ARRAYS AND VECTORS

Chapter 8

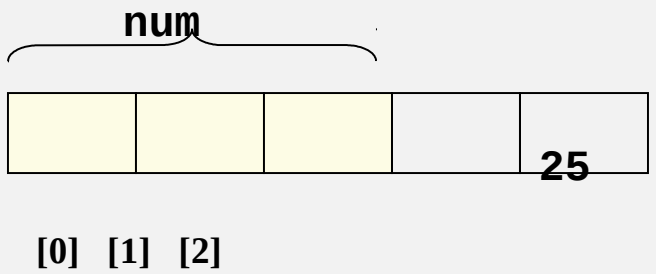
ARRAYS

- C++ arrays, for the most part, are just like C-style arrays.
  - Allow you to store and work with multiple values of the same data type
  - Declared the same
  - Access elements the same
  - Store and display data the same

ARRAYS

- C++ does not perform bounds checking
  - This means C++ does not check to see if an array subscript is in range
- An invalid array subscript can cause a program to overwrite other memory
- Example:

```
const int ISIZE = 3;
int num[ISIZE];
num[4] = 25;
```



VECTOR

- Advantages over arrays
  - Do not have to declare the number of elements that the vector will have
  - If you add a value to a vector that is already full, the vector will automatically increase its size to accommodate the new value
  - Vectors can report the number of elements they contain, pointers can not

VECTOR

- #include <vector>
- How to create a vector

important

  - vector<int> numbers; //defines a vector of int
  - vector<int> numbers(10); //defines a vector of 10 ints
  - vector<int> numbers(10,2);//defines a vector of 10 ints and initializes them to the value of 2.
  - Vector<int> set2(set1);//the vector set2 will have the same number of elements and hold the same set of values as set1

VECTORS - PUSH_BACK AND RESIZE

- A vector can grow in size as needed. Arrays can not
- In order to add an element to a vector when the vector is full **OR** had no initial defined size you will need to use a function called push_back
- You can also change the size of a vector by using a function provided resize(N)
- Ex. vector<int> test;
  test.push_back(75); //This adds 75 to the end of the vector
- Ex. vector<int> scores(15);
  scores.resize(25); adds 10 to the size of the vector.

DETERMINE SIZE OF A VECTOR

- Use size member function to determine **number of elements** currently in a vector

  Ex. howbig = scores.size();

REMOVING AN ELEMENT FROM A VECTOR

- Use pop_back member function to remove last element from a vector

  Ex. scores.pop_back();//Does NOT return a value only removes the last element
- To remove all contents of a vector, use the function **clear()** – leaves the size of the vector to be 0

  scores.clear();
- To determine if vector is empty, use the function **empty()**

  while (!scores.empty()) ...
- prog8_25.cpp
- prog8_27.cpp
- resize.cpp vectorPractice.cpp

VECTOR OF OBJECTS

- We can also have a vector of objects.
- Let’s think about a Birthday class
- vector<Birthday> bDays(2);
- You can also pass a contructor to a call to push_back

  bDays.push_back(Birthday(3,3,2017));

ARRAY OF OBJECTS

- Just as you can have an array of int’s you can have an array of objects.

```
class Square
{ private:
    int side;
  public:
    Square(int s = 1)
    { side = s; }
    int getSide()
    { return side; }
};
Square shapes[10];  // Create array of 10
                    // Square objects
```

ARRAYS OF OBJECTS

- Like an array of structures, use an array subscript to access a specific object in the array
- Then use dot operator to access member methods of that object

```
for (i = 0; i < 10; i++)
  cout << shapes[i].getSide() << endl;
```

INITIALIZING ARRAYS OF OBJECTS

- We can use default constructor to initialize the array of objects.
- We can use an initialization list – only works if there is only one data member in the class

  Square shapes[5] = {1,2,3,4,5};
- Default constructor is used for the remaining objects if initialization list is too short - only works if there is only one data member in the class

  Square boxes[5] = {1,2,3};

INITIALIZING ARRAYS OF OBJECTS

- If an object is initialized with a constructor that takes > 1 argument, the initialization list must include a call to the constructor for that object

  Rectangle spaces[3] = { Rectangle(2,5), Rectangle(1,3), Rectangle(7,7) };

STL (C++ STANDARD TEMPLATE LIBRARY) ITERATORS

- The Standard Template Library provides us with pointers to assist us in iterating through a vector or other containers provided by the STL
  - These are called iterators
  - iterator.cpp

DYNAMICALLY ALLOCATING MEMORY IN C++

DYNAMIC MEMORY ALLOCATION

- We know how to dynamically allocate memory in C.
- In C++ we use the **new** operator
  - double *dptr;
  - dptr = new double;
  - **new** returns an address of the memory location
  - Can use new to allocate an array or a specific amount of something
    - arrayPtr = new double[25];
    - now can access memory using [] array notation

RELEASING DYNAMIC MEMORY

- Use delete to free dynamic memory

  delete count;

- Use delete [] to free dynamic array memory

  delete [] arrayptr;

- Only use delete with dynamic memory!

DANGLING POINTERS AND MEMORY LEAKS

- A pointer is dangling if it contains the address of memory that has been freed by a call to **`delete`**.
  - Solution: set such pointers to 0 as soon as memory is freed.
- A memory leak occurs if no-longer-needed dynamic memory is not given back to the OS. The memory is unavailable for reuse within the program.
  - Solution: Give the memory back to the OS (delete in C++, free in C)
  - prog10_14.cpp

DYNAMIC MEMORY WITH OBJECTS

- Can allocate dynamic structure variables and objects using pointers:

```
stuPtr = new Student;
```

- Can pass values to constructor:

```
squarePtr = new Square(17);
```

- delete causes destructor to be invoked:

```
delete squarePtr;
```

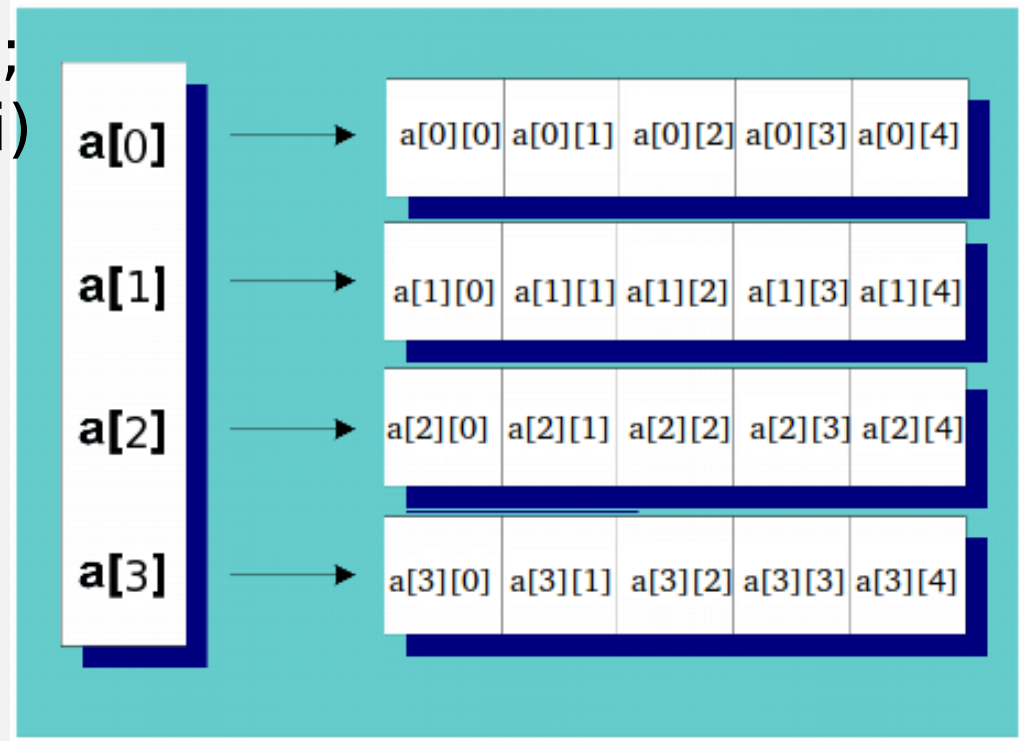
CONTROLLING MEMORY LEAKS

- Memory that is allocated with **`new`** should be de-allocated with a call to **`delete`** as soon as the memory is no longer needed. This is best done in the same function as the one that allocated the memory.
- For dynamically-created objects, **`new`** can be used in the constructor
- **`delete`** should be in the **destructor** if new was called in the constructor

Prog10_18.cpp

ALLOCATING FOR A 2D ARRAY IN C++

```
int** ary = new int*[rowCount];
for(int i = 0; i < rowCount; ++i)
{
    ary[i] = new int[colCount];
}
To delete
for(i =0; i< rowCount;  i++)
{
    delete [] ary[i];
}
delete [] ary;
```



INITIALIZING VALUES IN C++11

- In C++ 11, putting empty { } after a variable definition indicates that the variable should be initialized to its default value
  - int x {}; this tells the complier to initialize **x** to the standard default value
  - This will also work with a pointer
  - int *x {}
- In C we initialize a pointer to either 0 or NULL
  - The problem with this is this indicated the address is 0 (NULL translates to 0)
- C++ 11 also has the key word **`nullptr`** to indicate that a pointer variable does not contain a valid memory location
  - nullptr translates to “no valid address”
  - null.cpp nullptr.cpp

SMART POINTER C++11

- Objects that work like pointers but have the ability to automatically delete dynamically allocated memory
- They can be used to solve the following problems in a large software project
  - dangling pointers – pointers whose memory is deleted while the pointer is still being used
  - memory leaks – allocated memory that is no longer needed but is not deleted
  - double-deletion – two different pointers de-allocating the same memory

SMART POINTER C++11

- **Smart pointers** are objects that work like pointers.
- Unlike regular (raw) pointers, smart pointers can automatically delete dynamic memory that is no longer being used.
- There are three types of smart pointers:
  - unique pointers(unique_ptr)
  - shared pointers(shared_ptr)
  - weak pointers(weak_ptr) This will not be covered in this class

UNIQUE POINTER C++11

- #include <memory>
- A unique pointer points to a dynamically allocated object that has a single owner
- Ex. unique_ptr<int> uptr1(new int);

This points to an int. When this pointers goes out of scope the the unique_ptr class will delete the memory associated with this pointer.

- Another ex. unique_ptr<int> uptr2;

    uptr2 = unique_ptr<int> (new int);

UNIQUE POINTER C++11

- To avoid memory leaks, objects that are managed by smart pointers should have no other references to them.
- Should **NOT** do the following:

```
int *p = new int;
unique_ptr<int> uptr(p);
```

- Smart pointer do not support pointer arithmetic (uptr++, uptr = uptr +2)
- Does support dereferencing: (* and ->)
- Can not initialize a unique_ptr with the value of another unique_ptr object
  - Unique_ptr<int> uptr1(new int);
  - Unique_ptr<int> uptr2 = uptr1; not allowed.

UNIQUE POINTER C++11
MOVE()

- C++ provides a the move() function that transfers ownership from one unique pointer to another

```
unique_ptr <int> uptr1(new int);
*uptr1 = 15;
unique_ptr<int> uptr2(new int);
uptr2 = move(uptr1);  //what happens:  uptr2 is deallocated and then uptr1 gives ownership of the memory to uptr2.
uptr1 = nullptr;
cout << *uptr2 << endl;   // this will print 15
```

CLEARING A UNIQUE POINTER

- Unique pointers deallocate the memory for their objects when they go out of scope.
- To manually deallocate memory, use:
  - uptr = nullptr; or uptr.reset();
  - C++14 has made additional changes to unique pointers. We will not discuss this at this time.

unique_ptr.cpp

UNIQUE POINTER AND ARRAYS

- Use array notation when using an unique pointer to allocate memory for an array
  - Unique_ptr<int[]> uptr(new int[5]);
- Doing so ensures that the proper deallocation(delete[] instead of delete) will be used.

SHARED POINTERS

- A **shared pointer** points to a dynamically allocated object that may have multiple owners.
- A control block manages the reference count of the number of shared owners and also possibly the raw pointer if one exists.

sharedTest.cpp

SHARED POINTERS

- Be careful that all references to a dynamic object are tracked in the same control block
- In the code below:

```
int * rawPtr = new int;
shared_ptr<int> uptr4(rawPtr);
shared_ptr<int> uptr5(rawPtr);
```

- Two control blocks are created. This can cause a dangling pointer.
- Cannot create an array of shared pointers.