Knowledge Update Course
for Secondary School Computer and IT Teachers

Computer Programming

Day Two

Jasper Wong
email: icjwong@polyu.edu.hk

Industrial Centre
The Hong Kong Polytechnic University

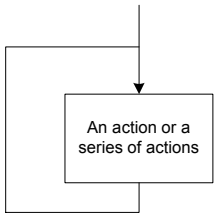
June, 2003

Day Two Agenda

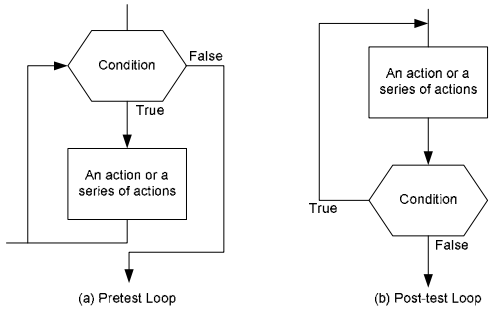
- Repetition: loops and recursion
- Arrays
- Sorting
- Searching
- Pointers
- Derived Types: enumerated, structure and union

Concept of a Loop

- the action is repeated over and over again.
- It never stops if no controls
- Pre-test loop, check before loop starts to execute or terminate the loop
- Post-test loop, code executed once, then check for loop execution or termination



Pretest and Post-Test Loops

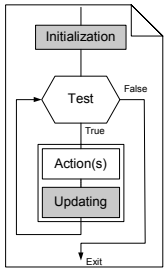


Initialization and Updating

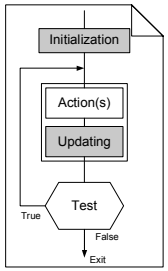
- In addition to the loop control expression, there are two other processes associated with almost all loops:
  - Initialization
  - Updating

Loop Initialization and update

- Initialization before loop execution
- Explicitly initialize loop variables.
- Implicit initialization by preexisting values
- Update control conditions to avoid infinite loop
- Event-controlled loops and counter-controlled loops



(a) Pretest Loop

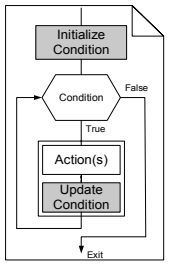


(b) Post-test Loop

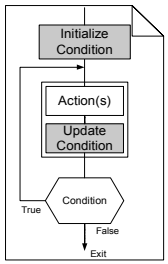
Event-Controlled Loops

- In an event-controlled loop, an event changes the loop control expression from true to false.
- For example, when reading data, reaching the end of the data changes the loop control expression from true to false.
- In event-controlled loops, the updating process can be explicit or implicit.
  - If it is explicit, such as finding a specific piece of information, it is controlled by the loop.
  - If it is implicit, such as the temperature of a batch of chemicals reaching a certain point, it is controlled by some external condition.

Event-Controlled Loops



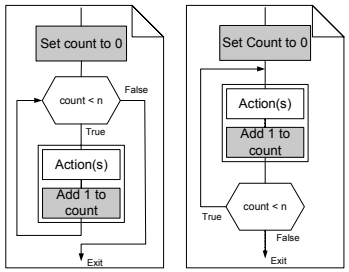
(a) Pretest Loop



(b) Post-test Loop

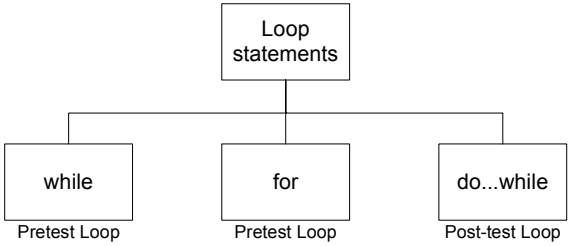
Counter-Controlled Loops

- Number of an action to be repeated is known
- Must initialize, update, and test the counter.
- Increment/Decrement counter update



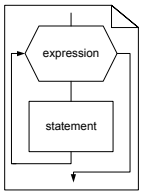
(a) Pretest Loop (b) Post-test Loop

Loop Statements



while Loop

- A pretest loop, expression is tested before every loop iteration
- No semicolon belongs for the while statement.
- The semicolon belongs to the statement



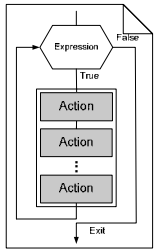
(a) Flowchart

```
while (expression)
statement
```

(b) Sample Code

while Loop

- A compound statement is required for the multiple statements loop



(a) Fowchart

```
while (expression)
{
    Action
    Action
    ⋮
    Action
}
// End of while
```

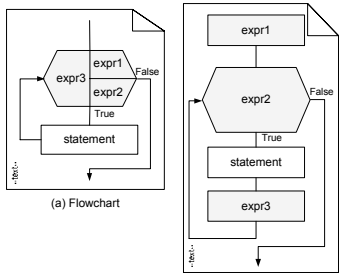
(b) C++ language

for Loops

- The for statement is a pretest loop that uses three expressions.
  - The first contains any initialization statements
  - The second contains the terminating expression
  - The third contains the updating expression

for Loops

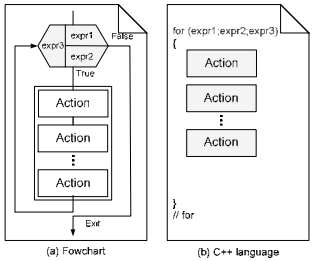
- Expression 1 is executed when the for starts.
- Expression 2 is the limit test expression.
- Expression 3 is the update expression.




```
for (expr1;expr2;expr3)
    statement
```

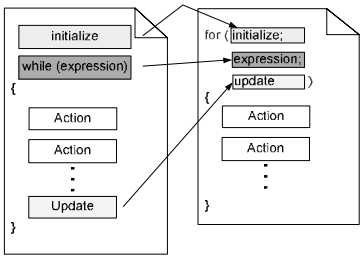

for Loops

- Use compound statement to include multiple statements



for Loops

- for loop contains the initialization, update code, and limit test in one statement.
- Compact in coding



The for Loop

- while loop and for loop for solving the same problem

```
i = 1;
sum = 0;
while (i <= 20)
{
  cin >> a;
  sum += a;
  i ++;
} // while
```

```
sum = 0;
for (i = 1; i <= 20; i++)
{
  cin >> a;
  sum += a;
} // for
```


The for Loop

- Example1: print odd numbers; change for statement

```
for (i = 1; i <= limit; i += 2)
cout << "\t" << i << endl;
```

- Example2: print numbers back; change for statement

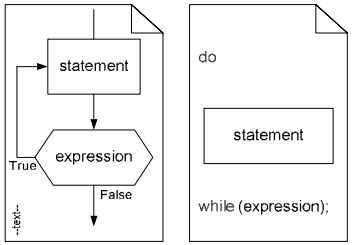
```
for (i = limit; i >= 1; i --)
cout << "\t" << i << endl;
```

- Example3: print numbers in two columns with odd number in the 1st column; change for statement and print statement

```
for (i = 1; i <= limit; i += 2)
cout << setw(2) << i << setw(2) << i+1 << endl;
```


The do … while Loop

- While statements ends with a semicolon

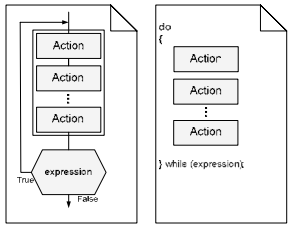


(a) Flowchart (b) Sample Code

The do … while Loop

- While statements ends with a semicolon
- Example

```
do
{
  cout << "Enter a number
between 10 & 20: ";
  cin >> a;
}   while ( a < 10 || a > 20 );
```



(a) Fowchart (b) C++ language

The Comma Expression

- A comma expression is a complex expression made up of two expressions separated by commas.
- It can legally be used in many places, it is generally used only in for statements.
- The expressions are evaluated left to right.
- The value and type of the expression is the value and type of the right expression.
- The other expression is included for its side effect.
- The comma expression has the lowest precedence of all expressions, priority 1.

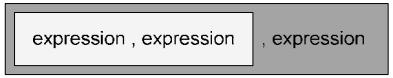
The Comma Expression

- It uses a comma expression to initialize the accumulator, sum, and the index, i, in the loop.
- the value of the comma expression is discarded.
- This is a common use of the comma operator.

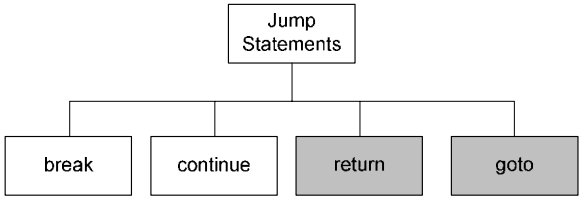
```
for (sum = 0, i = 1; i <= 20; i++)
{
  cin >> a;
  sum += a;
} // for
```


The Comma Expression

- Comma expressions can be nested.
- all expression values other than the last are discarded.
- the value of the expression is the value of the rightmost expression.
- if a comma expression is the second expression in a for loop, make sure that the loop control is the last expression.



Other Statements Related to Looping



break

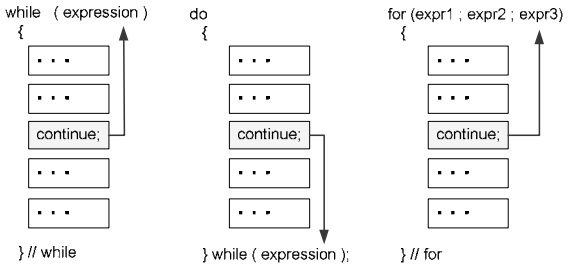
- In a loop, the break statement causes a loop to terminate.
- In a series of nested loops, break terminates only the inner loop.

```
while (condition)
{
  …
  for (…;…;…)
  {
    …
    if (otherConditon)
      break;
    …
  } // for
  // more while processing
  …
} // while
```

The break statement takes you out of the inner loop (the for loop). The loop is still active.

continue

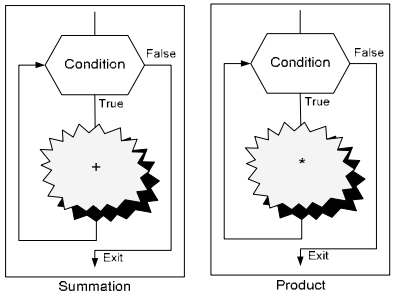
- It does not terminate the loop, but simply transfers to the testing expression



Looping Applications

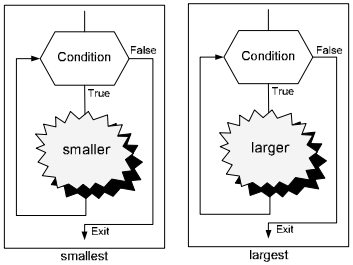
- We examine four common applications for loops:
  - Summation
  - Product
  - Smallest
  - Smallest or largest
  - Inquiries

Summation and Product



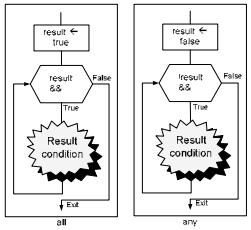
Smallest and Largest

- Determine a smallest/largest number
- Example: result = a < b ? a : b;



Inquires

- Inquiry types:
  - Any – at east one meets a given criteria
  - All – all meet the same specified criteria



Recursion

- Two approaches to write repetitive algorithms.
  - One uses loops
  - The other uses recursion.
- Recursion is a repetitive process in which a function calls itself.
- Either approach can be converted to the other's
- Some older languages do not support recursion, e.g. Cobol

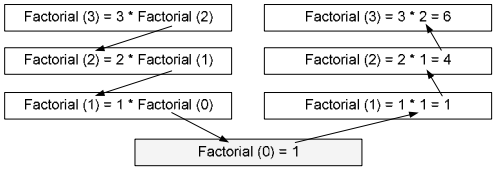
Iterative Definition

- A repetitive function is defined iteratively whenever the definition involves only the parameter and not the function itself.
- Example: factorial (4) = 4 * 3 * 2 * 1 = 24
- A repetitive function is defined recursively whenever the function appears within the definition itself.
- Example:

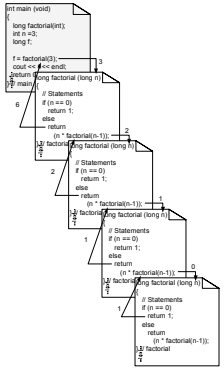
$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n == 0 \\ n * (\text{Factorial}(n-1)) & \text{if } n > 0 \end{cases}$$

Recursive Definition

- The decompostion of factorial (3), the recursive solution for a problem involves a two –way journey.
- Computer solution is much easier



Recursive Definition



- mechanism with the parameters for each individual call.

Designing Recursive Functions

- Every recursive function must have a base case.
- The rest of the function is known as the general case.
- In our factorial example, the base case is factorial (0). The general case is n * factorial (n-1).
- The general case contains the logic needed to reduce the size of the problem.
- Every recursive call must either solve part or the problem or reduce the size of the problem.

Designing Recursive Functions

- In the factorial problem, once the base case has been reached, the solution begins.
- The program has found one part of the answer and can return that part to the next more general statement.
- calculate factorial (0) is 1, that value is returned.
- That leads to solving the next general case, factorial (1) → 1 * factorial (0) → 1 * 1 → 1
- The value of factorial (1) is returned to the more general case, factorial (2), which we know to be factorial (2) → 2 * factorial (1) → 2 * 1 → 2
- Each general case is solved in turn, the next higher general case can be solved until finally the most general case, the original problem is solved.

Designing Recursive Functions

- Rules for designing a recursive function.
  - First determine the base case.
  - Then determine the general case.
  - Combine the base case and general case into a function.
- In combining the base and general cases into a function, you must pay careful attention to the logic.
- Each call must reduce the size of the problem and move it toward the base case.
- The base case, when reached, must terminate without a call to the recursive function, it must execute a return.

Fibonacci Numbers

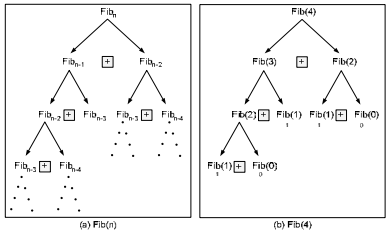
- Let's look at another example of recursion, a function that generates Fibonacci numbers.
- Named after an Italian mathematician, Leonardo Fibonacci, who lived in the early 13th century, Fibonacci numbers are a series in which each number is the sum of the previous two numbers.
- The first few numbers in the Fibonacci series are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Fibonacci Numbers

- Fibonacci numbers are a series in which each number is the sum of the previous two numbers.
- The first few numbers in the Fibonacci series are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
- To start the series, we need to know the first two numbers.
- As you see from the above series, they are 0 and 1.
- We discussing recursion, you should recognize these two numbers as the base cases.
- We can generalize the Fibonacci series as follows:
  Given:
  $\text{Fibonacci}_0 = 0$
  $\text{Fibonacci}_1 = 1$
  then
  $\text{Fibonacci}_n = \text{Fibonacci}_{n-1} + \text{Fibonacci}_{n-2}$

Fibonacci Numbers

- The left half of the figure shows the components of Fibonacci4 using a general notation.
- The right half of the figure shows the components as they would be called to generate the numbers in the series.

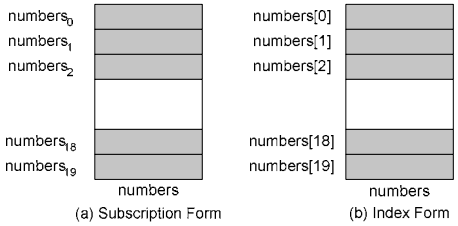


Concepts of Array

- To print 20 integers, we can declare and define 20 variables, each with a different name
- This approach may be acceptable for 20 integers, it is definitely not acceptable for 200 or 2,000 or 20,000 integers.
- To process large amounts of data we need a powerful data structure, such as an array.
- An array is a fixed-size, sequenced collection of elements of the same data type.

Concepts of Array

- The elements of the array are individually addressed through their subscripts, a concept shown graphically.

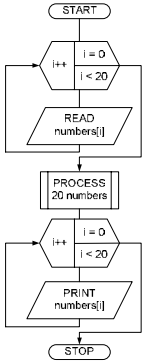


Concepts of Array

- We can use loops to read and write the elements in an array.
  - To add, subtract, multiply, and divide the elements.
  - more complex processing such as calculating averages.
- Now it does not matter if there are 2, 20, 200, 2000, or 20000 elements to be processed, because loops make it easy to handle them all.

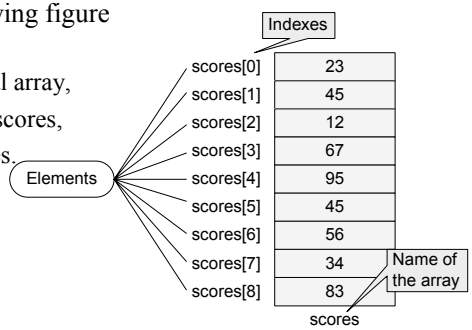
Concepts of Array

- The flowchart showing the loop used to process our 20 numbers using as array is seen in following figure.



Using Arrays in C++

- The following figure shows
  - A typical array,
  - Named scores,
  - Its values.

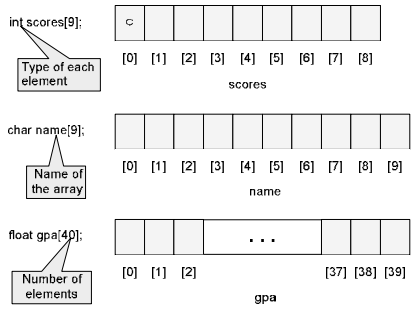


Declaration and Definition

- An array must be declared and defined before it can be used.
- Declaration and definition tell the compiler
  - The name of the array
  - The type of each element

Declaration and Definition

- The following figure shows three different array declarations:
  - Integers
  - Characters
  - Floating-point

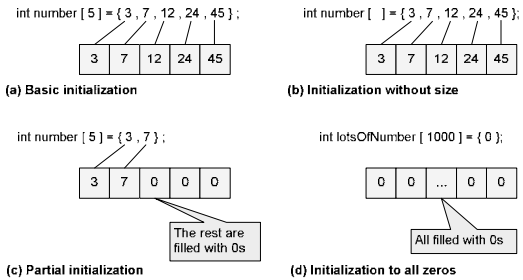


Initialization

- Initialization of all elements in an array can be done at the time of declaration and definition, just as with variables.
- For each element in the array we provide a value.
- The only difference is that the values must be enclosed in braces and, if there are more than one, separated by commas.
- It is a compile error to specify more values than there are elements in the array.

Initialization

- Examples of array initialization.



Assigning Values

- Individual elements can be assigned values using the assignment operator. Scores[4] = 23;
- You cannot assign one array to another array, if they match fully in type and size. You have to copy arrays at the individual element level.
- For example, to copy an array of 25 integers to second array of 25 integers, you could use a loop, as shown below.

```
for (i=0; i<25; i++)
    second [i] = first [i];
```


Assigning Values

- The value of an array follow a pattern, we can use a loop to assign values.
- For example, the following loop assigns a value that is twice the index number to array scores.

```
for (i=0; i<9; i++)
    scores [i] = i * 2;
```

- For another example, the following code assigns the odd numbers 1 through 17 to the elements of scores.

```
for (i=0; i<9; i++)
    scores [i] = (i * 2) +1;
```


Input and Output Values

- Typical way to input value from keyboard with an array

```
for (i=0; i<9; i++)
cin >> scores [i];
```

- Typical way to print the contents of an array

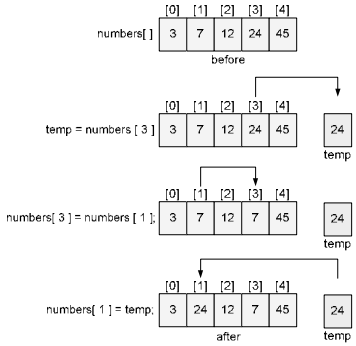
```
for (i=0; i<9; i++)
    cout << scores[i];
cout << endl;
```

Exchanging Values

- To exchange the values of two variables, we use a temporary variable to store the value in numbers[3] before moving the data from number[1]

```
temp          = numbers [3];
numbers [3]   = numbers [1];
numbers [1]   = temp;
```


Exchanging Values



Index Range Checking

- The C++ language does not check the boundary of an array, so it is the programmer's job to ensure that all references to indexed elements are valid and within the range of the array.
- Usually, but not always, your program will continue to run and either produce unpredictable results or eventually abort.
- Two common Index mistakes as follows:

```
for (i=1; i<=9; i++)
cin >> scores[i];
```


Index Range Checking

- The result of this error is that the data stored in memory after the scores array is erroneously destroyed.
- In this error, the first element of the array was not initialized.
- The problems created by unmanaged indexes are among the most difficult to solve, even with today's powerful programming workbenches.
- So you want to plan your array logic carefully and fully test it.

Arrays and Functions

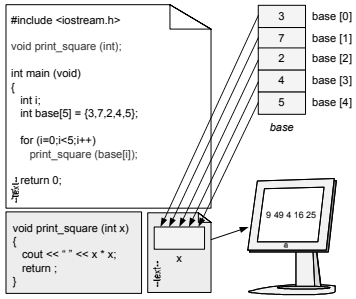
- To process array in a large program, you have to be able to pass them to functions.
- You can do this either by
  - passing individual elements
  - passing the whole array

Passing Individual Element

- Individual elements can be passed to a function like any ordinary variable.
- As long as the array element type matches the function parameter type, it can be passed.
- It will be passed as a value parameter, which means that the function cannot change the value of the element in the calling function.

Passing Individual Element

- We have a function, print_square, that receives an integer and print its square on the system console.
- Using the array, base, we can loop through the array, passing each element in turn to print_square.



Passing the Whole Array

- If we want the function to operate on the whole array, we must pass the whole array.
- A function would use a lot of memory and time to pass large arrays around every time we wanted to use one in a function.
- So, instead of passing the whole array, C++ passes the address of the array.

Passing the Whole Array

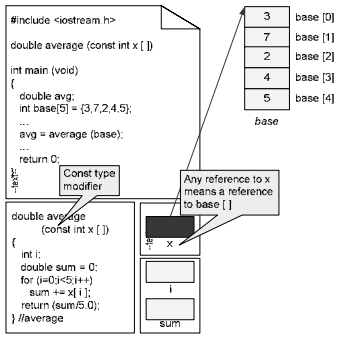
- The name of an array is a primary expression whose value is the address of the first element in the array.
- Since indexed references are simply calculated, all we need to refer to any of the elements in the array is the address of the array.
- Because the name of the array is in fact its address, passing the array name, as opposed to a single element, allows the called function to refer to the array back in the calling function.

Passing the Whole Array

- To pass a whole array in the calling function,
  - use the array name as the actual parameter.
  - declare the corresponding formal parameter is array.
  - no need to specify the number of elements in the array.
- Two rules associated with passing a whole array:
  - The function is called by passing the array name.
  - In the function definition, the formal parameter must be an array type; the size of the array does not need to be specified.

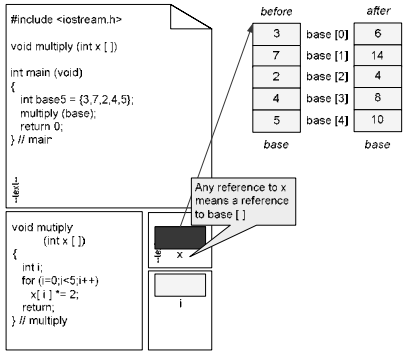
Passing Arrays as Constants

- Passing array name base and return the average of the inteers in the array
- The array is renamed as x
- Does not modify the original array - average



Passing Arrays for Updating

- Change the value of an array element by passing the array name without the constant modifier



Passing Arrays for Updating

- In this example, because we do not use the constant type modifier, we are telling the compiler that the array can be changed.
- Note that in addition to the array parameter, there is a local variable in the called function that is used to "walk" through the array.

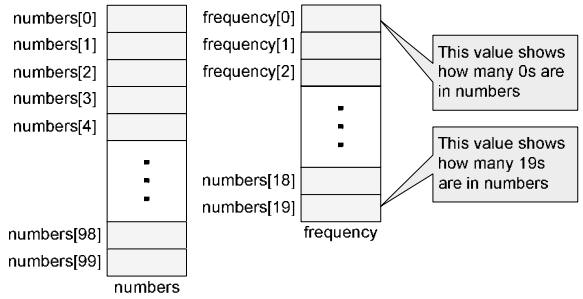
Two Common Array Applications

- Two common statistical applications that use arrays are
  - Frequency distributions
    - A frequency array shows the number of elements with an identical value found in a series of numbers.
  - Histograms.
    - A histogram is a pictorial representation of a frequency array.

Frequency Array

- Suppose we have taken a sample of 100 values between 0 and 19.
- We want to know
  - How many of the values are 0,
  - How many are 1,
  - How many are 2, and so forth up through 19.
- We can read these numbers into an array called numbers. We then create an array of 20 elements that will show the frequency of each number in the series.

Frequency Array

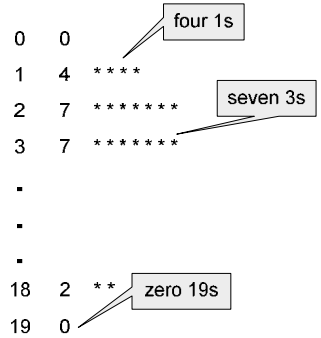


Frequency Array

- Use a for loop to examine each value in the array of 100 elements
- Assign the value from the data array to an index and then use the index to access the frequency array.
- The code for this technique is
  f = number [i];
  frequency [f]++;
- use the value from the array as the index for the array.
- the value of numbers[i] is determined first, and that value is used to index into frequency. frequency [numbers [i]]++;

Histograms

- Instead of printing the values of the elements to show the frequency of each number, we can print a histogram in the form of a bar chart.



Sorting

- Sorting is a process through which data are arranged according to their values.
- If the data were not ordered, it would take us hours and hours to find a single piece of information.
- Two sorting algorithms will be discussed:
  - Bubble sort
  - Insertion sort

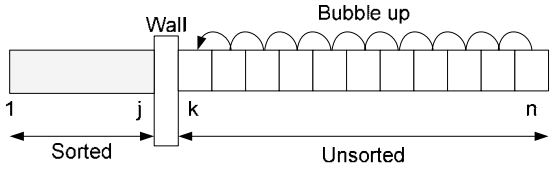
Bubble Sort

- In the bubble sort method, the list is divided into two sublists: sorted and unsorted.

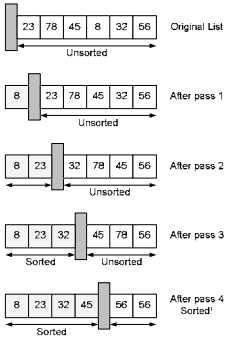
Bubble Sort

- The list is divided into sorted and unsorted list.
- The smallest element is bubbled from the unsorted sublist and moved to the sorted sublist.
- After the smallest element has been moved to the sorted list, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Given list of n elements, the bubble sort requires up to n-1 passes to sort the data
- Each time an element moves from the unsorted sublist to the sorted sublist, one sort pass is completed

Bubble Sort



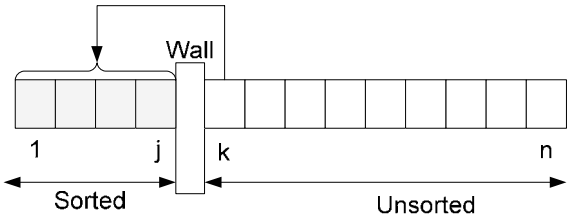
Bubble Sort



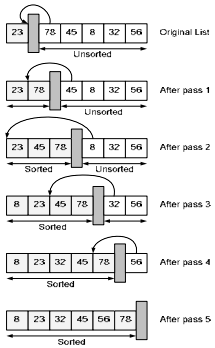
Insertion Sort

- In the insertion sort, the list is divided into sorted and unsorted lists.
- In each pass, the first element of the unsorted sublist is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of n elements will take at most n-1 passes to sort the data

Insertion Sort



Insertion Sort

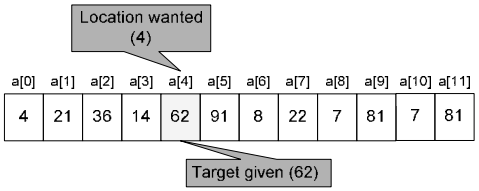


Sort Conclusions

- Bubble sort
  - is implemented in production systems.
  - is the foundation for Quicksort and Quickersort.
- Insertion sort
  - is used as a subfunction in both Quicksort and Singleton's variation,
  - is the foundation of a sorting method called Shell Sort.

Searching

- Searching is a process used to find the location of a target among a list of objects.
- In the case of an array, searching is to find the location (index) of the first element in the array.



Searching

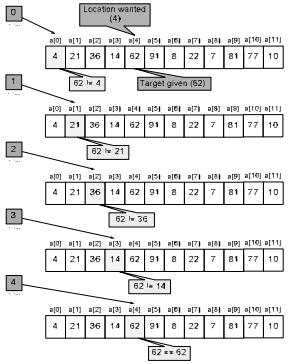
- The algorithm used to search a list depends to large extent on the structure of the list. Since our structure is currently limited to arrays, we will study searches that work with arrays.
- There are two basic searches for arrays, the sequential search and the binary search.
  - The sequential search can be used to locate an item in any array
  - The binary search requires the list to be sorted.

Sequential Search

- is used in an un-ordered list.
- Is used in a small list or list that is frequently searched.
- start searching for the target from the beginning of the list, and continue until the target is found or not in the list

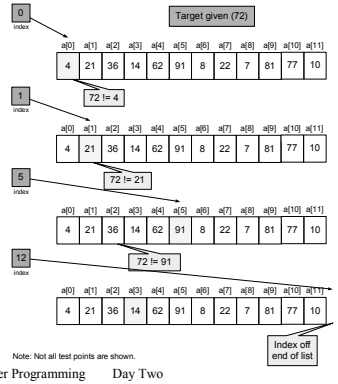
Sequential Search

- to find the value 62.
- first check the data at index 0, and, then at 1, 2, and 3 before finding the 62 in the fifth element (index 4).



Sequential Search

- The following figure traces the search for a target or 72 to show an example of how this works.



Note: Not all test points are shown.

Binary Search

- The sequential search algorithm is very slow. If we have an array of one million elements, we must do one million comparisons in the worst case.
- If the array is not sorted, this is the only solution.
- If the array is sorted, we can use a more efficient algorithm call the binary search.
- A binary search is used when a list starts to become large or more 16 elements.

Binary Search

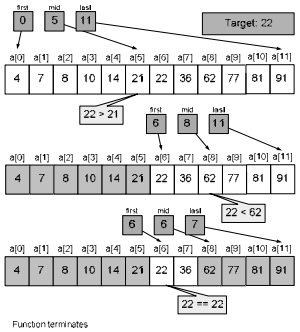
- The binary search starts by testing the data in the element at the middle of the array. This determines if the target is in the first half or the second half of the list.
  - If it is in the first half, there is no need to check the second half any more.
  - If it is in the second half, there is no need to test the first half any more.
- In other words, we eliminate half the list from further consideration.
- We repeat this process until we find the target or satisfy ourselves that it is not in the list.

Binary Search

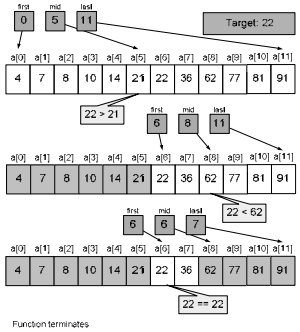
- To find the middle of the list, we need three variables:
  - one to identify the beginning of the list.
  - one to identify the middle of the list.
  - one to identify the end of the list.
- We will analyze two cases:
  - (1) the target is in the list and
  - (2) the target is not in the list.

Target Found

- Search 22 in a sorted array.
- 3 indexes first, mid, and last.
- Given first as 0 and last as 11, we can calculate mid as follow:
  mid = (first + last) / 2;

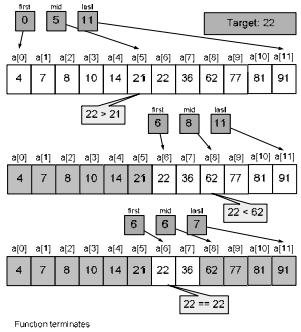


Target Found



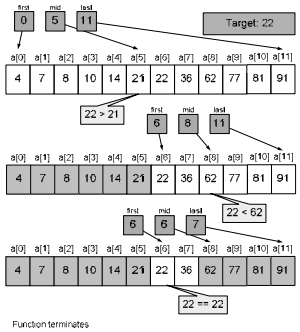
- Since the index mid is an integer, the result will be the integral value of the quotient.
- It truncates rather than rounds the calculation.
- Given the data in following figure, mid becomes 5 as a result of the first calculation.
  Mid = (0 + 11) / 2
  = 11 / 2 = 5

Target Found



- At index location 5, the target is greater than the list value (22>21).
- eliminate the array locations 0 through 5.
- assign mid + 1 to first and repeat the search.
- The next loop calculates mid with the new value for first and determines that the midpoint is now 8.
  mid = (6 + 11) / 2 = 17 / 2
  = 8

Target Found



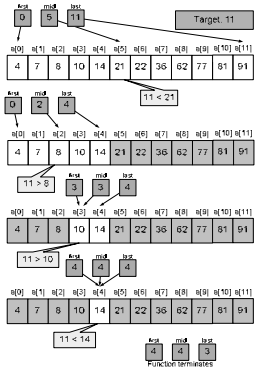
- Again we test the target to the value at mid, and this time we discover that the target is less than the list value (22<62).
- This time we adjust the ends of the list by setting last to mid – 1 and recalculate mid.
- This effectively eliminates elements 8 through 11 from consideration.
- We have now arrived at index location 6, whose value matches our target.
- This stops the search.

Target Not Found

- If the target is not in the list, the search process should stop when all possible locations are checked.
- This is done by testing for first and last crossing.
- Two conditions terminate the binary search algorithm: the target is found or first becomes larger than last.

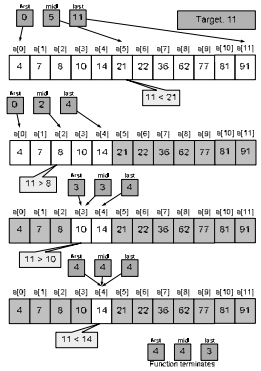
Target Not Found

- Imagine we want to find 11 in our binary search array.



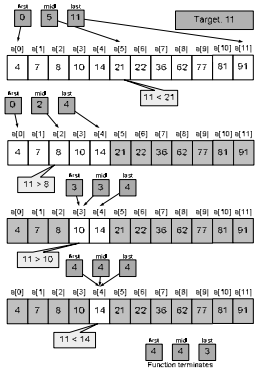
Target Not Found

- the loop continues to search until index locations 3 and 4.
- first and last set the mid index to 3.
  mid = (3 + 4) / 2 = 7 / 2 = 3



Target Not Found

- The test at index location 3 indicates that the target is greater than the list value, so we set first to mid + 1 or 4.
- We test the data at location 4 is 11 < 14.
  mid = (4 + 4) / 2 = 8 / 2 = 4

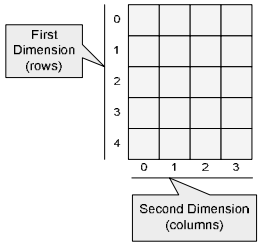


Target Not Found

- At this point, the target should be between two adjacent values; in the other words, it is not in the list.
- We see this algorithmically because last is set to mid – 1, which makes first greater than last, the signal that the value we are looking for is not in the list.

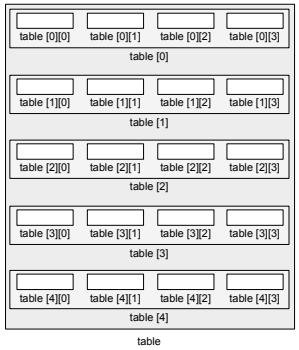
Two-Dimensional Arrays

- In one-dimensional arrays, data are organized linearly in only one direction.
- Many applications require that data be stored in more than one dimension. One common example is a table, which is an array that consists of rows and columns.



Two-Dimensional Arrays

- two-dimensional array is an array of arrays.

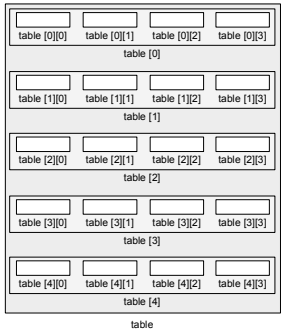


Declaring and Defining \ Two-Dimensional Arrays

- two-dimensional arrays must be declared and defined before being used
- Declaration and definition tell the compiler
  - The name of the array
  - The type of each element
  - The size of each dimension.

Declaring and Defining Two-Dimensional Arrays

- the array shown in the figure can be declared and defined as int table [5] [4];
- By convention,
  - The first dimension specifies the number of rows in the array.
  - The second dimension specifies the number of columns in each row.



Initialization

- Declaration and definition only reserve memory for the elements in the array.
- No values will be stored.
- All arrays should be initialized else the contents are unpredictable.
- Initialization of the array elements can be done when the array is defined.

```
int table[5][4] =  {0,1,2,3,10,11,12,13,20,21,22,
                    23,30,31,32,33,40,41,42,43};
```


Initialization

- A better way:

```
int table [5][4] =
    {
     {  0,  1,  2,  3 },
     { 10, 11, 12, 13 },
     { 20, 21, 22, 23 },
     { 30, 31, 32, 33 },
     { 40, 41, 42, 43 }
    };
```


Initialization

- In this example, we define each row as a one-dimensional
- Array of four elements enclosed in braces. The array of five rows also has its set of braces.
- There are commas between the elements in the rows and also commas between the rows.

Initialization

- In out discussion of one-dimensional arrays, we said that if the array is completely initialized with supplied values, it is not necessary to specify the size of the array.
- This concept carries forward to multidimensional arrays, except that only the first dimension can be omitted.
- All others must be specified.

Initialization

- The format is shown below:

```
int table [5][4] =
    {
     {  0,  1,  2,  3  },
     { 10, 11, 12, 13 },
     { 20, 21, 22, 23 },
     { 30, 31, 32, 33 },
     { 40, 41, 42, 43 }
    };
```

- To initialize the whole array to zeros, we specify only the first values, as shown below:
  int table [5][4] = {0};

Inputting Values

- For a two-dimensional array this usually requires nested for loops.
- If the array is an n by m array, the first loop varies the row from 0 to n-1.
- The second loop varies the column from 0 to m-1.

Inputting Values

- The code fill the array , is shown below:

```
for (row = 0; row < 5; row++)
    for (column = 0; column < 4; column++)
        cin >> table[row][column]);
```

- When the program runs, we enter the 20 values for the elements, and they are stored in the appropriate locations.

Outputting Values

- We can also print the value of the elements one by one using two nested loops.
- Again, the first loop controls the printing of the rows, and the second loop controls the printing of the columns.
- To print the table in its table format, a new line is printed at the end of each row.

Outputting Values

- The code to print, is shown below:

```
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 4; column++)
        cout << setw(8) << table[row][column]);
    cout << endl;
}
```

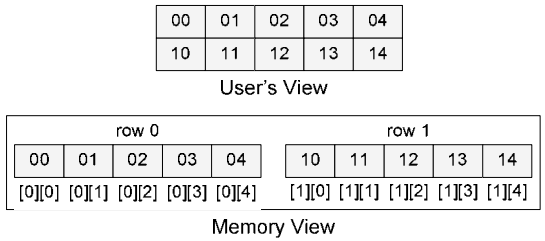

Accessing Values

- Individual elements can be initialized using the assignment operator.
  table [2] [0] = 23;
  table [0] [1] = table [3] [2] + 15;

Memory Layout

- The indexes in the definition of a two-dimensional array represent rows and columns.
- This format maps to the way the data are laid out in memory.
- If we were to consider memory as a row of bytes with the lowest address on the left and the highest address on the right, then an array would be placed in memory with the first element to the left and the last element to the right.
- Similarly, if the array is a two-dimensional array, then the first dimension is a row of elements that are stored to the left.

Memory Layout



Passing a Two-Dimensional Array to a Function

- With two-dimensional arrays, there are three choices for passing parts of the array to a function.

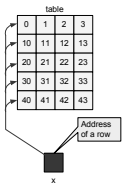
  1. Pass individual elements.
  2. Pass a row of the array.
  3. Pass the whole array.

Passing a Row

- The second case, passing a row of the array.
- We pass a whole row by indexing the array name with only the row number.

```
const int MAX_ROWS = 5;
const int MAX_COLS = 4;
void print_sqr (const int [ ]);
int main(void)
{
   int row;
   int table [MAX_ROWS] [MAX_COLS] =
            {
            {0,1,2,3},
            {10,11,12,13},
            {20,21,22,23},
            {30,31,32,33},
            {40,41,42,43}
            };
      ...
   for (row = 0; row < MAX_ROWS; row++)
      print_sqr (table [row]);
      ...
   return 0;
} // main
```

```
void print_sqr (const int x [ ])
{
   int col;
   for (col = 0; col < MAX_COLS; col++)
      cout << setw(6) << x [col] * x [col];
   cout << endl;
   return;
} // print_sqr
```

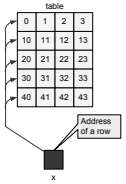


Passing a Row

- When we pass the row, the receiving function receives a on-dimensional array of four integers.
- The for loop in print_sqr prints the square of each of the four elements.

```
const int MAX_ROWS = 5;
const int MAX_COLS = 4;
void print_sqr (const int [ ]);
int main(void)
{
   int row;
   int table [MAX_ROWS] [MAX_COLS] =
            {
            {0,1,2,3},
            {10,11,12,13},
            {20,21,22,23},
            {30,31,32,33},
            {40,41,42,43}
            };
      ...
   for (row = 0; row < MAX_ROWS; row++)
      print_sqr (table [row]);
      ...
   return 0;
} // main
```

```
void print_sqr (const int x [ ])
{
   int col;
   for (col = 0; col < MAX_COLS; col++)
      cout << setw(6) << x [col] * x [col];
   cout << endl;
   return;
} // print_sqr
```

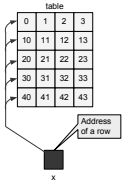


Passing a Row

- After printing all the values, the function advances to the next line on the console and returns.
- The for loop in main calls print_sqr five times so that the final result is a table of the values squared shown on the monitor.

```
const int MAX_ROWS = 5;
const int MAX_COLS = 4;
void print_sqr (const int [ ]);
int main(void)
{
   int row;
   int table [MAX_ROWS] [MAX_COLS] =
            {
            {0,1,2,3},
            {10,11,12,13},
            {20,21,22,23},
            {30,31,32,33},
            {40,41,42,43}
            };
      ...
   for (row = 0; row < MAX_ROWS; row++)
      print_sqr (table [row]);
      ...
   return 0;
} // main
```

```
void print_sqr (const int x [ ])
{
   int col;
   for (col = 0; col < MAX_COLS; col++)
      cout << setw(6) << x [col] * x [col];
   cout << endl;
   return;
} // print_sqr
```

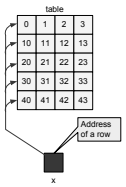


Passing a Row

- We defined the array dimensions as memory constants. This allows us to symbolically refer to the limits of the array in both the array definition and the for loops.
- Now, if we need to change the size of the array, all that is necessary is to change the constant values and recompile the program.

```
const int MAX_ROWS = 5;
const int MAX_COLS = 4;
void print_sqr (const int [ ]);
int main(void)
{
  int row;
  int table [MAX_ROWS] [MAX_COLS] =
          {
          {0,1,2,3},
          {10,11,12,13},
          {20,21,22,23},
          {30,31,32,33},
          {40,41,42,43}
          };
      ...
  for (row = 0; row < MAX_ROWS; row++)
    print_sqr (table [row]);
      ...
  return 0;
} // main
```

```
void print_sqr (const int x [ ])
{
  int col;
  for (col = 0; col < MAX_COLS; col++)
    cout << setw(6) << x [col] * x [col];
  cout << endl;
  return;
} // print_sqr
```



Passing the Whole Array

- When we pass a two-dimensional array to a function, we use the array name as the actual parameter just as we did with one-dimensional arrays.
- The formal parameter in the called function header. The function header must indicate that the array has two dimensions.
- This is done by including two sets of brackets, one for each dimension, as shown below:
  Double average (table [ ] [MAX_COLS])

Passing the Whole Array

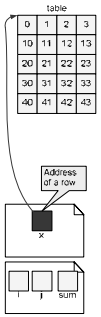
- We do not have to specify the number of rows. It is necessary to specify the size of the second dimension, we specified the number of columns in the second dimension (MAX_COLS).
- In summary, to pass two-dimensional array to functions,
  - The function must be called by passing only the array name.
  - In the function definition, the formal parameter is a two- dimensional array, with the size of the second dimension required.

Passing the Whole Array

- we can use a function to calculate the average of the integers in an array.
- In this case, we pass the name of the array to the function.

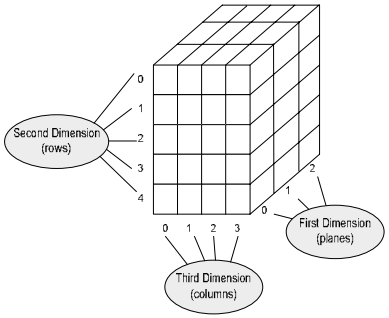
```
const int MAX_ROWS = 5;
const int MAX_COLS = 4;
double average (int [ ] [MAX_COLS]);
int main(void)
{
  int avg;
  int table [MAX_ROWS] [MAX_COLS] =
          {
          {0,1,2,3},
          {10,11,12,13},
          {20,21,22,23},
          {30,31,32,33},
          {40,41,42,43}
          };
      ...
  avg = average(table);
      ...
  return 0;
} // main
```

```
double average (int x [ ] [MAX_COLS]);
{
  int i;
  int j;
  double sum = 0;
  for (i = 0; i < MAX_ROWS; i++)
    for (j = 0; j < MAX_COLS; j++)
      sum += x [i] [j];
  return (sum / (MAX_ROWS * MAX_COLS));
} // average
```

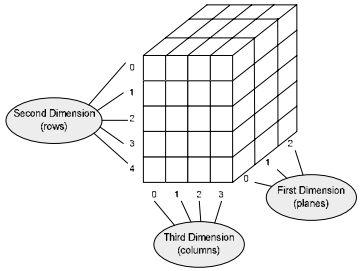


Multidimensional Arrays

- Multidimensional arrays can have three, four, or more dimensions.



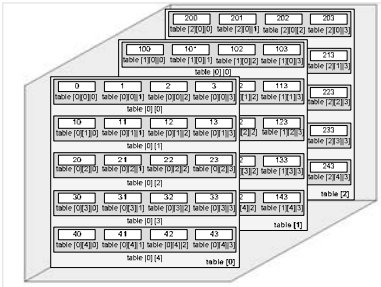
Multidimensional Arrays



Multidimensional Arrays

- C++ takes the three-dimensional array to be an array of two-dimensional arrays, and it consider the two-dimensional array to be an array of one-dimensional arrays.
- In other words a three-dimension array in C++ is an array of array of arrays.
- This concept also holds true for arrays of more than three dimensions.

Multidimensional Arrays

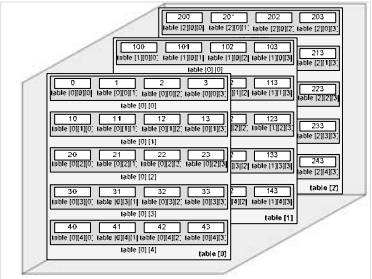


Declaring and Defining Multidimensional Arrays

- Like one-dimensional arrays, multidimensional arrays must be declared and defined before being used.
- Declaration and definition tell the compiler the name of the array, the type of each element, and the size of each dimension.
- The size of the array is a constant and must have a value at compilation time.

Declaring and Defining Multidimensional Arrays

int table [3] [5] [4];



Initialization

- declaration and definition only reserve space for the elements in the array.
- No values will be stored in the array.
- If we want to store values, we must either initialize the elements, read values from the keyboard, or assign values to each individual element.

Initialization

- The initialization of multidimensional arrays relies on a simple extension of the concept used for initializing a two-dimensional array.
- For the three-dimensional array, we nest each plane in a set of braces. For each plane, we bracket the row as we did for the for the two-dimensional array.

Initialization

```
Int table [3][5][4] =
{       { // Plane 0
                { 0,    1,    2,    3 }        //Row 0
                {10,    11,   12,   13 }       //Row 1
                {20,    21,   22,   23 }       //Row 2
                {30,    31,   32,   33 }       //Row 3
                {40,    41,   42,   43 }       //Row 4
        { // Plane 1
                {100 ,  101,  102 ,  103 }     //Row 0
                {110,   111,  112,   113 }     //Row 1
                {120,   121,  122,   123 }     //Row 2
                {130,   131,  132,   133 }     //Row 3
                {140,   141,  142,   143 }     //Row 4
        { // Plane 2
                {200 ,  201,  202,   203 }     //Row 0
                {210,   211,  212,   213 }     //Row 1
                {220,   221,  222,   223 }     //Row 2
                {230,   231,  232,   233 }     //Row 3
                {240,   241,  242,   243 }     //Row 4
} // table
```


Initialization

- As we saw previously, the plane's size, and only the plane's, does not have to be specified when we use explicit initialization.
- The size of all dimensions after the first must be explicitly stated.

Initialization

- If we want to initialize all the elements to 0, we can simply initialize only the first element to 0 and let the compiler generate the code to initialize the rest of the array of 0s.

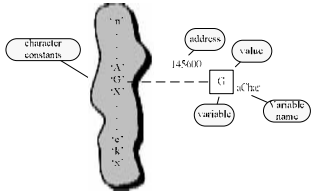
  int table [3] [5] [4] = {0}

Concepts of pointer

- A pointer is a derived data. It is a data type built from one of the standard types.
- Its value is any of the addresses available in the computer for storing and accessing data.
- Pointers are built on the basic concept of pointer constants.

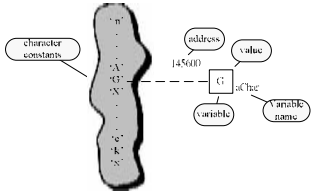
Pointer Constants

- We'll begin our discussion of pointers by comparing character constants and pointer constants.
- We have a character constant, such as any letter of the alphabet, that is drawn from a universe of all characters.



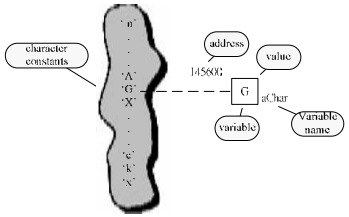
Pointer Constants

- In most computers, this universe is ASCII. A character constant can become a value and be stored in a variable.
- Although the character constant is unnamed, the variable has a name that is declared in the program.



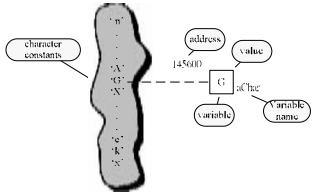
Pointer Constants

- there is a character variable, aChar.
- aChar contains the value 'G' that was drawn from the universe of character constants



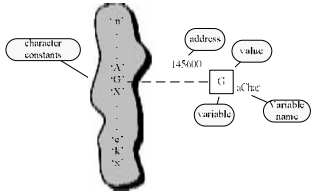
Pointer Constants

- The variable aChar has an address as well as a name.
- The name is created by the programmer; the address is the relative location of the variable with respect to the program's memory space.



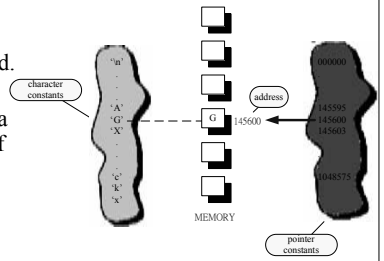
Pointer Constants

- Assume
  - We have a computer that has only one megabyte of memory (2^{20} bytes).
  - The computer has chosen the memory location 145600 as the byte to store this variable.



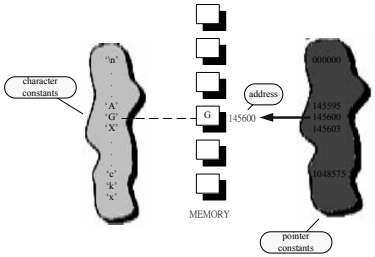
Pointer Constants

- Operating systems can put a program in memory wherever it is convenient when the program is started.
- aChar is stored at memory location 145600, which is a constant for the duration of the run, the next time the program is run it could be located at 876050.



Pointer Constants

- The addresses are constant, we cannot know what they will be, and therefore it is still necessary to refer to them symbolically.



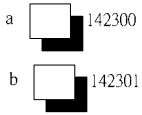
Pointer Values

- If we have a pointer constant, we should be able to save its value if we can identify it.
- The address operator (&) provides a pointer constant to any named location in memory. We nee a pointer value, therefore, all we must use the address operator.
- The address operator used with aChar is shown below: &aChar

Pointer Values

- define two character variables and prints their addresses as pointers.

```
// This program prints character addresses
#include <iostream.h>

int main ( void )
{
// Local Declarations
   char a;
   char b;

// Statements
   cout << &a << &b ;
   return 0;
}  // main
```



Pointer Values

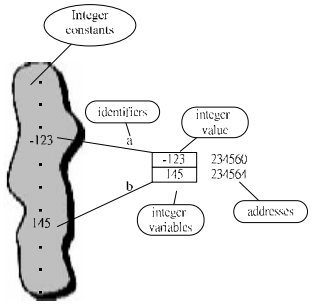
- Depending on the operating system, this program may print different numbers each time you run it.
- The addresses will be different in different computers. Most of the time, the computer allocates two adjacent memory locations because we define the two variables one after the other.

Pointer Values

- The situation changes slightly when integers are involved. In most computers, integers occupy either two or four bytes.
- Let us assume that we are working on a system with four-byte integers, which means that each integer variable occupies four memory locations.

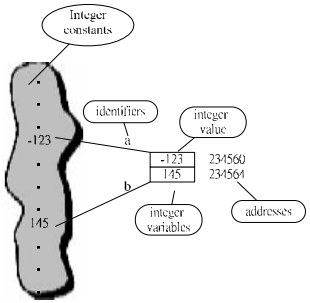
Pointer Values

- Which of these memory locations is used to find the address of the variable?
- In C++ the location of the first byte is used as the memory address.
  - For characters, there is only one byte, so its location is the address;
  - For integers, the address is the first byte of four.



Pointer Values

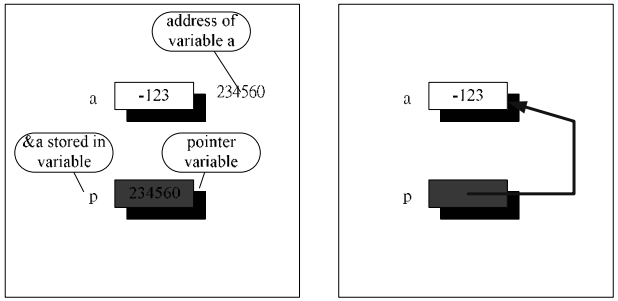
- The same system applies to floating-point and other data types.
- The address of a variable is the address of the first byte occupied by that variable.



Pointer Variables

- If we have pointer constants and pointer values, then we also have pointer variables.
- Thus, we can store the address of a variable into another variable, which is called a pointer variable.
- The concept is illustrated in following figure.

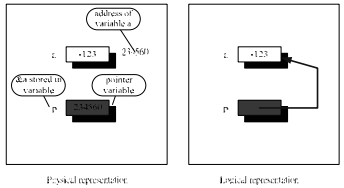
Pointer Variables



Physical representation Logical representation

Pointer Variables

- We must distinguish between a variable and its value. This figure details the differences.

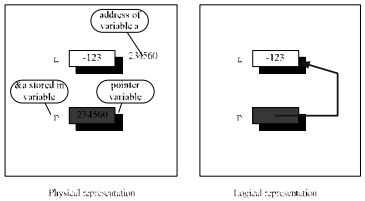


Physical representation Logical representation

- Variable a has its value, -123. the variable a is found at location 234560 in memory.
- Variable's name and location are constant, the value may change as the program executes.

Pointer Variables

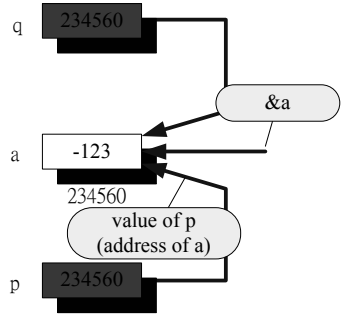
- there is a pointer variable, p. The pointer has a name and a location, both of which are constant.



- Its value at this point is the memory location 234560. This means that p is pointing to a.
- The physical representation shows how the data and pointer variables exist in memory.
- The logical representation shows the relationship between them without the physical details.

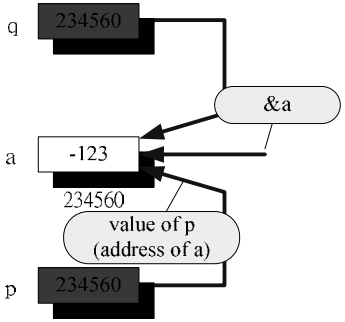
Pointer Variables

- We store a variable's address in two or more different pointer variables, as shown in following figure.



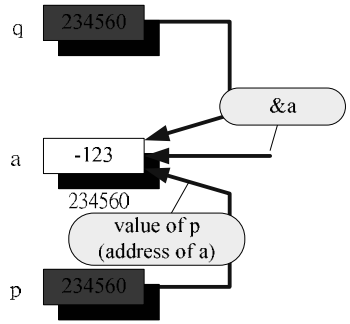
Pointer Variables

- There is a variable, a. and two pointers, p and q.
- Each pointer has a name and a location, both of which are constant.



Pointer Variables

- The value of p and q is the memory location 234560, which means that both p and q are pointing to a.
- There is no limit to the number of pointer variables that can point to a variable.



Pointer Variables

- C++ provides a special null constant, NULL, that can be used to set a pointer so that it points to nothing.
- You should always use NULL if a pointer does not contain an address.
- Similarly, when testing a pointer to determine if it is not active, you should test for NULL.

The Indirection Operator

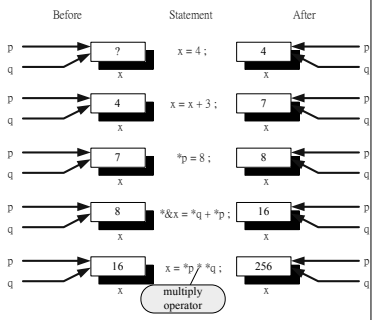
- C++ has an operator for this purpose, you will find the indirection operator (*).
- When you dereference a pointer, you are using its value to reference (address) another variable.
- The indirection operator is a unary operator whose operand must be a pointer value.
- The result is an expression that can be used to access the pointed variable for the purpose of inspection or alteration.

The Indirection Operator

- To access a through the pointer p, you simply code *p.
- The indirection operator is shown below
  *p
- Let us assume that we add 1 to the variable, a. We could do this with any of the following statements, assuming that the pointer, p, were properly initialized (p=&a).
  a++; a = a+1; *p = *p+1; (*p)++;

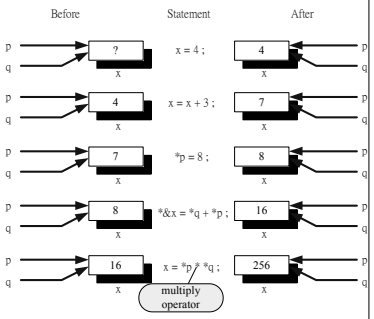
The Indirection Operator

- Let's assume that the variable x is pointed to by two pointers, p and q.
- The figure show, the expressions x, *p, *q all are expressions that allow the variable to be either inspected or changed.



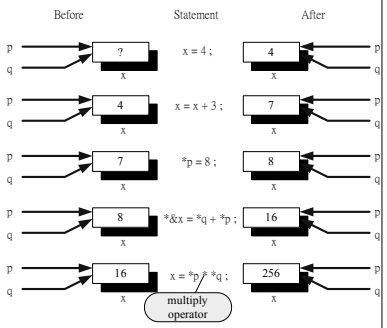
The Indirection Operator

- When used in the right-hand side of the assignment operator, they can only inspect (copy).
- When used in the left-hand side of the assignment operator, they alter the value of x.



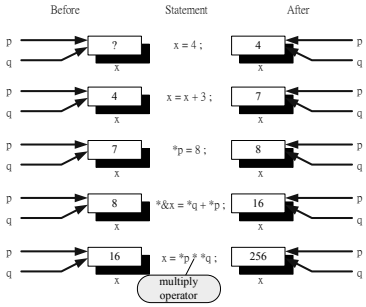
The Indirection Operator

- The indirection and address operators are the inverse of each other, and when they are combined in an expression, such as *&x, they cancel each other.
- Let's break down the expression and examine it.



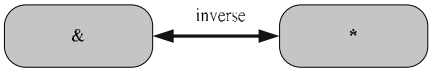
The Indirection Operator

- These two unary operators are evaluated form the right.
- The first expression is therefore &x, the address of x, which is a pointer value.
- The second expression, *(&x), dereferences the pointer constant, giving the variable (x) itself.

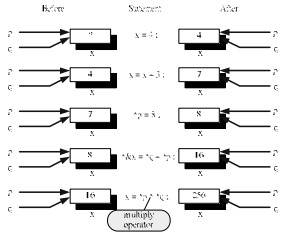


The Indirection Operator

- The operators effectively cancel each other.

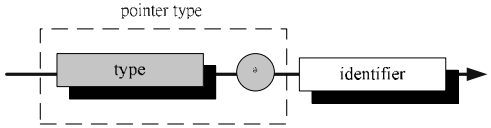


- We would never code the expression *&a in a program;



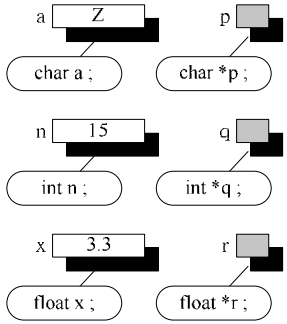
Pointer Declaration and Definition

- We use the indirection operator to define and declare pointer variables.
- Used in this way, it is really not an operator but rather a compiler syntactical notation.
- Making it the same token as the operator makes it easier to remember.



Pointer Declaration and Definition

- we declare different pointer variable. Their corresponding data variables are shown for comparison.
- In each case, the pointer is declared to be of a given type. Thus, p is a pointer to characters, q is a pointer to integers, and r is a pointer to floating-point variables.



Initialization of Pointer Variables

- In general, the C++ language does not initialize variables. Thus, when we start our program, all of our uninitialized variables have unknown garbage in them.
- The operating system often clears memory when it loads a program, but you can't count on this.

Initialization of Pointer Variables

- When the program starts, the pointers each have some unknown memory address in them.
- More precisely, they each have an unknown value that will be interpreted as a memory location.
- Most likely, the value will not be valid for the computer you are using, or it will not be valid for the memory you have been allocated.

Initialization of Pointer Variables

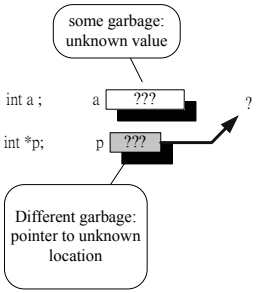
- If the address does not exist, you will get an immediate run-time error.
- If it is a valid address, you often, but unfortunately not always, get a run-time error.
- It is better to get the error when you use the invalid pointer than to have the program produce garbage.

Initialization of Pointer Variables

- One of the most common causes of errors in programming, by novices and professionals alike, is uninitialized pointers.
- Such errors can be very difficult to debug because the effect of the error is often delayed until later in the program execution.

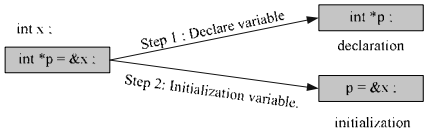
Initialization of Pointer Variables

- both an uninitialized variable and an uninitialized pointer.



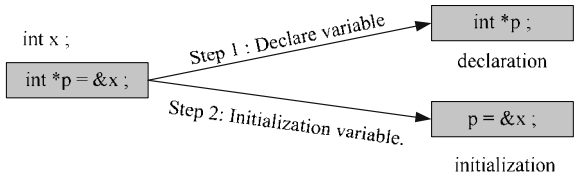
Initialization of Pointer Variables

- it is possible to initialize pointers when they are declared and defined.
- The data variable must be defined before the pointer variable.
- if we have an integer variable, x, and a pointer to integer, p, then to set p to point to x at declaration time, we code it as shown below.



Initialization of Pointer Variables

- The initialization involves two different steps.
  - First, the variable is declared.
  - Second, the assignment statement to initialize it is generated.



Initialization of Pointer Variables

- Some style experts suggest that you should not use an initializer in this way.
- Some argument is that it saves no code.That the initializer statement is required either as a part of the declaration and initialization or as a separately coded statement in the statement section of the function
- Putting the initialization in the declaration section tends to hide it and make program maintenance more difficult

Initialization of Pointer Variables

- We set a pointer to NULL, either during definition or during execution.
- The following statement demonstrates how we could define a pointer with an initial value of NULL:
  int *p = NULL;
- If you dereference p when it is NULL, you will most likely get a run-time error! NULL is not a valid address.
- The type of error you get will depend on the system you are using

Pointers and Functions

- One of the most useful application of pointers is in functions.
- When we discussed function earlier, we saw that C++ provides two ways to pass parameters to function:
  - Pass by value
  - Pass by address

Pointers and Functions

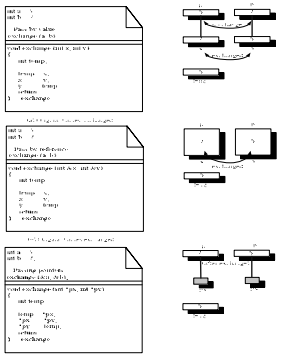
- When we passed by reference, C++ passes the address of the parameter variable, and the parameter name becomes an alias for the variable
- Any change made using the alias name resulted in a change to the original value.

Pointers and Functions

- We have studied pointers, we can all an alternative to pass by reference:
  - Pass a pointer
  - Use it to change the original variable
- The difference between pass by reference and passing pointers is that with pointers an alias is not created – we must use the dereference operator to effect the change.

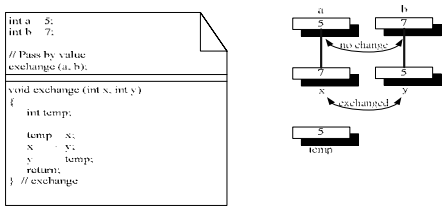
Pointers as Formal Parameters

- the three parameter formats with an exchange example.
- In all three examples, we call the exchange function, passing it two variables whose contents are to be exchange.



Pointers as Formal Parameters

- The data are exchanged in the called function, but nothing changes in the calling program. This obviously unworkable solution is illustrated in figure.



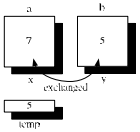
(a) Original values unchanged

Pointers as Formal Parameters

- The second example uses pass by reference. It this case, the values are exchanged using the alias names in the called function.

```
int a    5;
int b    7;

  Pass by reference
exchange (a, b);
```

```
void exchange (int &x, int &y)
{
    int temp;

    temp    x;
    x       y;
    y       temp;
    return;
}   exchange
```



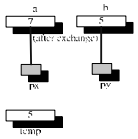
(b) Original values exchanged

Pointers as Formal Parameters

- The third example, we can pass pointers to the values. Once we have a pointer to a variable, it doesn't make any difference if it is local to the active function, if it is defined in main, or even if it is a global variable – we can change it.

```
int a    5;
int b    7;

  Passing pointers
exchange (&a, &b);
```

```
void exchange (int *px, int *py)
{
    int temp;

    temp    *px;
    *px     *py;
    *py     temp;
    return;
}   exchange
```



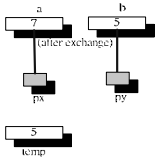
(c) Original values exchanged

Pointers as Formal Parameters

- We need to dereference a pointer to refer to the data it is pointing to.

```
int a    5;
int b    7;

// Passing pointers
exchange (&a, &b);
```

```
void exchange (int *px, int *pv)
{
    int temp;

    temp    *px;
    *px     *pv;
    *pv     temp;
    return;
}  // exchange
```



(c) Original values exchanged

Pointers as Formal Parameters

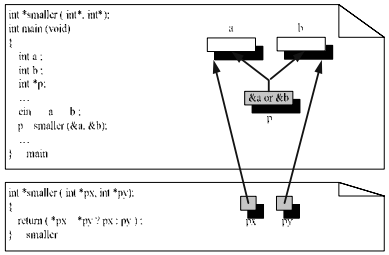
- when we must send back more than one value from a function, we have two choices:
  - Pass by reference
  - Pass by pointers
- Both accomplish the same result:
  - changing the values in the calling function's scope.
- Pass by reference is an easier, more natural way to work, and we recommend it over passing pointers.

Functions Returning Pointers

- Nothing prevents a function from returning a pointer to the calling function.
- In fact, as we shall see, it is quite common for functions to return pointers.
- As an example, let us write a rather trivial function to determine the smaller of two numbers.

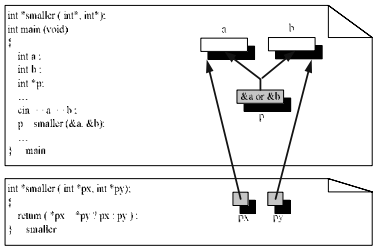
Functions Returning Pointers

- we need is a pointer to the smaller of two variables, a and b.
- We are looking for a pointer, we pass two pointers to the function, which uses a conditional expression to determine which value is smaller.



Functions Returning Pointers

- Once we know the smaller value, we can return the address of its location as a pointer.
- The return value is then placed in the calling function's pointer, p, so that after the call it points to either a or b based on their values.



Functions Returning Pointers

- When you return a pointer, it must point to data in the calling function or higher level functions.
- It is an error to return a pointer to a local variable in the called function because when the function terminates, its memory may be used by other parts of the program.

Functions Returning Pointers

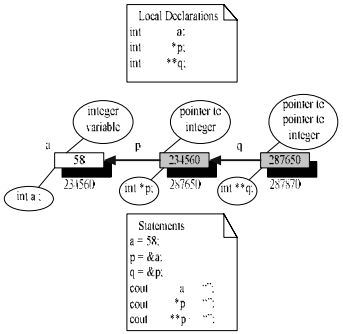
- A simple program might not “notice” the error because the space was not reused,
- A large program would get the wrong answer of fail when the memory being referenced by the pointer was changed.
- It is a serious error to return a pointer a local variable.

Pointers to Pointers

- Pointers, we have been using have pointed directly to data.
- It is possible to use pointer that point to other pointers.
- We can have a pointer pointing to pointer to an integer.

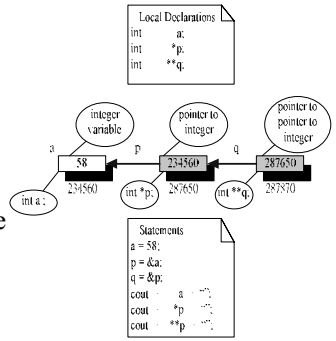
Pointers to Pointers

- There is no limit as to how many levels of indirection you can use.
- In practice, it seldom goes beyond two levels.

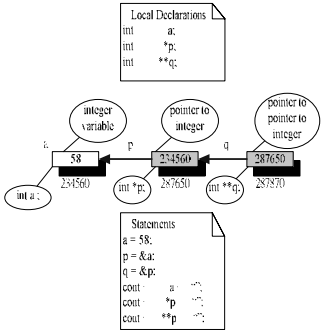


Pointers to Pointers

- Each level of pointer indirection requires a separate indirection operator when it is dereferenced.
- In figure, to refer to a using the pointer p, we have to dereference it once, as shown below. *p

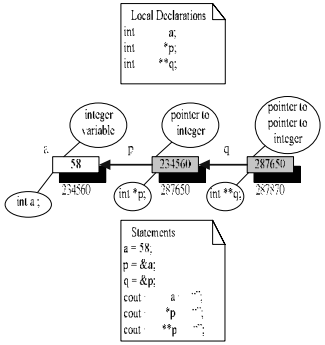


Pointers to Pointers



- To refer to a using the pointer q, we have to dereference it twice because there are two levels of indirection (pointers) involved.
- In other word, q is a pointer to a pointer to an integer.
- The double dereference is shown below
  **q

Pointers to Pointers



- Let's look at how we used these concepts in the C++ code fragment in figure.
- All three references in the cout statements refer to the variable a.
  - The first print statement prints the value of a directly.
  - The second uses the pointer p.
  - The third uses the pointer q.
- The result is that the value 58 prints three times, as shown below.
  58 58 58

Compatibility and the void pointer

- With one exception, it is invalid to assign to pointer of one type to a pointer of another type, though the values in both case are memory addresses and would seem to be fully compatible.
- The addresses may be compatible because they are drawn from the same set, what is not compatible is the underlying data type of the referenced object.
- In C++, we can't use the assignment operator with pointers to different types. If we try to , we get a compile error.

Compatibility and the void pointer

- The exception to the rule is the void pointer.
- The void pointer, known as the universal or generic pointer, can be used with any pointer, and any pointer can be assigned to a void pointer.
- A void pointer has no object type, it cannot be dereferenced. A void pointer is created as shown below.
  void *pVoid;

Casting Pointers

- It is possible to make an explicit assignment between incompatible pointer types by using a cast, just as it is possible to cast an integer to a float.
- For example, if for some unfathomable reason you decided that you had to use the character pointer, p, to point to an integer (a), you could cast it as shown below.

```
int     a;
char   *p;

p = ( char * ) &a;
```


Casting Pointers

- In this case, it is user beware! Unless you cast all operations that used p, you would most likely end up creating mounds of garbage.
- In fact, we will say that, with the exception of the void pointer, you should never cast a pointer.

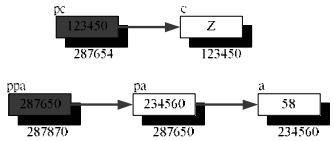
Casting Pointers

- The following assignments are all valid, but they are extremely dangerous and must be used with a very carefully though-out design.

```
//    Local Declarations
      void   *pVoid;
      char  *pChar;
      int     *pInt;
//    Statements
      pVoid= pChar;
      pInt    = pVoid;
      pInt    = (int *) pChar;
```

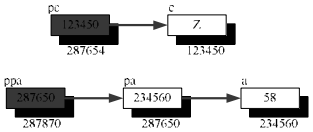

Casting Pointers

- Let's construct an example in which we have two variable:
  - One integer
  - One character
- The character has one pointer associated with if; the integer has two, one a second-level pointer.
- These variables and their pointers are shown in figure.




```
char c = 'z';
char *pc ;

int       a = 58;
int      *pa;
int     **ppa;

pc       = &c;    //Good and valid
pa       = &a;    //Good and valid
ppa      = &pa;   //Good and valid

// The following are invalid and will generate errors
pc       = &a;    //Different types
ppa      = &a;    //Different levels
```

Casting Pointers

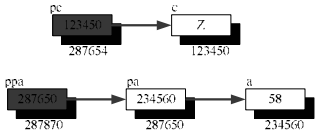


- Without casting the assignment, we cannot make the character pointer point to the integer value.
- For example, it is invalid and will result in a compile error to store the address on a in PC.

```
char c = 'z';
char *pc ;

int     a = 58;
int     *pa;
int     **ppa;

pc      = &c;   //Good and valid
pa      = &a;   //Good and valid
ppa     = &pa;  //Good and valid

// The following are invalid and will generate errors
pc      = &a;   //Different types
ppa     = &a;   //Different levels
```


Casting Pointers

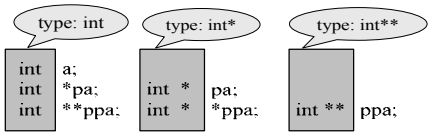
- When the pointers are associated with the same type, as seen in the integer pointers and examples in figure, any assignment made must be at the correct level.




```
char c = 'z';
char *pc ;

int     a = 58;
int     *pa;
int     **ppa;

pc      = &c;   //Good and valid
pa      = &a;   //Good and valid
ppa     = &pa;  //Good and valid

// The following are invalid and will generate errors
pc      = &a;   //Different types
ppa     = &a;   //Different levels
```


Casting Pointers

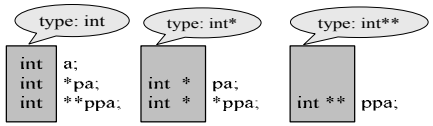
- It is an error to assign the address of a, even though it is an integer, to ppa.
- This is because ppa is a pointer to a pointer to an integer.




```
a       = 4;
*pa     = 4;    pa    = &a;
**ppa   = 4;    *ppa= &a;       ppa = &pa;
```


Casting Pointers

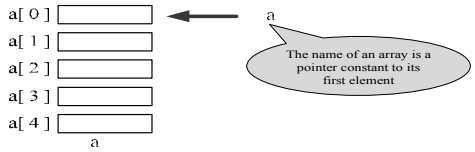
- Its type is pointer to pointer, not pointer to integer.
- It can only be assigned the address of a pointer to an integer.




```
a       = 4;
*pa     = 4;    pa    = &a;
**ppa   = 4;    *ppa= &a;       ppa = &pa;
```

Arrays and Pointers

- There is a very close relationship between arrays and pointers.
- The name of an array is a pointer constant to the first element.
- Because the array's name is a pointer constant, its value cannot be changed. The figure shows an array with the array name as a pointer constant.



Arrays and Pointers

- The array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory.
- We can use the array name anywhere we can use a pointer, as long as it is being used as an right-hand side value.

Arrays and Pointers

- Specifically, this means that we can use it with the indirection operator. When we dereference an array name, we are dereferencing the first element of the array.
- We are referring to array[0]. However, when the array name is dereferenced, it is referring only to the first element, not the whole array.
- The name of the array (a) is a pointer only to the first element – not the whole array.

Arrays and Pointers

- Prove this to yourself by writing program with the code block shown below. The block prints the address of the first element of the array (&a[0]) and the array name, which is a pointer constant.

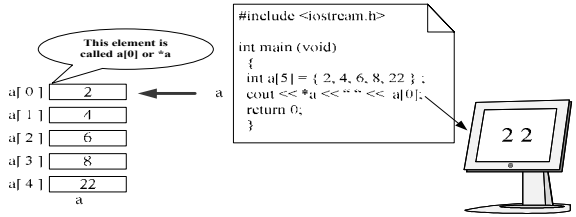
```
{ //Demonstrate array name is a pointer constant
  int a [5];
  cout << "Address of a[0]: "    << &a[0]
        << "Name as pointer: " << a << endl;
}
```

Arrays and Pointers

- The values printed by this code will be addresses in your computer.
- The first printed address (the address of the first element in the array) and the second printed address (the array pointer) will be the same, proving our point.

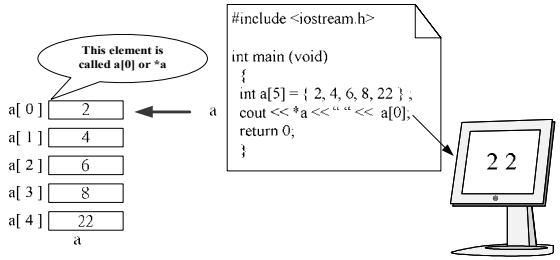
Arrays and Pointers

- A simple variation on this code is to print the value in the first element of the array using both a pointer and an index.



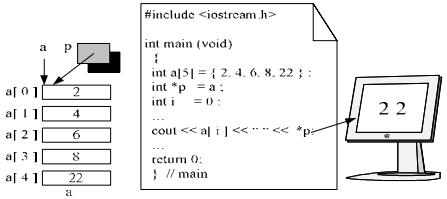
Arrays and Pointers

- The same value, 2, is printed in both cases, again proving our point that the array name is a pointer constant to the beginning of the array.



Arrays and Pointers

- We'll explore another point. If the name of an array is really a pointer, let's see if we can store this pointer in a pointer variable and use it in the same way we use the name of the array.



Arrays and Pointers

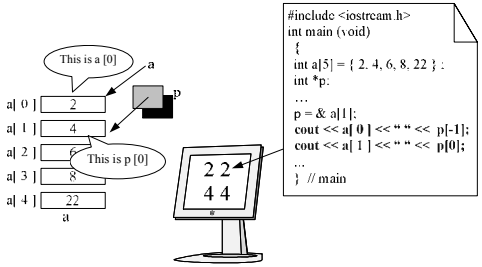
- Define a pointer and initialize it to point to the first element of the array by assigning the array name.
- The array name is unqualified, there is no address operator or index specification.
- We print the first element in the array, first using an index notation and then pointer notation.
- To access an array, any pointer to the first element can be used instead of the name of the array.

Arrays and Pointers

- The another example explores the close relationship between an array and a pointer.
- We store the address names to access each element. This does not mean that we have two arrays.
- It shows that a single array can be accessed through different pointers.

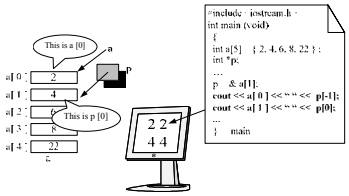
Arrays and Pointers

- use of multiple names for an array to reference different location at the same time.



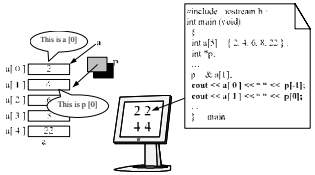
Arrays and Pointers

- First, we have the array name. We create a pointer to integer and set it to the second element of the array (a[1]).
- It is a pointer, we can use it as an array name and index it to point to different elements in the array.



Arrays and Pointers

- We demonstrate this by printing the first two elements using first the array name and then the pointer.
- When a pointer is not referencing the first element of an array, it can have a negative offset.
- This is shown in the reference to p[-1].



Pointer Arithmetic and Arrays

- Besides indexing, there is another powerful method of moving through an array: pointer arithmetic.
- Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the address in pointers.
- Pointer arithmetic is especially powerful when we want to move through an array from element to element, such as when we are searching an array sequentially.

Pointers and One-Dimensional Arrays

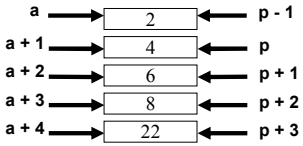
- If we have an array, a, then a is a constant pointing to the first element and a+1 is a constant pointing to the second element.
- Again, if we have a pointer, p, pointing to the second element of an array, then p-1 is a pointer to the previous (first) element and p+1 is a pointer to the next (third) element.

Pointers and One-Dimensional Arrays

- Furthermore, given a, a+2 is the address two elements from a and a+3 is the address three elements from a.
- We can generalize the notation, therefore, as follows: Given pointer, p, p±n is a pointer to the value n elements away.

Pointers and One-Dimensional Arrays

- It does not matter how a and p are defined or initialized; as long as they are pointing to one of the elements of the array, we can add or subtract to get the address of the other elements of the array.



Pointers and One-Dimensional Arrays

- The meaning of adding or subtracting here is different from normal arithmetic.
- When you add an integer n to a pointer value, you will get a value that corresponds to another index location n elements away.
- In other words, n is an offset from the original pointer. To determine the new value, C++ must know the size of one element.
- The size of the element is determined by the type of the pointer.
- This is one of the prime reasons that pointers of different types cannot be assigned to each other.

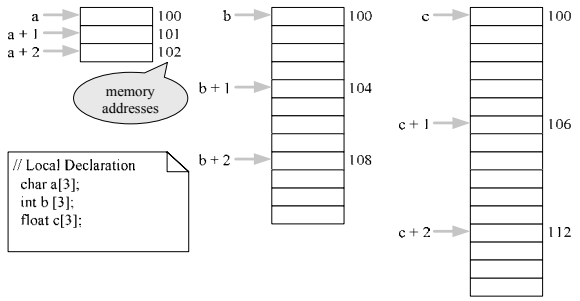
Pointers and One-Dimensional Arrays

- If the offset is 1, then C++ can simply add or subtract one element size from the current pointer value.
- This may make the access more efficient than the corresponding index notation.
- If the offset is more than 1, C++ must compute the offset by multiplying the offset by the size of one array element and adding it to the pointer value.
- This calculation is shown below.
  Address = pointer + (offset * size of element)

Pointers and One-Dimensional Arrays

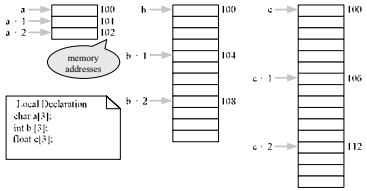
- Depending on the hardware, the multiplication in this formula can make it less efficient than simply adding 1, and the efficiency advantage of pointer arithmetic over indexing may be lost.

Pointers and One-Dimensional Arrays



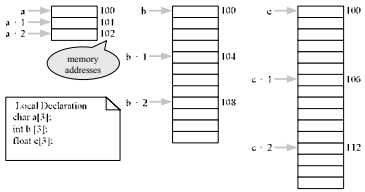
Pointers and One-Dimensional Arrays

- For char, which is usually implemented as one byte, adding 1 moves us to the next memory address (101).
- Assuming that integers are four bytes (b), adding 1 moves us four bytes in memory (104).



Pointers and One-Dimensional Arrays

- Finally, assuming the size of float is six bytes (c), adding 1 moves us six bytes in memory (106).
- In other words, a+1 means different things in different situations.

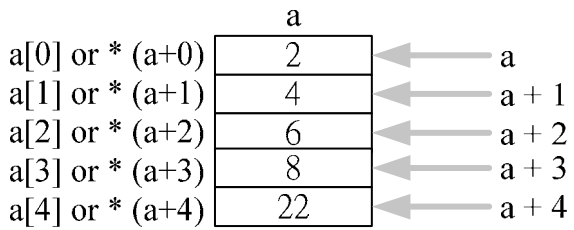


Pointers and One-Dimensional Arrays

- Let look at how we can use that value
- We have two choices:
  - First, we can assign it to another pointer. This is a rather elementary operation that uses the assignment operator, as shown below.
    p = aryName + 5;
  - Second, we can use it with the indirection operator to access or change the value of the element we can pointing to.

Pointers and One-Dimensional Arrays

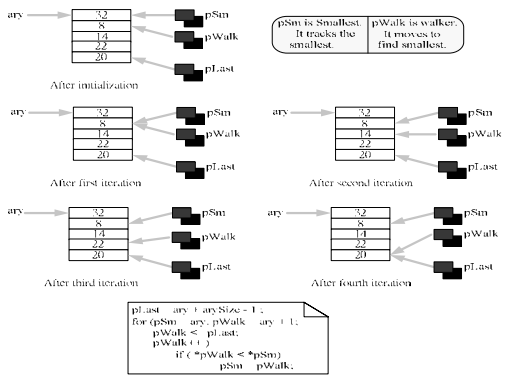
- This possibility is demonstrated in following figure.



Pointers and One-Dimensional Arrays

- To practice, let's use pointers to find the smallest number among five integers stored in an array.
- The following figure tracks the code as it works its way through the array.
- We start with the smallest pointer (pSm) set to the first element of the array.

Pointers and One-Dimensional Arrays



Pointers and One-Dimensional Arrays

- The function's job is to see if any of the remaining elements are smaller.
- Since we know that the first element is not smaller than itself, we set the walking pointer (pWalk) to the second element.
- The walking pointer then advances through the remaining elements, each time checking the element it is current element is smaller, its location is assigned to pSm.

Pointers and Other Operators

- Arithmetic operations involving pointers are very limited.
- Addition can be used only when one operand is a pointer and the other is an integer.
- Subtraction can be used only when both operands are pointers or when the first operand is a pointer and the second operand is an integer, such as an array index.
- You can manipulate a pointer with the postfix and unary increment and decrement operators.
- All of the following pointer arithmetic operations are valid: p+5 5+p p-5 p1-p2 p++ --p

Pointers and Other Operators

- When one pointer is subtracted from another, the result is an index representing the number of elements between the two pointers.
- Note, however, that the result is meaningful only if the two pointers are associated with the same array structure.

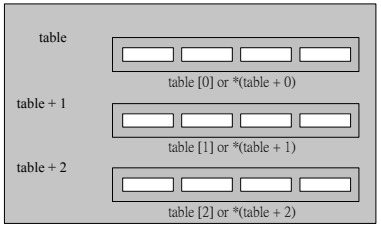
Pointers and Other Operators

- The relational operators are allowed only if both operands are pointers of the same type.
- Two pointer relational expressions are shown below.
  p1>=p2 p1!=p2
- The most common comparison is a pointer and the NULL constant, as shown in Table.

Long Form	Short Form
if (ptr==NULL)	if (!ptr)
if (ptr !=NULL)	if (ptr)

Pointers and Two-Dimensional Arrays

- This Figure contains a two-dimensional array and a code fragment to print the array.



int table[3][4]

```
for (i = 0; i < 3; i++ )
 {
   for ( j = 0; j < 4; j++ )
       cout << setw(6) << *(*(table+i)+j);
   cout << endl;
 }  // for i
```

Pointers and Two-Dimensional Arrays

- The complex expression shown below.
  *(*table + i) + j)
- This pointer notation is equivalent to the index syntax, table[i][j]. With multidimensional arrays, the pointer arithmetic has no efficiency advantage over indexing.
- Because the pointer notation for multidimensional arrays is so complex and there is no efficiency advantage, most programmers find it easier to use the index notation.

Passing an Array to a Function

- The name of an array is actually a pointer to the first element, we can send the array name to a function for processing.
- When we pass the array, we do not use the address operator.
- Remember, the array name is a pointer constant, so the name is already the address of the first element in the array.
- A typical call would look like the following:
  doIt (aryName);

Passing an Array to a Function

- The called program can declare the array in one of two ways.
- First, it can use the traditional array notation. This format has the advantage of telling the user very clearly that the program is dealing with an array rather than a single pointer.
- This is an advantage from a structured programming and human engineering point of view.
  int doIt (int ary [ ]);

Passing an Array to a Function

- You can declare the array in the prototype statement as a simple pointer.
- The disadvantage to this format is that, while it is technically correct, it actually masks the data structure (array).
- For one-dimensional arrays, it is the code of choice with professional programmers.
  int doIt (int *arySalary);

Passing an Array to a Function

- If you are passing a multidimensional array, you must use the array syntax in the header declaration and definition.
- The compiler needs to know the size of the dimensions after the first to calculate the offset for pointer arithmetic.
- To receive a three-dimensional array, you would use the following declaration in the function's header statement:
  float doIt (int bigAry [ ] [12] [5]);

Memory Allocation Functions

- We have two choices when we want to reserve memory locations for an object:
  - Static allocation
  - Dynamic allocation

Memory Allocation Functions

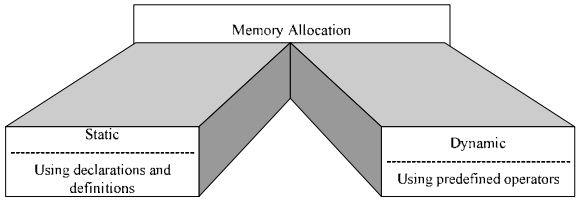
- Static memory allocation requires that the declaration and definition of memory be fully specified in the source program.
- The number of bytes reserved cannot be changed during run time.
- Static allocation is the technique we have used in this book to this point.
- It works fine as long as you know exactly what your data requirements are.

Memory Allocation Functions

- Dynamic memory allocation uses predefined operators to allocate and release memory for data while the program is running.
- This approach effectively postpones the data definition to run time.
- To use dynamic memory allocation, the programmer must use either standard data types or already must have declared any derived types.

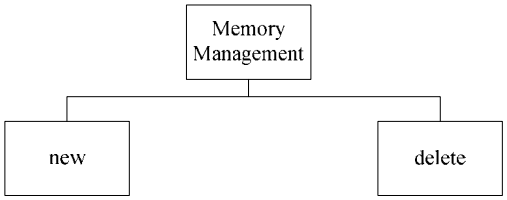
Memory Allocation Functions

- This figure shows the characteristics of memory allocation.



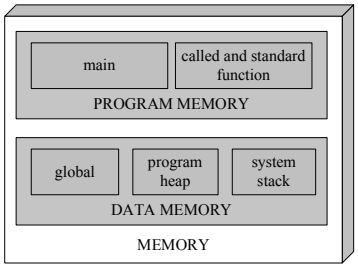
Memory Usage

- C++ uses two memory operators: new is used to allocate space from dynamic memory, and delete is used to return it for reuse, as shown in following figure.



Memory Usage

- The conceptual view of memory



Memory Usage

- Conceptually, we can say that memory is divided into
  - Program memory
  - Data memory
- Program memory consists of the memory used for main and all called functions.
- Data memory consists of permanent definitions, such as global data and constants, local definitions, and dynamic data memory

Memory Allocation (new)

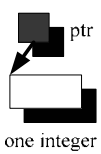
- The new operator allocates dynamic memory large enough to store the type being allocated and return its address as a pointer.
- The operator new is a unary operator; its operand is the object type for the memory being allocated.
- The compiler knows the size of all types, standard and derived, it knows the amount of space needed by the object's type.

Memory Allocation (new)

- The following example allocates memory for an integer and assigns its address to the pointer.
  int *ptr;
  ptr = new int;
- If there is not enough dynamic memory for an object, C++ returns a NULL pointer.

Memory Allocation (new)

- a typical new allocation.




```
int *ptr;
if ( ! (ptr = new int ))
        // No memory available
        exit (100);
// Memory available
        ...
```

- we are allocating one integer object.
  - If the memory is allocated successfully, ptr contains a value (address).
  - If the memory allocation is not successful (ptr contains NULL), there is no memory and we exit the program with error code 100.

Initialization of Dynamic Memory

- Allocations for standard data types can be initialized with the new operator.
- Initialization for derived types must be programmed.
- To initialize a standard type, we enclose the initializer in parentheses after the type.
- To initialize the dynamic memory integer we allocated earlier to the value 39, we would write the following code:
  int *ptr;
  ptr = new int(39);

Releasing Memory (delete)

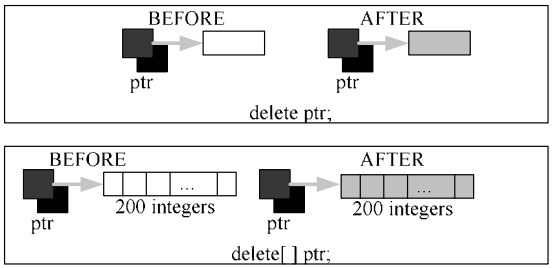
- When dynamic memory locations are no longer needed, they should be released using the delete operator.
- Delete is a unary operator. Its operand is a pointer that was previously used to dynamically allocate memory using the new operator.
- It is an error to delete memory that was not allocated with the new operator or to refer to memory after it has been released.
- The statement to delete our pointer to integer is shown below.
  delete ptr;

Releasing Memory (delete)

- To delete a dynamic array, we place brackets after the delete operator as shown below.
- It is an error to use the delete operator without brackets to delete an array.
  delete [ ] ptr;

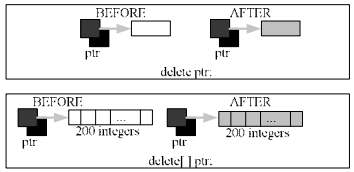
Releasing Memory (delete)

- The figure shows two delete example.



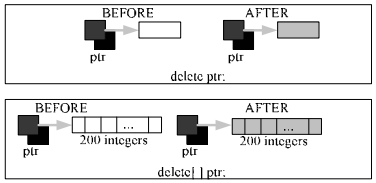
Releasing Memory (delete)

- The first one releases a single element, allocated back to dynamic memory.
- In the second example, the 200 elements were previously allocated.
- When we delete the pointer, all 200 elements are released



Releasing Memory (delete)

- You should note two things in this figure.
  - First, it is not the pointers that are being released but rather what they point to.
  - Second, to release a dynamic memory array, you need only release the pointer once



Releasing Memory (delete)

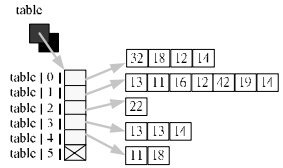
- Releasing memory does not change the value in a pointer. It still contains the address in the heap.
- It is a logic error to use the pointer after memory has been released.
- Your program may continue to run, but the data may be destroyed if the memory area is allocated for another use.
- This logic error is very difficult to trace in your program. We suggest that immediately after you free memory you also clear the pointer by setting it to NULL.

Array of Pointers

- Using a two-dimensional array to store these numbers would waste a lot of memory.
- The solution in this case is to create five one-dimensional arrays that are joined through an array of pointers

Array of Pointers

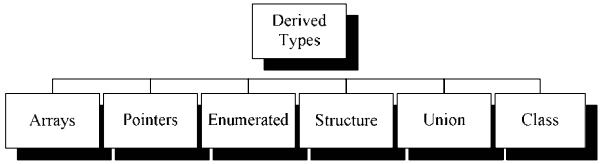
- This concept is shown in following figure along with the statements needed to allocate the arrays in the heap.
- table is a pointer to a pointer to an integer and must be declared as shown below, not as an array.
  int **table




```
table = new int* [rowNum + 1];
...
table[0] = new int[4];
table[1] = new int[7];
table[2] = new int[1];
table[3] = new int[3];
table[4] = new int[2];
table[5] = NULL;
```

Derived Types

- We will discuss three of the remaining derived types:
  - Enumerated
  - Structure
  - Union
- The derived types are shown in this figure.



Enumerated Types

- The enumerated type, enum, is derived from the integer type.
- In an enumerated type, each integer value is given an identifier called an enumeration constant.
- We can use symbolic names rather than numbers, which makes our programs much more readable
- For example, enumeration make it possible to give symbolic names to the case identifiers in a switch statement, thereby making the switch cases more readable.

Enumerated Types

- Once we have declared an enumerated type, we can create variables from the standard types.
- In fact, C++ allows the enumerated constants, or variables that hold enumerated constants to be used anywhere.
- As shown in this figure, there are two basic enumeration formats: the enumeration constant and the enumeration type.

```
enum                    { enumeration constants };

enum    type_name       { enumeration constants };
type_name       variable_name;
```


Enumerated Types

- Both enumeration formats start with the keyword enum.
- The enumeration constant is followed by a list of enumeration constants.
- For example, enumeration constants for on and off may be declared as shown below.
  enum {off, on};

Enumerated Types

- The only difference between the enumeration constant and the enumeration type is that the type is given a type name.
- The type name can be used to declare enumerated variables.
- An enumerated type for the months of the year is shown below, followed by a variable definition.

```
enum months {jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec};
months dateMonth;
```


Enumerated Types

- The list of constant identifiers provides the symbolic names we can use in our programs.
- The first thing you should notice is that the list is contained within braces.
- The purpose of an enumerated type is to assign names to integers, the question is what integers are we assigning these names to?
- When you don't tell C++ what values you want to use, it simply start at 0 and then equates each enumerated constant to the next higher integral number.
- In this case, jan equates to 0, feb equates to 1, and so forth, until we get to dec, which equates to 11.

Enumerated Types

- There's nothing wrong with the way we set up our months, unless we are asking users to given us a month.
- In that case, it's not a good idea to ask users to have to manipulate the data they give us because often this leads to errors.
- What we should do, therefore, is to equate each month to its normal value, such as the value 1 for January.
- This is done with an assignment operator, as shown below.

```
enum months { jan = 1, feb = 2, mar = 3,
              apr = 4, may = 5, jun = 6,
              jul = 7, aug = 8, sep = 9,
               oct = 10, nov = 11, dec = 12 };
```


Enumerated Types

- This is perfectly good code.
- We can shorten it a little by using the compiler.
- Because C++ assigns the next larger integer to each enumeration constant, we can specify the starting point and let C++ do the rest of the work.
- Here's months, coded the shorter way.

```
enum months { jan = 1, feb, mar,
              apr, may, jun,
              jul, aug, sep,
               oct, nov, dec };
```

- Now jan will start with a rather than 0, and all the other months will fall in line.

Enumerated Types

- Three final thoughts about enumerated types:
- First, C++ allows you to assign the same integer to multiple enumeration constants in the same definition.
- For example, in the enumerated colors below, we have assigned the same number to orange and tangerine.
  enum colors (green, blue, orange, tangerine = 2};

Enumerated Types

- Second, while the underlying type for enumeration is integer, C++ treats it as a separate type.
- Because C++ promotes an enumerated type to integer when required, you can assign an enumerated type to an integer.
- You cannot assign an integer to an enumerated type unless you cast it.
- Assigning orange to an integer is permitted, nut assigning 5 to a variable of type colors is not.
- This is true even if the integer is in the enumerated type' range.

Enumerated Types

- Finally, enumerated types are promoted to integers when they are printed.
- If we were to print shirt in this example.

```
colors      shirt;
int         x;

shirt = orange;      // OK
shirt = 2;                  // ERROR
shirt = (color) 2;          // OK
x     = orange;      //OK. x is 2
```

- We would see 2, not orange. If we want to print the colors, we would use a switch statement to determine the color and print the appropriate literal description

Structure

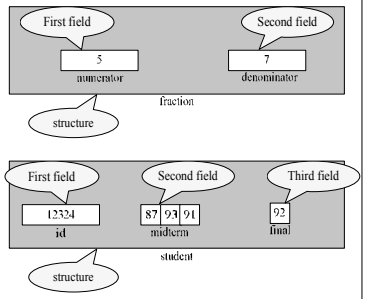
- A structure is a collection of related elements, possibly of different types, having a single name.
- Each element in a structure is called a field.
- A field is the smallest element of named data that has meaning.
- It has many of the characteristics of the variables you have been using in your programs.
- It has a type and it exists in memory.
- It can be assigned values, which in turn can be accessed for selection or manipulation.
- As an example, consider the data you might store about a student.
  - Several fields come to mind quickly: name, student number, address, major and so forth.

Structure

- We have studied another data type that can hold multiple pieces of data: the array.
- The difference between an array and a structure is that all elements in an array must be of the same type, while the elements in a structure can be of the same or different types.

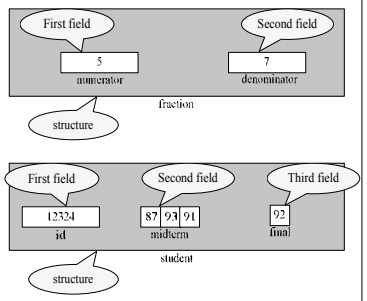
Structure

- This figure contains two examples of structures.
- The first example, fraction has two field, both of which are integer.
- The second example, student, has three field, two of which are integers and one an array.



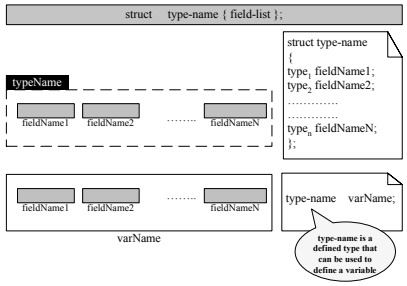
Structure

- One design caution to keep in mind concerning structures is that the data in a structure should all be related to one object.
- In this figure, the integers in the fraction in the first example both belong to the same fraction and the data in the second example all relate to one student.
- It is not good structured programming practice to combine unrelated data, even for programming expediency.



Structure Declaration and Definition

- Like all data types, structures must be declared and defined.
- To declare a structure type, we use the keyword struct followed by the name of the type and its field list



Structure Declaration and Definition

- To define variable at the same time you declare the structure, you can simply list the variables, separated by commas, after the closing brace.
- There are two reasons why we do not recommend that you do so.
- First, the proper place for structure declarations is in the global area of the program before main.
- This puts them within the scope of the entire program and is mandatory if the structure is to be shared by functions.
- In fact, on large projects, you will usually find the structures declared in a header file that is shared by all members of the project.

Structure Declaration and Definition

- The second reason is that it breaks the rule of putting multiple variable definitions in one statement.
- If you do define a variable when you define the structure, we strongly suggest that you define only one.

Structure Declaration and Definition

- Once you have declared a structure, you can use it to define variables.
- To declare and use the student structure, you would code it as follows.

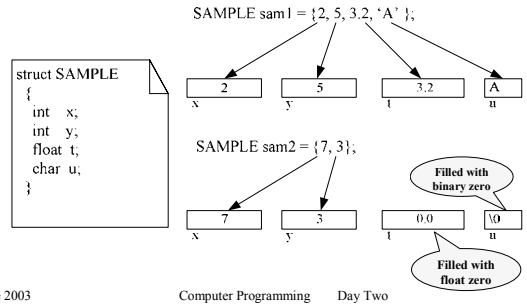
```
struct STUDENT
{
  intid;
  intmidterm[3];
  intfinal;
};

STUDENT aStudent;
…
void printStudent ( STUDENT Stu);
```


Initialization

- A structure can be initialized.
- The rules for structure initialization are similar to the rules for array initialization:
  1. The initializers are enclosed in braces and separated by commas
  2. The initializers must match their corresponding types in the structure declaration
  3. If you use a nested structure, the nested initializers must be enclosed in their own set of braces.

Initialization

- The figure shows two examples of structure initialization.



Initialization

- In the first example, there is an initializer for each field.
- How they are mapped to the structure in sequence.
- The second example demonstrates what happens when no all fields are initialized.
- As with arrays, when one or more initializers are missing, the structure elements will be assigned null values – 0 for integers and floating-point numbers, and '\0' for characters.

Accessing Structures

- We will first discuss how to access individual components of a structure and then consider the assignment of whole structures.
- After looking at how pointers are used with structures, we will conclude by examining arrays of structures.

Referencing Individual fields

- Each field in a structure can be accessed and manipulated using expressions and operators.
- Anything you can do with an individual variable can be done with a structure field.
- The only is problem is to identify the individual fields you are interested in.

Referencing Individual fields

- Each field in a structure has a name, we could simply use the name.
- The problem with such a simple approach is that if we wanted to compare a student's id in one structure to a student's id in another structure, the statement would end up being
  if (id == id)

Referencing Individual fields

- Which is an ambiguous expression.
- We need some way to identify the structures that contain the field identifiers, in this case, id.
- C++ uses an operator that is common to many other languages, the member operator, which is simple a period (.).

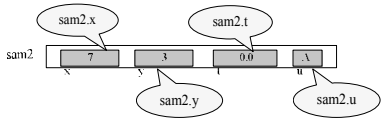
Referencing Individual fields

- Using the structure student, we can refer to the individual components as shown below.

  aStudent.id
  aStudent.midterm[1]
  aStudent.final

Referencing Individual fields

- This figure contains another example, using the structure sample.
- With this structure, we can use a selection statement to evaluate the character member, u, and, if it is an 'A', add the two integer elements and store the result in the first.
- This code is shown below.
  if (sam2.u == 'A')
    sam2.x += sam2.y;



Referencing Individual fields

- We can also read data into and write data from structure members just as we can from individual variables.
- For example, the value for the field of the sample structure can be read from the keyboard and placed in sam1 using the input statement below.

```
cin >> sam1.x >> sam1.y >> sam1.t >> sam1.u;
```


Precedence of Member Operator

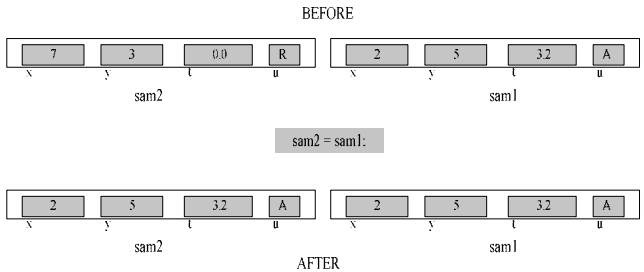
- The dot operator creates a postfix expression from a primary expression.
- When you use the dot operator, the value must be determined immediately.
- For example, consider the following statements:
  sam2.x++ ++sam2.x

Structure Operations

- The structure is an entity that can be treated as a whole.
- However, only one operation, assignment, is allowed on the structure itself.
- In other words, a structure can only be copied to another structure of the same type using the assignment operator.
- Rather than assign individual members when we want to copy one structure to another, we can simply assign one to the other.

Structure Operations

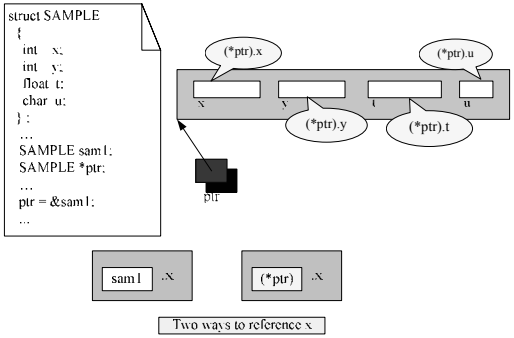
- This figure copies sam1 to sam2.



Pointer to Structures

- Structures, like other types, cal be accessed through pointers.
- In fact, this is one of the most common methods used to reference structures.
- For example, let's use our SAMPLE structure with pointers
- The first thing we must do is define a pointer for the structure, as shown below.
  SAMPLE *ptr;

Pointer to Structures



Pointer to Structures

- We now assign the address of sam1 to the pointer using the address operator (&) as we would with any other pointer.
  ptr = &sam1;
- Now we can access the structure itself and all the members using the pointer, ptr.
- The structure itself can be accessed like any object using the indirection operator (*).
  *ptr // Refer to whole structure

Pointer to Structures

- The pointer contains the address of the beginning of the structure, we do not use the structure name with the number operator, the pointer takes its place.
- The reference to each of the SAMPLE members is shown below.
  (*ptr).x (*ptr).y (*ptr).t (*ptr).u

Pointer to Structures

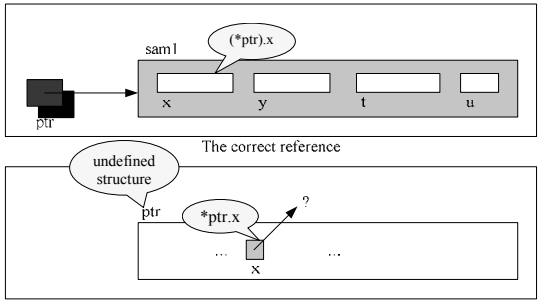
- The parentheses in the above expressions.
- They are absolutely necessary; however, omitting them is a very common mistake.
- They are required because the precedence priority of the member operator (17) is higher than the priority of the indirection operator (15).
- If you do not use the parentheses, C++ applies the dot operator first and the asterisk operator next.
- In other words,
  *ptr.x is interpreted as *(ptr.x)
  which is wrong.

Pointer to Structures

- The expression *(ptr.x) means that there is a completely different structure called ptr that contains a member, x, which must be a pointer.
- This is not the case, the result is a compile error.
- The correct notation, (*ptr).x, first resolves the primary expression (*ptr) and then applies the dereferenced value to the member, x.

Pointer to Structures

- This figure shows how this error is interpreted.



Selection Operator

- There is another operator that eliminates the problems with pointers to structures – the selection operator.
- The selection operator is at the same level as the member operator.

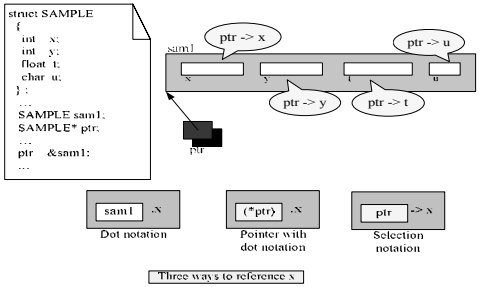
(*pointerName).fieldName
Same As
pointerName → fieldName

Selection Operator

- The token for the selection operator is an arrow formed by the minus sign and the greater than symbol (→).
- The token is placed immediately after the pointer identifier and before the member to be referenced.

Selection Operator

- We use this operator to refer to the members of our previously declared structure, sam1, in this figure.



Complex Structures

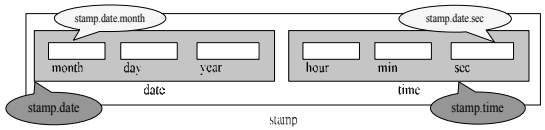
- The structures were designed to handle complex problems.
- The limitations on structures are not on the structures themselves but on the imagination of the software engineers who solve the problems
- Structures within structures, arrays within structures (nested structure), and arrays of structures are all common.

Nested Structures

- We have structure as members of a structure.
- When a structure includes another structure, it is a nested structure.
- There is no actual limit to the number of structures that can be nested, but in practice, the number seldom goes beyond three.

Nested Structures

- For example, we can have a structure called stamp that stores the date and the time.
- The date is in turn a structure that stores the month, day, and year.
- The time is also a structure, one that stores the hour, minute, and second.
- This structure design is shown in this figure.



Declaring Nested Structures

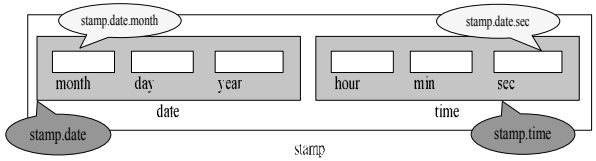
- It is possible to declare a nested structure with one declaration, this approach is not recommended.
- It is far simpler and much easier to follow the structure if each structure is declared separately and then grouped in the high-level structure.

Declaring Nested Structures

- When declaring structures separately, the most important point to remember is that nesting must be done from inside out – from the lowest level to the most inclusive level.
- In other word, the innermost structure must be declared first, then the next level, working upward toward the outer, most inclusive structure.

Declaring Nested Structures

- Consider the time stamp structure shown in this figure.



Declaring Nested Structures

- The inner two structures, DATE and TIME, must be declared before the outside structure, STAMP, is declared.
- We show the declaration of STAMP and a variable that uses it below.

```
struct DATE
{
   int month;
   int day;
   int year;
};
```

```
struct TIME
{
   int hour;
   int min;
   int sec;
};
```

```
struct STAMP;
{
   DATE date;
   TIME time;
};

STAMP stamp;
```


Declaring Nested Structures

- It is possible to nest the same structure type more than once in a declaration.
- For example, consider a structure that contains start and end times for a job.
- Using STAMP structure, we create a new declaration, as shown below, and then define a variable that uses it.

```
struct JOB
{
   …
   STAMP startTime;
   STAMP endTime;
};

JOB job;
```


Declaring Nested Structures

- The major advantage of declaring each of the structures separately is that it allows much more flexibility in working with them.
- For example, with DATE declared as a separate type declaration, it is possible to pass the date structure to a function without having to pass the rest of the STAMP structure.

Referencing Nested Structures

- When you access a nested structure, you must include each level from the highest (stamp) to the component being referenced.
- The complete set of references for stamp is shown below.
- The last two references are to job.

```
stamp
stamp.date
stamp.date.month
stamp.date.day
stamp.date.year
stamp.time
stamp.time.hour
stamp.time.min
stamp.time.sec

job.startTime.time.hour
job.endTime.time.hour
```

Nested Structure Initialization

- Initialization of a nested structure follows the rules mentioned for a simple structure.
- Each structure must be initialized completely before proceeding to the next member.
- Each structure is enclosed in a set of braces.
- For example, to initialize stamp, first we initialize date, and then time, separated by a comma.
- To initialize date, we provide values for month, day, and year, each separated by commas.
- We can initialize the members of time.
- A definition and initialization for stamp is shown below.
  STAMP stamp = { { 05, 10, 1936}, { 23, 45, 00 } };

Structures Containing Arrays

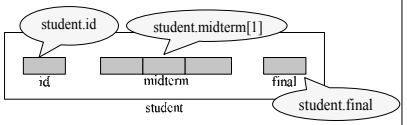
- Structures can have one or more arrays as members.
- The arrays can be accessed either through indexing or through pointers, as long as they are properly qualified with the member operator.

Declaring Arrays for Structures

- As we saw with nested structures, an array may be included within the structure or may be declared separately and then included.
- If it is declared separately, the declaration must be complete before it can be used in the structure

Declaring Arrays for Structures

- To see an example of this concept, consider the structure in this figure, which contains the student identifier, three midterm scores, and the final exam score.




```
// Global Declarations
  struct STUDENT
        {
            int    id;
            int    midterm[3];
            int    final;
        };
                .
                :
// Local Declarations
   STUDENT    student;
```

Referencing Arrays in Structures

- How we declared the structure, each element will have the same reference.
- We refer to the structure first and then to the array component.
- When we refer to the array, we can use either index or pointer notation.

Referencing Arrays in Structures

- The index applies to elements within an array, so it must follow the identifier of an array.
- In our student example, there is one array, an array of midterm scores.
- Each of its elements can be referenced with an index, as shown below.

```
student
student.id
// A pointer constant
student.midterm
student.midterm[ j ]
student.final
```


Referencing Arrays in Structures

- We have already shown how to refer to fields in a structure using the selection operator (→).
- When one structure contains an array, we can use a pointer to refer directly to the array elements.
- For example, given a pointer to integer, pScores, we could refer to the scores in student as shown on the following page.

```
pScores = student.midterm;
totalScores = *pScores + *(pScores + 1) + *(pScores + 2);
```


Array Initialization in Structure

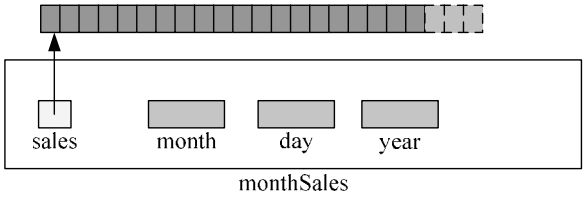
- The initialization of a structure containing an array simply requires extending the rules for structure initialization to include the initialization of the array.
- The array is a separate member, its values must be included in a separate set of braces.
- For example, the student structure can be initialized as shown below.
  STUDENT student = { 1234, { 92, 80, 70 }, 87 };

Structure Containing Pointers

- Not surprisingly, a structure can have pointers as members.
- In fact, pointers are very common in structures. The use of pointers can save memory.
- For example, suppose that we wanted to record the daily sales for a month.
- We could design a structure that contained an array of 31 floats, but the number of days ranges from 28 to 31, depending on the month.
- Another design would use a pointer to an array of floating-point variables that are dynamically allocated depending on the number of days in the month.

Structure Containing Pointers

- This structure is shown in following figure.



Structure Containing Pointers

- The structure declaration is shown below.

```
struct monthlySales
{
  float      *sales;
  int month;
  int day;
  int year;
};
```

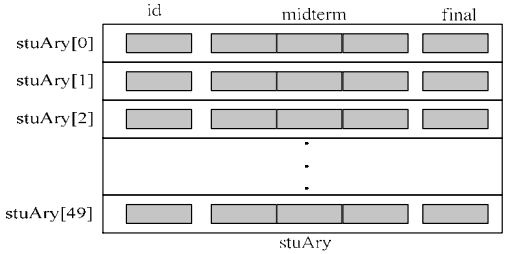
- Given a variable named may, we would then refer to a given day as shown below.
  may.sales + i
- In this case, i is the day of the month relative to zero.

Array of Structures

- As a programmer, you will encounter many situations that require you to create an array of structures.
- To name just one example, you would use an array of students when you are working with a group of students and the data are stored in a structure.
- By putting the data in an array, you can quickly and easily work with the data to calculate averages.

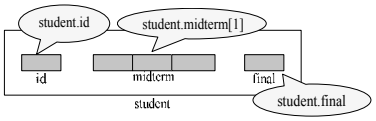
Array of Structures

- Let's create an array to handle the scores for up to 50 students in a class.
- This figure shows how such an array might look.



Array of Structures

- A structure is a type, you can create the array just as you would create an array of integers.
- The code is shown below.
  STUDENT stuAry[50];
- To access the data for one student, you have to refer only to the structure name with an index or a pointer, as shown below.
  stuAry[i] *pStu




```
// Global Declarations
 struct STUDENT
        {
          int    id;
          int    midterm[3];
          int    final;
        };
                 :
// Local Declarations
 STUDENT    student;
```


Array of Structures

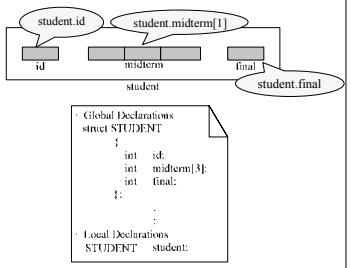
- For example, let's write a short segment of code to compute the average for the final exam.
- We use a for loop since we know the number of students in array.

```
int                totScore = 0;
float          average;
STUDENT  *pStu;
STUDENT  *pLastStu;
…
pLastStu     = stuAry + 49;
for (pStu = stuAry; pStu <= pLastStu; pStu++)
   totScore += pStu->final;
average       = totScore / 50.0;
```


Array of Structures

- To access as individual element in one of the student's arrays, such as the second midterm for the fifth student, you must use an index or pointer for each field, as shown below.

  stuAry[4].midterm[1];



Array of Structures

- To access students' midterms with pointers, you must use one index or pointer for the array.
- You must have a second index or pointer for the midterms.
- The code to compute the average for each midterm is shown below.
- We use a separate array, midTermAvrg, to store the average for the midterms.

Array of Structures

- In this example, we use indexes to access the midterms and pointers to access the students.

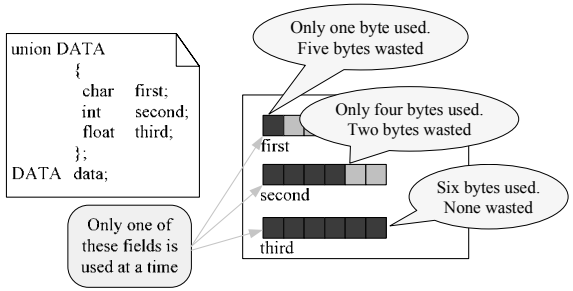
```
float       midTermAvrg[ 3 ];
int                 sum;
STUDENT             *pStu;
STUDENT             *pLastStu;
…
pLastStu    =   stuAry + 49;
for ( i = 0; i < 3; i++ )
{
  sum = 0;
  for ( pStu = stuAry; pStu <= pLastStu; pStu++ )
    sum += pStu->midterm[ i ];
  midTermAvrg[ i ] = sum / 50.0;
} // for i
```


Unions

- The union is a construct that allows a portion of memory to be shared by different types of data.
- Imagine that we must use a construct that can hold either an integer, a float, or a character, but not more than one at the same time.
- For example, at one point in the program the data might be a character and then later, in the same area, the data might be an integer.

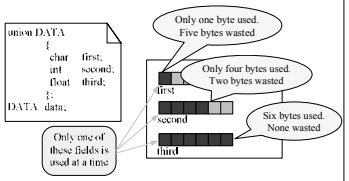
Unions

- This construct is shown in following figure.



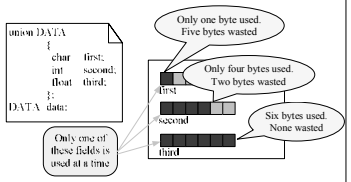
Unions

- When a union is defined, C++ reserve enough room to store the largest data object in the construct.
- In this figure, the size of the character is one, the size of the integer is four, and the size of the floating-point number is six.
- C++ will always reserve six bytes of storage for this construct, regardless of what type of data is currently stored in it.



Unions

- The format for the union should look familiar.
- With the exception of the keywords struct and union, these two structures are syntactically identical.
- The results are dramatically different.
- The declaration for the union is shown in this figure.

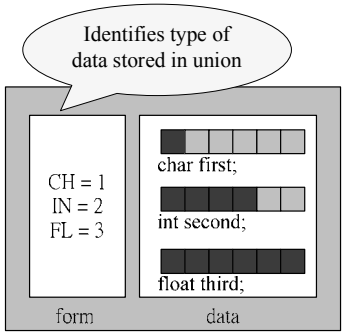


Unions

- We use the member operator (.) to reference any of the formats in the union.
- We did this because the union has the same form as the structure, and we must tell C++ which member we are referencing.
- In other words, all three members occupy the same space, we still need to identify which type format we are dealing with.

Unions

- The typical solution is to provide an embedded code in the structure to indicate the type of data present.
- Let's modify our data example to add a category code we will call form.
- The structure is shown in this figure.



Unions

- We implement this structure as shown below.

```
enum   FORM { CH = 1, IN = 2, FL = 3};
union   DATA
{
  char        first;
  int         second;
  float       third;
};

struct INFORMATION
{
  FORM     form;
  DATE     data;
};

INFORMATION information;
```


Unions

- First we create an enumerated type to give the forms mnemonic names.
- We create a structure that declares the three different types present.
- This structure is included in the INFORMATION structure.

Unions

- When our program assigns data to the union, it must also initialize the union's form.
- The program must know what type of data it is creating at any given point, this is not difficult.
- For example, assume that we are reading what we know must be a piece of character data into the union.
- We would use the following code:
  cin >> information.data.first;
  information.form = CH;