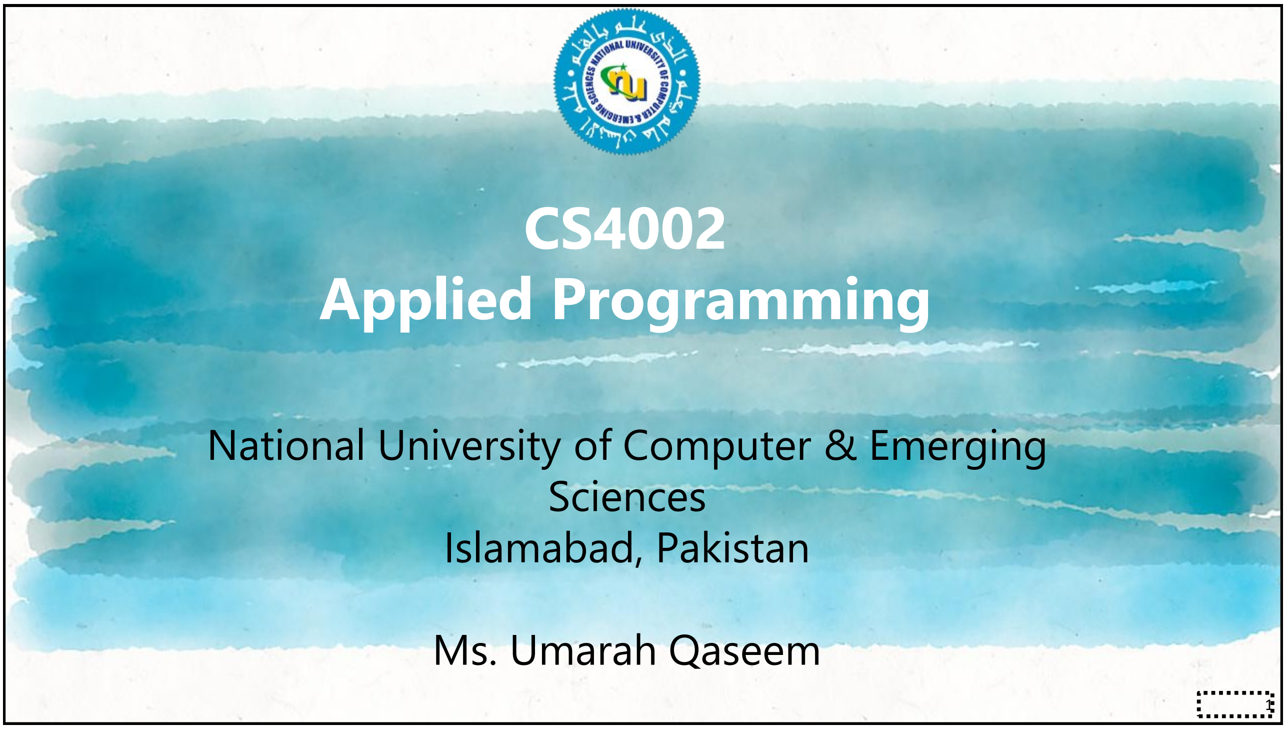
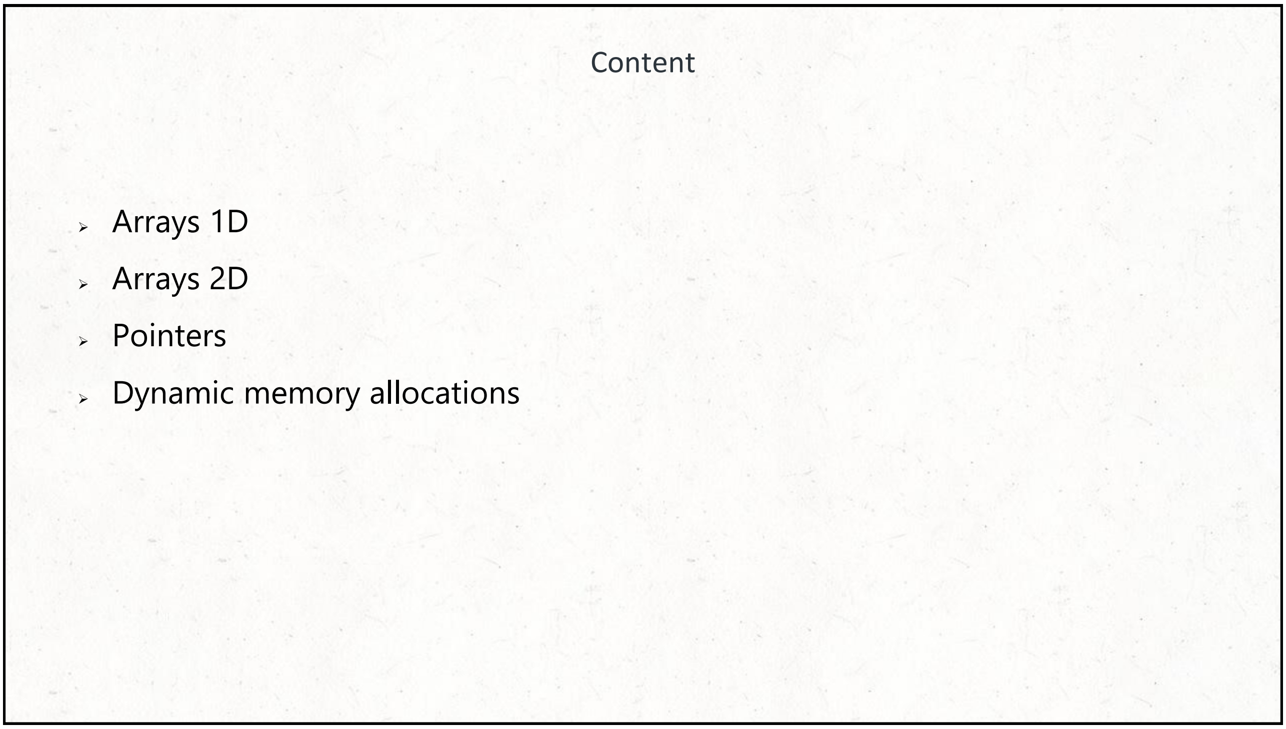


CS4002
Applied Programming

National University of Computer & Emerging Sciences
Islamabad, Pakistan

Ms. Umarah Qaseem

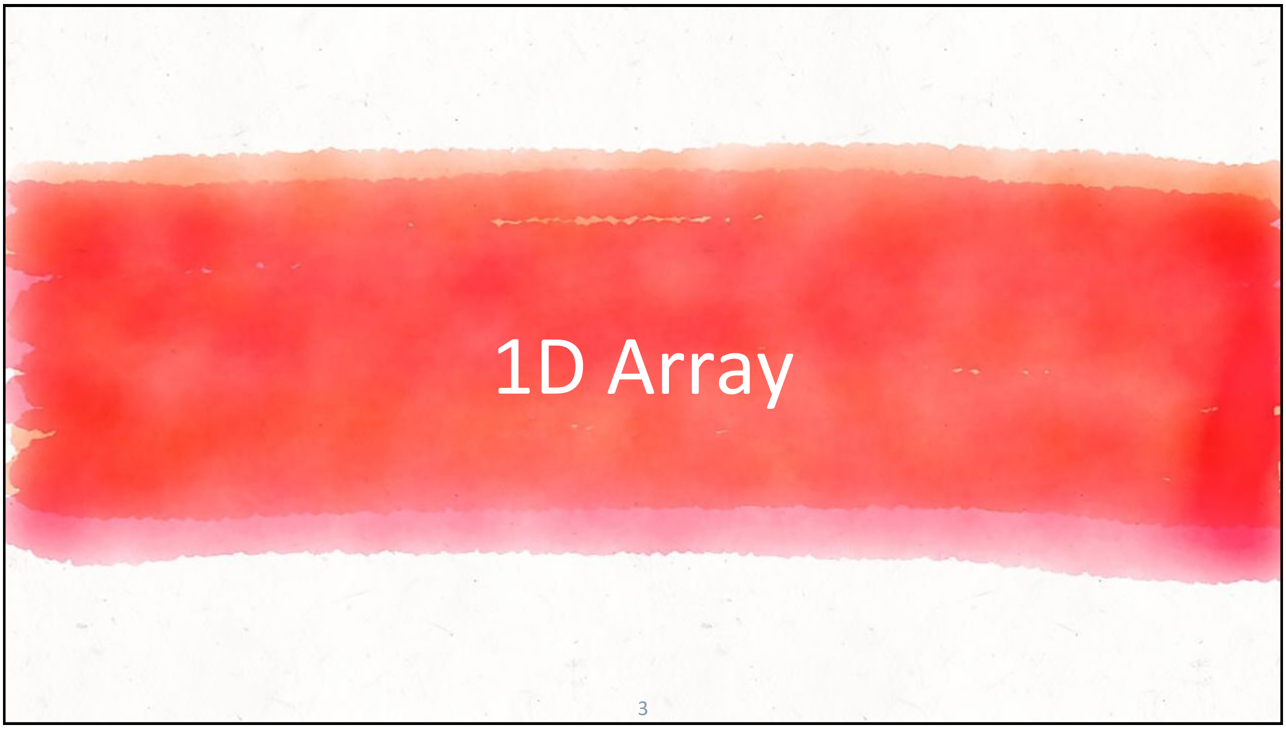
1



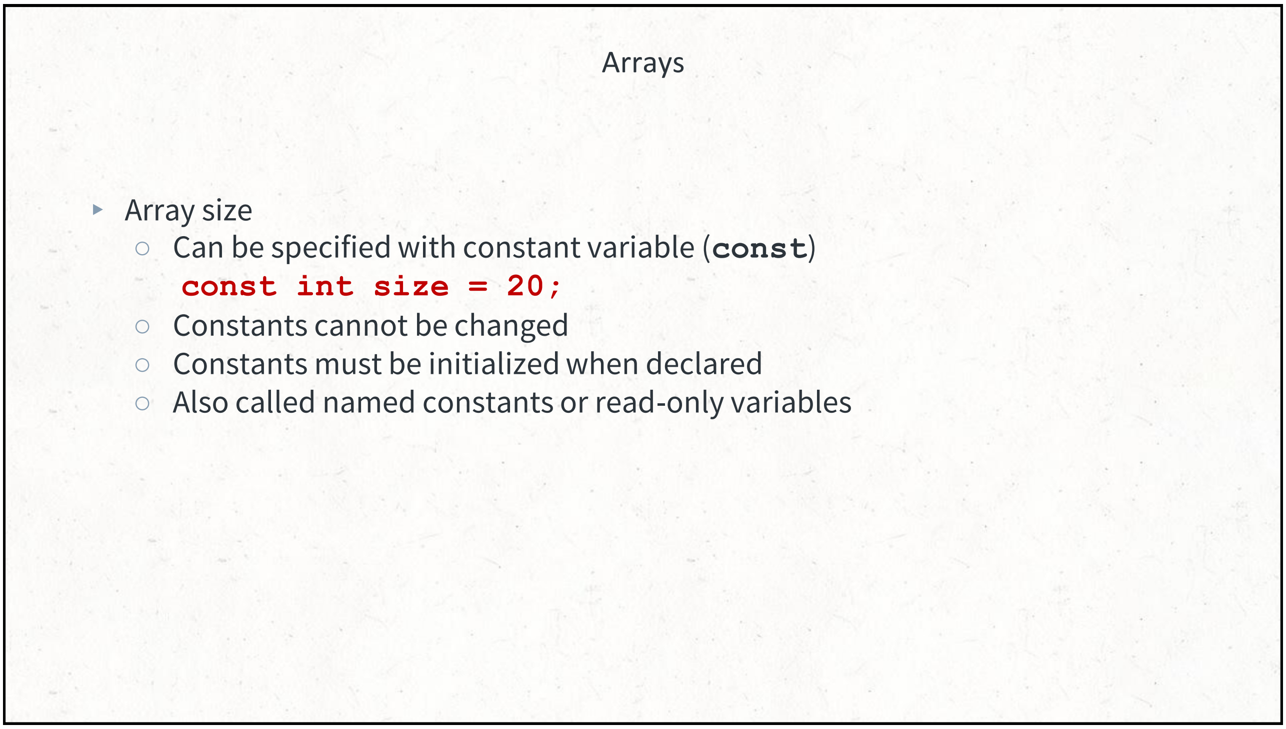
Content

- Arrays 1D
- Arrays 2D
- Pointers
- Dynamic memory allocations

2



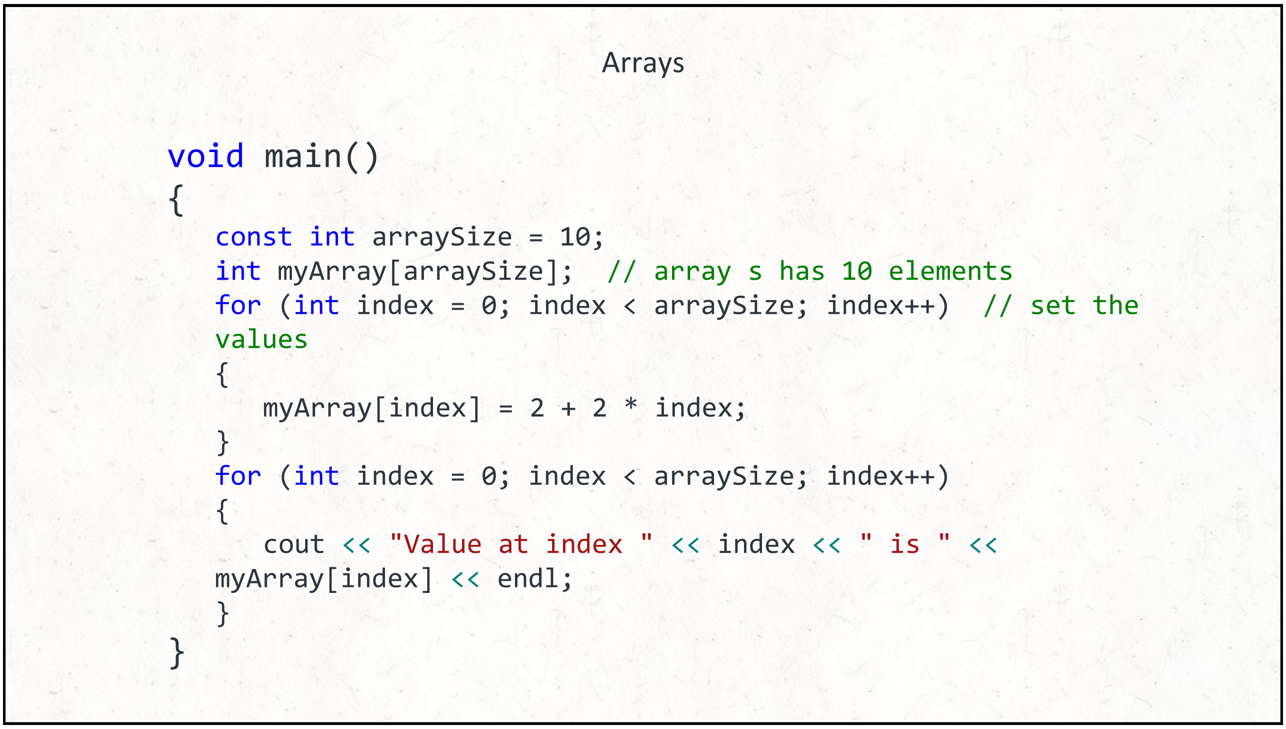
1D Array



Arrays

- Array size
  - Can be specified with constant variable (`const`)

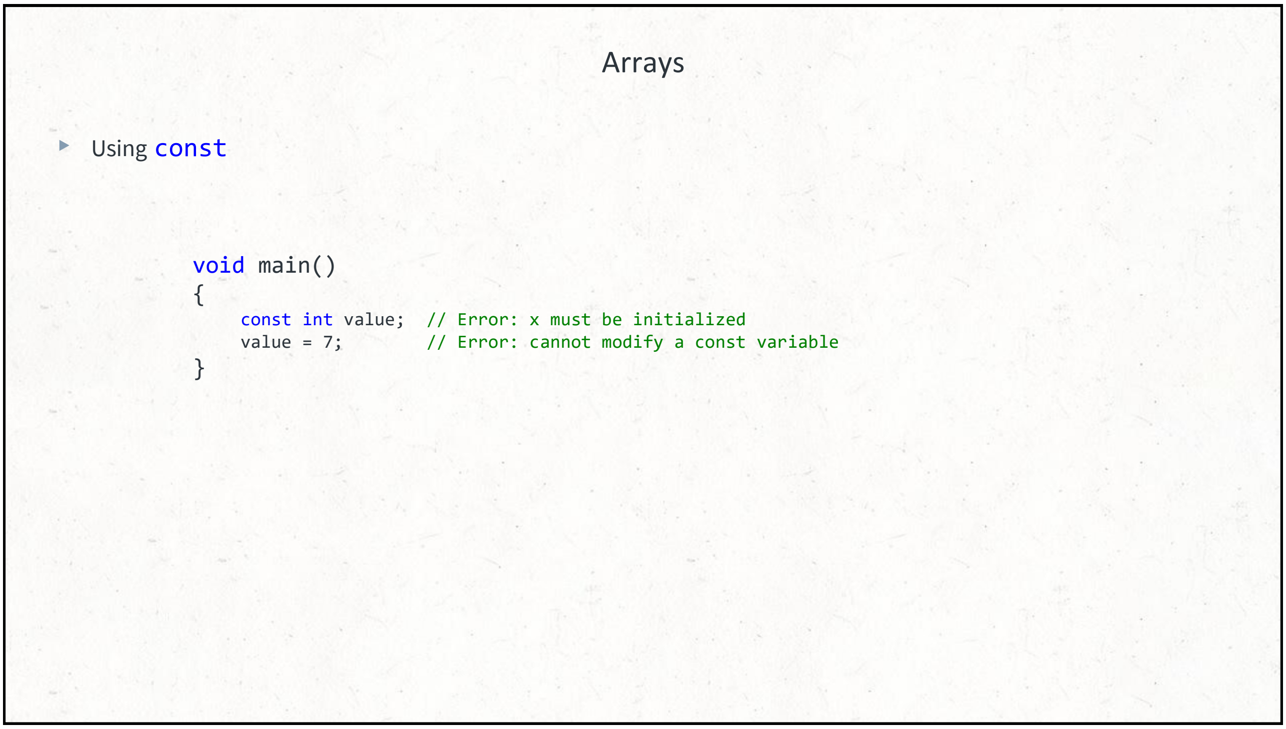
    ```
    const int size = 20;
    ```
  - Constants cannot be changed
  - Constants must be initialized when declared
  - Also called named constants or read-only variables



Arrays

```
void main()
{
   const int arraySize = 10;
   int myArray[arraySize];  // array s has 10 elements
   for (int index = 0; index < arraySize; index++)  // set the
   values
   {
      myArray[index] = 2 + 2 * index;
   }
   for (int index = 0; index < arraySize; index++)
   {
      cout << "Value at index " << index << " is " <<
   myArray[index] << endl;
   }
}
```

5

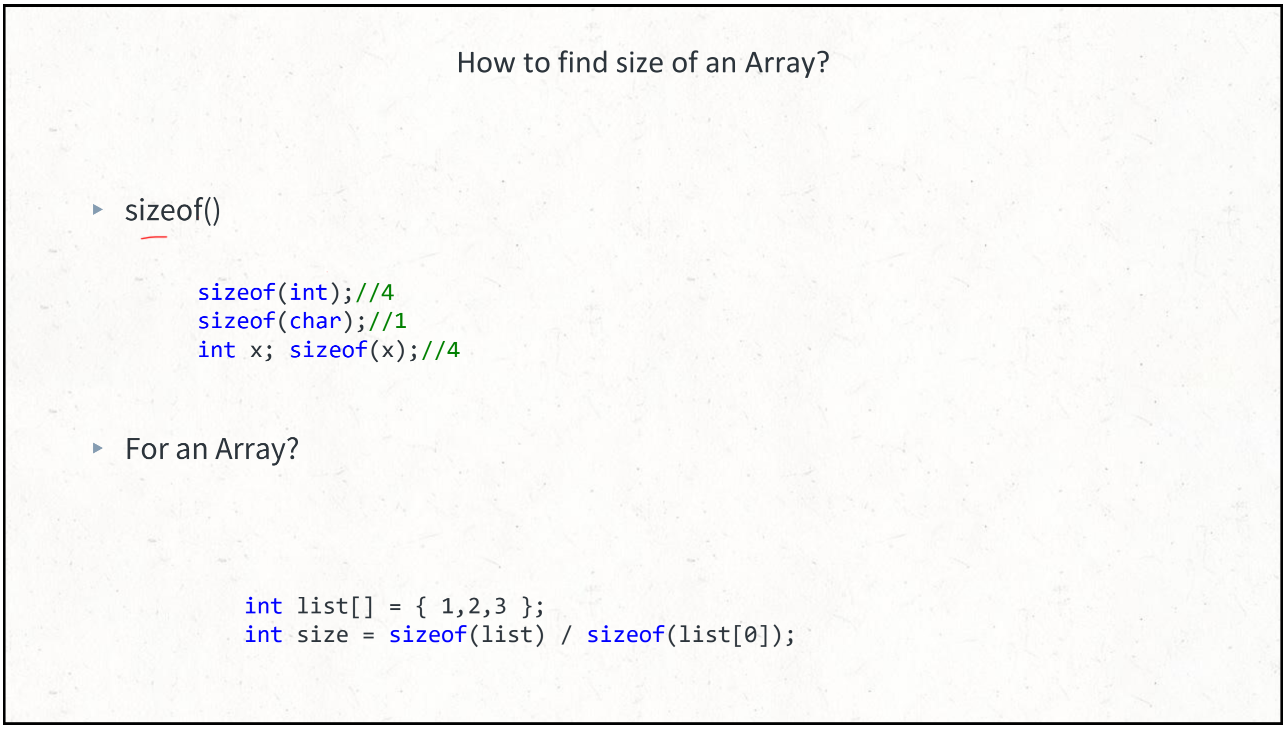


Arrays

- Using const

```
void main()
{
    const int value;  // Error: x must be initialized
    value = 7;        // Error: cannot modify a const variable
}
```

6



How to find size of an Array?

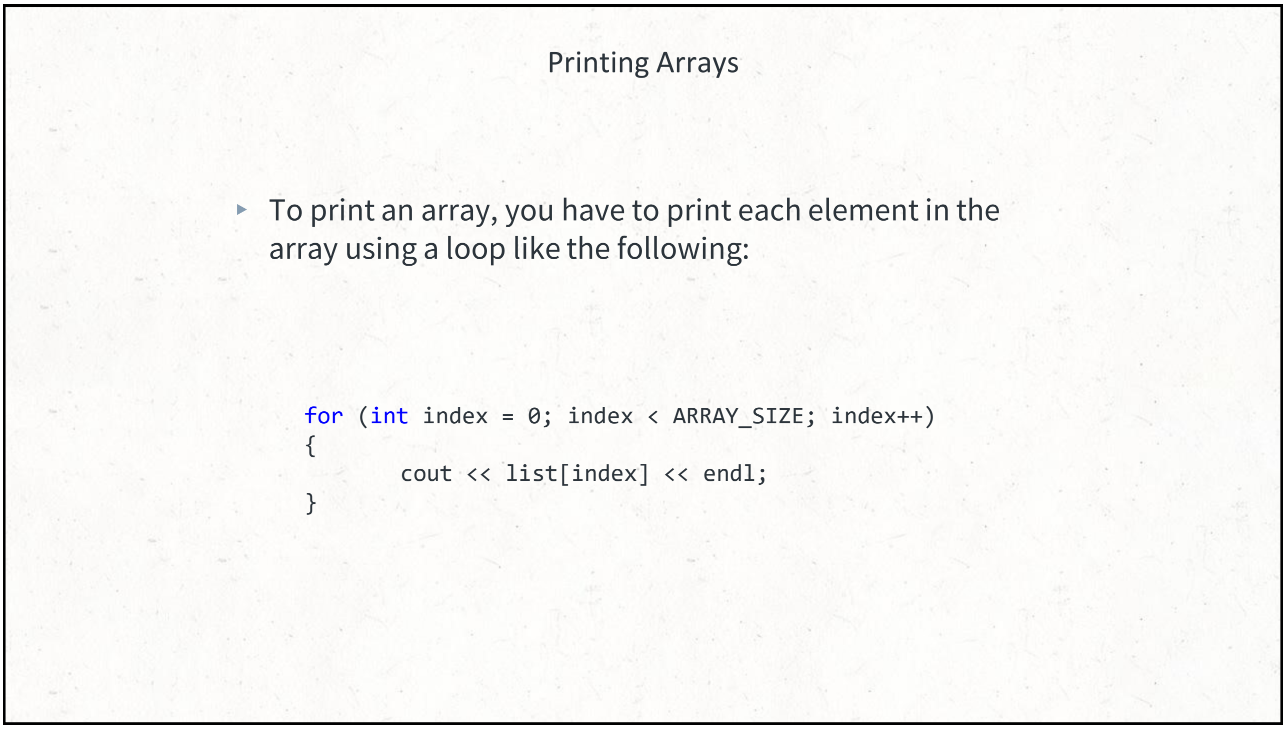
- sizeof()

```
sizeof(int);//4
sizeof(char);//1
int x; sizeof(x);//4
```

- For an Array?

```
int list[] = { 1,2,3 };
int size = sizeof(list) / sizeof(list[0]);
```

7

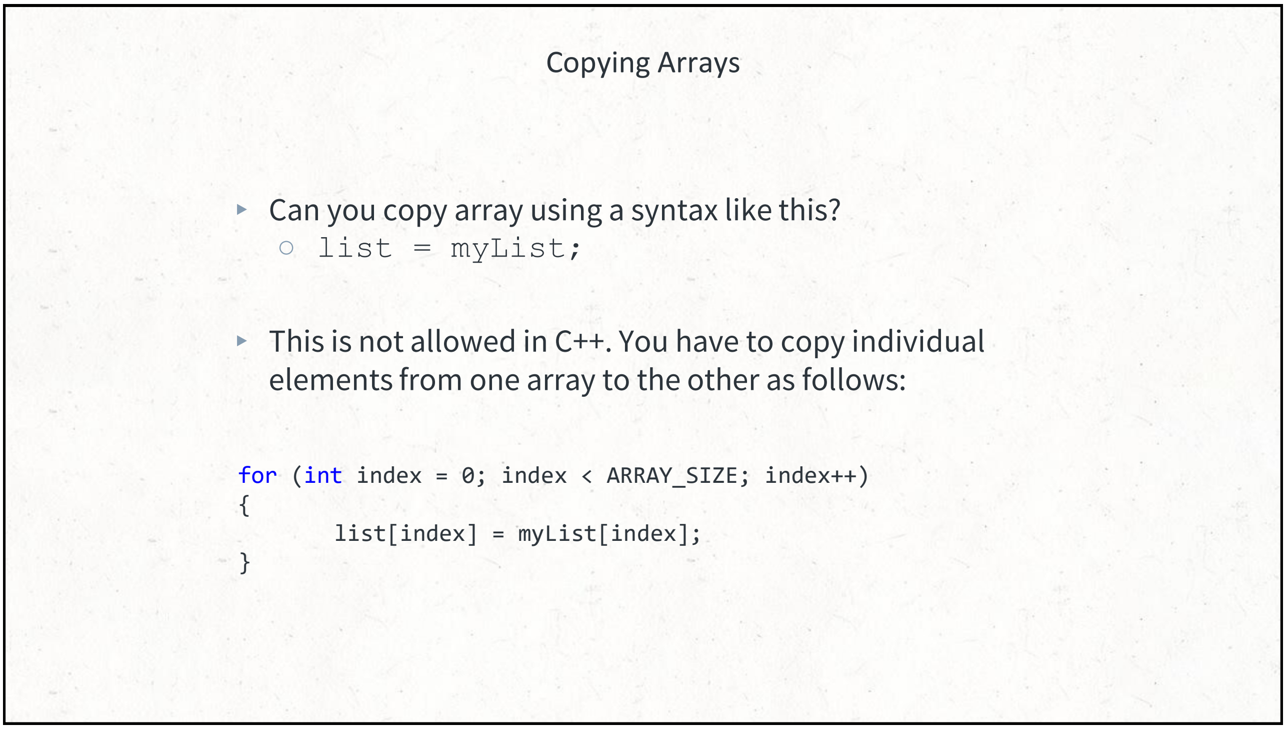


Printing Arrays

- To print an array, you have to print each element in the array using a loop like the following:

```
for (int index = 0; index < ARRAY_SIZE; index++)
{
        cout << list[index] << endl;
}
```

8

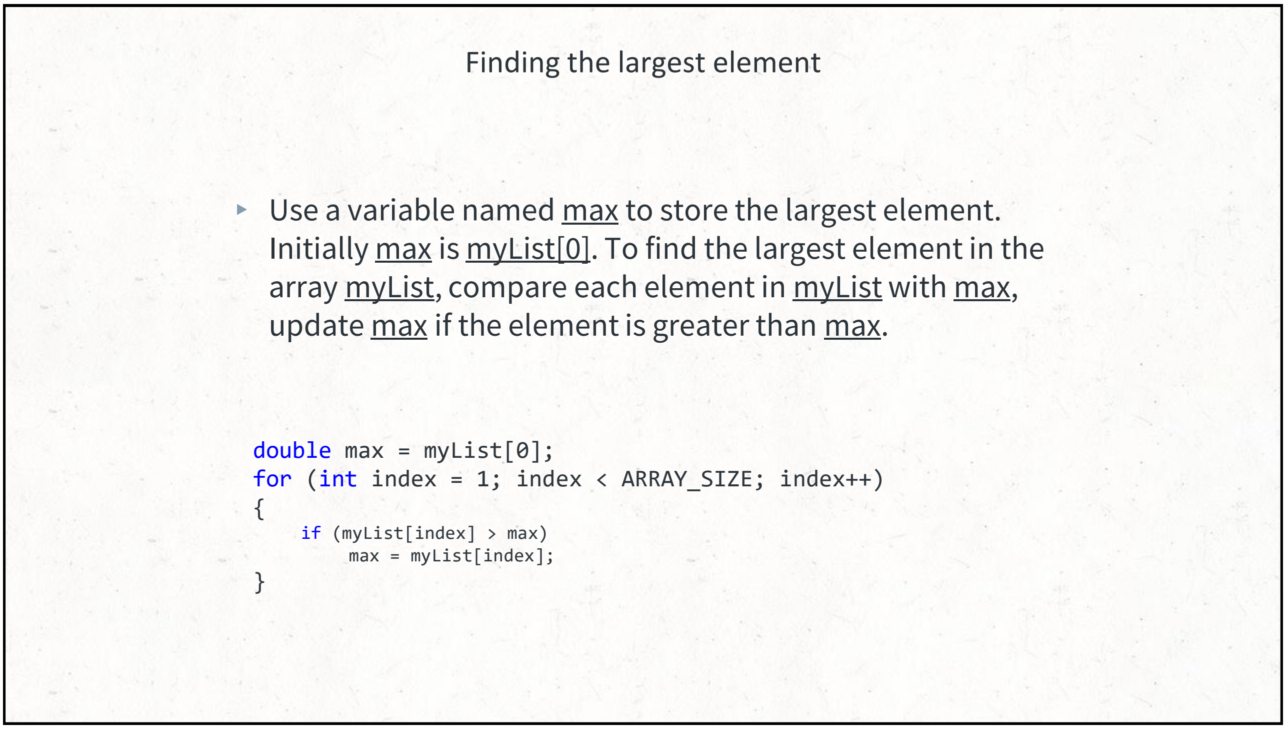


Copying Arrays

- Can you copy array using a syntax like this?
  - ○ `list = myList;`
- This is not allowed in C++. You have to copy individual elements from one array to the other as follows:

```
for (int index = 0; index < ARRAY_SIZE; index++)
{
        list[index] = myList[index];
}
```

9

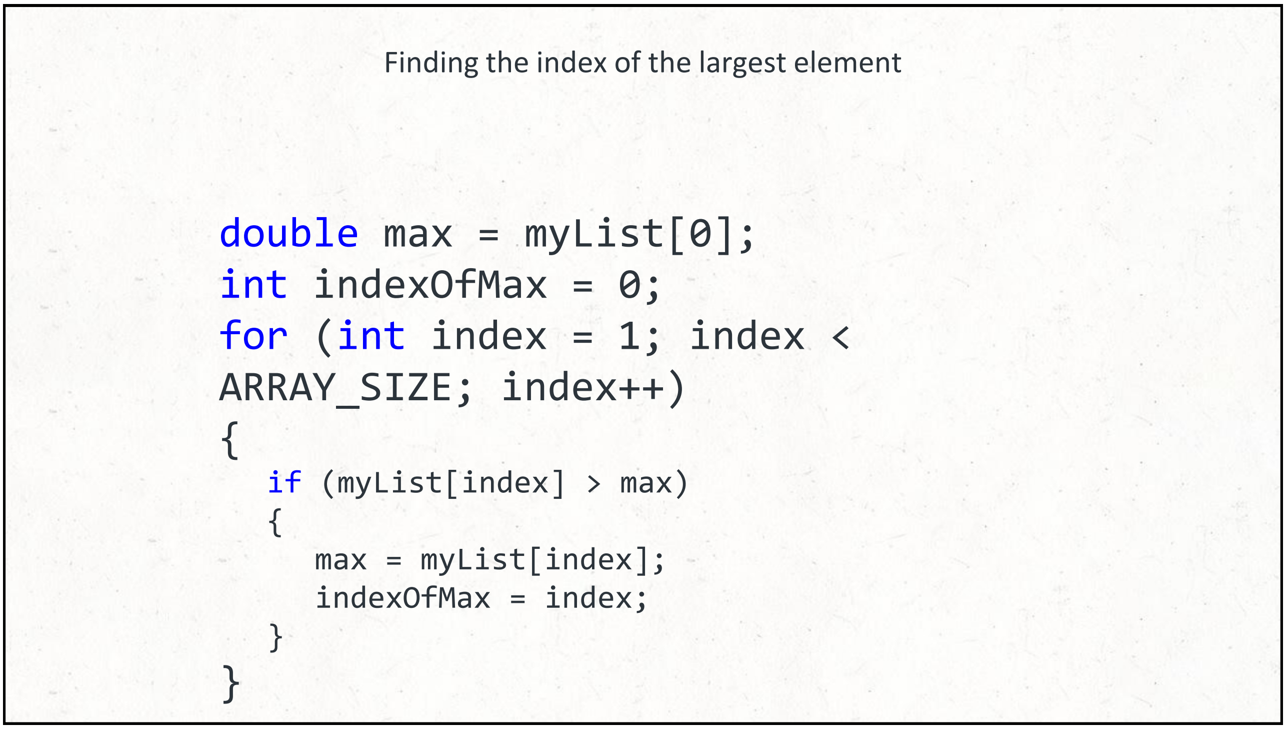


Finding the largest element

- Use a variable named max to store the largest element. Initially max is myList[0]. To find the largest element in the array myList, compare each element in myList with max, update max if the element is greater than max.

```
double max = myList[0];
for (int index = 1; index < ARRAY_SIZE; index++)
{
     if (myList[index] > max)
          max = myList[index];
}
```

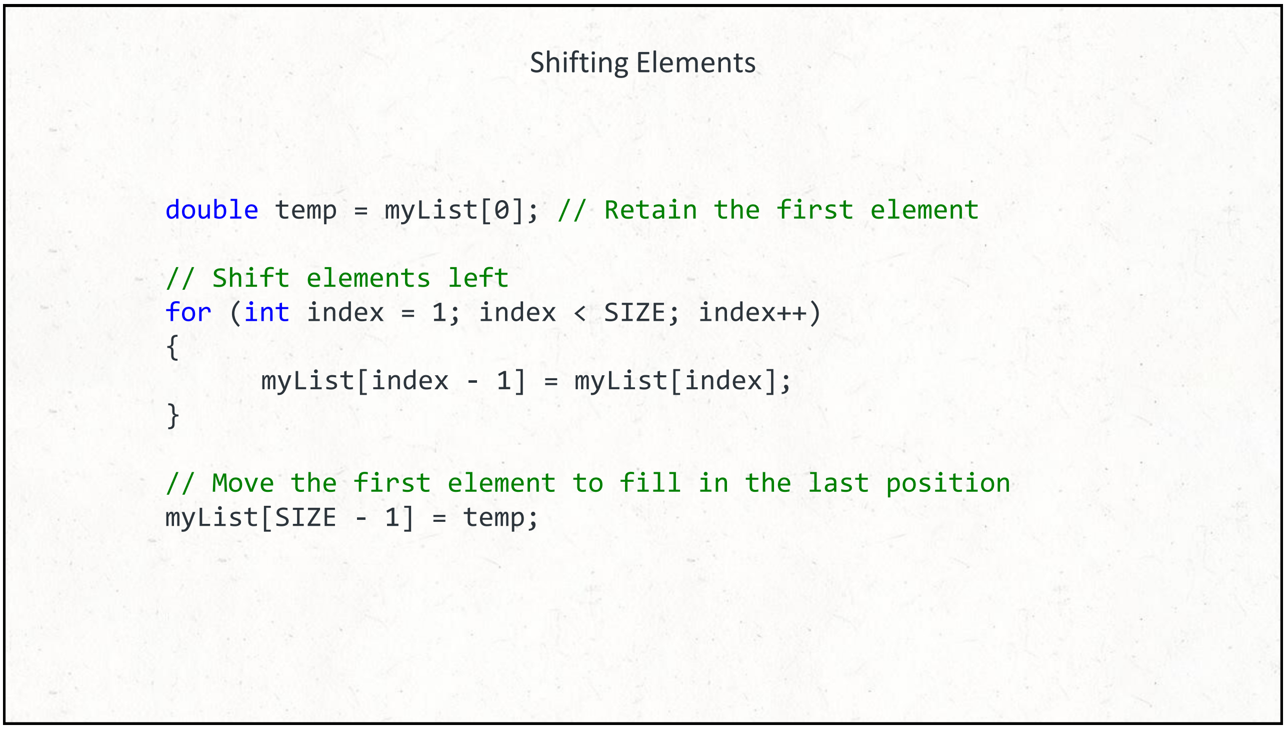
10



Finding the index of the largest element

```
double max = myList[0];
int indexOfMax = 0;
for (int index = 1; index <
ARRAY_SIZE; index++)
{
   if (myList[index] > max)
   {
      max = myList[index];
      indexOfMax = index;
   }
}
```

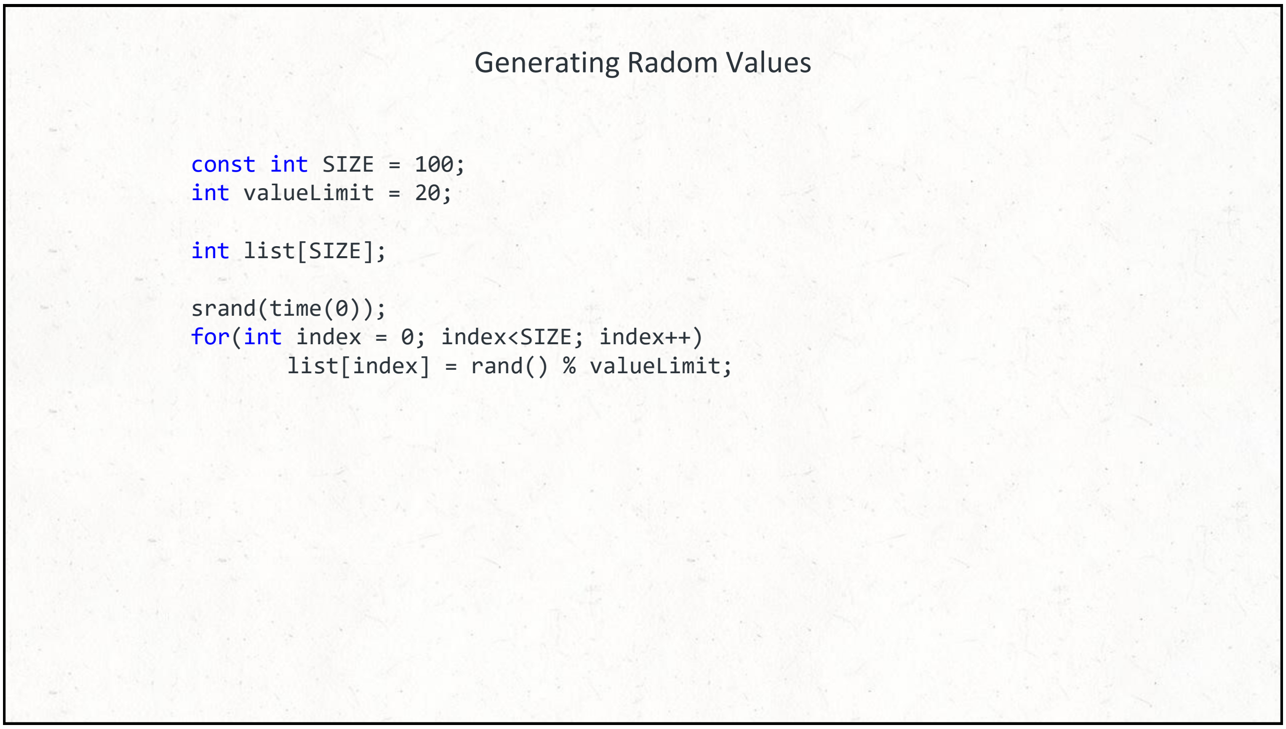
11



Shifting Elements

```
double temp = myList[0]; // Retain the first element

// Shift elements left
for (int index = 1; index < SIZE; index++)
{
      myList[index - 1] = myList[index];
}

// Move the first element to fill in the last position
myList[SIZE - 1] = temp;
```

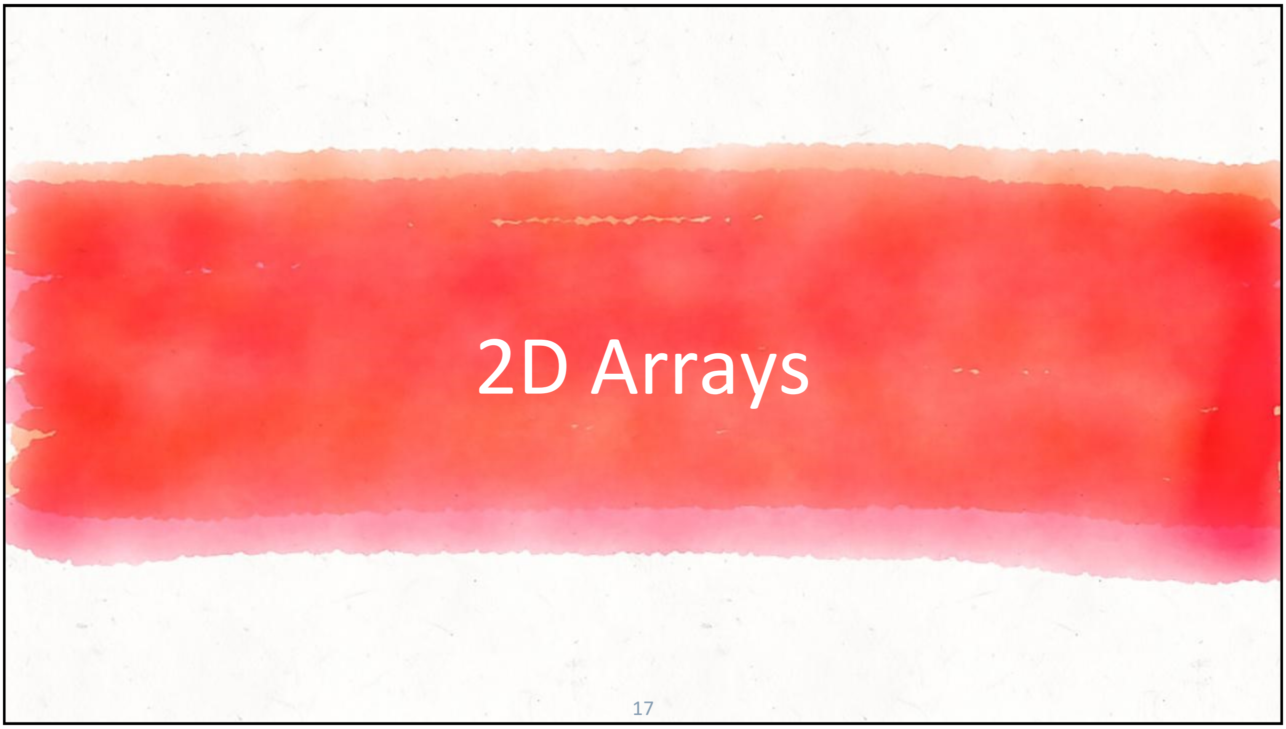
12



Generating Radom Values

```
const int SIZE = 100;
int valueLimit = 20;

int list[SIZE];

srand(time(0));
for(int index = 0; index<SIZE; index++)
        list[index] = rand() % valueLimit;
```

15



2D Arrays

17

Outline

- Two-Dimensional Array
- Declaring Two-Dimensional Array
- Initializing Two-Dimensional Array
- Accessing elements of Two-Dimensional Array
- Inputting elements of Two-Dimensional Array
- Outputting elements of Two-Dimensional Array
- Operations on two Dimensional arrays
- Matrices and operations performed

18

Two Dimensional Arrays

19

What is a Two Dimensional Array ?

- A two dimensional array is the arrangement of elements in rows and columns.
- It has two indices, one for row and other for column. First index represents the number of rows, second index represents the number of columns.

Example : int list[4][4];

ROWS

COLUMNS

78	90	56	45
89	88	76	99
22	69	97	24
34	60	44	56

Declaring a Two Dimensional Array

data-type name [no of rows] [no of columns];

integer constant/ literal

Example :

int List[4][4];

- Data type= integer
- Name= list
- No. of rows=4
- No. of columns=4

	0	1	2	3
0	List[0][0]	List[0][1]	List[0][2]	List[0][3]
1	List[1][0]	List[1][1]	List[1][2]	List[1][3]
2	List[2][0]	List[2][1]	List[2][2]	List[2][3]
3	List[3][0]	List[3][1]	List[3][2]	List[3][3]

Structure of a 2 D array

Two- and Multidimensional Arrays

- Two-dimensional array: collection of a fixed number of components (of the same type) arranged in two dimensions
  - ○ Sometimes called matrices or tables
- Declaration syntax:

dataType arrayName [intRowSize] [intColSize];

where `intRowSize` and `intColSize` are expressions yielding positive integer values, and specify the number of rows and the number of columns, respectively, in the array

Two-dimensional Arrays

sales	[0]	[1]	[2]	[3]	[4]
[0]					
[1]					
[2]					
[3]					
[4]					
[5]					
[6]					
[7]					
[8]					
[9]					

Accessing Array Components

- Syntax: **arrayName [intRowIndex] [intColIndex];**

  where `intRowIndex` and `intColIndex` are expressions yielding nonnegative integer values, and specify the row and column position

24

Accessing Array Components (cont'd.)

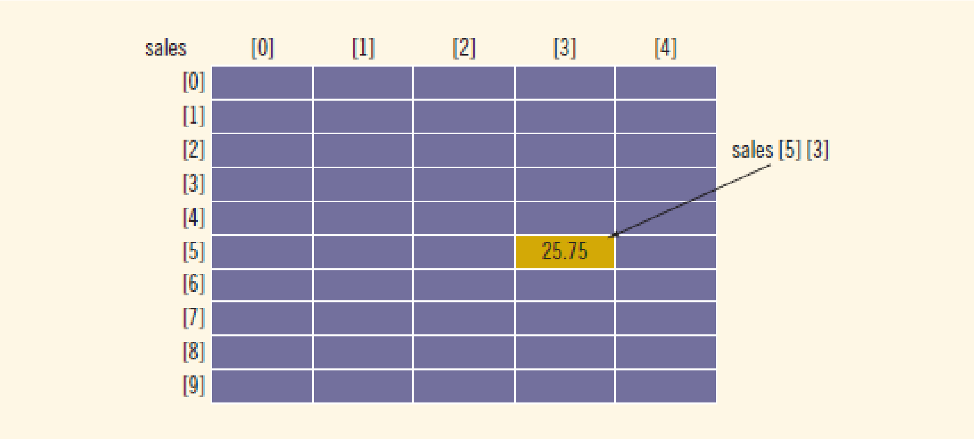
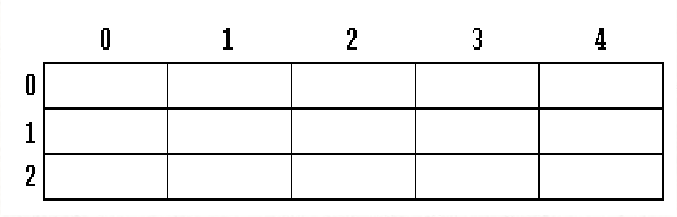


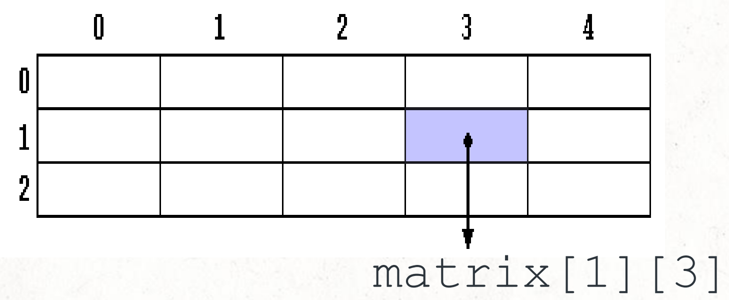
FIGURE 9-11 `sales[5][3]`

25

Multi-dimensional Arrays



```
int matrix[3][5]
```



26

Processing Two-Dimensional Arrays (cont'd.)

```
const int rowSize = 7;
const int colSize = 6;

int matrix[rowSize][colSize];
```

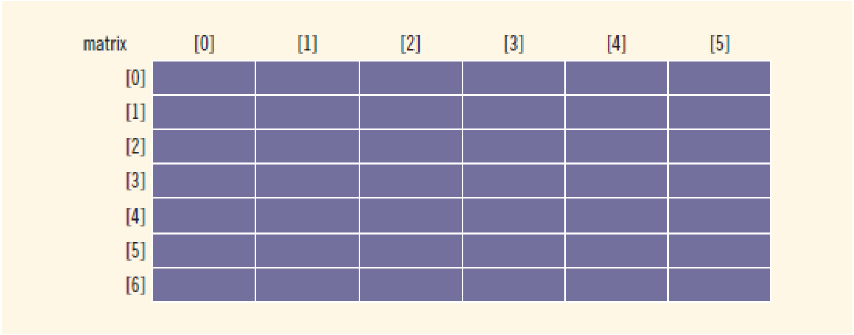


FIGURE 9-15 Two-dimensional array `matrix`

27

Multi-dimensional Arrays

- Very useful and practical, e.g.
  - Matrices
  - Images

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

1	2	3
4	5	6
7	8	9

  - Multiple loops to access individual elements

28

Initialization

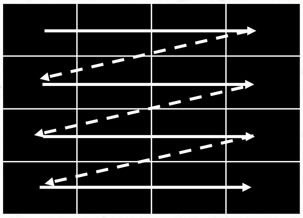
- To initialize the entire matrix to 0:

```
const int rowSize = 7;
const int colSize = 6;
int matrix[rowSize][colSize];

for (int row = 0; row < rowSize; row++)
{
    for (int col = 0; col < colSize; col++)
    {
        matrix[row][col] = 0;
    }
}
```

29

Initialization

Row wise input



- To initialize the matrix with user input:

```
const int rowSize = 7;
const int colSize = 6;
int matrix[rowSize][colSize];

for (int row = 0; row < rowSize; row++)
{
   for (int col = 0; col < colSize; col++)
   {
      cout << "Enter value for Row " << row << "and Column " << col;
      cin>>matrix[row][col];
    }
    cout << endl;
}
```

30

Activity: Initialize with random values

- To initialize the matrix with random numbers:

```
srand(time(0));
int randLimit = 100;

const int rowSize = 7;
const int colSize = 6;
int matrix[rowSize][colSize];

for (int row = 0; row < rowSize; row++)
{
   for (int col = 0; col < colSize; col++)
   {
      matrix[row][col] = 1 + rand() % randLimit;
   }
}
```

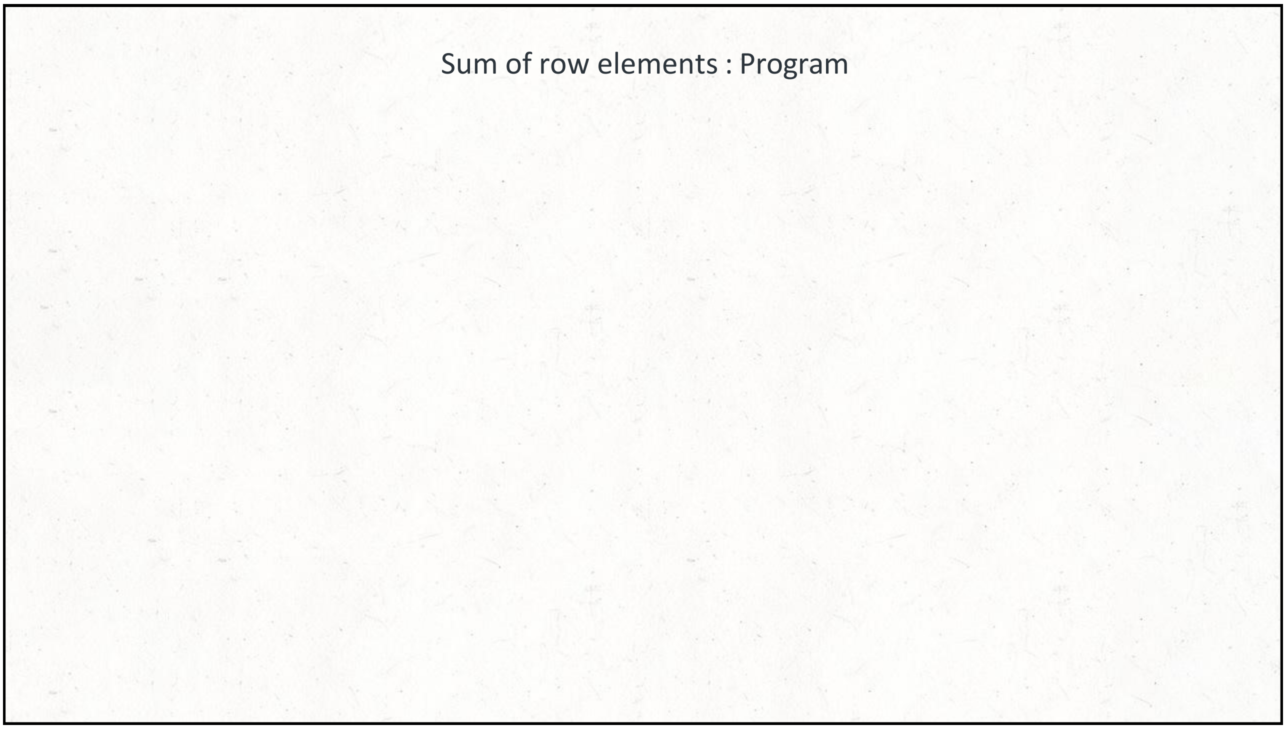
31

Print

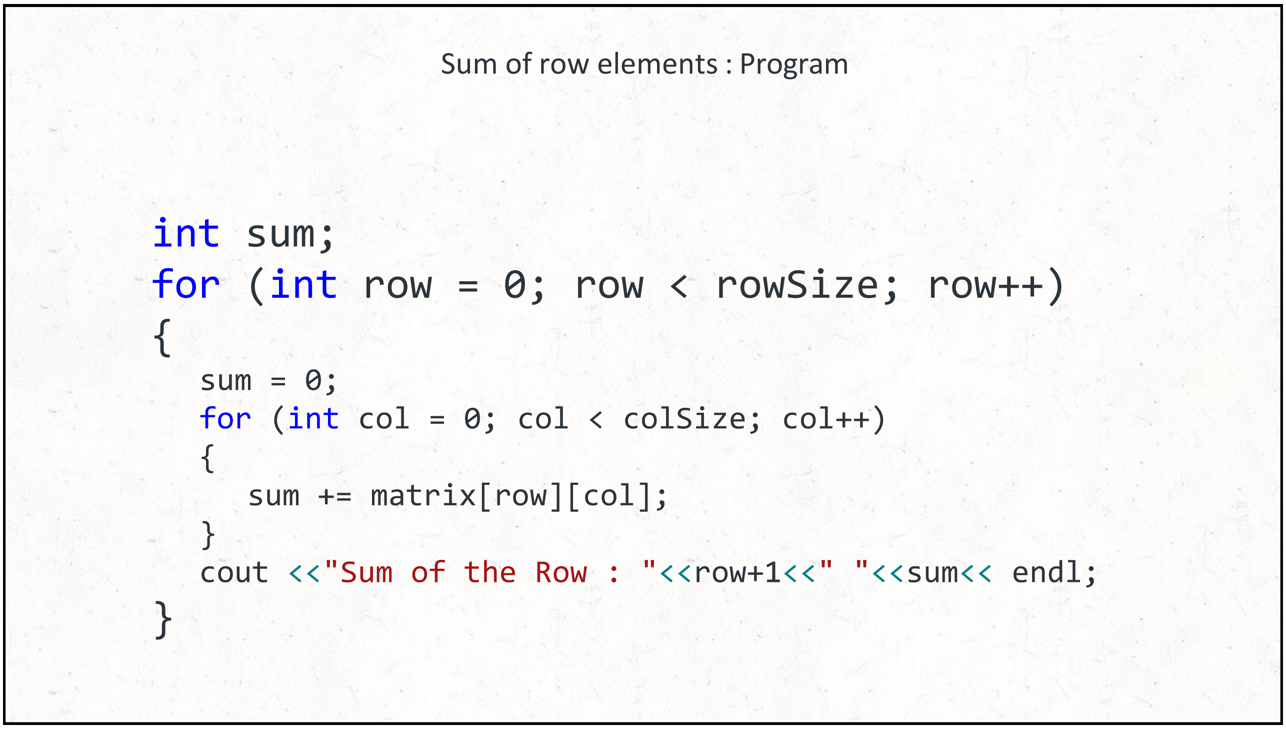
- To output the components of matrix:

```
for (int row = 0; row < rowSize;
row++)
{
   for (int col = 0; col < colSize; col++)
   {
      cout<<setw(5)<<matrix[row][col]<<" ";
   }
   cout << endl;
}
```

Operations on 2 D Arrays



Sum of row elements : Program



Sum of row elements : Program

```
int sum;
for (int row = 0; row < rowSize; row++)
{
   sum = 0;
   for (int col = 0; col < colSize; col++)
   {
      sum += matrix[row][col];
   }
   cout <<"Sum of the Row : "<<row+1<<" "<<sum<< endl;
}
```

Activity Sum of col elements : Program

36

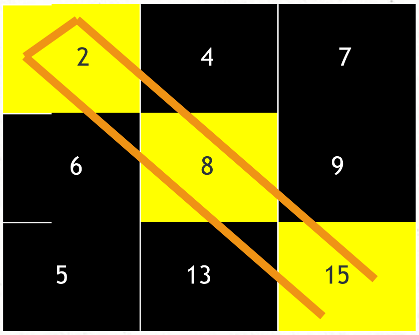
Solution: Sum of col elements : Program

```
int sum;
for (int col = 0; col < colSize; col++)
{
   sum = 0;
   for (int row = 0; row < rowSize; row++)
   {
      sum += matrix[row][col];
   }
   cout <<"Sum of the Column : "<<col+1<<" "<<sum<< endl;
}
```

37

Home Task: Matrix : Diagonal Elements

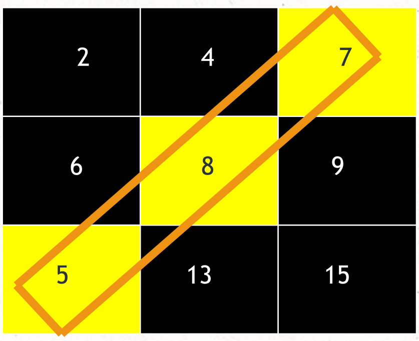
Elements of main diagonal



2	4	7
6	8	9
5	13	15

Sum = A[0][0]+A[1][1]+A[2][2]
= 2+8+15=25

Elements of alternate diagonal



2	4	7
6	8	9
5	13	15

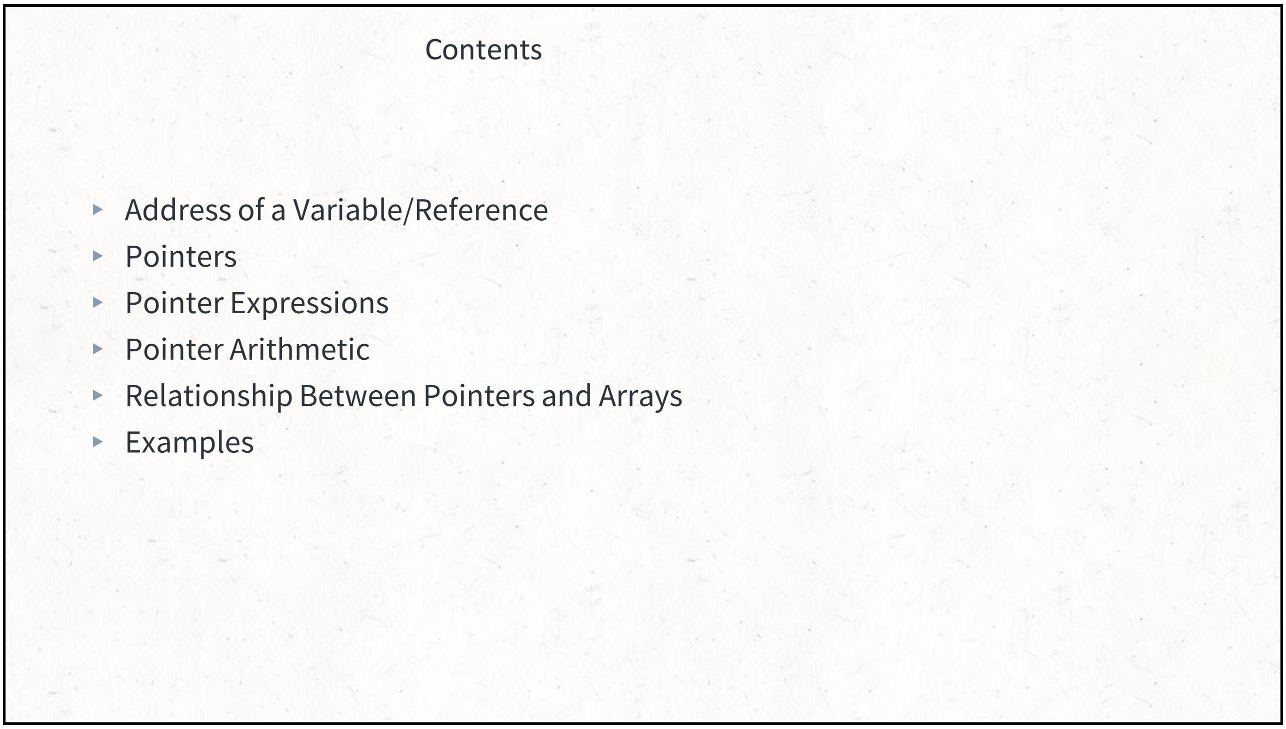
Sum = A[0][2]+A[1][1]+A[2][0]
= 7+8+5=20

38

Pointers

39

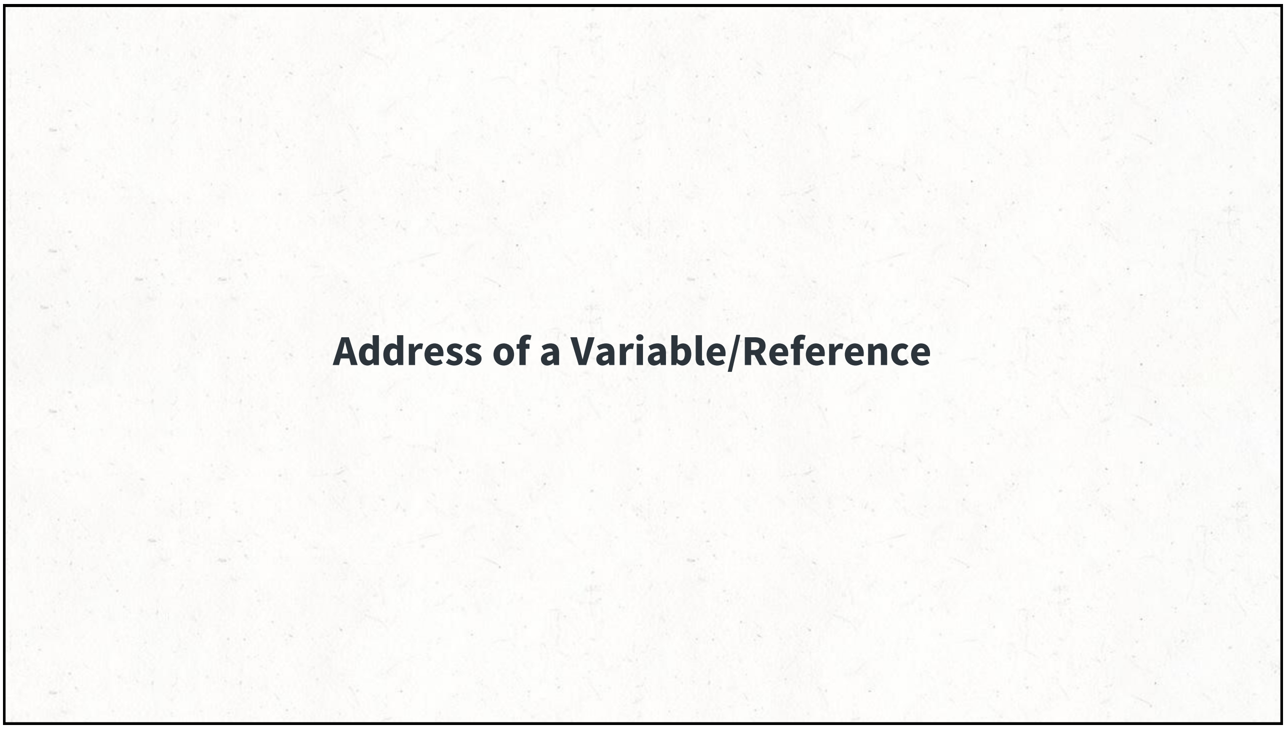
39



Contents

- Address of a Variable/Reference
- Pointers
- Pointer Expressions
- Pointer Arithmetic
- Relationship Between Pointers and Arrays
- Examples

40

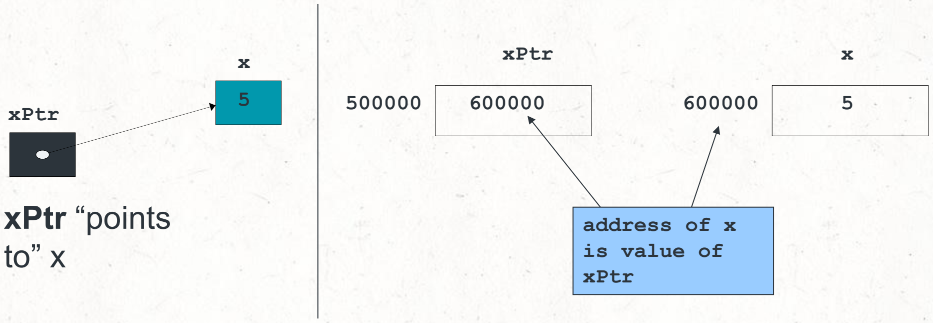


Address of a Variable/Reference

42

Pointers

```
void main()
{
    int x = 5;
    int* xPtr;
    xPtr = &x;
    cout << xPtr << endl;
}
```



x

5

xPtr

xPtr “points to” x

xPtr

x

500000

600000

600000

5

address of x is value of xPtr

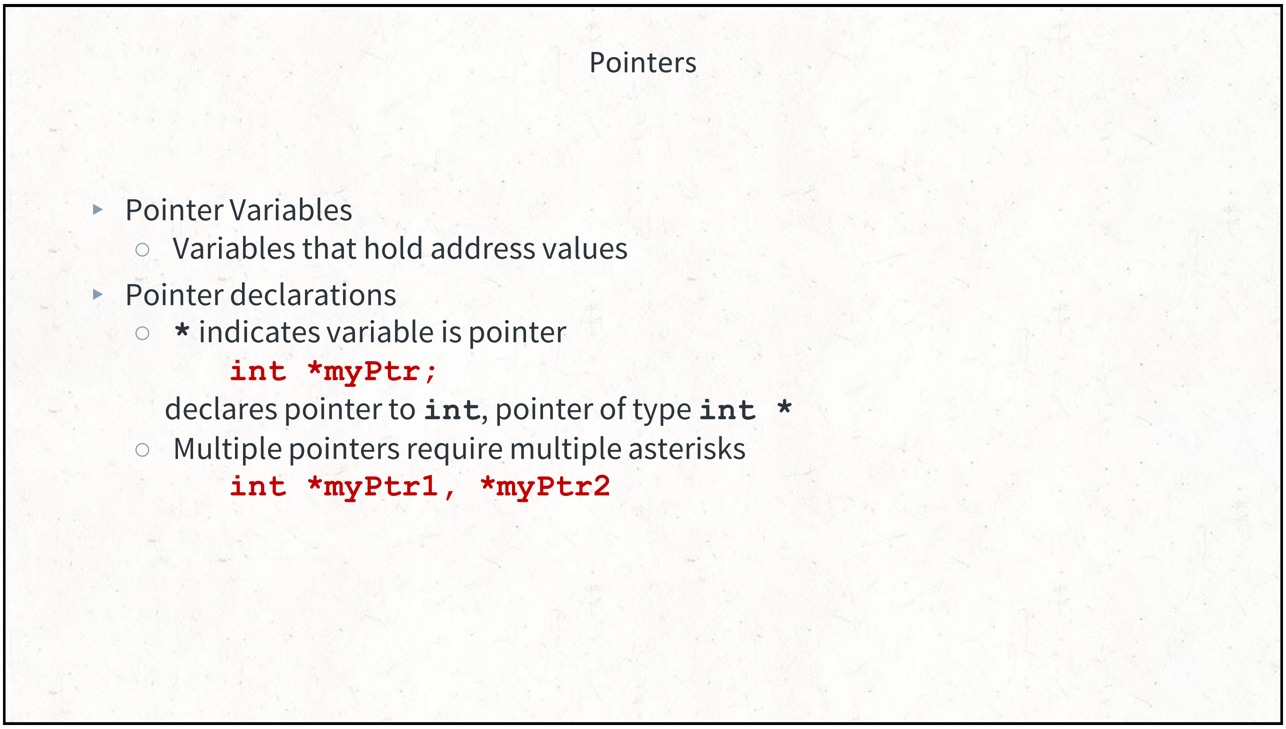
43

Pointers

- Addresses and Pointers
- The “*Address of” (Reference)* Operator &

```
int myVariable1 = 5;
char myChar = 'B';
cout << myVariable1 << endl;
double myDoubleValue = 12.56;

cout << &myVariable1 << endl;
cout << &myDoubleValue << endl;
cout << &myChar << endl;


int* myPrt = &myVariable1;
double* doublePrt = &myDoubleValue;
char* charPrt = &myChar;

cout << myPrt<<endl;
cout << doublePrt << endl;
cout << charPrt << endl;
```

44



Pointers

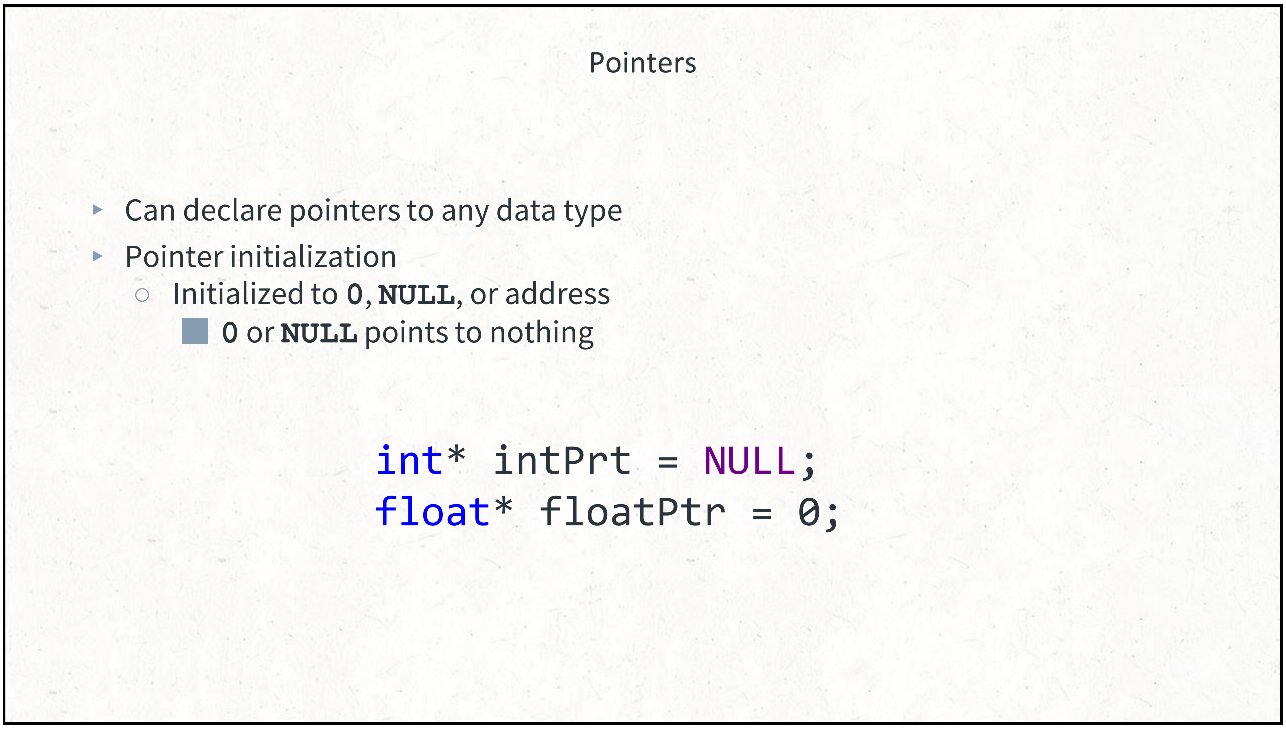
- Pointer Variables
  - Variables that hold address values
- Pointer declarations
  - ***** indicates variable is pointer

    ```
    int *myPtr;
    ```

    declares pointer to **`int`**, pointer of type **`int *`**
  - Multiple pointers require multiple asterisks

    ```
    int *myPtr1, *myPtr2
    ```

45

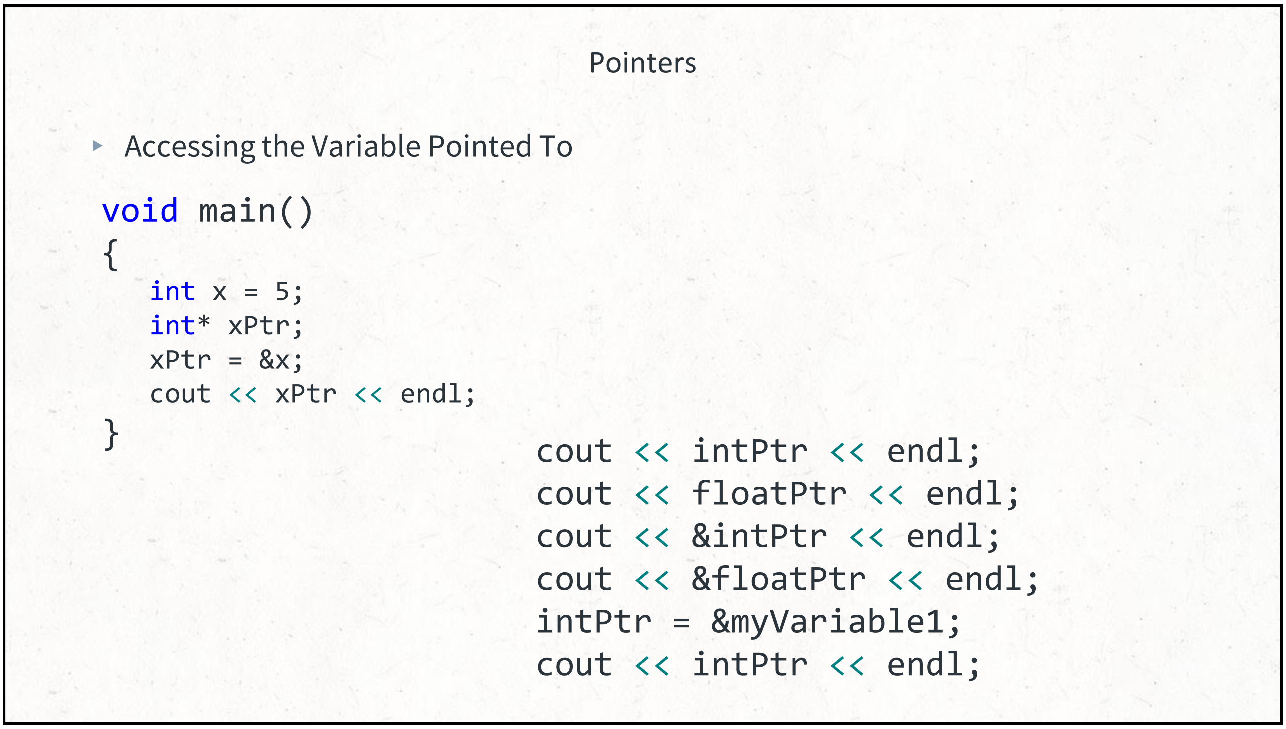


Pointers

- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **`0`**, **`NULL`**, or address
    - **`0`** or **`NULL`** points to nothing

```
int* intPrt = NULL;
float* floatPtr = 0;
```

46



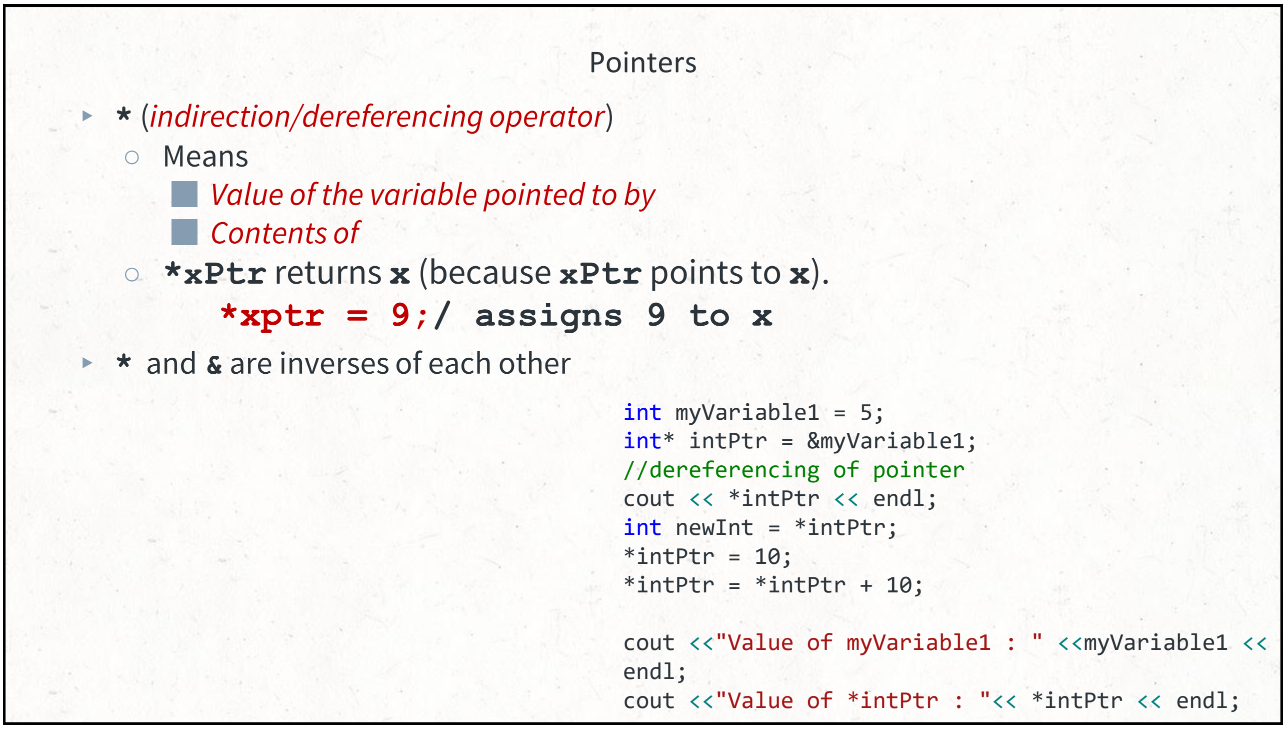
Pointers

- Accessing the Variable Pointed To

```
void main()
{
    int x = 5;
    int* xPtr;
    xPtr = &x;
    cout << xPtr << endl;
}
```

```
cout << intPtr << endl;
cout << floatPtr << endl;
cout << &intPtr << endl;
cout << &floatPtr << endl;
intPtr = &myVariable1;
cout << intPtr << endl;
```

47



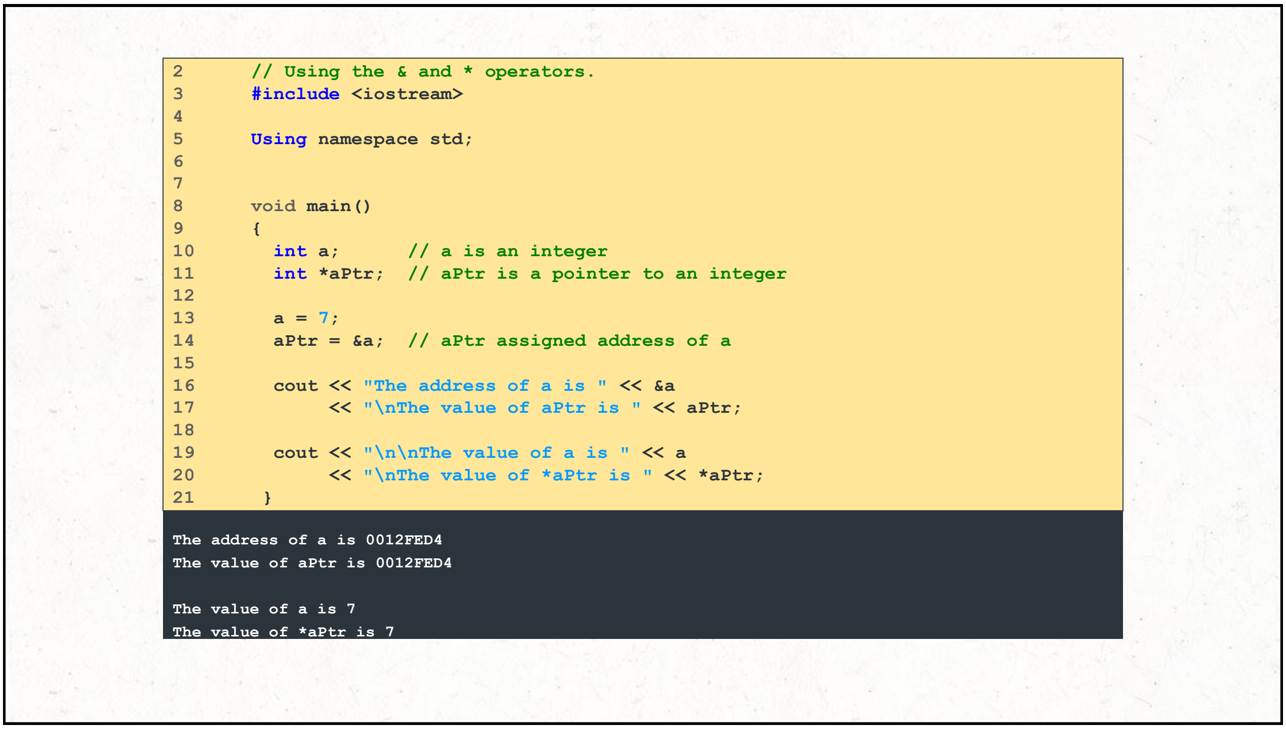
Pointers

- ***** (*indirection/dereferencing operator*)
  - Means
    - *Value of the variable pointed to by*
    - *Contents of*
  - ***xPtr** returns **x** (because **xPtr** points to **x**).

    ***xptr = 9;/ assigns 9 to x**
- ***** and **&** are inverses of each other

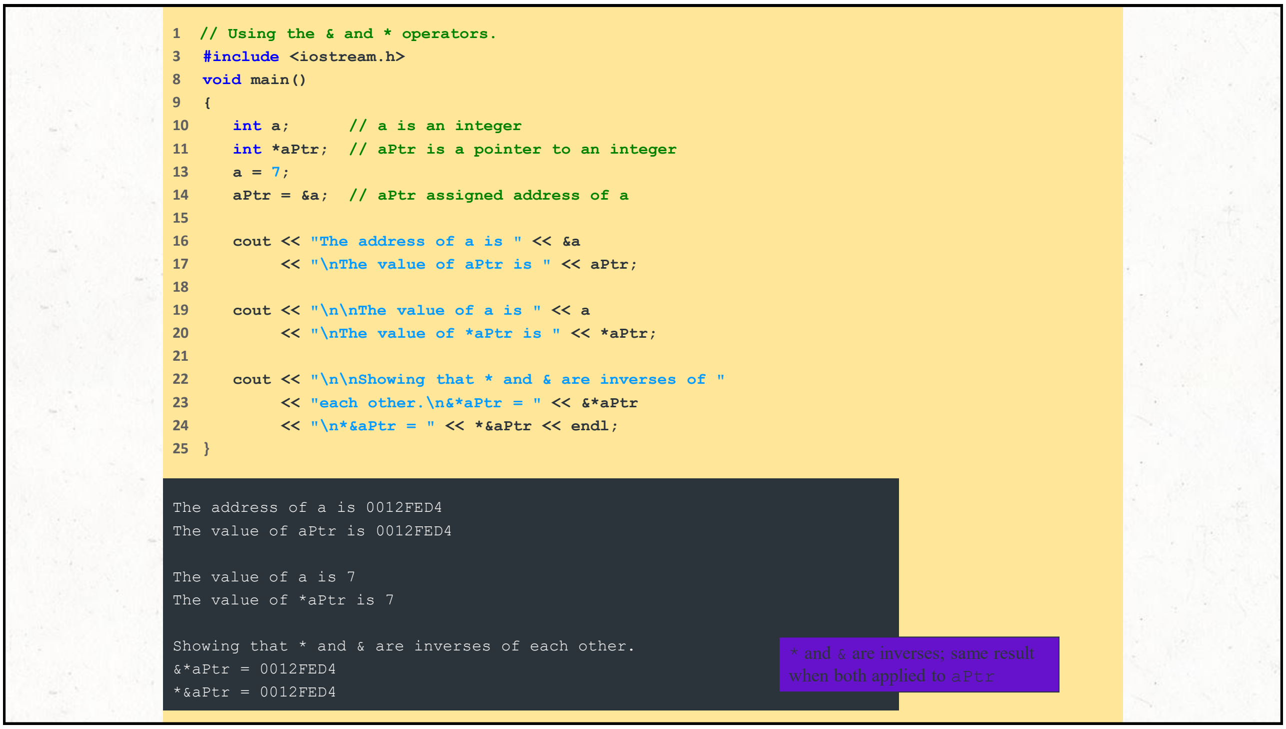
```
int myVariable1 = 5;
int* intPtr = &myVariable1;
//dereferencing of pointer
cout << *intPtr << endl;
int newInt = *intPtr;
*intPtr = 10;
*intPtr = *intPtr + 10;

cout <<"Value of myVariable1 : " <<myVariable1 <<
endl;
cout <<"Value of *intPtr : "<< *intPtr << endl;
```

48



```
// Using the & and * operators.
#include <iostream>

Using namespace std;


void main()
{
  int a;       // a is an integer
  int *aPtr;  // aPtr is a pointer to an integer

  a = 7;
  aPtr = &a;  // aPtr assigned address of a

  cout << "The address of a is " << &a
       << "\nThe value of aPtr is " << aPtr;

  cout << "\n\nThe value of a is " << a
       << "\nThe value of *aPtr is " << *aPtr;
 }
```

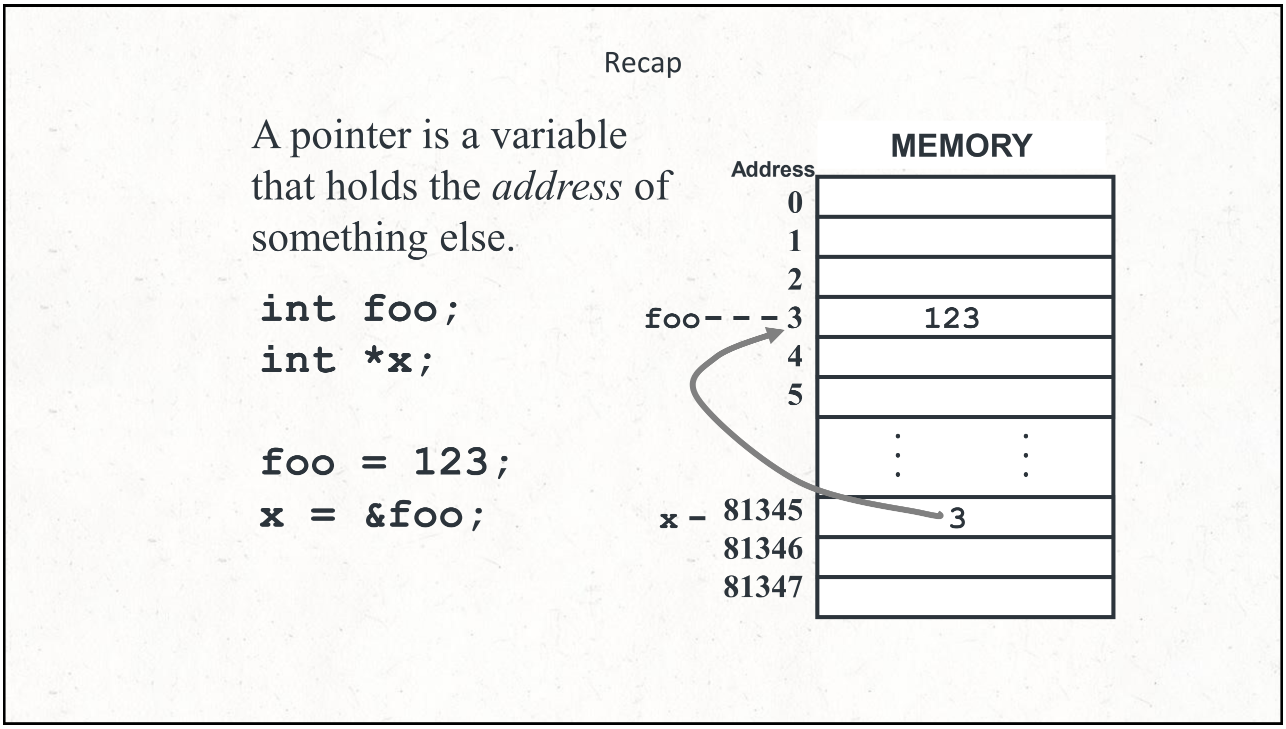
```
The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7
```



```
// Using the & and * operators.
#include <iostream.h>
void main()
{
   int a;      // a is an integer
   int *aPtr;  // aPtr is a pointer to an integer
   a = 7;
   aPtr = &a;  // aPtr assigned address of a

   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr;

   cout << "\n\nThe value of a is " << a
        << "\nThe value of *aPtr is " << *aPtr;

   cout << "\n\nShowing that * and & are inverses of "
        << "each other.\n&*aPtr = " << &*aPtr
        << "\n*&aPtr = " << *&aPtr << endl;
}
```

```
The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4
```

* and & are inverses; same result when both applied to aPtr

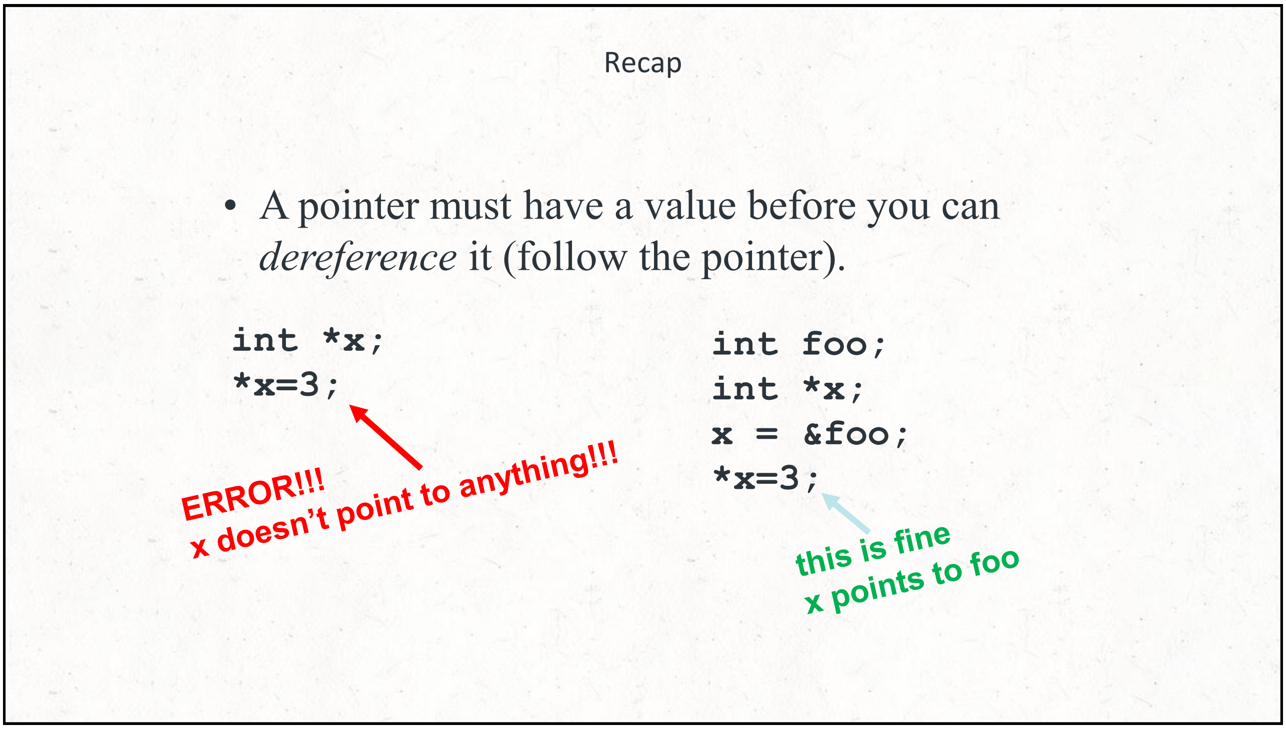


Recap

A pointer is a variable that holds the *address* of something else.

```
int foo;
int *x;

foo = 123;
x = &foo;
```

51



Recap

- A pointer must have a value before you can *dereference* it (follow the pointer).

```
int *x;
*x=3;
```

ERROR!!!
x doesn't point to anything!!!

```
int foo;
int *x;
x = &foo;
*x=3;
```

this is fine
x points to foo

52

Pointer Expressions and Pointer Arithmetic

- Pointer arithmetic
  - Increment/decrement pointer **(++** or **--**)
  - Add/subtract an integer to/from a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Pointer arithmetic meaningless unless performed on pointer to array

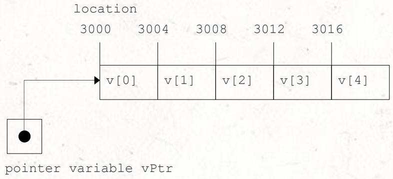
53

```
int myArray[5] = {7,5,8,7,9};
int* intPtr = &myArray[0];
//int* intPtr = myArray;


/*The array variable holds the address of
first element of array*/
cout << intPtr << endl; // address of 7
cout << myArray << endl; // address of starting location of
array

for (int index = 0; index < 5; index++)
       cout << &myArray[index] << "  ";
cout << endl;
for (int index = 0; index < 5; index++)
       cout << &intPtr[index] << "  ";
```

54

Pointer Expressions and Pointer Arithmetic

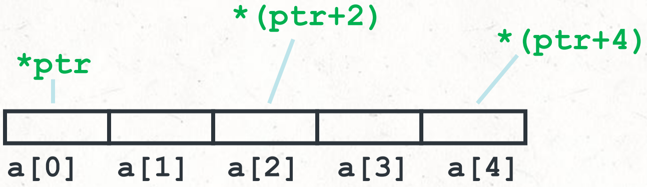
- 5 element **`int`** array with 4 bytes **`int`**s
  - **`vPtr`** points to first element **`v[ 0 ]`**, which is at location 3000
    `vPtr = 3000`
  - **`vPtr += 2`**; sets **`vPtr`** to **3008**
    `vPtr` points to **`v[ 2 ]`**

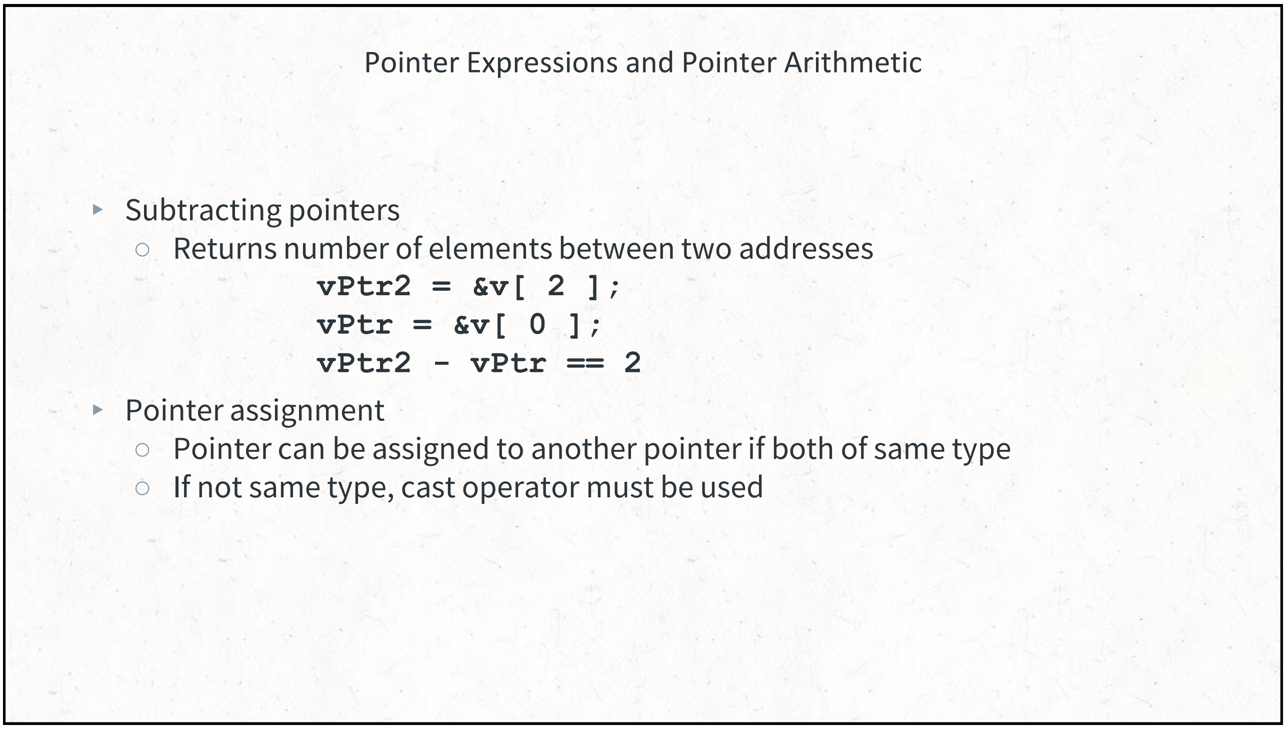


Pointer Expressions and Pointer Arithmetic

- If you increment a pointer, it will be increased by the size of whatever it points to.

```
int a[5];
int *ptr = a;
```





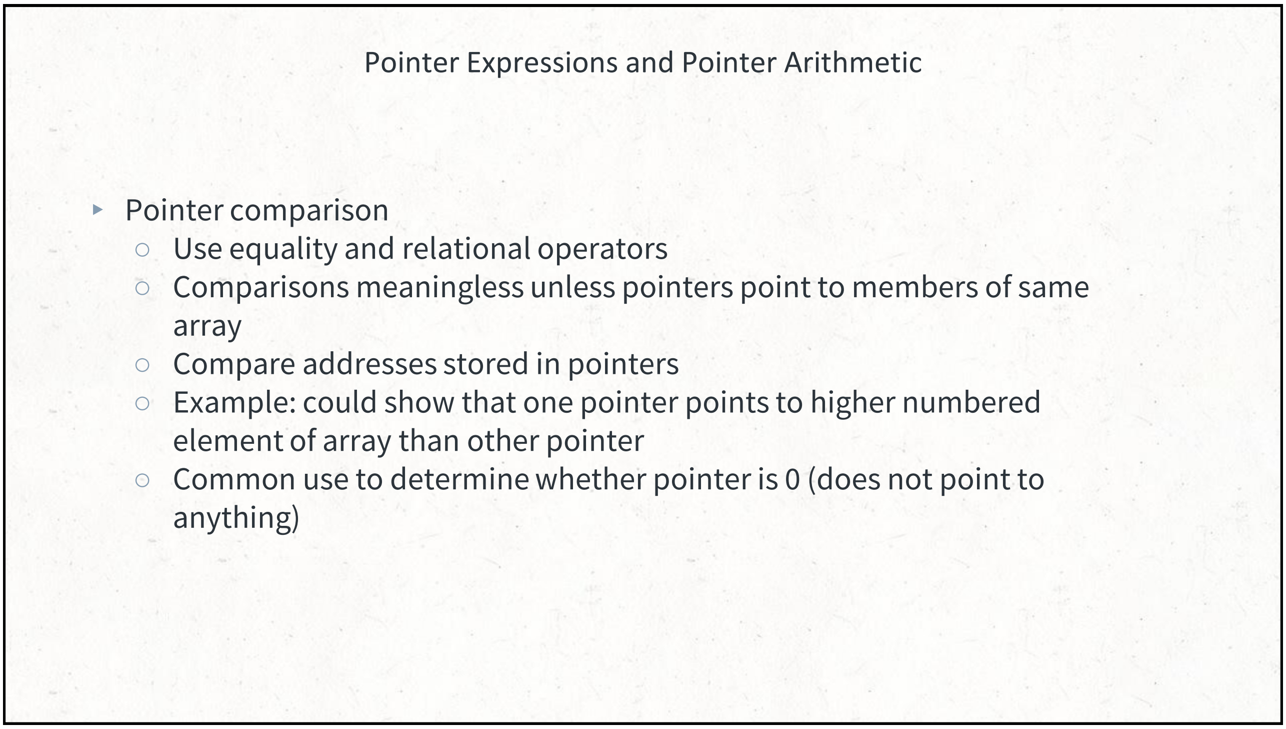
Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements between two addresses

```
vPtr2 = &v[ 2 ];
vPtr = &v[ 0 ];
vPtr2 - vPtr == 2
```

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used

57



Pointer Expressions and Pointer Arithmetic

- Pointer comparison
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
  - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Common use to determine whether pointer is 0 (does not point to anything)

58

Relationship Between Pointers and Arrays

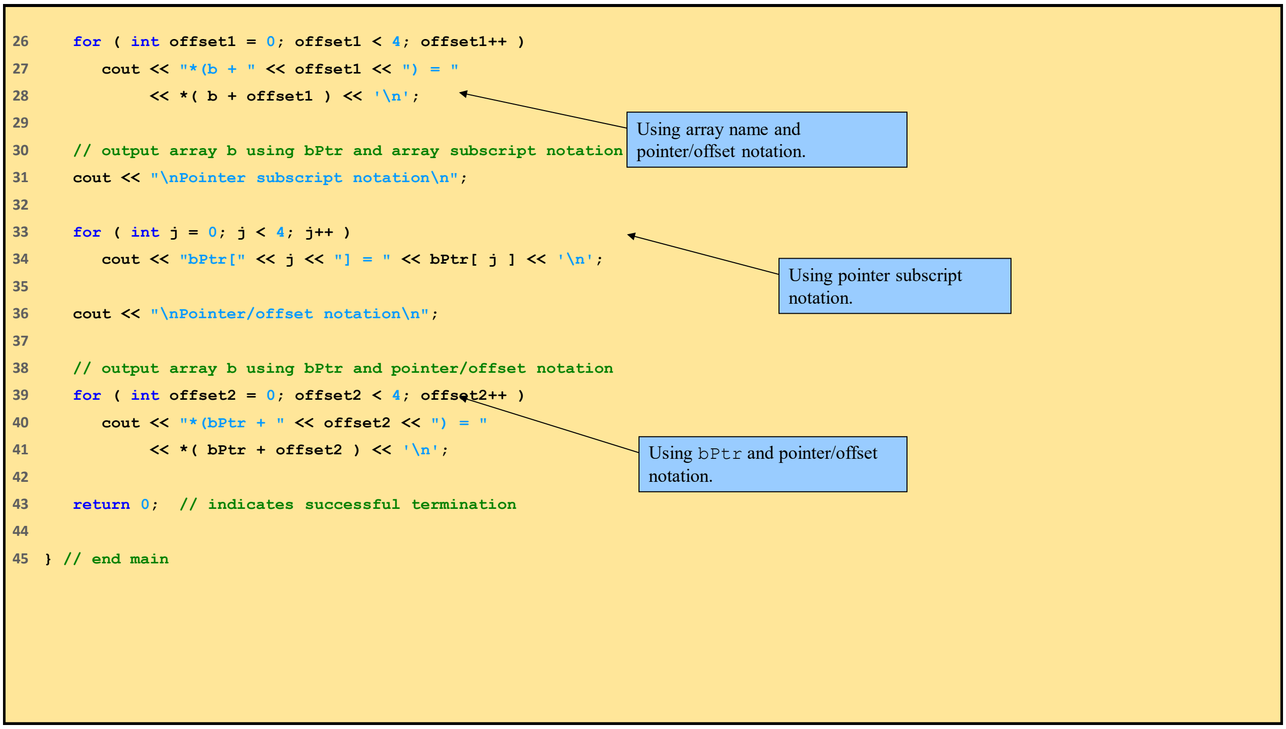
- Arrays and pointers closely related
  - Array name like constant pointer
  - Pointers can do array subscripting operations
- Accessing array elements with pointers
  - Element `b[ n ]` can be accessed by `*( bPtr + n )`
    - Called pointer/offset notation
  - Addresses
    - `&b[ 3 ]` same as `bPtr + 3`
  - Array name can be treated as pointer
    - `b[ 3 ]` same as `*( b + 3 )`
  - Pointers can be subscripted (pointer/subscript notation)
    - `bPtr[ 3 ]` same as `b[ 3 ]`

59

```
// Using subscripting and pointer notations with arrays.

#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int b[] = { 10, 20, 30, 40 };
   int *bPtr = b;   // set bPtr to point to array b

   // output array b using array subscript notation
   cout << "Array b printed with:\n"
        << "Array subscript notation\n";

   for ( int i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   // output array b using the array name and
   // pointer/offset notation
   cout << "\nPointer/offset notation where "
        << "the pointer is the array name\n";
```

Using array subscript notation.

60



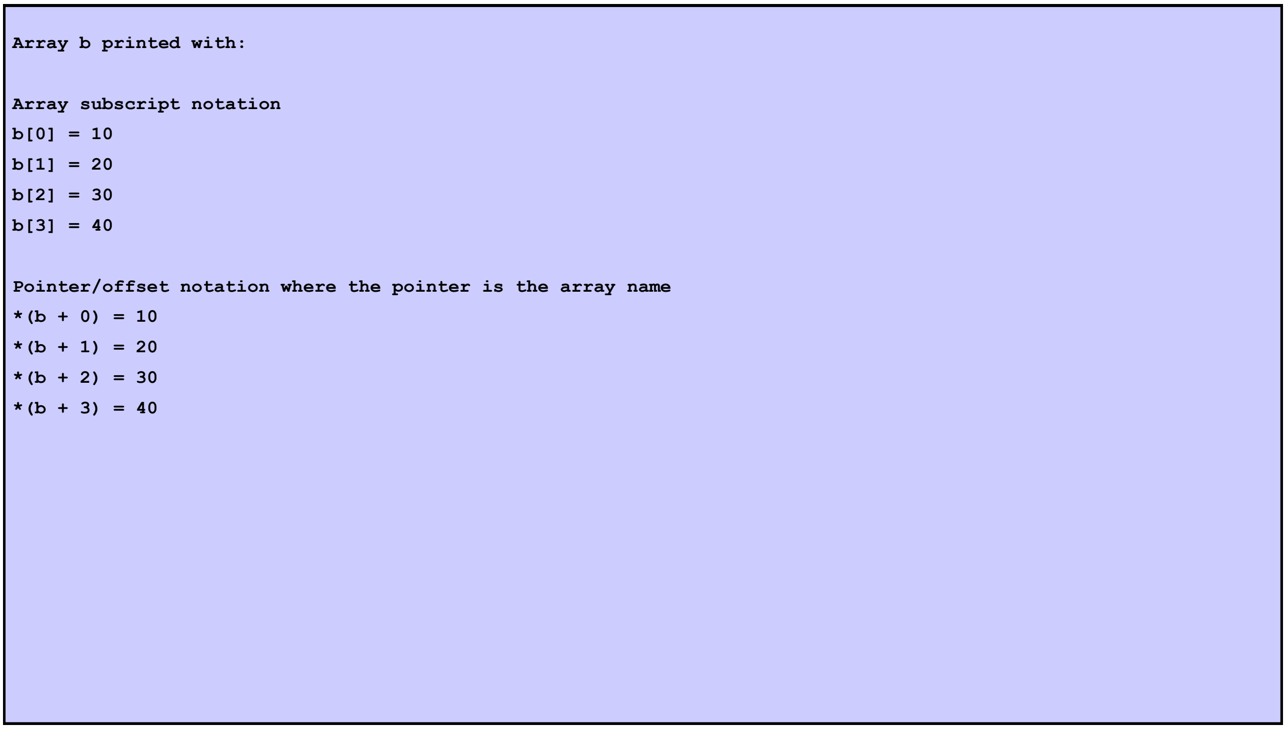
```
   for ( int offset1 = 0; offset1 < 4; offset1++ )
      cout << "*(b + " << offset1 << ") = "
           << *( b + offset1 ) << '\n';

   // output array b using bPtr and array subscript notation
   cout << "\nPointer subscript notation\n";

   for ( int j = 0; j < 4; j++ )
      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';

   cout << "\nPointer/offset notation\n";

   // output array b using bPtr and pointer/offset notation
   for ( int offset2 = 0; offset2 < 4; offset2++ )
      cout << "*(bPtr + " << offset2 << ") = "
           << *( bPtr + offset2 ) << '\n';

   return 0;  // indicates successful termination

} // end main
```

Using array name and pointer/offset notation.

Using pointer subscript notation.

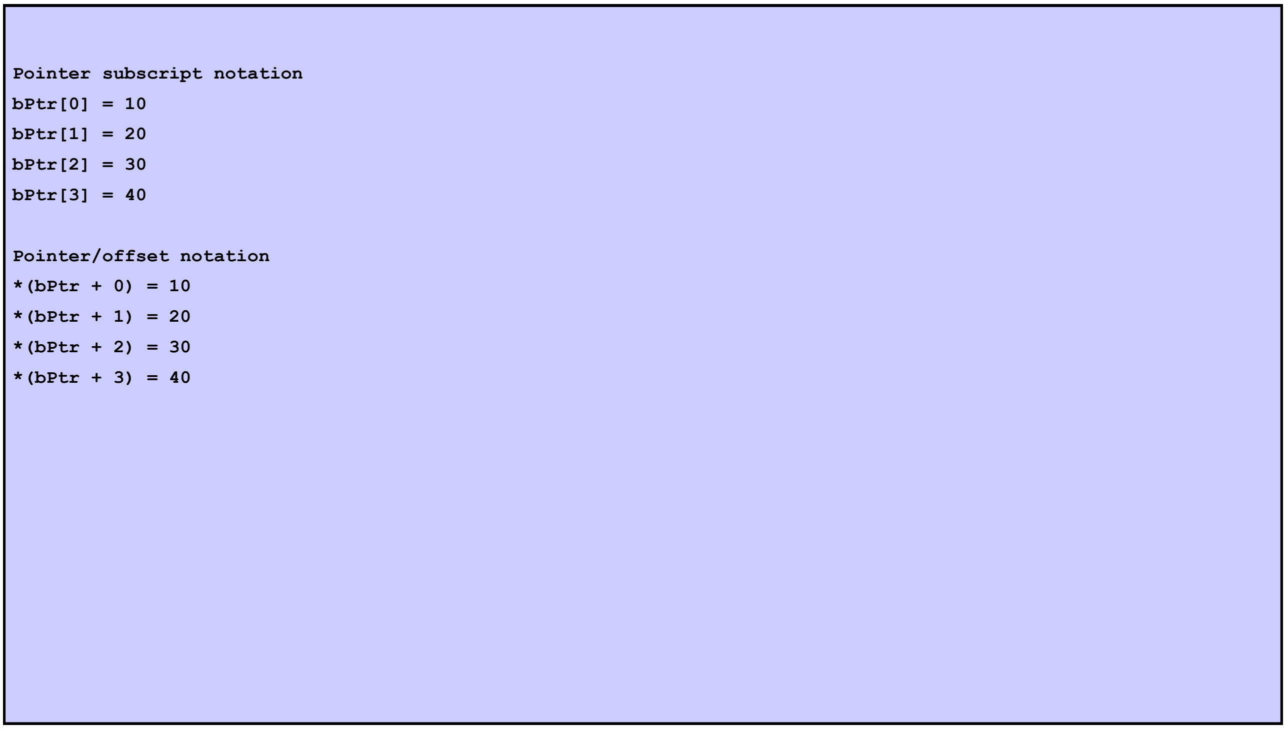
Using bPtr and pointer/offset notation.

61



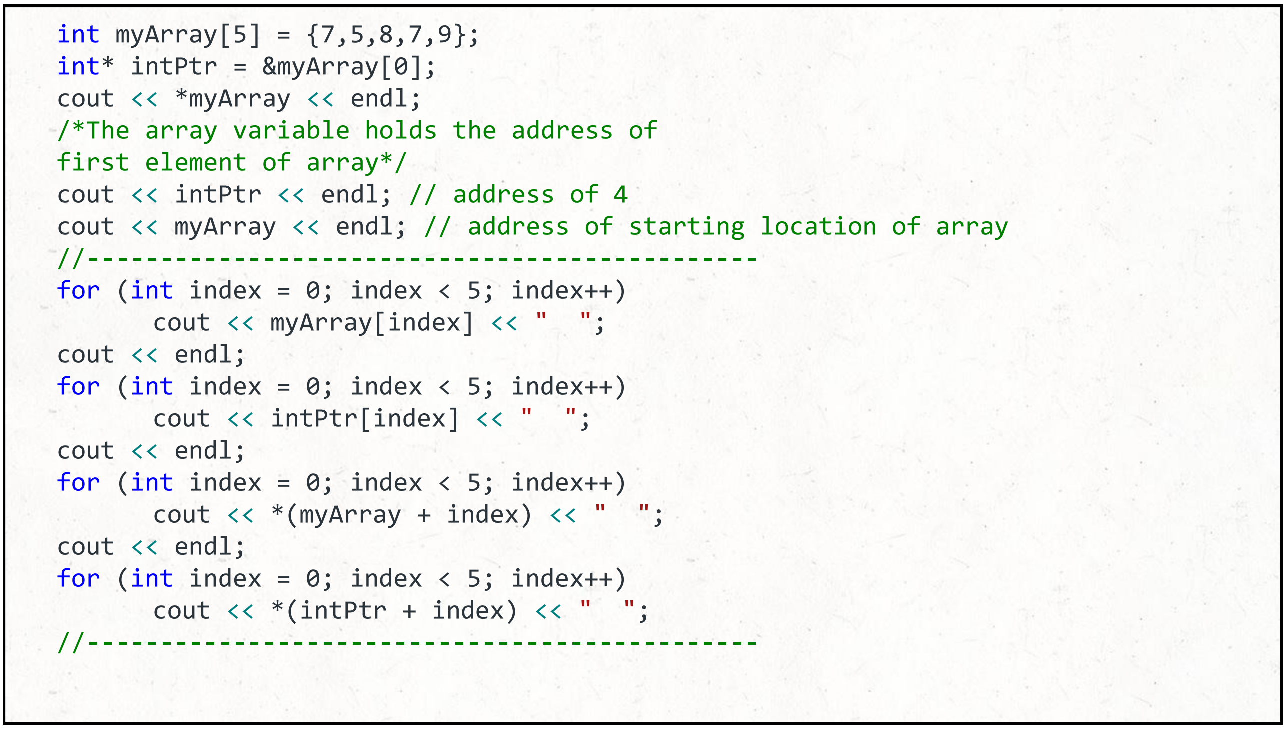
```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40
```

62



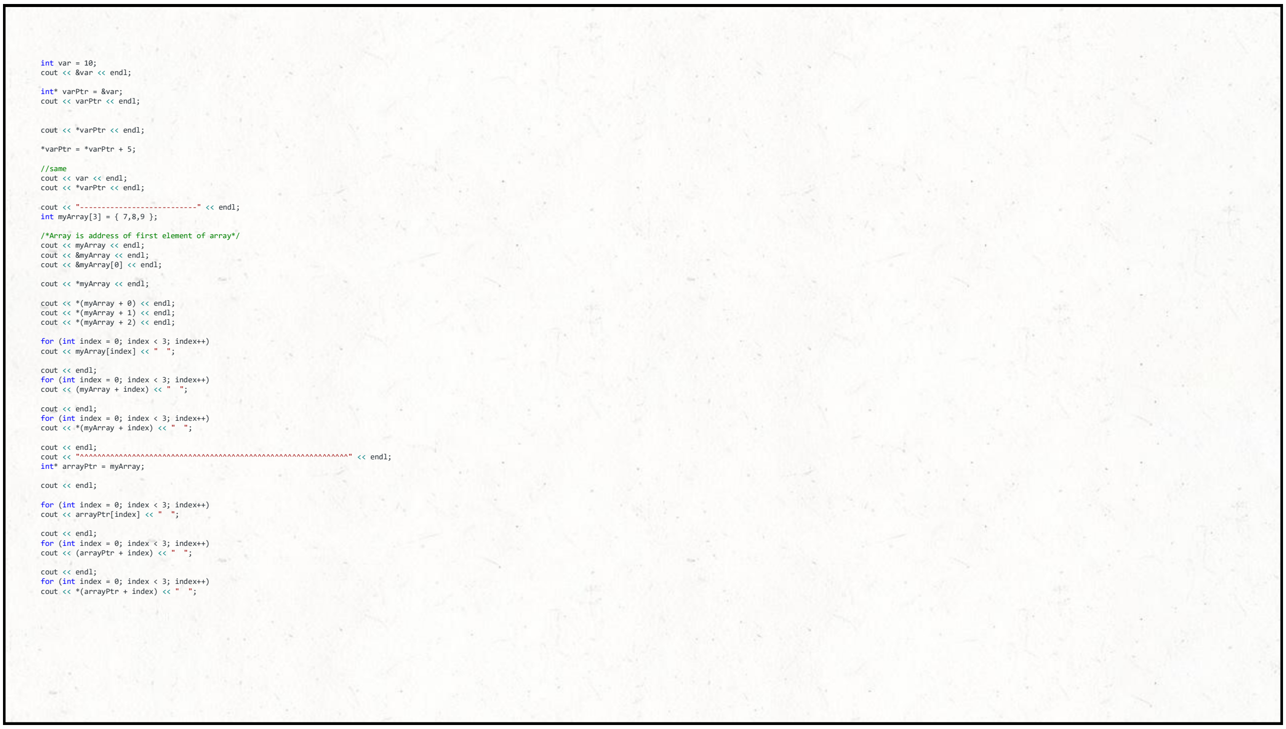
```
Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

63



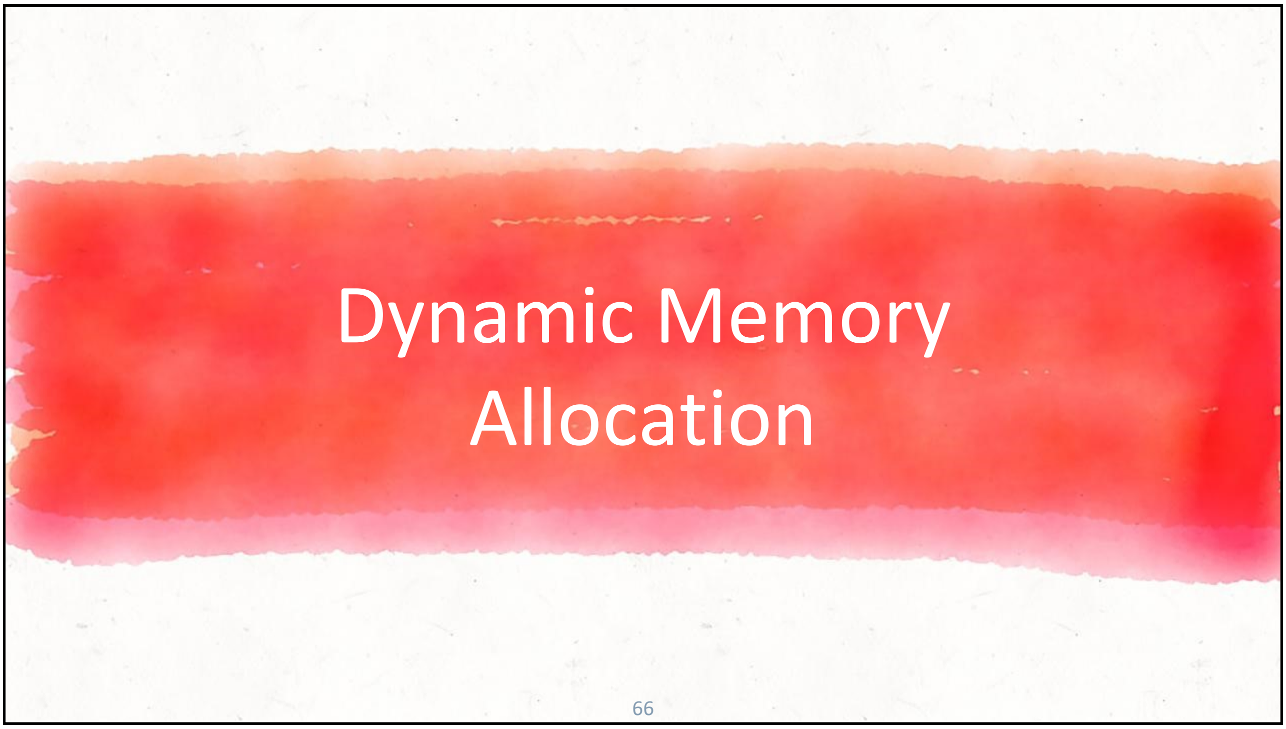
```
int myArray[5] = {7,5,8,7,9};
int* intPtr = &myArray[0];
cout << *myArray << endl;
/*The array variable holds the address of
first element of array*/
cout << intPtr << endl; // address of 4
cout << myArray << endl; // address of starting location of array
//----------------------------------------------
for (int index = 0; index < 5; index++)
       cout << myArray[index] << "   ";
cout << endl;
for (int index = 0; index < 5; index++)
       cout << intPtr[index] << "   ";
cout << endl;
for (int index = 0; index < 5; index++)
       cout << *(myArray + index) << "   ";
cout << endl;
for (int index = 0; index < 5; index++)
       cout << *(intPtr + index) << "   ";
//----------------------------------------------
```

64



```cpp
int var = 10;
cout << &var << endl;

int* varPtr = &var;
cout << varPtr << endl;


cout << *varPtr << endl;

*varPtr = *varPtr + 5;

//same
cout << var << endl;
cout << *varPtr << endl;

cout << "---------------------------" << endl;
int myArray[3] = { 7,8,9 };

/*Array is address of first element of array*/
cout << myArray << endl;
cout << &myArray << endl;
cout << &myArray[0] << endl;

cout << *myArray << endl;

cout << *(myArray + 0) << endl;
cout << *(myArray + 1) << endl;
cout << *(myArray + 2) << endl;

for (int index = 0; index < 3; index++)
cout << myArray[index] << "  ";

cout << endl;
for (int index = 0; index < 3; index++)
cout << (myArray + index) << "  ";

cout << endl;
for (int index = 0; index < 3; index++)
cout << *(myArray + index) << "  ";

cout << endl;
cout << "^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^" << endl;
int* arrayPtr = myArray;

cout << endl;

for (int index = 0; index < 3; index++)
cout << arrayPtr[index] << "  ";

cout << endl;
for (int index = 0; index < 3; index++)
cout << (arrayPtr + index) << "  ";

cout << endl;
for (int index = 0; index < 3; index++)
cout << *(arrayPtr + index) << "  ";
```

65

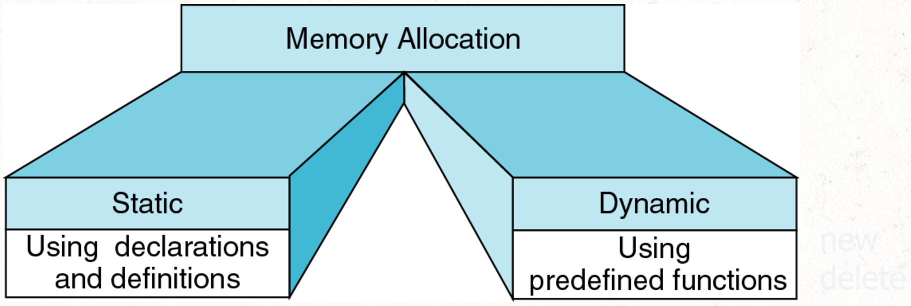


Dynamic Memory Allocation

66

66

Memory Allocation

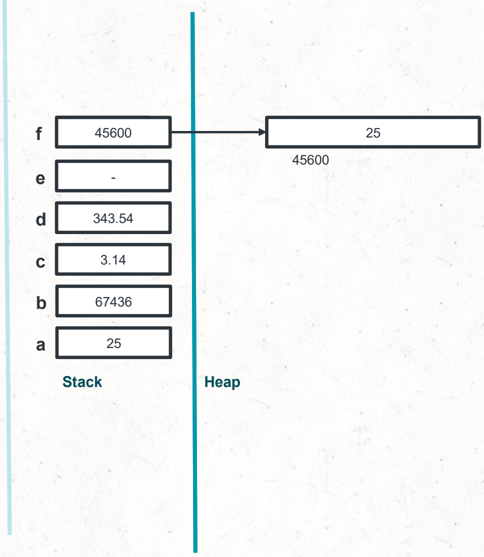

Memory Allocation
Static
Using declarations and definitions
Dynamic
Using predefined functions
new
delete

```
{
   int a[200];
   …
}
```

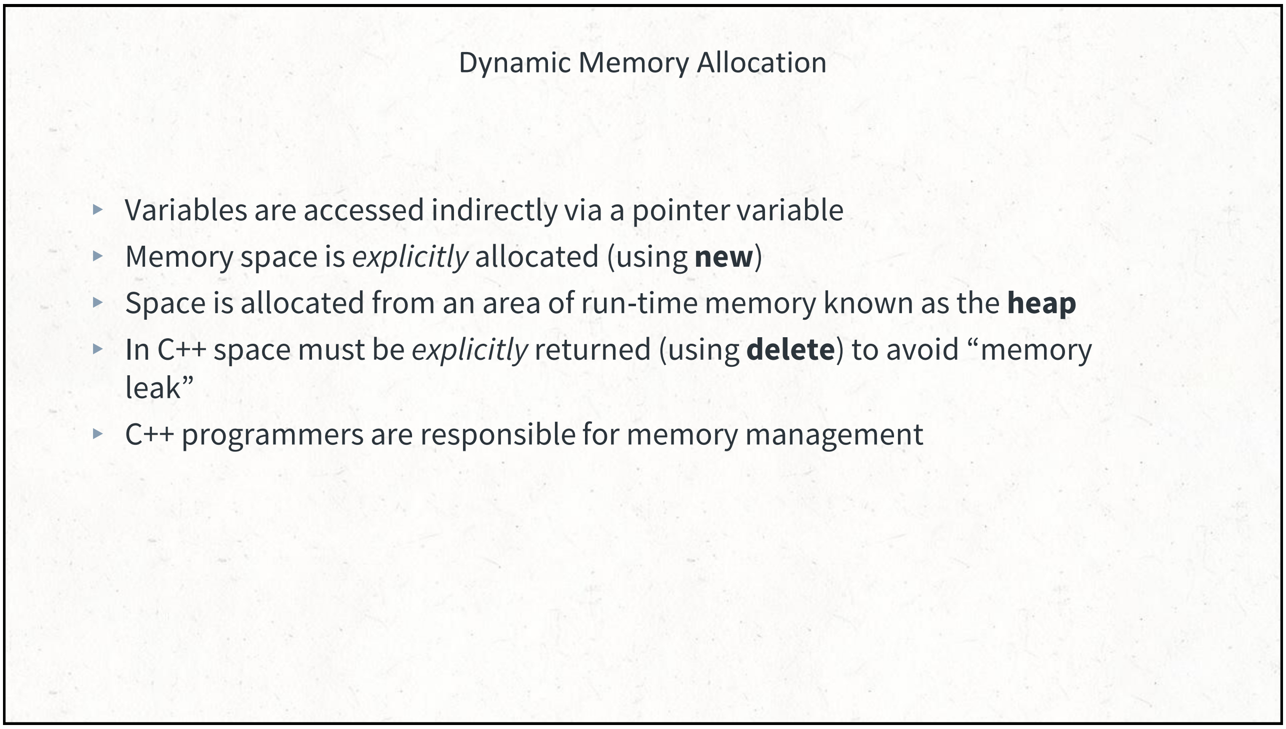
```
int* ptr;
ptr = new int[200];
…
delete [] ptr;
```

67

```
int a;              //32 bits
long b;             //64 bits
float c;            //32 bits
double d;           //64 bits
char e;             //16 bits
int* f;             //16 bits

a = 25;
b = 6743674654783474;
c = 3.14F;
d = 4443534534.54;
e = '-';
f = new int;
*f = 25;
```



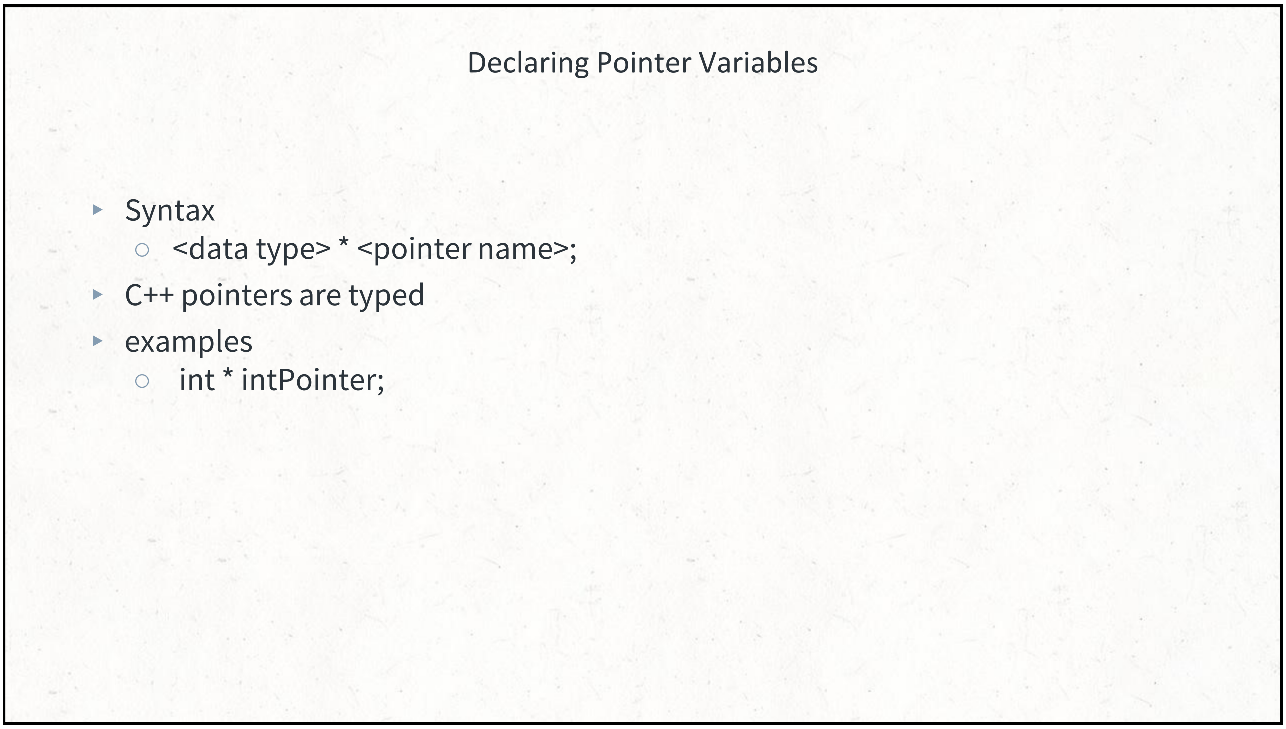
68



Dynamic Memory Allocation

- Variables are accessed indirectly via a pointer variable
- Memory space is *explicitly* allocated (using **new**)
- Space is allocated from an area of run-time memory known as the **heap**
- In C++ space must be *explicitly* returned (using **delete**) to avoid “memory leak”
- C++ programmers are responsible for memory management

69



Declaring Pointer Variables

- Syntax
  - <data type> * <pointer name>;
- C++ pointers are typed
- examples
  - int * intPointer;

70

Assigning a value to a pointer variable

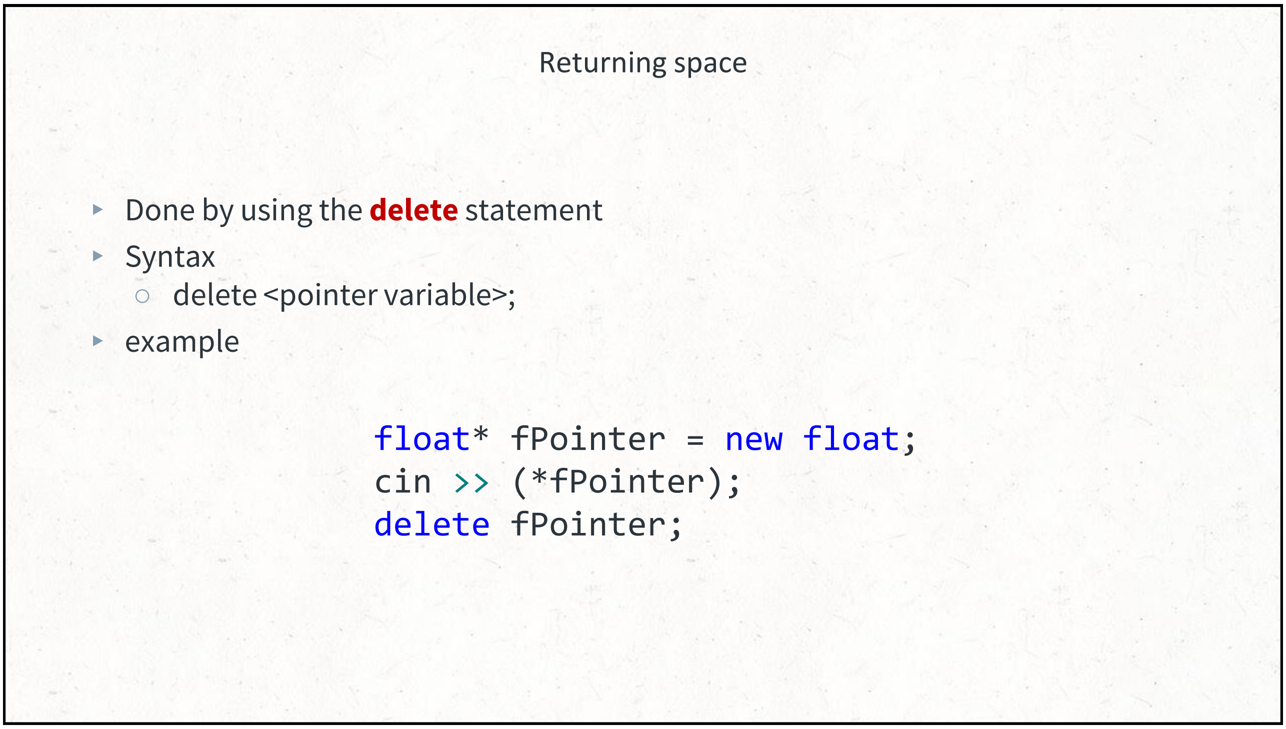
- The **value** of a pointer variable is a memory address
- Assign address of an existing variable
  - int number;
  - intPointer = &number;
- Use "new" to allocate space in the heap
  - intPointer = new int;
  - Address of heap memory space allocated becomes the value of the pointer variable

71

Dereferencing

- Heap variables do not have a name of their own
  - **Anonymous** variables
- *intPointer refers to the value pointed to by intPointer
- what happens?
- intPointer = new int;
  - *intPointer = 36;
  - cout << *intPointer;
  - intPointer = null;

72

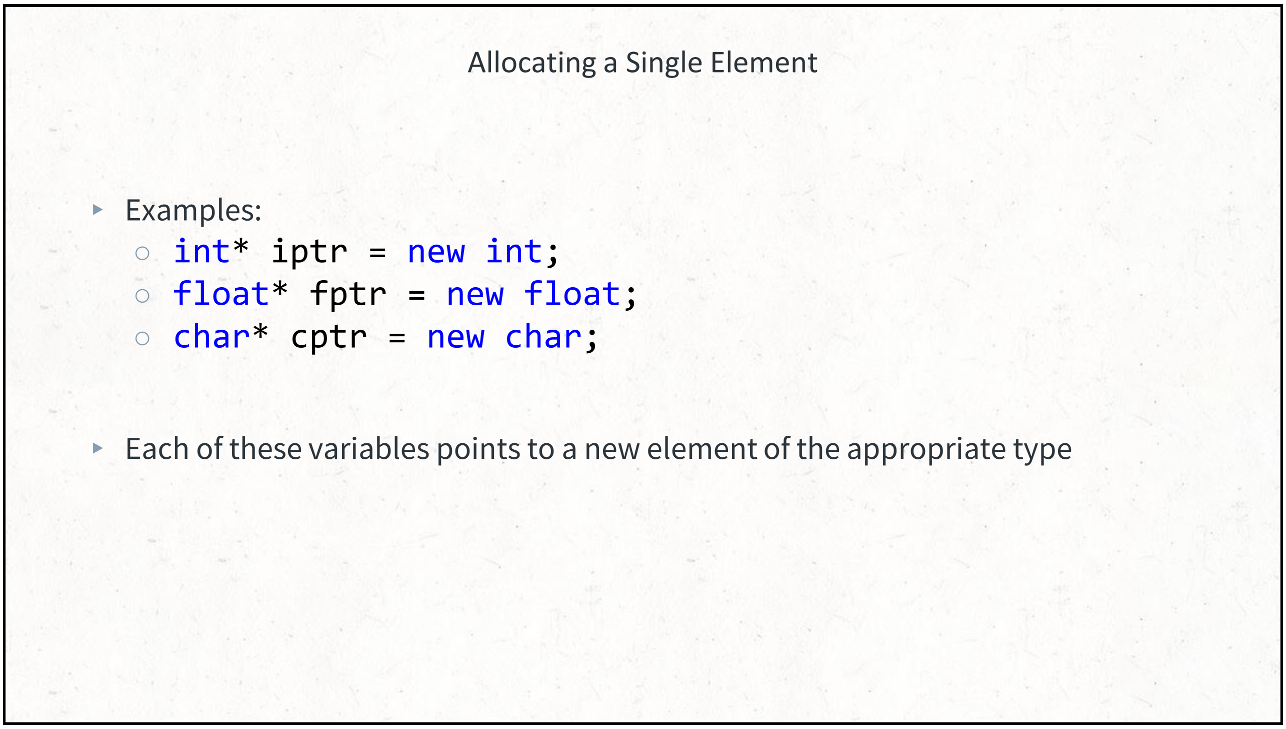


Returning space

- Done by using the **delete** statement
- Syntax
  - delete <pointer variable>;
- example

```
float* fPointer = new float;
cin >> (*fPointer);
delete fPointer;
```

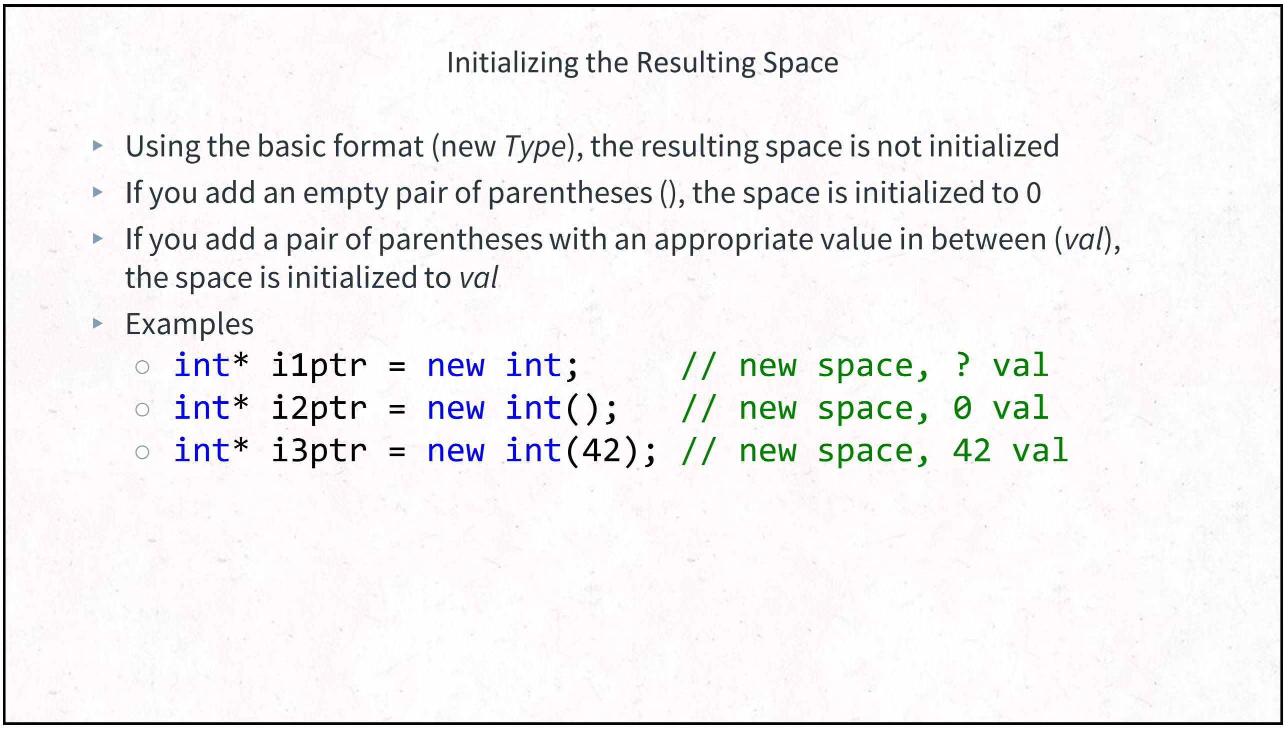
74



Allocating a Single Element

- Examples:
  - `int* iptr = new int;`
  - `float* fptr = new float;`
  - `char* cptr = new char;`
- Each of these variables points to a new element of the appropriate type

75

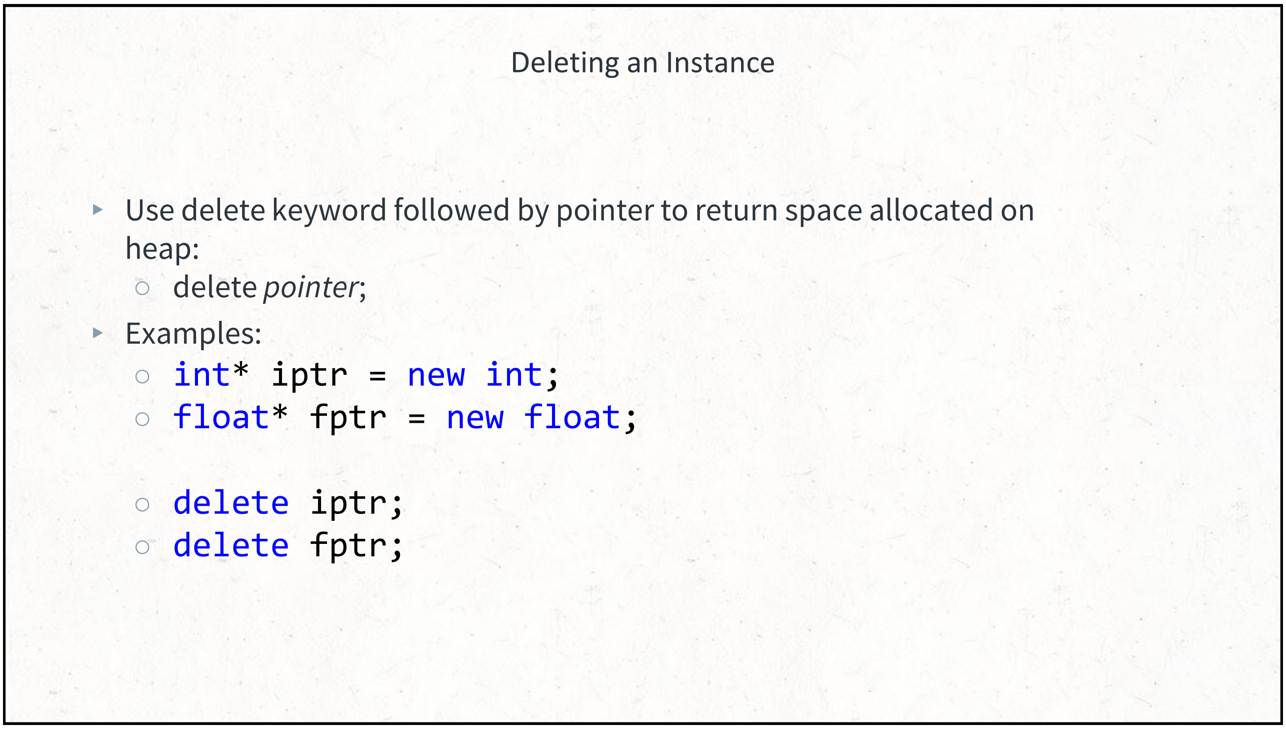


Initializing the Resulting Space

- Using the basic format (new *Type*), the resulting space is not initialized
- If you add an empty pair of parentheses (), the space is initialized to 0
- If you add a pair of parentheses with an appropriate value in between (*val*), the space is initialized to *val*
- Examples

```
○ int* i1ptr = new int;      // new space, ? val
○ int* i2ptr = new int();    // new space, 0 val
○ int* i3ptr = new int(42); // new space, 42 val
```

76

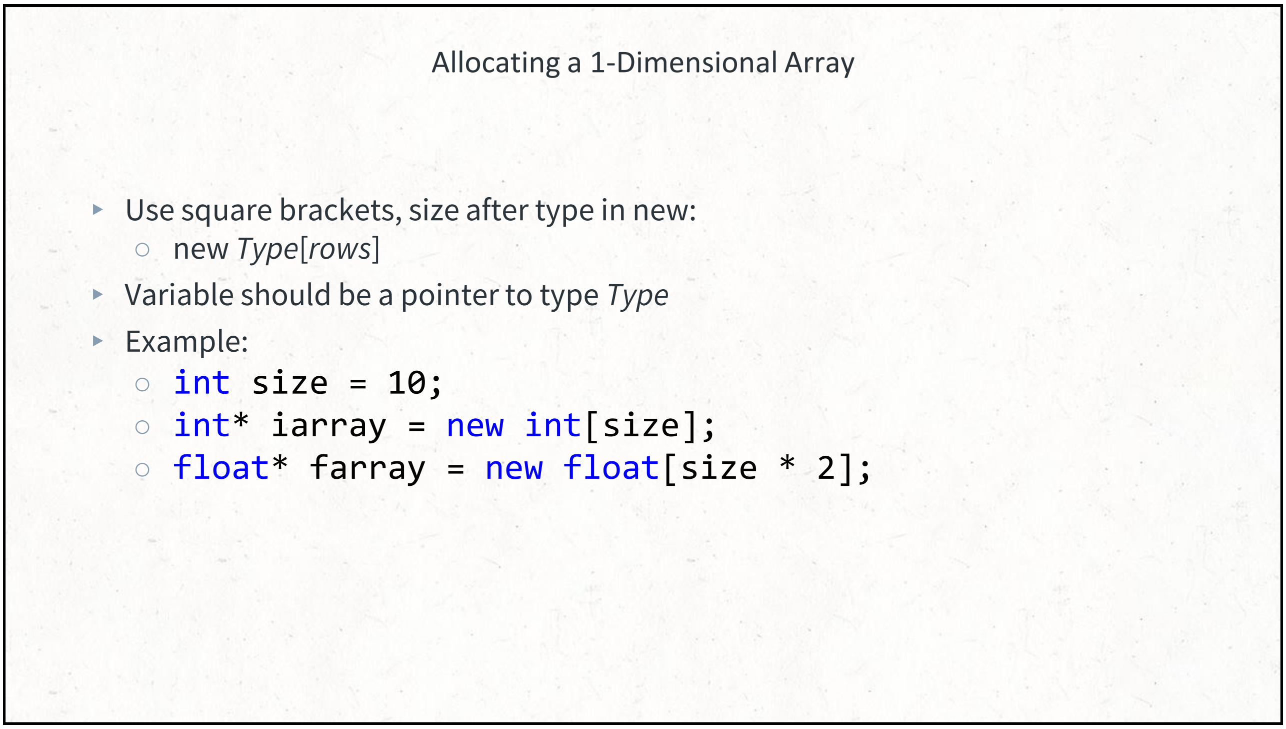


Deleting an Instance

- Use delete keyword followed by pointer to return space allocated on heap:
  - delete *pointer*;
- Examples:

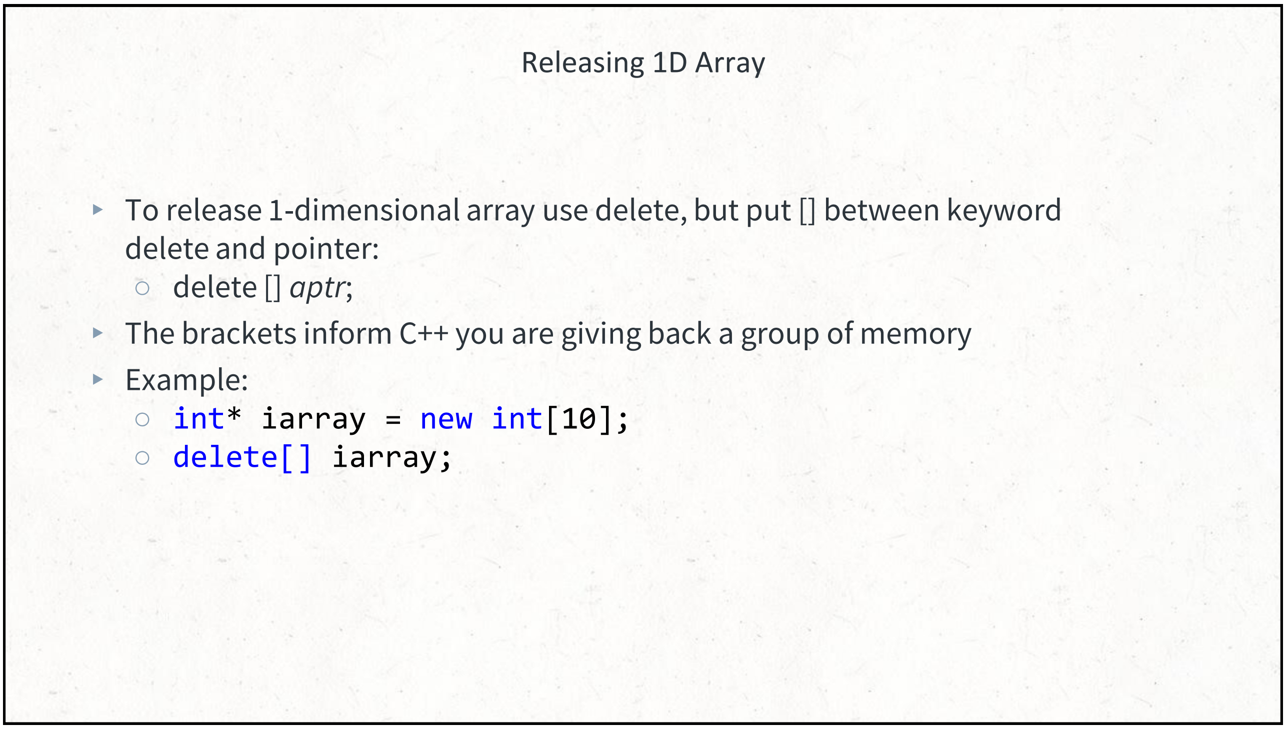
```
○ int* iptr = new int;
○ float* fptr = new float;

○ delete iptr;
○ delete fptr;
```

77



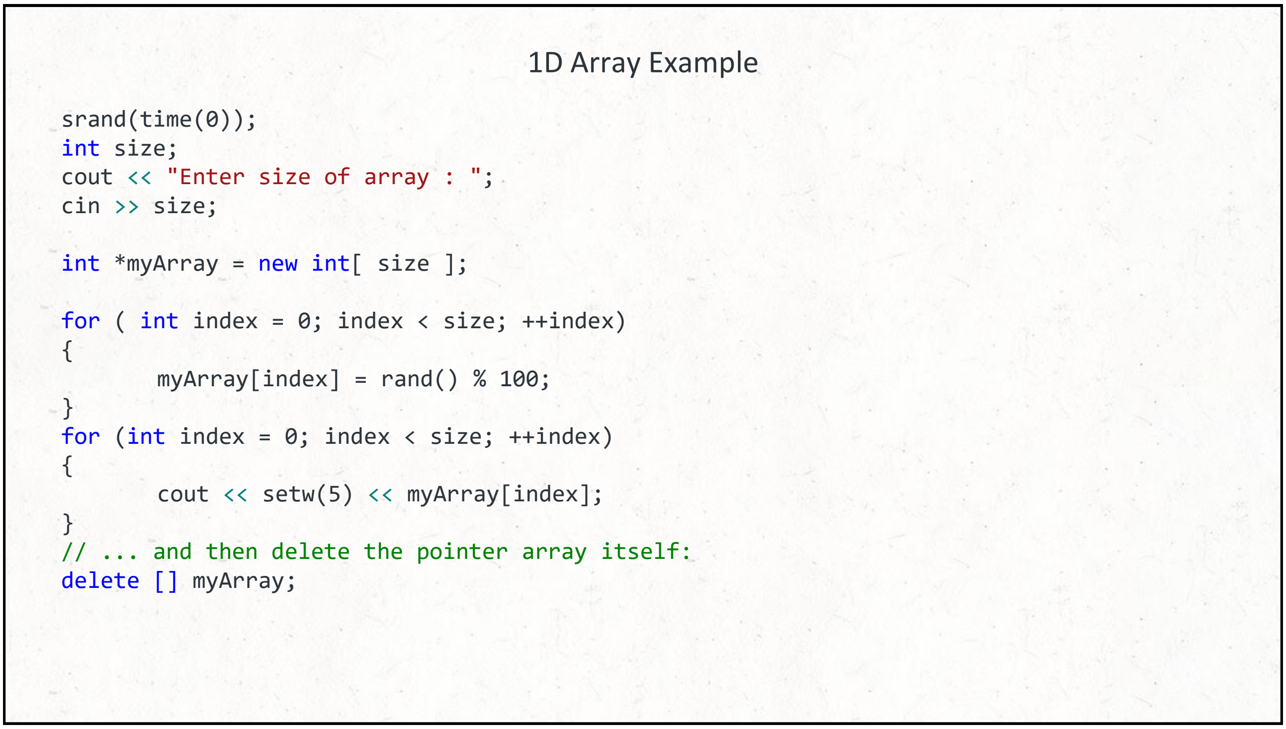
Allocating a 1-Dimensional Array

- Use square brackets, size after type in new:
  - new *Type*[*rows*]
- Variable should be a pointer to type *Type*
- Example:
  - `int size = 10;`
  - `int* iarray = new int[size];`
  - `float* farray = new float[size * 2];`



Releasing 1D Array

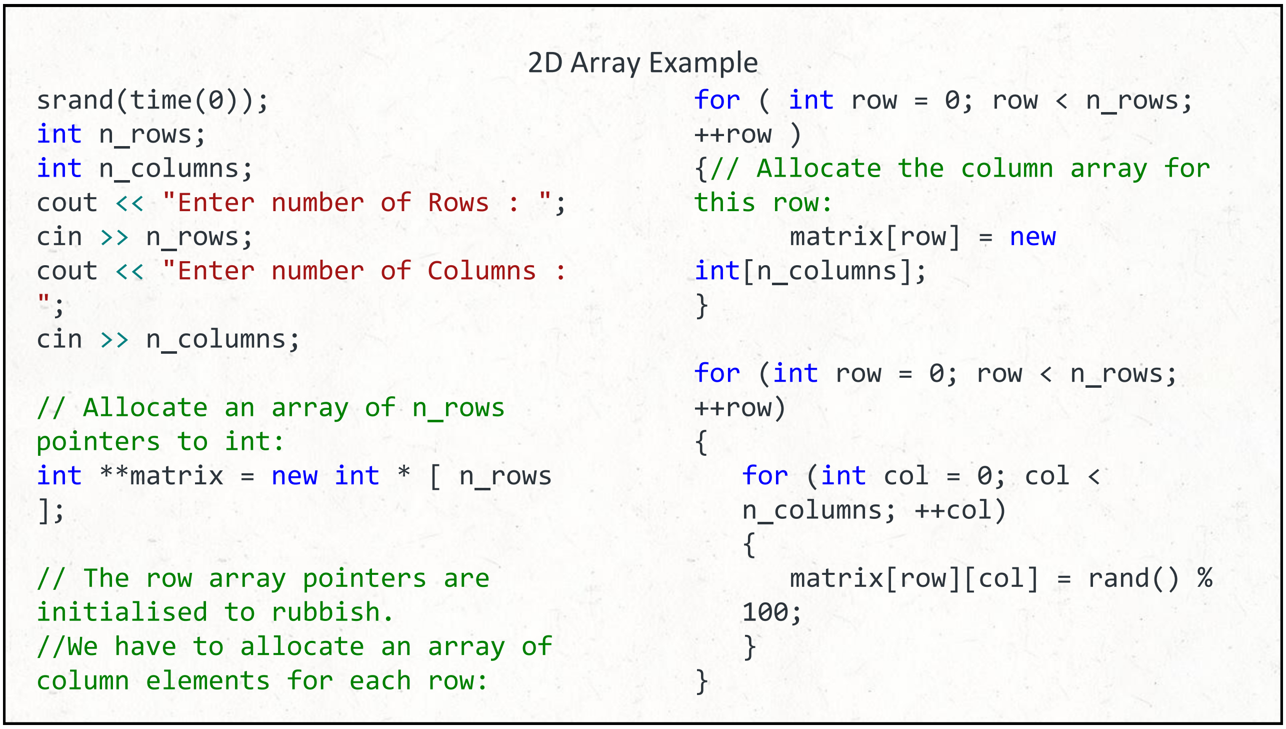
- To release 1-dimensional array use delete, but put [] between keyword delete and pointer:
  - delete [] *aptr*;
- The brackets inform C++ you are giving back a group of memory
- Example:
  - `int* iarray = new int[10];`
  - `delete[] iarray;`



1D Array Example

```
srand(time(0));
int size;
cout << "Enter size of array : ";
cin >> size;

int *myArray = new int[ size ];

for ( int index = 0; index < size; ++index)
{
        myArray[index] = rand() % 100;
}
for (int index = 0; index < size; ++index)
{
        cout << setw(5) << myArray[index];
}
// ... and then delete the pointer array itself:
delete [] myArray;
```

80

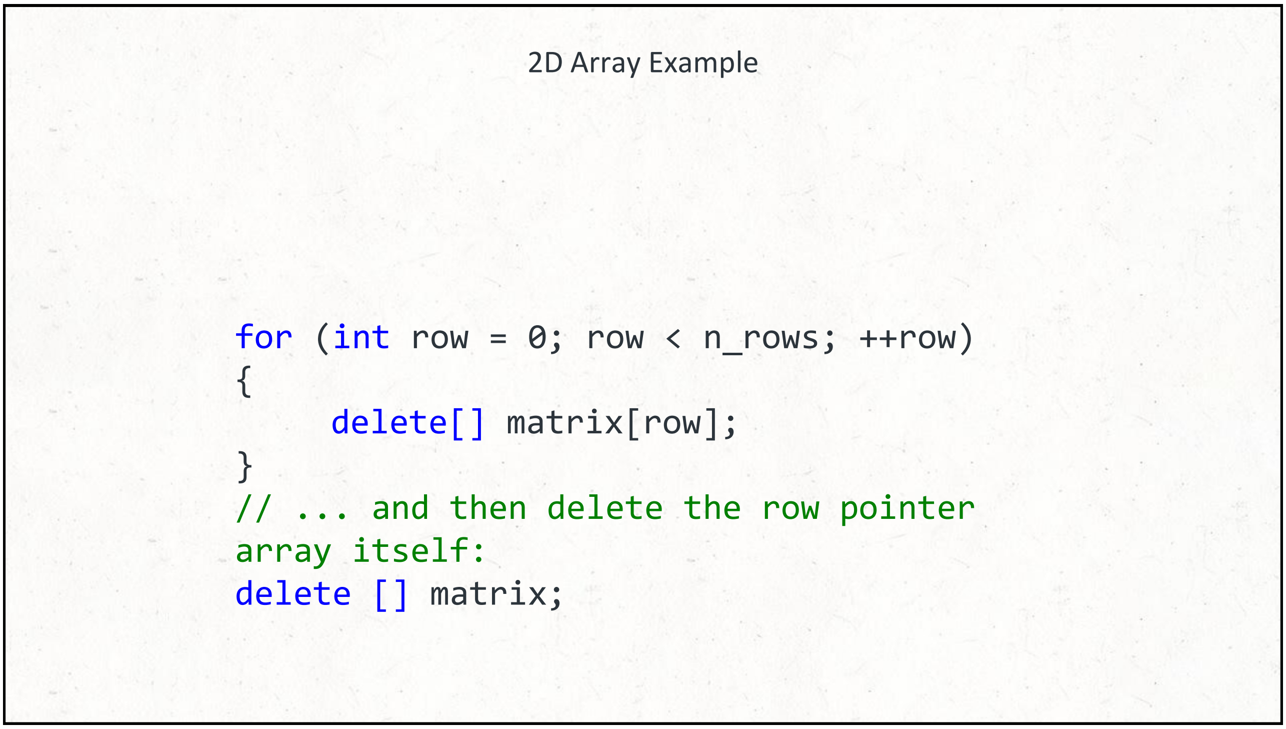


2D Array Example

```
srand(time(0));
int n_rows;
int n_columns;
cout << "Enter number of Rows : ";
cin >> n_rows;
cout << "Enter number of Columns :
";
cin >> n_columns;

// Allocate an array of n_rows
pointers to int:
int **matrix = new int * [ n_rows
];

// The row array pointers are
initialised to rubbish.
//We have to allocate an array of
column elements for each row:
```

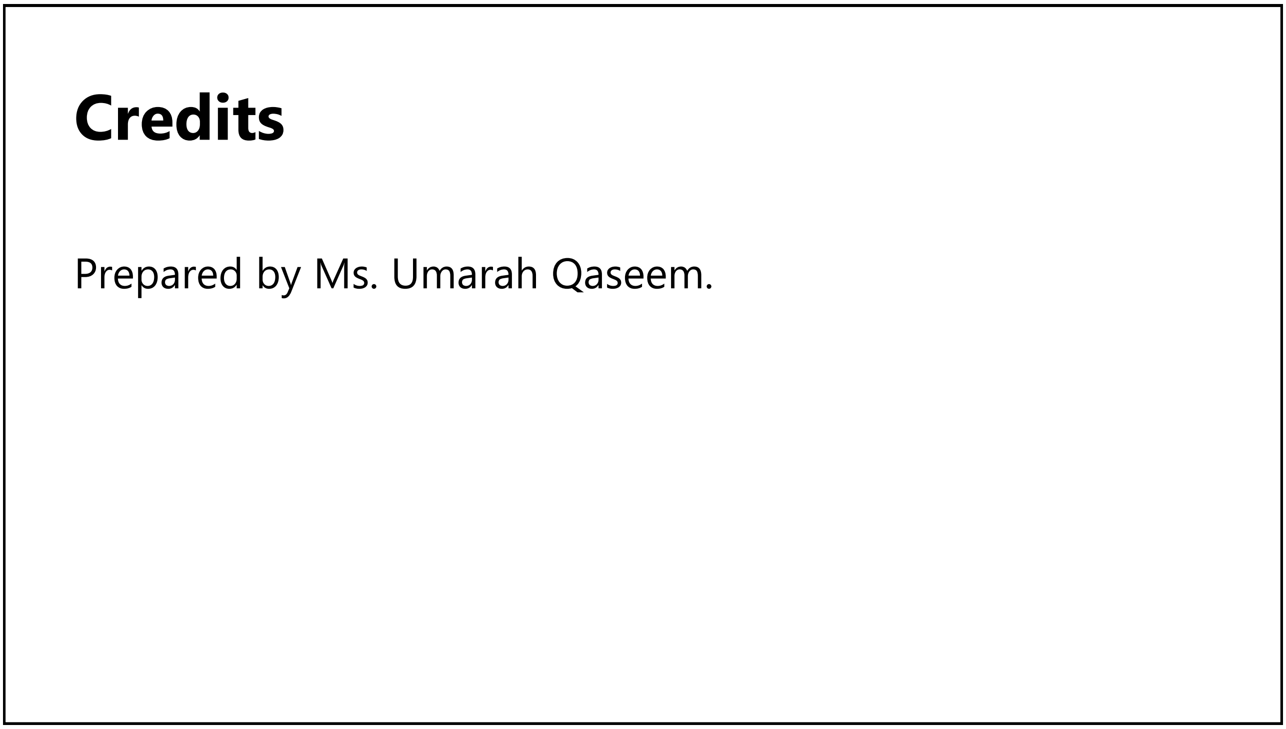
```
for ( int row = 0; row < n_rows;
++row )
{// Allocate the column array for
this row:
      matrix[row] = new
int[n_columns];
}

for (int row = 0; row < n_rows;
++row)
{
   for (int col = 0; col <
   n_columns; ++col)
   {
      matrix[row][col] = rand() %
   100;
   }
}
```

81



2D Array Example

```
for (int row = 0; row < n_rows; ++row)
{
     delete[] matrix[row];
}
// ... and then delete the row pointer
array itself:
delete [] matrix;
```



Credits

Prepared by Ms. Umarah Qaseem.