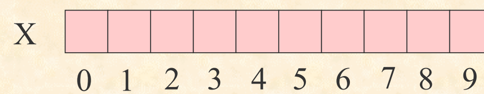
ARRAYS - INTRODUCTION

- An array is a **collection** of variables of the **same type** referenced by a **single name**
  - Consists of **contiguous** memory locations
  - May be single or multidimensional
  - Each array element is **indexed** by its position in the array
    Here is an array X with 10 elements



0 1 2 3 4 5 6 7 8 9

- C++ supports array types allowing array variables to be declared

ARRAY DECLARATION

- Type **name**[size]; // for 1 dimensional array
  where size is a non negative constant
- An array of 10 integers is declared as

```
int x[10];
```

  - The square brackets denote that the variable x is of an array type
  - The constant 10 denotes that the array has ten elements – the array size is 10
  - The size of the array given in the declaration must always be a constant

Accessing Array Elements

- To access a particular array element the index operator is used

```
int a;
a = x[1];
```

- will get the value stored in an array x at index 1

- The index operator is denoted by square brackets and takes an integer expression as its argument

- A value can be stored into an array using the same operator on the left side of an assignment

```
x[4] = 10;
```


Notes on C++ Arrays

- Given the declaration

```
int x[10];
```

  the array x is indexed from 0 - 9

  C++ array elements begin at index 0 and end at index size-1, where size is the number of elements in the array

  Hence

  x[0], x[1], . . ., x[8], x[9], are all **valid**

  x[10], x[11], . . . are all **invalid**

  x[-1], x[-2], . . . are all **invalid**

- **BE VERY CAREFUL** - these are not reported as errors by the compiler and can cause runtime side-effects which will probably crash the program

C++ Index

- The index expression used must evaluate to an integer

  Therefore the following expressions are valid given a, b, c, y are of type int:

  x[a*4];

  x[a + b + c];

  x[y[5]];

- **Note** - It is up to the programmer to make sure that the index expression denotes a valid element

Array Program Example

```
#include  <iostream.h>
const int ARRAY_SIZE = 10;
void main ()
{
    int numbers[ARRAY_SIZE];
    int i;
    //  store values in array
    for (i=0; i < ARRAY_SIZE; i++)
        numbers[i] = i*i + 10;
    //  print values from array
    for (i=0; i<ARRAY_SIZE; i++)
        cout << numbers[i] << endl;
}
```

Array Types

❒ Arrays in C++ can be constructed of any type

```
long n[10];
double x[100];
char c[256];
int I[130];
```

❒ Storage Requirements to hold an array

total bytes = size_of_array * **sizeof**(type)

e.g. Total bytes for array I is

130 * **sizeof**(int)

C++ Array Differences

❒ Note - unlike Pascal and Modula II

- the index starts at zero to size-1
- index is any constant expression that evaluates to type int (may be negative)
- no bounds checking

these give faster compilation and execution

Array Addressing

- The array name is the **address** (position) of the first element

  Given

  int x[10];

  the array x is stored at memory location **base_address**

  so x[index] is stored at memory location

  base_address + index

  Therefore
  - x[0] is stored at memory location **base_address**
  - x[4] is stored at memory location **base_address + 4**

Character Arrays

- An array of characters can store a **string**

  char str[100];

  A **string** is a sequence of characters terminated by the null character (zero), '\0'

- **Example**

  "hello" is an array of 6 characters (not 5)

  'h' , 'e' , 'l' , 'l' , 'o' , '\0'

- **String constants** appear in double quotes

  "this is a string constant"

Initializing Arrays

- As with variables arrays may be initialized on definition

```
int numbers[10] = {0,1,2,3,4,5,6,7,8,9};
```

This sets the value of the array element to the index

- The declarations may be **unsized**

```
int numbers[] = {0,1,2,3,4,5,6,7,8,9};
```

The compiler determines the size of the array from the number of elements

- A string constant syntax may be used

```
char string1[] = {'h','e','l','l','o','\0'};
char string2[]= "hello";
```

In string2 a **null** character is automatically added to the end of the sequence to make its size 6

- This is **not** a string but an array of characters with size 5

```
char letters[] = {'h','e','l','l','o'}
```


Using Arrays

- Consider the problem of counting the frequency each letter of the alphabet appears in a piece of text

To count letter frequencies

1. create 26 counters and set them to zero
2. while there are characters that can be read in
   - 2.1 If the character is a letter
   - 2.2 Add one to the corresponding counter
3. Output the values in each of the counters

A Bad Program

```
#include <iostream.h>
int main ()
{
    int count_a=0, count_b=0, count_c=0, etc;
    char ch;
    while (cin.get(ch))
        if (ch == 'a')
            count_a += 1;
        else if (ch == 'b')
            count_b += 1;
        and so …
    cout << "a " << count_a << endl;
    cout << "b " << count_b << end;
    and so on …
    return 0;
}
```


A Better Program Using Arrays

```
#include <iostream.h>
#include <ctype.h>    // char functions
int main ()
{
    const int n_letters = 26;
    //  declare and zero the array of counters
    int counts[n_letters];
    for (int i=0; i < n_lettters; i++)
        counts[i] = 0;
    // count the letter frequencies
    char c;
    while (cin.get( c ))
        if (isupper( c ))
            counts[int( c ) - int('A')] += 1;
        else if (islower( c ))
            counts[int( c ) - int('a')] += 1;
    //  List the results
    for (int ch_no = 0; ch_no < n_letters; ch_no++)
        if (counts[ch_no] != 0)  {
            cout << "char  "<< char(ch_no + int('A')) << " ";
            cout << counts[ch_no] << endl;
        }
    return 0;
}
```

A Program Using Stings

```
//  This program creates two arrays with a-z and A-Z
#include <iostream.h>
int main ()
{
char lowerAlpha[27];
char upperAlpha[27];
for (char I=0; I <26; I++)  {
lowerAlpha[I] = I + ‘a’;
upperAlpha[I] = I + ‘A’;
}
// add terminating null character
lowerAlpha[26] = ‘\0’;
upperAlpha[26] = ‘\0’;
cout << “lower case characters are: “ << lowerAlpha << endl;
cout << “upper case characters are: “ << upperAlpha << endl;
return 0;
}
```


Two Dimensional Arrays

- C++ supports 2D arrays

  2D arrays are equivalent to 1D array each of whose elements is an array, that is an **array of arrays**

Columns

Rows	0	1	2	3	4
0					
1					
2					

- A 2D array is declared in the following way:

  type **name**[row_size][column_size];

Two Dimensional Arrays

- Each dimension is specified by giving the size in square brackets
  Therefore a 2D array of integers may be
  int x[10][20];

- **Note**
  - The **index operator** is applied **twice** to access elements of a 2D array
    - The **first** application selects the **row** and the **second** application selects **element** in that row

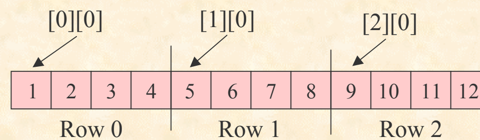
```
int x[10][20];          // declare 2D array
int y;
x[3][4] = 15;           // assign to element {3,4}
y = x[3][4];            // access element {3,4}
```


Program to Read Numbers Into 2D Array

```
#include <iostream.h>
int const rowsize = 3;
int const colsize = 4;
void main()
{
    int num[rowsize][colsize];
    //  read numbers 1-12 into array num
    for (int row = 0; row < rowsize; row++)
            for (int col = 0; col < colsize; col++)
    num[row][col] = (row*colsize) + col + 1;
    // print out array values
    for (int row = 0; row < rowsize; row++)
            for (int col = 0; col < colsize; col++)
    cout << num[row][col] << "  " << endl;
    }
}
```

2D Array Layout

- 2D and multidimensional arrays are stored in memory as single dimensional array
- They are stored in **row major order**



- Array element position is calculated using the following formula

  address = base_address

  + (number_of_columns * row_index + column_index)

  where base_address is the position [0][0]

  The required storage in bytes of a 2D array is

  row_size * column_size * sizeof(type)

N-Dimensional Arrays

- An array of any dimensions can be declared by specifying all the dimension sizes

```
int x[10][20][30];
int y[5][10][15][20];
```

- N-dimensional arrays are implemented as an array of array of array, etc
- It is always possible to take an n-dimensional slice from an array

```
x[1][2];     // Gives an array of int
y[1][2];     // Gives an array of arrays of int
```

Notes on Multidimensional Arrays

- Not popular because of their large storage requirements

  int array_4D[10][6][9][4];

  requires 2160 * sizeof(int) = 8640 given 4 bytes per int

- Storage increases exponentially with no. of dimensions

- Indexing is slower than with 1D arrays - involves more calculations

Initializing Multidimensional Arrays

- int [2][3] = {

      {1,2,3},
      {4,5,6}
  };

- char months[][15] = {

  "invalid month", "January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November",

  "December" };

  An **advantage** of unsized declarations:

  may lengthen or shorten the table without changing array dimensions

Array Parameters

- It is possible to pass an array as a function parameter

```
int function(int x[10])
{
        x[1] = 1; //  Watch out
}
```

  - **But** arrays **CANNOT** be passed by value so the array named is NOT a copy of the actual parameter
  - Instead x names the **SAME** array as the actual argument
  - Changing an element of x changes the actual argument array

- **Note** - For a one dimensional array parameter the array size can be omitted

```
int func(int x[]);
```

  func can be called with any one dimensional array as the argument

Array Parameters

- **Note** - it is up to the programmer to ensure that only valid array accesses are made in the function body

- N-dimensional arrays can be passed as argument

```
int func(int x[10][20]);
```

  When called the programmer must make sure that the actual argument array is of the correct size

  The first dimension size can be omitted but not the second

```
int func(int x[][20]);        // n by 20 elements
int func(int x[][]);          // error
```

- In the function body enough must be known about the size of the array to perform the indexing operations