CS2024 – C++ PROGRAMMING

Lecture #8: Arrays and Vectors
C++: How to Program – Chapter 7

STANDARD TEMPLATE LIBRARY

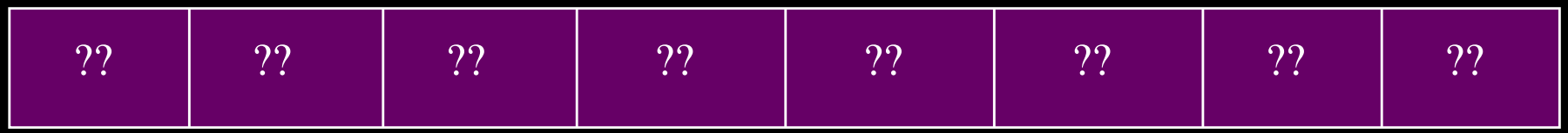
- Last lecture we touched on the concept of *template functions*
- They are “blueprints” that let us create new, overloaded versions of functions based on the data types we pass into them
- The concept of *templates* extends into classes as well and we will cover this generically in a future lecture
- You can imagine that a *class template* would allow you to have different versions of a class based on the data types used with it
- C++ has an entire library of such classes (in addition to you being able to create your own such classes)
- Today we will cover two such classes: array and vector
- The array type was available as part of the *Boost Libraries* in some previous versions of C++ but is now available in the std:: library with C++11

ARRAYS

- What is an Array?
  - A “chunk” of memory which contains consecutive instances of a given data type
  - Think of it as a “list” of data, each data item of the same data type

```
array<int,8> iArray;      // Available as std::array in C++11
```

- At this point, iArray might look like this:



- Memory is allocated, but not initialized (hence the ??s)

ARRAYS

- In order to make use of the array data type, we must include the corresponding header

```
#include <iostream>
#include <array>          // Allows us to use array
using namespace std;

int main(int argc,char *argv[])
{
  array<int,5> myArray;

  for (int i{0}; i<5; i++)
    cout << myArray[i] << endl;
}
```

- Memory still not initialized, let's see what happens

DEMONSTRATION #1

An uninitialized array

ARRAYS

- As stated previously, the memory needed for the array is allocated, but we haven't stored any data in there yet. Let's look at a couple of ways to store data

```
#include <iostream>
#include <array>          // Allows us to use array (C++11)
using namespace std;

int main(int argc,char *argv[])
{
  array<int,5> myArray;   // Still uninitialized
  array<int,5> primeArray{2,3,5,7,11}  // Initialize C_++11 way

  // Initialize myArray to prime number + 1
  for (int i{0}; i<5; i++)
    myArray[i] = primeArray[i]+1;
}
```

GOOD PROGRAMMING PRACTICE

- In the previous examples I have been passing “hard coded integers” (such as “5”) as the array size
- This is syntactically acceptable, but it doesn’t make for readable code
  - If you are looking at my code and see a “5” you don’t know if…
    - 5 was an arbitrary number
      - In the old days we could be lax about the size of things
      - “Eh, 5 should be enough”
    - 5 was meaningful as the maximum size of something

GOOD PROGRAMMING PRACTICE

- If we use a variable to hold the size of the array we want to allocate the variable name could tell us something

```
int main()
{
  // how many times did you visit the gym this year?
  int numberOfMonths{12};
  array<string,numberOfMonths> gymVisitsThisYear;

  // What are you assignment grades for this class?
  int numberOfAssignments = 12;
  array<double,numberOfAssignments> myGrades;

  // rest of code…
}
```

GOOD PROGRAMMING PRACTICE

- Using this method, our code (and our intent) is clearer.
- It also means that we can re-use the constant in other parts of our code…

```
int main()
{
  // What are your assignment grades for this class?
  int numberOfAssignments = 12;
  array<double,numberOfAssignments> myGrades;

  //  rest of program happens here…

  // print out my grades
  for (int i=0; i<numberOfAssignments; i++)  // re-use var
    cout << "Assignment #" << i << ": " << myGrades[i] << endl;
}
```

BETTER PROGRAMMING PRACTICE

- There is one addition we can make that will improve our code even more
- Mark `numberOfAssignments` as `const`

```
int main()
{
  // What are you assignment grades for this class?
  const int numberOfAssignments = 12;
  array<double,numberOfAssignments> myGrades;

  //  rest of program happens here…

  // print out my grades
  for (int i=0; i<numberOfAssignments; i++)  // re-use var
    cout << "Assignment #" << i << ": " << myGrades[i] << endl;
}
```

BETTER PROGRAMMING PRACTICE

- Marking a variable as `const` means that any attempts to modify it will result in a compiler error.
- Since we are using the variable to represent a size that shouldn't change during the execution of our program, this is a good thing!
- `const` can be used in many situations to mark something as "something that shouldn't change"
- We'll see more of this later, but for now let's look at it in the context of our array example…

DEMONSTRATION #2

Array initializations and the `const` keyword

USING ARRAYS – BOUNDS CHECKING

- What do you suppose happens when we do the following?

```
array<int,5> someArray;
cout << someArray[6] << endl;
```

- Our array is only “allocated” with 5 elements.
- We’ve asked for element 6 which is really the 7th element.
- C++ will not flag this code as an error, nor will there be anything that happens at runtime to signal an error.
- Instead, C++ will calculate the memory address of the 7th element and return whatever is there (even though it isn’t part of the data structure (array) we allocated

USING ARRAYS – BOUNDS CHECKING

- This is worse:

```
array<int,5> someArray;
someArray[6] = 5;
```

- Now we are attempting to *write* to an element in the array that technically doesn't exist
- This memory could belong to another variable and/or data structure which would be "mangled" by our erroneous code
- This type of error (though very obvious in the above example) is the cause of some of the hardest to find bugs that can happen in C/C++

USING ARRAYS – BOUNDS CHECKING

- The previous two examples can be summarized by saying that C++ does no *bounds checking* when accessing array items
- You are responsible to know how big the array is and only access elements that are within its bounds.
- If you don’t have a local (const) variable handy to tell you how big the array is, you can always use the size() method on the array variable:

DEMONSTRATION #3

Using arrays with the `size()` method

RANGE BASED FOR

- In C++11, there is a new twist on the for loop.
- Instead of the standard 3-expression format we've used so far, we can specify a new *range based* format…

```
#include <iostream>
#include <array>
using namespace std;

int main(int argc, char *argv[])
{
    array<int, 5> myArray{ 5,8,2,4,3 };

    for (int item : myArray)
      cout << "Next item is: " << item;
}
```

SORTING AND SEARCHING

- Two common operations you might perform on arrays would be *sorting* and *searching*
- Sorting refers to re-arranging the elements in the array as per some ordering scheme
  - Alphabetize
  - Arrange “lowest” to “highest”
- Searching refers to determining if a given element exists in the array

SORTING

- To sort an array, we use the Standard Template Library method sort()

```
#include <iostream>
#include <array>
using namespace std;

int main(int argc, char *argv[])
{
    array<int, 5> myArray{ 5,8,2,4,3 };   // C++11 initialization

    sort(myArray.begin(),myArray.end());  // let's sort!

    for (int num : myArray)   // New range-based format
      cout << "Next item: " << num << endl;
}
```

SORTING

- Don’t worry about the exact meaning of the `begin()` and `end()` methods
  - They are common/available to all Standard Template Library types
  - The sort() method takes the starting point and ending point on which to apply the sort
  - `begin()` and `end()` specify the start and end of the array, respectively
- So, the following:
  - myArray.sort(myArray.begin(),myArray.end());
  - Sorts the entire array
- More on begin/end later this semester

SEARCHING

- To search an array, we use the binary_search() method

```
#include <iostream>
#include <array>
using namespace std;

int main(int argc, char *argv[])
{
    array<int, 5> myArray{ 5,8,2,4,3 };   // C++11 initialization

    bool found = binary_search(myArray.begin(),myArray.end(),2);

    if (found)
      cout << “We found 2” << endl;
    else
      cout << “2 was NOT found” << endl;
}
```

SEARCHING

- `binary_search()` takes the same first two parameters as the sort() method, but adds a third parameter: the element to search for
- So, the following:
  - myArray.sort(myArray.begin(),myArray.end(),2);
  - Looks for the element “2” in myArray
- Returns a “boolean” (true/false)
  - Returns `true` if the element was found
  - Returns `false` if not

DEMONSTRATION #4

Sorting, Searching and range-based `for` loops

MULTI-DIMENSIONAL ARRAYS

- You can have an array of arrays to create a *two-dimensional array*

```
#include <iostream>
#include <array>
using namespace std;

int main(int argc, char *argv[])
{
  const int columns = 3; // our multi-dimensional array will
  const int rows = 2;    // have 3 columns and 2 rows
  array<array<int,columns>,rows> = array1{1,2,3,4,5,6};

  for (array<int,3> aRow: array1)  // iterate through rows
    for (int x : aRow)                 // iterate through cols
      cout << x << endl;
}
```

DEMONSTRATION #5

Multi-Dimensional Arrays

INTRODUCTION TO VECTORS

- A `vector` is a container holding an arbitrary number of like-typed objects
- This is very similar to the Java equivalent
- A vector is a "higher level" array as you don't need to specify how big it is when declaring it
- C++ will also do *bounds checking* to prevent you from reading or writing from memory you shouldn't ☺
- Objects in a vector, however, may be accessed as if the vector was an array (using the a[n] notation)
- Like the `array` class we just looked at, you must specify the type of the items in the `vector` when you declare it.

INTRODUCTION TO VECTORS

A vector is declared as follows:

```
vector<int> intVector;  // declares a vector of ints
vector<storage type> variableName;
```

- We use the keyword `vector` followed by a left angle bracket, followed by a data type, followed by a right angle bracket
- Then we use whitespace followed by a variable name
- This declares a `vector` represented by the variable name specified which contains an arbitrary number of objects of the data type specified
- In our specific example above, `intVector` is declared as a variable that holds a vector of `int` objects
- How do we get data in and out of a `vector`?

INTRODUCTION TO VECTORS

- Keep in mind that a vector is a C++ class that has member functions.
- There are multiple ways to get data in and out.
- One set of operations are “stack” operations (push, pop)

```
vector<int> intVector;  // declares a vector of ints

// call the push_back method to put an element in the vector
intVector.push_back(1);
intVector.push_back(4);

// Use the back() method to retrieve reference to last object
cout << “last item is: “ << intVector.back() << endl;
// pop_back() removes the last element
intVector.pop_back();
```

INTRODUCTION TO VECTORS

- back() return the last element of the vector
- pop_back removes the last element of the vector
- To verify, we enlist the help of additional methods
  - Size() -- returns how many elements are in the vector
  - isEmpty() – returns true if the vector is empty

```
vector<int> intVector;  // declares a vector of ints

// call the push_back method to put an element in the vector
intVector.push_back(1);
intVector.push_back(4);
Cout << "There are " << intVector.size() << " items\n";
Cout << "last one is: " << intVector.back() << endl;
intVector.pop_back();
Cout << "There are now " << intVector.size() << " items\n";
```

INITIALIZING VECTORS

- You can use C++11 unified initialization syntax to supply to initial elements in the vector
- You can utilize a constructor that takes an integer to specify how much room there is in the vector

```
// declare a vector of prime integers
vector<int> primeVector{2,3,5,7,11,13};

// declare a vector with room for 5 elements
vector<int> intVector(5);

cout << “size of primeVector is “ << primeVector.size() << endl;
cout << ”size of intVector is “ << intVector.size() << endl;
```

USING [] WITH VECTORS

- You may also use array notation ([]) to access elements in the vector.
- Be careful. There is no bounds checking and crashes can result if you write to an element that is bigger than the size of the vector

```
// declare a vector of prime integers
vector<int> primeVector{2,3,5,7,11,13};

// print the primes
for (int i=0; i<primeVector.size(); i++)
    cout << primeVector[i] << endl;

// But be careful!
primeVector[6] = 17;    // This will likely crash!
```

USING [] WITH VECTORS

- Using vector methods that assume there is content in the vector when there isn't will lead to *undefined results* which will likely involve your application *crashing*
- You can use the at() method to get bounds checking

```
// declare a vector of prime integers
vector<int> primeVector{2,3,5,7,11,13};

// print the primes
for (int i=0; i<primeVector.size(); i++)
    cout << primeVector.at(i) << endl;

// You can catch the following, but more on that later...
primeVector.at(6) = 17;    // This will result in exception
```

DEMONSTRATION #6

Vectors

FINAL THOUGHTS

- Assignment #4 posted
  - You still only need to submit an a4.cpp file and a writeup
  - You will need to have separate functions in your a4.cpp file