Review

Outline

• Array
• Pointer
• Object-Oriented Programming
Array

- Arrays are data structures containing related data items of same type.
- An array is a consecutive group of memory locations.

Elements of An Array

- To refer to a particular location or element in the array, we specify the name of the array and the index of the element.
- The index must be integer or integer expression, its value must be larger than or equal to 0.
- First element has index zero.
- The element could be used as a variable.
- Example
  \[ c[\ 0\ ] += 2; \]
  \[ c[\ a + b\ ] = 3; \]
Example

- An array \( c \) has 12 elements (\( c[0], c[1], \ldots, c[11] \)); the value of \( c[0] \) is \(-45\).

Declare An Array

- Declaration of an array:
  \[
  \text{type arrayName[ arraySize ]};
  \]
- Example
  \[
  \text{int c[ 12 ]};
  \]
- Array’s size must be an integer constant greater than zero.
- Arrays can be declared to contain values of data types like \text{int}, \text{float}, \text{double}, \text{char}, \text{etc.}
Initialize An Array (1)

• With an initializer list
  – Initializer list
    • Items enclosed in braces ({}) and separated by commas.
  – Examples
    • `int n[5] = {10, 20, 30, 40, 50};`
    • `int m[5] = {10, 20, 30};`
      – The remaining elements are initialized to zero.
    • `int p[] = {10, 20, 30, 40, 50};`
      – Because array size is omitted in the declaration, the compiler determines the size of the array based on the size of the initializer list.

Initialize An Array (2)

• Using a loop to initialize the array’s elements
  – Declare array, specify number of elements
  – Use repetition statement to loop for each element

• Example:
  ```c
  int n[10];
  for ( int i = 0; i < 10; i++)
  {
    n[ i ] = 0;
  }
  ```
Using An Array

• Usually use `for` loop to access each element of an array.

• C++ has **no array bounds checking**
  – Referring to an element outside the array bounds is an execution-time logic error. It is not a syntax error.
  – You need to provide the correct array size.

Using An Array – Example 1

• Summing the elements of an array

• Example:

```c++
const int arraySize = 6;
int a [ arraySize ] = { 87, 69, 45, 45, 43, 21 };
int total = 0;
for ( int i = 0; i < arraySize; i++)
{
    total += a[i];
}
cout <<"Total of array elements is " <<total<<endl;
```
Passing Arrays as Parameters

- In C++, arrays are always passed by reference, and & is not used with the formal parameter type.
- Whenever an array is passed as a parameter, its base address is sent to the called function.
- Example:
  ```cpp
  // prototype
  float SumValues ( float values [],
                   int numOfValues );
  ```

C++ Tips

Pass-by-value sends a copy of the contents of the actual parameter

SO,
the actual parameter cannot be changed by the function
C++ Tips

Pass-by-reference sends the location (memory address) of the actual parameter

SO,
the actual parameter can be changed by the function

const array parameter

Because arrays are always passed as reference parameters, you can protect the actual parameter from unintentional changes by using const in formal parameter list and function prototype.

FOR EXAMPLE . . .

```cpp
// prototype
float SumValues(const float values[],
    int numOfValues);
```
// Pre: values[0] through values[numOfValues-1]
//      have been assigned
// Returns the sum of values[0] through
//      values[numOfValues-1]

float SumValues (const float values[],
                int numOfValues )
{
    float sum = 0;
    for ( int index = 0; index < numOfValues; index++ )
    {
        sum += values[ index ];
    }
    return sum;
}

Multidimensional Array

- Multidimensional arrays with two dimensions
  - Called two dimensional or 2-D arrays
  - Represent tables of values with rows and columns
  - Elements referenced with two subscripts ([x] [y])
  - In general, an array with \( m \) rows and \( n \) columns is called an \( m\text{-by-}n \) array
- Multidimensional arrays can have more than two dimensions
Declaring & Initializing Two-Dimensional Arrays

- Declaring two-dimensional array `b`
  - `int b[3][2] = {{1, 2}, {3, 4}};`
  - 1 and 2 initialize `b[0][0]` and `b[0][1]`
  - 3 and 4 initialize `b[1][0]` and `b[1][1]`
  - 0 and 0 initialize `b[2][0]` and `b[2][1]` (implicitly).

```
Row 0
    | 1 | 2 |
Row 1
    | 3 | 4 |
Row 2
    | 0 | 0 |
```
Passing Multidimensional Arrays to Functions

- Multidimensional array parameters
  - Size of first dimension is not required
    - As with a one-dimensional array
  - Size of subsequent dimensions are **required**
    - Compiler must know how many elements to skip to move to the second element in the first dimension
  - Example
    - `void printArray( const int a[][ 3 ];`
      - Function will skip row 0's 3 elements to access row 1's elements (`a[ 1 ][ x ]`)
Multidimensional-Array Manipulations

- Commonly performed with `for` statements
  - Example
    - Modify all elements in a row
      ```c
      for ( int col = 0; col < 4; col++ )
          a[ 2 ][ col ] = 0;
      ```
  - Example
    - Total all elements
      ```c
      total = 0;
      for ( int row = 0; row < 3; row++ )
          for ( int col = 0; col < 4; col++ )
              total += a[ row ][ col ];
      ```

Pointer Types

**Pointer variable**

- A variable whose value is the address of a location in memory

```c
int* intPointer
```
Pointer Types

```c
int alpha;
int* intPointer;
intPointer = &alpha;
```

*If alpha is at address 33, memory looks like this*

---

```c
int x;
x = 12;

int* ptr;
ptr = &x;
```

NOTE: Because ptr holds the address of x, we say that ptr “points to” x
Pointer Types

• **Dereference operator (*)**
  – An operator that, when applied to a pointer variable, denotes the variable to which the pointer points

```cpp
int x;
x = 12;

int* ptr;
ptr = &x;
std::cout << *ptr;
```

*ptr is the value in the place to which ptr points
### Pointer Types

```c
int x;
x = 12;

int* ptr;
ptr = &x;
*ptr = 5;

// changes the value
// at address ptr to 5
```

---

```c
char ch;
ch = ‘A’;

char* q;
q = &ch;
*q = ‘Z’;
char* p;
p = q;

// the right side has value 4000
// now p and q both point to ch
```
Pointer Types

- **Dynamic allocation (new operator)**
  - Allocation of memory space for a variable at run time (as opposed to static allocation at compile time)

```c
int * intPointer;
intPointer = new int;
```
Pointer Types

• **NULL Pointer**
  – A pointer that points to nothing; available in `<cstdlib>`
  – NULL is defined to be 0;
  – But NULL is not memory address 0;
  int * intPtr = NULL;
  Float * money = NULL;

![Diagram of intPtr and money]

• **Memory Leak**
  – The loss of available memory space that occurs when memory is allocated dynamically but never deallocated
  float * myMoney = new float;
  float * money = new float;
  myMoney = money; //wrong

• **Garbage**
  – Memory locations that can no longer be accessed
  • To avoid memory leaks, use `delete`.
  delete myMoney; //correct
  myMoney = money;
Pointer and Array

• Dynamically create array with pointers

```c
int * foo;
foo = new int[5];
delete [] foo;
```

– The system dynamically allocate space for 5 elements of type int and returns the memory address of the first element of the array, which is assigned to foo (the pointer).

```
+-------------------+-------------------+-------------------+
|                   |                   |                   |
|                   |                   |                   |
|                   |                   |                   |
+-------------------+-------------------+-------------------+
```

```
   int
     
   foo
```

Object-Oriented Programming

• objects, sending a message, methods, instance variables….

• **Object**
  – An instance of a class

• **Method**
  – A public member function of a class

• **Instance variable (Data Member)**
  – A private data member of a class
C++ class data type

- A class is an unstructured type that encapsulates a fixed number of data components (data members) with the functions (called member functions) that manipulate them.

```cpp
class DateType // declares a class data type
{
public:
    DateType (int newMonth, int newDay, int newYear); // constructor
    int getYear() const; // returns year
    int getMonth() const; // returns month
    int getDay() const; // returns day

private:
    int year;
    int month;
    int day;
};
```
2 separate files for class type

// SPECIFICATION FILE (datatype.h)
// Specifies the data and function members.
class DateType
{
    public:
        .
    private:
        .
};

// IMPLEMENTATION FILE (datatype.cpp)
// Implements the DateType member functions.
.
.

Code Using DateType

#include "datatype.h"  // includes specification of the class
#include <iostream>
using namespace std;

int main()
{
    // declares 2 objects of DateType
    DateType startDate (6, 30, 1998);
    DateType endDate (10, 31, 2002);
    bool retired = false;

    cout << startDate.getMonth() << "/" << startDate.getDay() << "/" << startDate.getYear() << endl;
    while(!retired)
    {
        finishSomeTask();
    }
    return 0;
}
Implementation of member functions

```cpp
// IMPLEMENTATION FILE  (datatype.cpp)
#include "datatype.h"  // also must appear in client code

DataType :: DataType ( int  newMonth, int  newDay,
                        int  newYear )
// Post:  year is set to newYear.
// month is set to newMonth.
// day is set to newDay.
{
    year    =  newYear ;
    month   =  newMonth ;
    day      =  newDay ;
}

int DateType :: getMonth (  ) const
// Accessor function for data member month
{
    return  month ;
}

int DateType :: getYear (  ) const
// Accessor function for data member year
{
    return  year ;
}

int DateType :: getDay (  ) const
// Accessor function for data member day
{
    return  day ;
}
```
Object-Oriented Programming

• Three ingredients in any object-oriented language
  – encapsulation
  – inheritance
  – polymorphism

Just as a capsule protects its contents, the class construct protects its data members, but what are inheritance and polymorphism?

Object-Oriented Programming

• Inheritance
  – A mechanism used with a hierarchy of classes in which each descendant class inherits the properties (data and operations) of its ancestor class

• Base class
  – The class being inherited from

• Derived class
  – the class that inherits
Object-Oriented Programming

Inheritance is an “is-a” relationship:
- a wheeled vehicle is a vehicle;
- a bicycle is a wheeled vehicle;
- a four-door car is a car…

Object-Oriented Programming

- **Overriding**
  - Function with `virtual` keyword in base class.
  - Derived classes override function as appropriate.
  - An overridden function in a derived class has the same signature and return type (i.e. prototype) as the function it overrides in its base class.

- **Polymorphism**
  - The ability to determine which of several operations with the same name is appropriate; a combination of static and dynamic binding
Each class has a method `Print`:
- `Person.Print` just prints the name
- `Employee.Print` prints the name and job title
- `Manager.Print` prints name, job title, and department

`Print` is overridden.

Static binding is when the compiler can tell which `Print` to use; dynamic binding is when the determination cannot be made until runtime.

Object-Oriented Programming

- **Inheritance** and **polymorphism** work together
- **How?**
- They combine to allow the programmer to build useful **hierarchies of classes** that can be put into a library to be reused in different applications
Reference

• Reproduced by permission of Jones and Bartlett Publishers International.