Introduction to C Programming (Part B)

Overview (King Ch. 8-12)

- Arrays (Ch. 8)
- Functions (Ch. 9)
- Program Organization (Ch. 10)
- Pointers (Ch. 11)
- Pointer Arithmetic (Ch. 12)

Chapter 8

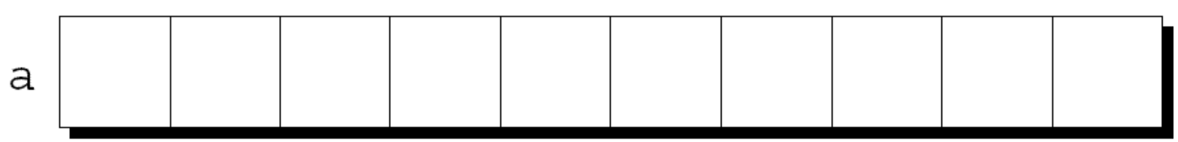
Arrays

Scalar Variables vs Aggregate Variables

- So far, the only variables we've seen are ***scalar:*** capable of holding **a single value**.
- C also supports ***aggregate*** variables, which can store **collections of values**.
- There are two kinds of aggregates in C:
  - **Arrays**
  - **Structures** (later)

One-Dimensional Arrays

- An ***array*** is a data structure containing a number of data values, all of which have the **same type**.
- These values, known as ***elements,*** can be individually selected by their position within the array.
- The simplest kind of array has just one dimension.
- The elements of a one-dimensional array `a` are conceptually arranged one after another in a single row (or column):



One-Dimensional Array Declaration

- To declare an array, we must specify the *type* of the array's elements and the *number* of elements:

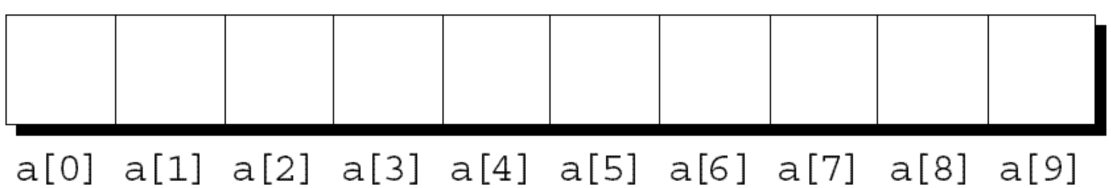
```
int a[10];
```

- Using a macro to define the length of an array is an excellent practice:

```
#define N 10
…
int a[N];
```

Array Subscripting

- To access an array element, write the array name followed by an integer value in square brackets.
- This is referred to as ***subscripting*** or ***indexing*** the array.
- The elements of an array of length *n* are indexed from **0** to ***n* – 1**.
- If `a` is an array of length 10, its elements are designated by `a[0]`, `a[1]`, …, `a[9]`:



Array elements are lvalues

- Expressions of the form `a[i]` are **lvalues**, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

Typical operations on an array

- Many programs contain `for` loops whose job is to perform some operation on every element in an array.
- Examples of typical operations on an array `a` of length `N`:

```
for (i = 0; i < N; i++)
  a[i] = 0;                 /* clears a */
```

```
for (i = 0; i < N; i++)
  scanf("%d", &a[i]);   /* reads data into a */
```

```
for (i = 0; i < N; i++)
  sum += a[i];             /* sums the elements of a */
```

Array subscript bounds

- C doesn't require that subscript bounds be checked; if a subscript goes out of range, the program's **behavior is undefined**.
- A common mistake: forgetting that an array with *n* elements is indexed from 0 to *n* – 1, not 1 to *n*:

```
int a[10], i;

for (i = 1; i <= 10; i++)
  a[i] = 0;
```

Array Initialization

- The most common form of ***array initializer*** is a list of constant expressions enclosed in braces and separated by commas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

- If the initializer is shorter than the array, the remaining elements of the array are given the value 0:

```
int a[10] = {1, 2, 3, 4, 5, 6};
/* initial value of a is {1,2,3,4,5,6,0,0,0,0} */
```

- Using this feature, we can easily initialize an array to all zeros:

```
int a[10] = {0};
/* initial value of a is {0,0,0,0,0,0,0,0,0,0} */
```

- The length of the array may be omitted:

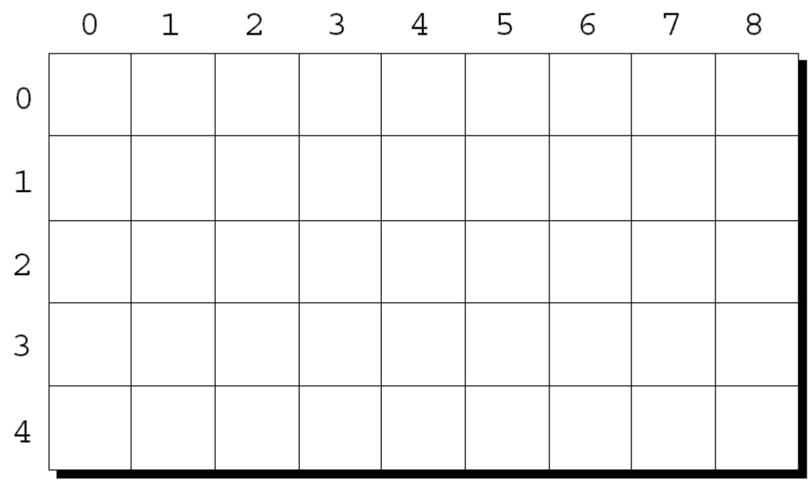
```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Multidimensional Arrays

- An array may have any number of dimensions.
- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

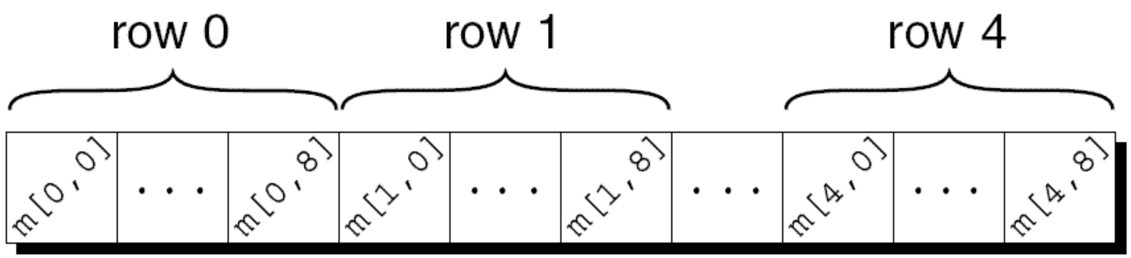
  `int m[5][9];`

- `m` has 5 rows and 9 columns, both indexed from 0
- `m[i][j]` will access the element in row `i`, column `j`



Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Initializing a Multidimensional Array

- We can create an initializer for a two-dimensional array by nesting one-dimensional initializers:

```
int m[5][9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
               {0, 1, 0, 1, 0, 1, 0, 1, 0},
               {0, 1, 0, 1, 1, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 0, 1, 0},
               {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

- Initializers for higher-dimensional arrays are constructed in a similar fashion.
- We can omit the inner braces (risky).
- C provides a variety of ways to abbreviate initializers for multidimensional arrays

Multidimensional Arrays

- Nested `for` loops are ideal for processing multidimensional arrays.
- Consider the problem of initializing an array for use as an identity matrix. A pair of nested `for` loops is perfect:

```
#define N 10

double ident[N][N];
int row, col;

for (row = 0; row < N; row++)
  for (col = 0; col < N; col++)
    if (row == col)
      ident[row][col] = 1.0;
    else
      ident[row][col] = 0.0;
```

Chapter 9

Functions

Introduction

- A function is a series of statements that have been grouped together and given a name.
- Each function is essentially a small program, with its own declarations and statements.
- Advantages of functions:
  - A program can be divided into small pieces that are easier to understand and modify.
  - We can avoid duplicating code that's used more than once.
  - A function that was originally part of one program can be reused in other programs.

Function Definitions

- General form of a ***function definition:***

  return-type function-name （ *parameters* ）
  {
  declarations
  statements
  }

Program: Computing Averages

- A function named `average` that computes the average of two `double` values:

```
double average(double a, double b){
  return (a + b) / 2;
}
```

  - `double` is the ***return type*** of the function.
  - The identifiers `a` and `b` are the function's ***parameters***.

Function Calls

- A function call (inside main or another function) consists of a function name followed by a list of **arguments**, enclosed in parentheses:

```
average(x, y)
print_count(i)
print_pun()
```

Function Declarations

- A ***function declaration*** provides the compiler with a brief glimpse at a function whose full definition will appear later.
- General form of a function declaration:

  return-type function-name `(` *parameters* `)` `;`
- The declaration of a function must be consistent with the function's definition.
- Here's the `average.c` program with a declaration of `average` added.

Function Declarations

```
#include <stdio.h>

double average(double a, double b);    /* DECLARATION */

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}


double average(double a, double b){     /* DEFINITION */
  return (a + b) / 2;
}
```

Arguments

- In C, arguments are ***passed by value:*** when a function is called, each argument is evaluated and its value ***assigned*** to the corresponding ***parameter***.
- Since the **parameter contains a copy of the argument's value**, any changes made to the parameter during the execution of the function don't affect the argument.

Array Arguments

- When a function parameter is a one-dimensional array, the length of the array can be left unspecified:

```
int f(int a[]){  /* no length specified */
   …
}
```

- If the function needs the length of the array we have to supply it as argument or compute it inside the function.

Array Arguments

- Example:

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

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

  return sum;
}
```

- Since `sum_array` needs to know the length of `a`, we must supply it as a second argument.

Array Arguments

- A function that 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;
}

int main(void){
    int b[100];
    store_zeros(b, 100);
}
```

Array Arguments

- The ability to modify the elements of an array argument may seem to **contradict** the fact that C passes arguments by value.
- There's actually no contradiction. What happens?
  - The name of the array serves as a pointer to the first element of the array (see later).
  - The pointer is **passed by value**
  - The array elements are **not copied**: modifying an array element inside a function affects the element value in the original array

Multidimensional Array Arguments

- If a parameter is a multidimensional array, only the length of the first dimension may be omitted.
- If we revise `sum_array` so that `a` is a two-dimensional array, we must specify the number of columns in `a`:

```
#define LEN 10

int sum_two_dimensional_array(int a[][LEN], int n)
{
  int i, j, sum = 0;

  for (i = 0; i < n; i++)
    for (j = 0; j < LEN; j++)
      sum += a[i][j];

  return sum;
}
```

The **`return`** Statement

- The `return` statement has the form

  `return` *expression* `;`

- The expression is often just a constant or variable:

```
return 0;
return status;
```

- `return` may appear in functions who return `void`, provided that no expression is given:

```
return;  /* return in a void function */
```

Program Termination

- The value returned by `main` is a status code that can be tested when the program terminates.
- `main` should return:
  - 0 if the program terminates normally
  - a value other than 0 to indicate abnormal termination

The `exit` Function

- Another way to terminate a program is by calling the `exit` function, which belongs to `<stdlib.h>`.
- To indicate normal termination, we'd pass 0:

```
exit(0);    /* normal termination */
```

- The difference between `return` and `exit`
  - `exit` causes program termination regardless of which function calls it.
  - The `return` statement causes program termination only when it appears in the `main` function.

Recursion

- A function is ***recursive*** if it calls itself.
- The following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

```
int fact(int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fact(n - 1);
}
```

Recursion

- To see how recursion works, let's trace the execution of the statement

  `i = fact(3);`

`fact(3)` finds that 3 is not less than or equal to 1, so it calls

`fact(2)`, which finds that 2 is not less than or equal to 1, so it calls

`fact(1)`, which finds that 1 is less than or equal to 1, so it returns 1, causing

`fact(2)` to return 2 × 1 = 2, causing

`fact(3)` to return 3 × 2 = 6.

Recursion

- We can condense the `power` function by putting a conditional expression in the `return` statement:

```
int power(int x, int n)
{
  return n == 0 ? 1 : x * power(x, n - 1);
}
```

- Both `fact` and `power` are careful to test a "**termination condition**" as soon as they're called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Recursion and Divide-and-Conquer

- Methods exhibit recursive behavior when they can be defined by two properties:
  - A simple **base case**
  - A set of rules that reduce all other cases toward the base case
- Recursion often arises as a result of an algorithm design technique known as ***divide-and-conquer,*** in which a large problem is divided into smaller pieces that are then tackled by the same algorithm.

Chapter 10

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 compound statement that it appears in.

Local Variables

- Since C99 doesn't require variable declarations to come at the beginning of a function, it's possible for a local variable to have a very small scope:

```
void f(void)
{
  …
  int i;    ┐
  …         ├ scope of i
}           ┘
```

- **Parameters** are treated as local variables:
  - automatic storage duration and block scope
  - initialized automatically when a function is called

External Variables

- Passing arguments is one way to transmit information to a function.
- Functions can also communicate through ***external variables***—variables that are declared outside the body of any function (a.k.a. ***global variables***).
- Properties of external variables:
  - Static storage duration – static memory address
  - **File scope** - visible from its point of declaration to the end of the enclosing file.

Scope

- In a C program, the same identifier may have several different meanings.
- C’s scope rules enable the programmer (and the compiler) to determine which meaning 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, the 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.

YORK UNIVERSITÉ UNIVERSITY

Scope Example

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

void f(int i )          /* Declaration 2 */
{
  i = 1;
}

void g(void)
{
  int i = 2;            /* Declaration 3 */

  if (i > 0) {
    int i ;             /* Declaration 4 */

    i = 3;
  }

  i = 4;
}

void h(void)
{
  i = 5;
}
```

Scope

- In the example on the previous slide, the identifier `i` has four different meanings:
  - In Declaration 1, `i` is a variable with static storage duration and file scope.
  - In Declaration 2, `i` is a parameter with block scope.
  - In Declaration 3, `i` is an automatic variable with block scope.
  - In Declaration 4, `i` is also automatic and has block scope.
- C’s scope rules allow us to determine the meaning of `i` each time it’s used (indicated by arrows).

YORK UNIVERSITÉ UNIVERSITY

Organizing a C Program

- There are several ways to organize a program. One possible ordering:
  - `#include` directives
  - `#define` directives
  - Type definitions
  - Declarations of external variables
  - Prototypes/declarations of functions other than `main`
  - Definition of `main`
  - Definitions of other functions

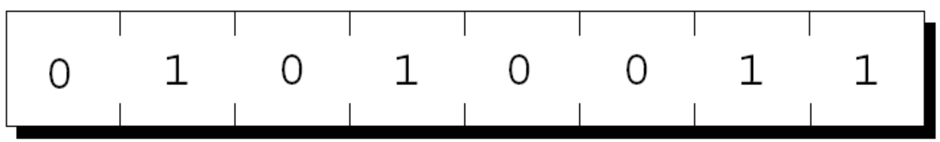
Chapter 11

Pointers

Pointer Variables

- The first step in understanding pointers is visualizing what they represent at the machine level.
- In most modern computers, main memory is divided into ***bytes,*** with each byte capable of storing eight bits of information:

- Each byte



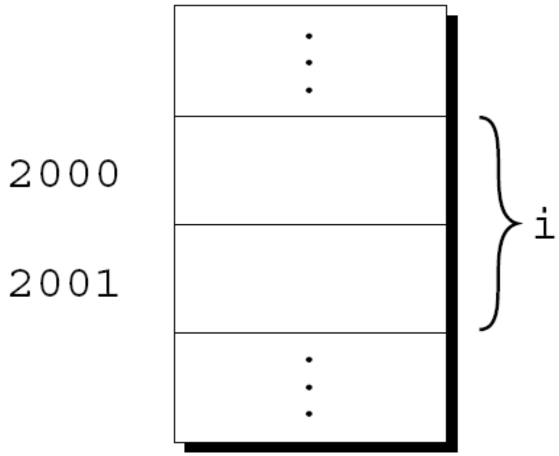
Pointer Variables

- 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

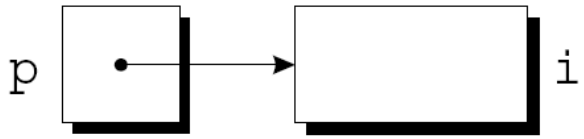
Pointer Variables

- Each variable in a program occupies one or more bytes of memory.
- The address of the first byte is said to be the address of the variable.
- In the following figure, the address of the variable `i` is 2000:



Pointer Variables

- Addresses can be stored in special ***pointer variables.***
- When we store the address of a variable `i` in the pointer variable `p`, we say that `p` "points to" `i`.
- A graphical representation:



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`.
- We use the term *object* instead of *variable* since `p` might point to an area of memory that doesn't belong to a variable.

Declaring Pointer Variables

- 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 */
```

- There are no restrictions on what the referenced type may be.

The Address and Indirection Operators

- C provides a pair of operators designed specifically for use with pointers.
  - To find the address of a variable, we use the **& (address)** operator.
  - To gain access to the object that a pointer points to, we use the *** (*indirection*)** operator.

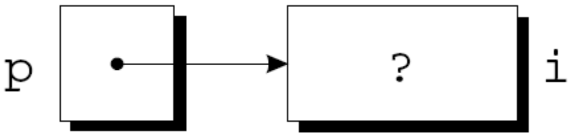
The Address Operator

- Declaring a pointer variable sets aside space for a pointer but doesn't make it point to an object:

```
int *p;  /* points nowhere in particular */
```

- To initialize a pointer variable assign it the address of a variable:

```
int i, *p;
…
p = &i;
```



The Indirection Operator

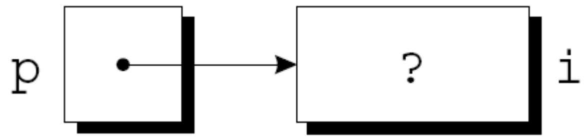
- Once a pointer variable points to an object, we can use the *** (indirection) operator** to access what's stored in the object.
- E.g., we can use the pointer to print the value of `i`:

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

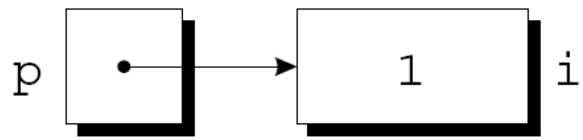
- As long as `p` points to `i`, `*p` is an ***alias*** for `i`.
  - `*p` has the same value as `i`.
  - Changing the value of `*p` changes the value of `i`.

The Indirection Operator

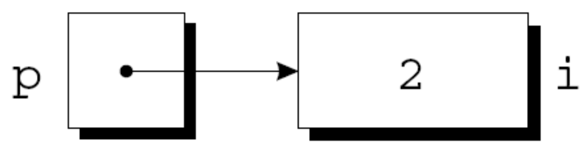
```
p = &i;
```




```
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

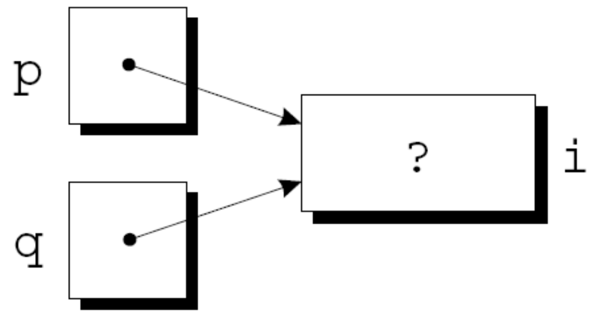
- Assume that the following declaration is in effect:

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

- Example of pointer assignments:

```
p = &i;
q = p;
```

`q` now points to the same place as `p`:

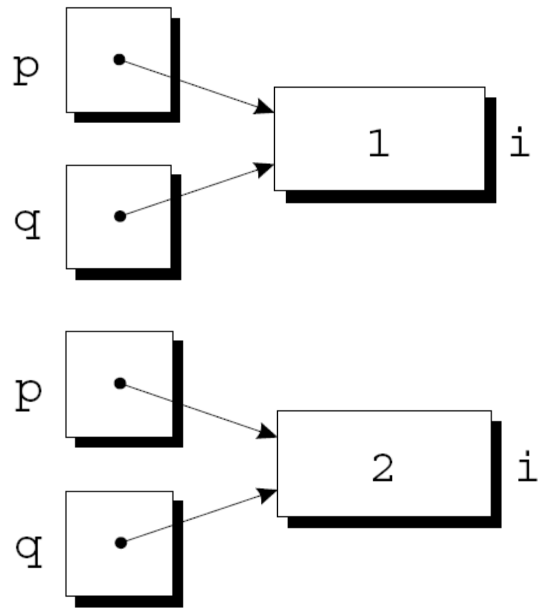


Pointer Assignment

- 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

- Be careful not to confuse

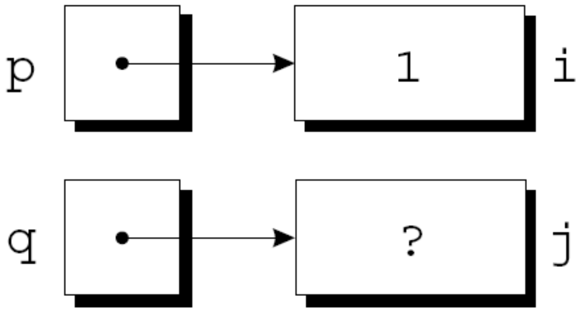
  `q = p;`

  with

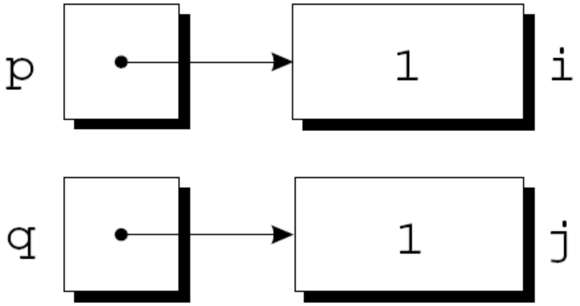
  `*q = *p;`
- The first statement is a pointer assignment, but the second is not.
- The example on the next slide shows the effect of the second statement.

Pointer Assignment

```
p = &i;
q = &j;
i = 1;
```




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



Pointers as Arguments

- How to swap the values of two integers?

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

int main(void){
    int i = 5, j = 7;
    swap(i,j);
    printf("i:%d, j:%d \n", i, j);
    return 0;
}
```

Wrong: arguments are passed by value

Pointers as Arguments

- How to swap the values of two integers?

```
void swap (int *p, int *q){
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main(void){
    int i = 5, j = 7;
    swap(&i,&j);
    printf("i:%d, j:%d \n", i, j);
    return 0;
}
```

Correct

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`.

York Université University

Chapter 12

Pointers and Arrays

Introduction

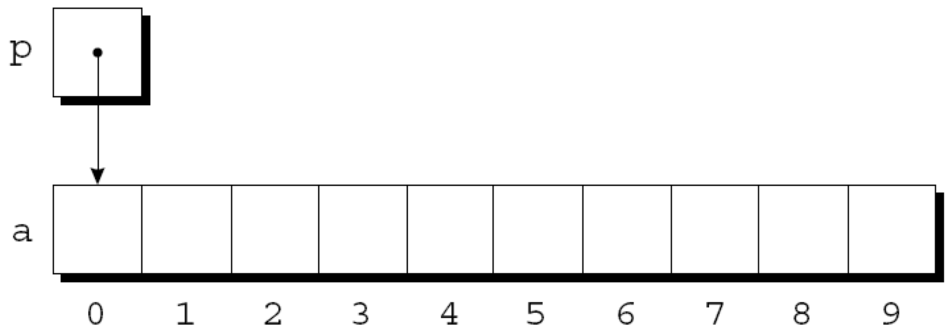
- C allows us to perform arithmetic—addition and subtraction—on pointers to array elements.
- This leads to an alternative way of processing arrays in which pointers take the place of array subscripts.
- The relationship between pointers and arrays in C is a close one.
- Understanding this relationship is critical for mastering C.

Pointer Arithmetic

- We know that pointers can point to array elements:

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

- A graphical representation:

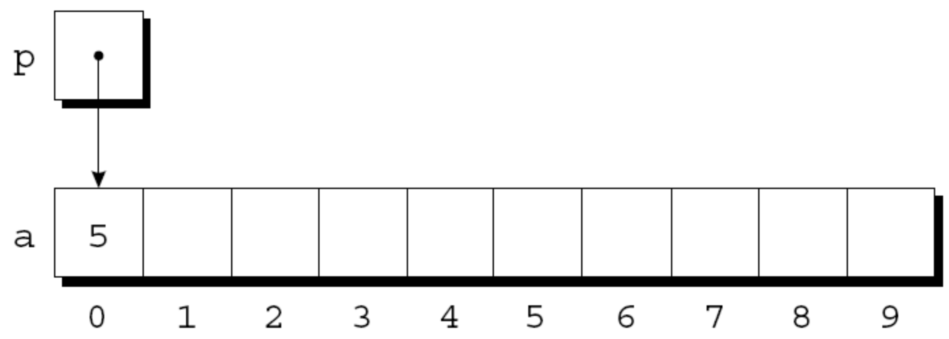


Pointer Arithmetic

- 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 (and only 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

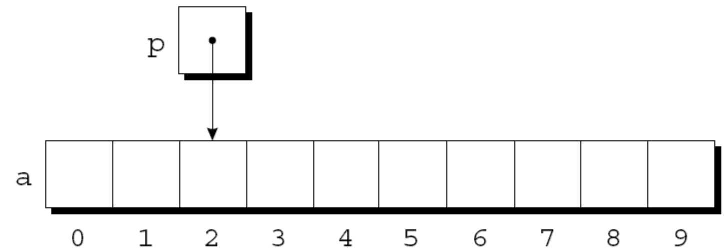
- Adding an integer `j` to a pointer `p` yields a pointer to the element `j` places after the one that `p` points to.
- More precisely, if `p` points to the array element `a[i]`, then `p + j` points to `a[i+j]`.
- Assume that the following declarations are in effect:

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

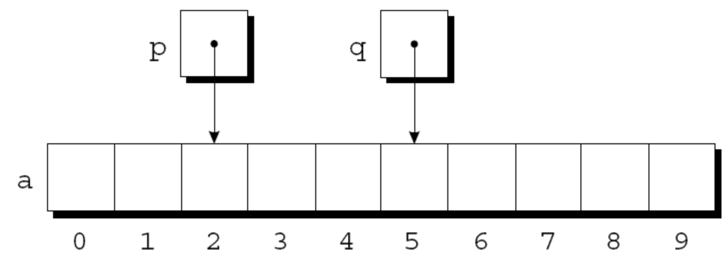
Adding an Integer to a Pointer

- Example of pointer addition:

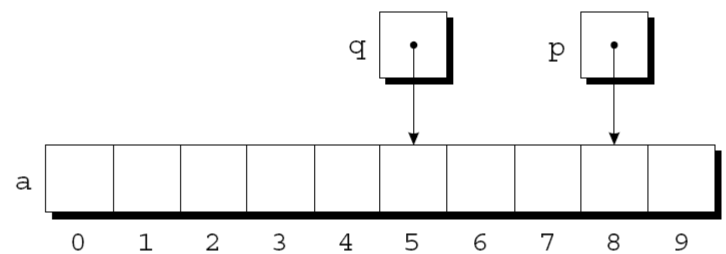
```
p = &a[2];
```




```
q = p + 3;
```




```
p += 6;
```



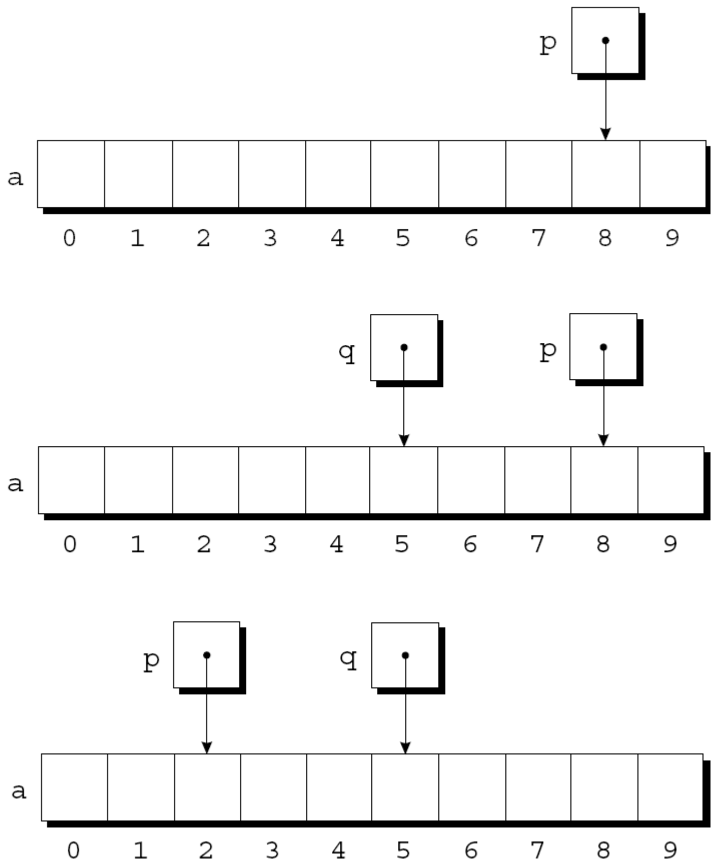
Subtracting an Integer from a Pointer

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

```
p = &a[8];
```

```
q = p - 3;
```

```
p -= 6;
```

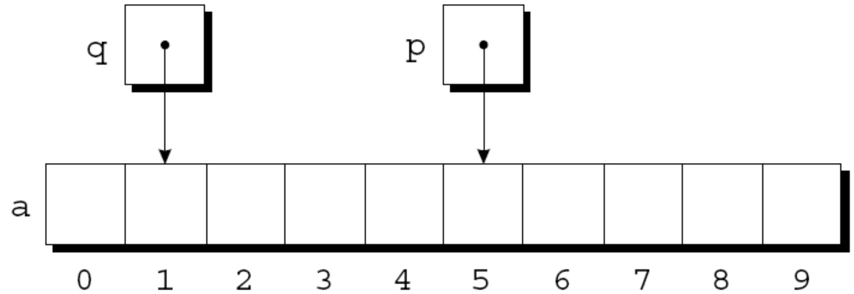


UNIVERSITÉ
UNIVERSITY

Subtracting One Pointer from Another

- When one pointer is subtracted from another, the result is the distance (measured in array elements) between the pointers.
- 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 */
```

Comparing Pointers

- Pointers can be compared using the relational operators (<, <=, >, >=) and the equality operators (== and !=).
  - 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 and the value of `p >= q` is 1.

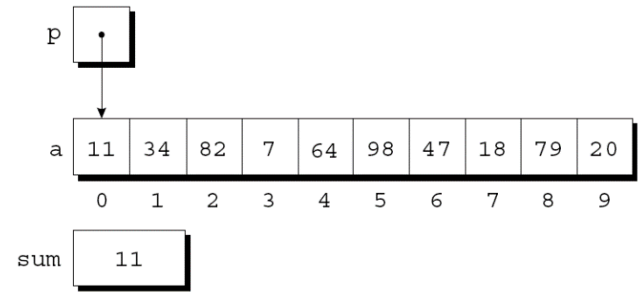
Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array `a`:

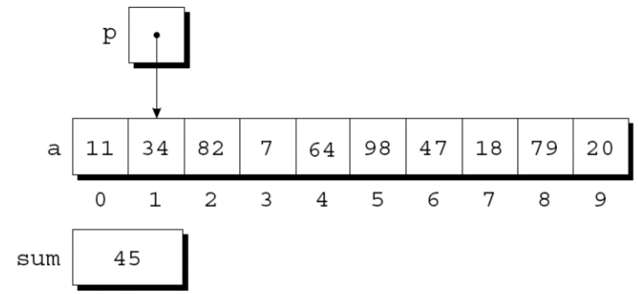
```
#define N 10
…
int a[N], sum, *p;
…
sum = 0;
for (p = &a[0]; p < &a[N]; p++)
   sum += *p;
```

Using Pointers for Array Processing

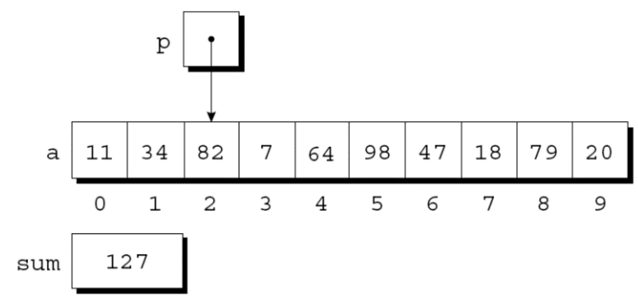
At the end of the first iteration:



At the end of the second iteration:



At the end of the third iteration:



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++;
```

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;
```

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.

Array Arguments (Revisited)

- *Consequence 1:* An array used as an argument isn't protected against change.
  - 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)

- *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.

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; the compiler treats the declarations as though they were identical.

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)

- In the latter case, `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);
```