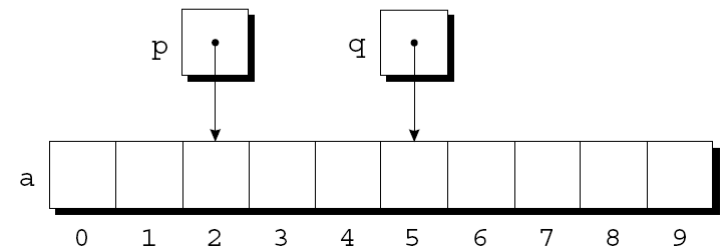
Subtracting One Pointer from Another

❑ Subtraction = the distance between the pointers

  – measured in array elements

❑ If `p` points to `a[i]` and `q` points to `a[j]`, then `p - q` is equal to `i - j`.

❑ Example:

```
p = &a[5];
q = &a[1];
```
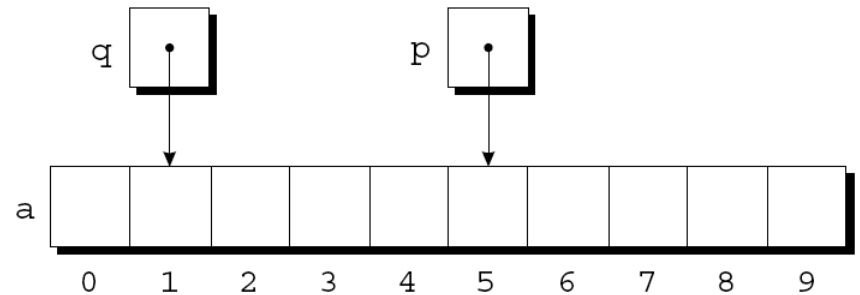


```
i = p - q;    /* i is 4 */
i = q - p;    /* i is -4 */
```

Subtracting Pointer from Another

❑ Operations that cause undefined behavior:

- Performing arithmetic on a pointer that doesn't point to an array element

- Subtracting pointers unless both point to elements of the same array

Comparing Pointers

❑ Pointers can be compared via relational operators ($<$, $<=$, $>$, $>=$) and the equality operators ($==$ $!=$).

  – Using relational operators is meaningful only for pointers to elements of the same array.

❑ The outcome of the comparison depends on the relative positions of the two elements in the array.

❑ After the assignments

```
p = &a[5];
q = &a[1];
```

the value of `p <= q` is 0

the value of `p >= q` is 1.

Pointers to Compound Literals (C99)

❑ It's legal for a pointer to point to an element within an array created by a compound literal:

```
int *p = (int []){3, 0, 3, 4, 1};
```

❑ Using a compound literal saves us the trouble of first declaring an array variable and then making `p` point to the first element of that array:

```
int a[] = {3, 0, 3, 4, 1};

int *p = &a[0];
```

Using Pointers for Array Processing

❑ Use pointer arithmetic to visit the elements of an array by repeatedly incrementing a pointer variable.

❑ A loop that sums the elements of an array `a`:

```
#define N 10

…

int a[N], sum, *p;

…

sum = 0;
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

Using Pointers for Array Processing

At the end of the first iteration:

p [·]

a | 11 | 34 | 82 | 7 | 64 | 98 | 47 | 18 | 79 | 20 |
     0   1   2   3   4   5   6   7   8   9

sum [11]
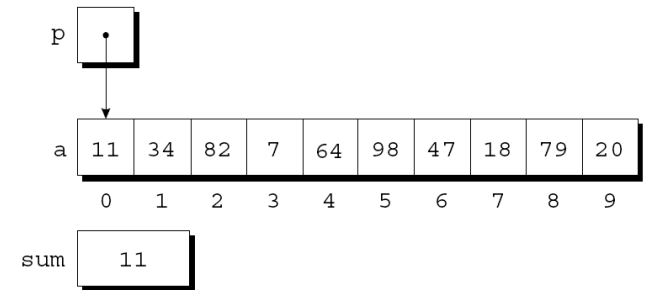
At the end of the second iteration:

p [·]

a | 11 | 34 | 82 | 7 | 64 | 98 | 47 | 18 | 79 | 20 |
     0   1   2   3   4   5   6   7   8   9

sum [45]
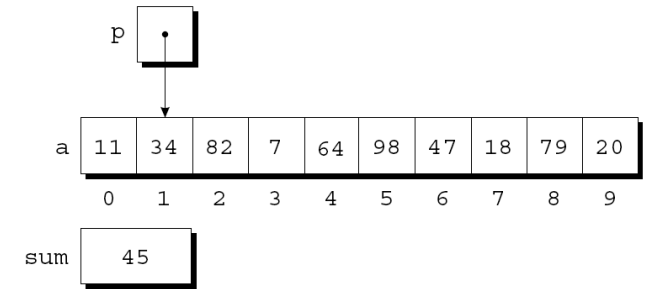
At the end of the third iteration:

p [·]

a | 11 | 34 | 82 | 7 | 64 | 98 | 47 | 18 | 79 | 20 |
     0   1   2   3   4   5   6   7   8   9

sum [127]
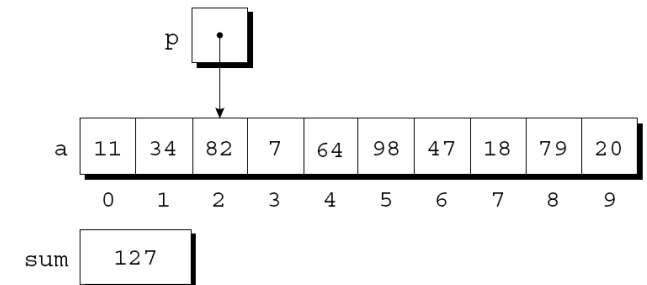
Using Pointers for Array Processing

❏ The condition `p < &a[N]` in the `for` statement deserves special mention.

❏ It's legal to apply the address operator to `a[N]`, even though this element doesn't exist.

❏ Pointer arithmetic may save execution time.

❏ However, some C compilers produce better code for loops that rely on subscripting.

Combining the * and ++ Operators

❑ C programmers often combine the * (indirection) and ++ operators.

❑ A statement that modifies an array element and then advances to the next element:

```
a[i++] = j;
```

❑ The corresponding pointer version:

```
*p++ = j;
```

❑ Because the postfix version of ++ takes precedence over *, the compiler sees this as

```
*(p++) = j;
```

Combining the * and ++ Operators

❑ Possible combinations of `*` and `++`:

Expression	Meaning
`*p++` or `*(p++)`	Value of expression is `*p` before increment; increment `p` later
`(*p)++`	Value of expression is `*p` before increment; increment `*p` later
`*++p` or `*(++p)`	Increment `p` first; value of expression is `*p` after increment
`++*p` or `++(*p)`	Increment `*p` first; value of expression is `*p` after increment

Combining the * and ++ Operators

❑ The most common combination of `*` and `++` is `*p++`, which is handy in loops.

❑ Instead of writing

```
for (p = &a[0]; p < &a[N]; p++)
  sum += *p;
```

to sum the elements of the array `a`, we could write

```
p = &a[0];
while (p < &a[N])
  sum += *p++;
```

Combining the * and ++ Operators

❏ The `*` and `--` operators mix in the same way as `*` and `++`.

❏ For an application that combines `*` and `--`, let's return to the stack example of Chapter 10.

❏ The original version of the stack relied on an integer variable named `top` to keep track of the "top-of-stack" position in the contents array.

❏ Let's replace `top` by a pointer variable that points initially to element 0 of the `contents` array:

```
int *top_ptr = &contents[0];
```

Combining the * and ++ Operators

❑ The new `push` and `pop` functions:

```
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    *top_ptr++ = i;
}

int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return *--top_ptr;
}
```

Using an Array Name as a Pointer

❑ Pointer arithmetic is one way in which arrays and pointers are related.

❑ Another key relationship:

The name of an array can be used as a pointer to the first element in the array.

❑ This relationship simplifies pointer arithmetic and makes both arrays and pointers more versatile.

Using an Array Name as a Pointer

❑ Suppose that `a` is declared as follows:

```
int a[10];
```

❑ Examples of using `a` as a pointer:

```
*a = 7;    /* stores 7 in a[0] */
*(a+1) = 12;    /* stores 12 in a[1] */
```

❑ In general, `a + i` is the same as `&a[i]`.

– Both represent a pointer to element `i` of `a`.

❑ Also, `*(a+i)` is equivalent to `a[i]`.

– Both represent element `i` itself.

Using an Array Name as a Pointer

❑ The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.

❑ Original loop:

```
for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
```

❑ Simplified version:

```
for (p = a; p < a + N; p++)
    sum += *p;
```

Using an Array Name as a Pointer

❑ Although an array name can be used as a pointer, it's not possible to assign it a new value.

❑ Attempting to make it point elsewhere is an error:

```
while (*a != 0)
  a++;                  /*** WRONG ***/
```

❑ This is no great loss; we can always copy `a` into a pointer variable, then change the pointer variable:

```
p = a;
while (*p != 0)
  p++;
```

Program: Reversing a Series of Numbers

❑ The `reverse.c` program of Chapter 8 reads 10 numbers, then writes the numbers in reverse order.

❑ The original program stores the numbers in an array, with subscripting used to access elements of the array.

❑ `reverse3.c` is a new version of the program in which subscripting has been replaced with pointer arithmetic.

reverse3.c

```c
/* Reverses a series of numbers (pointer version) */

#include <stdio.h>

#define N 10

int main(void)
{
  int a[N], *p;

  printf("Enter %d numbers: ", N);
  for (p = a; p < a + N; p++)
    scanf("%d", p);

  printf("In reverse order:");
  for (p = a + N - 1; p >= a; p--)
    printf(" %d", *p);
  printf("\n");

  return 0;
}
```

Another example about the pointer and address

```c
// pnt_add.c -- pointer addition
#include <stdio.h>
#define SIZE 4
int main(void)
{
    short dates [SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;

    pti = dates;    // assign address of array to pointer
    ptf = bills;
    printf("%23s %15s\n", "short", "double");
    for (index = 0; index < SIZE; index ++)
        printf("pointers + %d: %10p %10p\n",
               index, pti + index, ptf + index);

    return 0;
}
```

Here is sample output:

```
                          short          double
pointers + 0: 0x7fff5fbff8dc 0x7fff5fbff8a0
pointers + 1: 0x7fff5fbff8de 0x7fff5fbff8a8
pointers + 2: 0x7fff5fbff8e0 0x7fff5fbff8b0
pointers + 3: 0x7fff5fbff8e2 0x7fff5fbff8b8
```

pointer addition increase by 2
since `pti` is type `int`

pti   pti + 1   pti + 2   pti + 3
▼   ▼   ▼   ▼

56014   56015 56016   56017   56018   56019 56020   56021 —— machine address

dates[0]   dates[1]   dates[2]   dates[3] —— array elements
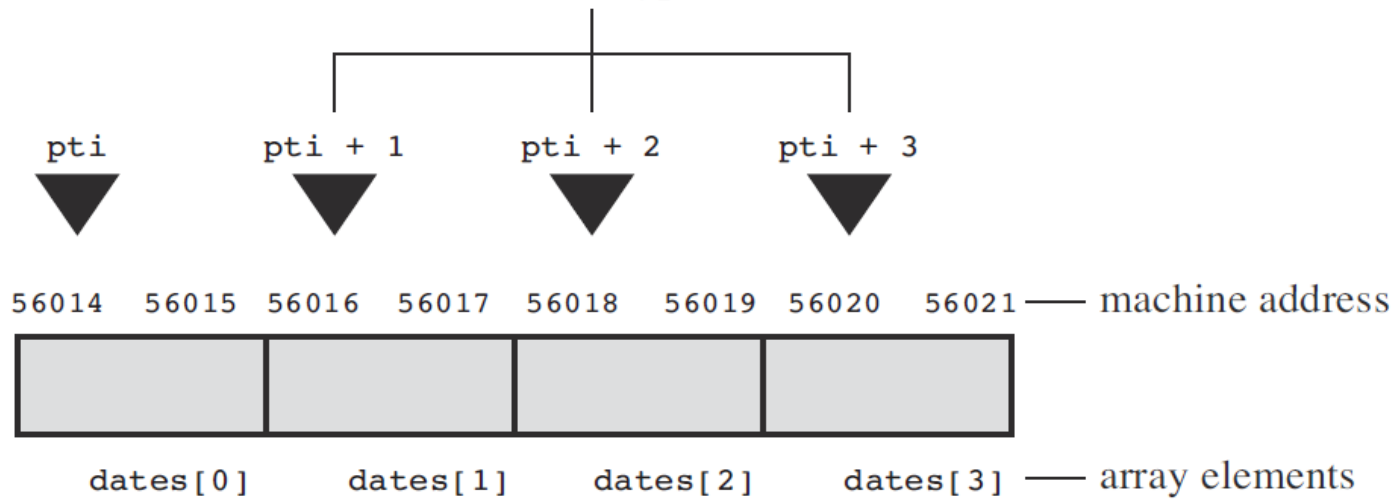
```
int dates[y], *pti;
pti = dates; (or pti = & dates[0];)
```
▲

pointer variable `pti` is assigned the
address of the first element of the array `dates`

As a result of C's cleverness, we have the following equalities:

```
dates + 2 == &date[2]        // same address
*(dates + 2) == dates[2]     // same value
```

All about pointers

Array Arguments (Revisited)

❑ When passed to a function, an array name is treated as a pointer.

❑ Example:

```
int find_largest(int a[], int n)
{
  int i, max;

  max = a[0];
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
```

❑ A call of `find_largest`:

```
largest = find_largest(b, N);
```

This call causes a pointer to the first element of b to be assigned to a; the array itself isn't copied.

Consequences of Array Arguments

❑ *Consequence 1:* When an ordinary variable is passed to a function, its value is copied; any changes to the corresponding parameter don't affect the variable.

— In contrast, an array used as an argument isn't protected against change.

❑ *Consequence 2:* The time required to pass an array to a function doesn't depend on the size of the array.

— There's no penalty for passing a large array, since no copy of the array is made.

❑ *Consequence 3:* An array parameter can be declared as a pointer if desired.

Array Arguments (Revisited)

❑ For example, the following function modifies an array by storing zero into each of its elements:

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

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

Array Arguments (Revisited)

❑ To indicate that an array parameter won't be changed, we can include the word `const` in its declaration:

```
int find_largest(const int a[], int n)
{
    …
}
```

❑ If `const` is present, the compiler will check that no assignment to an element of `a` appears in the body of `find_largest`.

Array Arguments (Revisited)

❑ *Consequence 3:* An array parameter can be declared as a pointer if desired.

❑ `find_largest` could be defined as follows:

```
int find_largest(int *a, int n)

{

   …

}
```

❑ Declaring `a` to be a pointer is equivalent to declaring it to be an array

Array Arguments (Revisited)

❑ Although declaring a *parameter* to be an array is the same as declaring it to be a pointer, the same isn't true for a *variable*.

❑ The following declaration causes the compiler to set aside space for 10 integers:

```
int a[10];
```

❑ The following declaration causes the compiler to allocate space for a pointer variable:

```
int *a;
```

Array Arguments (Revisited)

❑ The following declaration give a pointer variable:

```
int *a;
```

❑ `a` is not an array; attempting to use it as an array can have disastrous results.

❑ For example, the assignment

```
*a = 0;    /*** WRONG ***/
```

will store 0 where `a` is pointing.

❑ Since we don't know where `a` is pointing, the effect on the program is undefined.

Array Arguments (Revisited)

❑ *Consequence 4:* A function with an array parameter can be passed an array "slice"—a sequence of consecutive elements.

❑ An example that applies `find_largest` to elements 5 through 14 of an array `b`:

```
largest = find_largest(&b[5], 10);
```

Using a Pointer as an Array Name

❑ C allows us to subscript a pointer as though it were an array name:

```
#define N 10

…

int a[N], i, sum = 0, *p = a;

…

for (i = 0; i < N; i++)

   sum += p[i];
```
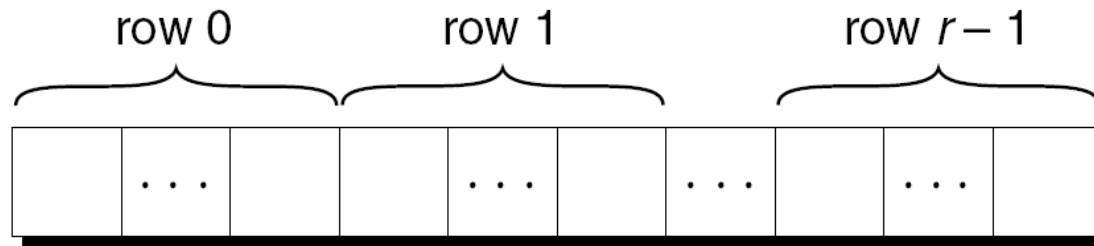
The compiler treats `p[i]` as `*(p+i)`.

Processing the Elements of a Multidimensional Array

❑ C stores two-dimensional arrays in row-major order.

❑ Layout of an array with *r* rows:

❑ If `p` initially points to the element in row 0, column 0, we can visit every element in the array by incrementing `p` repeatedly.

Processing Multidimensional Array

❑ Consider the problem of initializing all elements of the following array to zero:

```
int a[NUM_ROWS][NUM_COLS];
```

❑ The obvious technique would be to use nested `for` loops:

```
int row, col;
…
for (row = 0; row < NUM_ROWS; row++)
  for (col = 0; col < NUM_COLS; col++)
    a[row][col] = 0;
```

❑ If we view `a` as a one-dimensional array of integers, a single loop is sufficient:

```
int *p;
…
for (p = &a[0][0]; p <= &a[NUM_ROWS-1][NUM_COLS-1]; p++)
  *p = 0;
```

Processing Multidimensional Array

❑ A pointer variable `p` can also be used for processing the elements in just one *row* of a two-dimensional array.

❑ To visit the elements of row `i`, we'd initialize `p` to point to element 0 in row `i` in the array `a`:

```
p = &a[i][0];
```

or we could simply write

```
p = a[i];
```

Processing the row of Multidimensional Array

❑ For any two-dimensional array `a`, the expression `a[i]` is a pointer to the first element in row `i`.

```
int a[20][10];
```

❑ Why? `a[i]` is equivalent to `*(a + i)`.

❑ Thus, `&a[i][0]` is the same as `&(*(a[i] + 0))`, which is equivalent to `&*a[i]`.

❑ This is the same as `a[i]`, since the `&` and `*` operators cancel.

Processing the row of Multidimensional Array

❑ A loop that clears row `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], *p, i;

…

for (p = a[i]; p < a[i] + NUM_COLS; p++)
  *p = 0;
```

❑ Since `a[i]` is a pointer to row `i` of the array `a`, we can pass `a[i]` to a function that's expecting a one-dimensional array as its argument.

❑ A function designed to work with one-dimensional arrays will also work with a row belonging to a two-dimensional array.

Processing Row of a Multidimensional Array

❑ Consider `find_largest`, which was originally designed to find the largest element of a one-dimensional array.

❑ We can just as easily use `find_largest` to determine the largest element in row `i` of the two-dimensional array `a`:

```
largest = find_largest(a[i], NUM_COLS);
```

Processing the row of Multidimensional Array

❑ Processing the elements in a *column* of a two-dimensional array isn't as easy, because arrays are stored by row, not by column.

❑ A loop that clears column `i` of the array `a`:

```
int a[NUM_ROWS][NUM_COLS], (*p)[NUM_COLS], i;

…

for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

Using the Name of a Multidimensional Array as a Pointer

❑ The name of *any* array can be used as a pointer, regardless of dimensions, but some care is required.

❑ Example:

```
int a[NUM_ROWS][NUM_COLS];
```

a is *not* a pointer to `a[0][0]`; instead, it's a pointer to `a[0]`.

❑ C regards `a` as a one-dimensional array whose elements are one-dimensional arrays.

❑ When used as a pointer, `a` has type `int (*)[NUM_COLS]` (pointer to an integer array of length `NUM_COLS`).

Using the Name of a Multidimensional Array as a Pointer

❑ `a` points to `a[0]` is useful for simplifying loops that process the elements of a two-dimensional array.

❑ Instead of writing

```
for (p = &a[0]; p < &a[NUM_ROWS]; p++)
  (*p)[i] = 0;
```

to clear column `i` of the array `a`, we can write

```
for (p = a; p < a + NUM_ROWS; p++)
  (*p)[i] = 0;
```

Pointers and Variable-Length Arrays (C99)

❑ Pointers are allowed to point to elements of variable-length arrays (VLAs).

❑ An ordinary pointer variable would be used to point to an element of a one-dimensional VLA:

```
void f(int n)
{
  int a[n], *p;
  p = a;
  …
}
```

Pointers and Variable-Length Arrays (C99)

❑ When the VLA has more than one dimension, the type of the pointer depends on the length of each dimension except for the first.

❑ A two-dimensional example:

```
void f(int m, int n)
{
  int a[m][n], (*p)[n];

  p = a;

  …
}
```

Since the type of `p` depends on `n`, which isn't constant, `p` is said to have a ***variably modified type.***

Pointers and Variable-Length Arrays (C99)

❑ The validity of an assignment such as `p = a` can't always be determined by the compiler.

❑ The following code will compile but is correct only if `m` and `n` are equal:

```
int a[m][n], (*p)[m];

p = a;
```

❑ If `m` is not equal to `n`, any subsequent use of `p` will cause undefined behavior.