Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

June 29, 2018

Vectors

- This lecture contains a brief introduction to **C++ vectors.**
- A vector is essentially a dynamically resizeable array.
- Vectors are simple but very useful things, which we shall employ in many applications.
- Vectors are part of something known as the **Standard Templates Library (STL).**
- The STL is an important part of the C++ language.
- The STL is a large body of software.
- This lecture offers only a small glimpse of a large body of software.

1 Arrays

- Consider an array a of integers, of length 10. We would declare it as follows.

```
int a[10];
```

- This is straightforward enough.
- However, the length of the array is fixed at 10, a number we need to know in advance.
- Suppose, as a simple and easily visualized example, we wish to have arrays x and y of type `double`, to hold the (x, y) coordinates of the vertices of a polygon.
  1. *How many vertices does the polygon have?*
  2. In general, a polygon has n vertices, where n is a positive integer and $n \geq 3$.
- ***However, and this is the key point, the value of n is not known in advance.***
  1. C++ does not permit us to declare arrays of unknown length.
  2. The following array declaration is not legal according to the C++ standard.

```
int n;                          // not initialized yet, and not a constant
double x[n], y[n];              //  *** n = undefined value ***
n = 10;
x[2] = 1.2;
y[3] = 3.4;
```

  3. Some older compilers will compile the above statements. However, the program will generate a run time fault if executed. *(I know, I tried.)*
  4. Moreover, we cannot declare empty arrays and fill in the size later.

```
//  no size specified, fill it in later
double x[], y[];                //  *** COMPILATION ERROR ***
```

  5. *Actually, we can, but not by using arrays as declared above.*
  6. We shall learn later how to do so.
- Given all of the above weak points, what alternatives do we have?
- **This is where C++ vectors enter the picture.**

2 Digression: vectors, strings & C++ classes

- Before we proceed further with vectors, it is probably helpful to digress briefly to comment about **strings.**
- You have already used strings in previous programming courses.
- For example, you should understand (and be able to write) the following simple program.

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

int main()
{
  string s("abcd");
  cout << s << endl;

  s.insert(0, "ABCD");
  cout << s << endl;
  return 0;
}
```

- Both vectors and strings are examples of **C++ classes.**
- You have already come across strings and can perform various operations with strings.
- Here we shall learn how to perform various operations with vectors.
- We introduce vectors at this early stage because they are useful, just as strings are useful.
- Later, we shall begin a formal study of C++ classes.

3 Vectors: introduction

- In C++, we can declare things called **vectors** as follows.

```
vector<double> x;
vector<double> y;
```

- At the simplest level, ***which is all we shall study in this lecture,*** a vector is a "generalized array" with the following properties:

  1. The length need not be specified in advance (default = 0).
  2. The length can be specified later, during program execution.
  3. *The length can change, during program execution.*
  4. We are permitted to "resize" a vector (and it can be resized to a *smaller value,* if desired.)
  5. As opposed to a C++ array, ***a vector knows its length.***
  6. A vector contains internal information, including its length.
  7. This is part of the notion of a generalized array: a vector contains extra information.

- It is therefore more accurate to say that a vector *contains an array,* and supports the functionality of an array.

- A vector obviously also contains much more internal structure.

- However, we shall only treat vectors as generalized arrays for now.

- In this lecture we shall learn about simple but important and useful properties of vectors.

4 Vectors: declaration

- There are clearly many things to learn about vectors, so let us begin with the simplest.
- ***How do we declare a vector?***
- First of all, how do we declare ordinary C++ arrays?
- Let us declare arrays of type `char`, `int` and `double`, all of length 10.

```
char   arr_c[10];
int    arr_i[10];
double arr_d[10];
```

- We declare arrays slightly differently, with an admittedly strange notation. We write

```
vector<char>    vec_c;
vector<int>     vec_i;
vector<double>  vec_d;
```

- The data type `char`, `int` and `double` is written in angle brackets.
- We shall learn why later, but will accept it as a peculiar feature for now.
- The length was not specified, and the above vectors all have default lengths of zero.
- We can specify a length in the declaration.

```
int n = 10;
vector<char>    vec_c(n);
vector<int>     vec_i(n);
vector<double>  vec_d(n);
```

- **Note the syntax: it is parentheses, i.e. (n), not square brackets $[n]$.**
- The notation "(n)" looks more like a function call rather than an array declaration.
- *And in fact it is, but this is another fact about a "generalized array" that we must accept for now and study in more detail later.*
- *If the value of n changes during program execution, the lengths of the above vectors are not affected.*
- **As stated above, a vector knows its length.**
- The above value of n is used merely to initialize the vector lengths (to 10 in this case).
- *The lengths of the vectors do not depend on the value of n at later times during program execution.*

5 Vectors: declaration with initialization

- An ordinary C++ array contains uninitialized memory when it is declared.
- However, it is also possible to supply initialization values.

```
int a[] = {1, 2, 4, 8, 16};        // declaration with initialization
```

- **We can declare a vector and also specify an initial value for its elements.**

```
int n = 10;
char c = ' ';
vector<char>    vec_c(n, c);
vector<int>     vec_i(n, 1);
vector<double>  vec_d(n, 1.23456);
```

- If we do not specify data values, they will be defaulted to zero.

```
int n = 10;
vector<char>    vec_c(n);          // n elements, all zero
vector<int>     vec_i(n);          // n elements, all zero
vector<double>  vec_d(n);          // n elements, all zero
```

6 Vectors: get/set elements

- Before we bore ourselves to death with vector declarations, let us *do something* with vectors.
- Let us get some data in/out of vectors.
- Here is a simple but working C++ program to get/set the values of the elements of a vector.

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

int main()
{
  int n = 10;
  vector<int>     vec_i(n, 1);
  vector<double>  vec_d(n, 1.23456);

  // print initial values
  for (int i = 0; i < n; ++i) {
    cout << i << "    " << vec_i[i] << "     " << vec_d[i] << endl;
  }
  cout << endl;

  // set the values
  for (int i = 0; i < n; ++i) {
    vec_i[i] = 2*i;
    vec_d[i] = i + 0.5;
  }

  // print updated values
  for (int i = 0; i < n; ++i) {
    cout << i << "    " << vec_i[i] << "     " << vec_d[i] << endl;
  }

  return 0;
}
```

- **We require the "vector" header file `#include <vector>`.**
- **The syntax to get/set values is the same as for ordinary C++ arrays.**
- The expressions "`vec_i[i]`" and "`vec_d[i]`" get the values of the vector elements.
- The expressions "`vec_i[i] = 2*i`" and "`vec_d[i] = i + 0.5`" set the values of the vector elements.

7 Vectors: index out of bounds error

- *However, things cannot be so simple.*
- It was stated above thae we can declare a vector without specifying an initial length, in which case the length defaults to zero.
- Hence what would happen if we attempted to get/set array elements beyond the end of the vector (array out of bounds)?
- This would cause a run-time fault.
- Let us declare a vector without specifying an initial length.
- **The C++ program below compiles successfully, but generates a run-time error.**

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

int main()
{
  int n = 10;
  vector<int> vec_i;                          // length not specified, default = 0

  for (int i = 0; i < n; ++i) {
    cout << i << "    " <<  vec_i[i] << endl; // RUN TIME ERROR ARRAY OUT OF BOUNDS
  }
  cout << endl;

  return 0;
}
```

- **We must populate the vector before we attempt to get/set its elements.**
- The following code, to set the element values, would also generate a run-time error.

```
  vector<int> vec_i;                          // length not specified, default = 0
  vec_i[0] = 3;                               // RUN TIME ERROR ARRAY OUT OF BOUNDS
  vec_i[1] = 4;                               // RUN TIME ERROR ARRAY OUT OF BOUNDS
}
```

- Since the initial length of the vector is zero, the operation of populating the vector also increases its length.
- In this respect, a vector is significantly different from an ordinary array.
- **The length of a vector can be changed dynamically, during program execution.**

8 Vectors: populate a vector

- ***How do we populate a vector?***
- **New elements can be added to a vector only at the end of the vector.**
  1. **Elements can also be removed from the end of a vector.**
  2. In other words, we can both increase *and decrease* the length of a vector, during program execution.
  3. We shall study the removal of elements later.
- **To add extra elements to a vector, we employ the `push_back()` function.**
- The syntax of `push_back()` is a little strange.
- It will help to see a working example.
- The C++ program below pushes elements onto a vector.
- It also shows how to query the length of the vector.
- We only print data elements up to the known length of the vector.
- **The program below compiles and executes correctly.**

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

int main()
{
  vector<int> vec_i;          // length not specified, default = 0

  vec_i.push_back(5);         // push end of vector, length = 1
  vec_i.push_back(7);         // push end of vector, length = 2
  vec_i.push_back(6);         // push end of vector, length = 3
  vec_i.push_back(12);        // push end of vector, length = 4

  int n = vec_i.size();       // get length of vector

  // print values
  for (int i = 0; i < n; ++i) {
    cout << i << "    " <<  vec_i[i] << endl;
  }

  return 0;
}
```

- **See next page(s).**

8.1 Adding elements: `push_back`

- There is no simple way to explain the `push_back` function.
- Since this is a vector of elements of type `int`, the data values we "push" onto the end of the vector are obviously integers.
- In the above example, we "push back" four integers 3, 7, 6 and 12.
- After each `push_back` operation, the length of the vector increases by one.
- All of that is obvious enough.
  1. However, note the syntax.

     ```
     vec_i.push_back(5);
     vec_i.push_back(7);
     vec_i.push_back(6);
     vec_i.push_back(12);
     ```

  2. The name of the vector comes first.
  3. Then a dot. *Why?*
  4. Then the "function name" `push_back`, with the data value in parentheses.
- ***What is the dot? What does the dot do?***
  1. We cannot answer that until we learn about C++ classes.
  2. Actually the dot notation has been encountered before, when manipulating strings.
  3. However, the dot was (probably) employed without explanation for strings either.
- If `push_back` were a function, the logical syntax should be a function call:

  ```
  push_back(vec_i, 3);
  push_back(vec_i, 7);
  push_back(vec_i, 6);
  push_back(vec_i, 12);
  ```

- ***Hence*** `push_back` ***is not a function in the usual sense.***
- Clearly, to push a `double` onto a vector of type `double`, the syntax is:

  ```
  vector<double> vec_d;
  vec_d.push_back(1.2345);
  vec_d.push_back(6.789);
  // etc
  ```

- For now we shall just have to accept the peculiar syntax of `push_back`.

9 Length of vector: `size()`

- The above program also shows how to query a vector to get its length.
- The value is given by the `size()` function.

```
int sz = vec_i.size();
```

- As with `push_back`, so also `size()` is not a function in the usual sense.
- Once again we write the name of the vector, followed by a dot.
- Then "`size()`" with no arguments in the parentheses.
- Since `size()` returns an integer (technically unsigned, but never mind), the actual data type of the vector does not matter,
- For a vector `vec_d` of elements of type `double`, the syntax is the same:

```
int sz = vec_d.size();
```

- **Note that the length of the vector grows every time we push back an element, and the value returned by `size()` will update accordingly.**
- All of the internal information is automatically kept up to date internally by the vector.

10 Vectors: size and capacity

- There is another parameter associated with the memory allocated for a vector.
- It is called **capacity.**
  1. The **capacity** is the number of elements allocated for the vector.
  2. The **size** is the number of elements actually used.
  3. Therefore `capacity >= size`.
  4. The vector allocates extra space in case we wish to grow the vector (for example, using `push_back`).
- The value of the capacity is given by the `capacity()` function.

```
int cap = vec_i.capacity();
```

- **In general, the capacity is not used much in programming.**
  1. If the size grows too large, the vector allocates extra memory automatically.
  2. Therefore always `capacity >= size`.

11 Vectors: clear

- One of the most important operations is to **clear a vector.**
- This resets the size of the vector to zero.
- The relevant function is called **clear()** and the code is obvious:

```
vector<int> vec_i;
vector<double> vec_d;
// etc

vec_i.clear();
vec_d.clear();
```

- Here is a working C++ program demonstrating the use of `clear()`.

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

int main()
{
  vector<int> vec_i;

  vec_i.push_back(5);
  vec_i.push_back(7);
  vec_i.push_back(6);
  vec_i.push_back(12);

  cout << "size  = " << vec_i.size()  << endl;
  cout << endl;

  vec_i.clear();                              // clear the vector

  // print updated value
  cout << "size  = " << vec_i.size()  << endl;  // size is zero

  return 0;
}
```

12 Vectors: first and last elements: `front()` and `back()`

- The first and last elements of a vector: `front()` and `back()`.
- For whatever reason, C++ provides two functions to return the first and last data values in a vector.
- They have the names `front()` and `back()`.
- Here is a working C++ program demonstrating their use:
- It is essentially the same program as before.

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

int main()
{
  vector<int> vec_i;         // length not specified, default = 0

  vec_i.push_back(5);        // push end of vector, length = 1
  vec_i.push_back(7);        // push end of vector, length = 2
  vec_i.push_back(6);        // push end of vector, length = 3
  vec_i.push_back(12);       // push end of vector, length = 4

  int n = vec_i.size();      // get length of vector
  int f = vec_i.front();     // first element of vector
  int b = vec_i.back();      // last element of vector

  cout << "size  = " << n << endl;
  cout << "front = " << f << endl;
  cout << "back  = " << b << endl;

  return 0;
}
```

13 Vectors: removing elements

- **We remove an element from the end of a vector via `pop_back()`.**
- This decreases the length of the vector by one.
- The "popped" data element is effectively destroyed (goes out of scope).
- Here is a working C++ program demonstrating the use of `pop_back()`.

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

int main()
{
  vector<int> vec_i;
  vec_i.push_back(5);
  vec_i.push_back(7);
  vec_i.push_back(6);
  vec_i.push_back(12);
  cout << "size  = " << vec_i.size()  << endl;
  cout << "front = " << vec_i.front() << endl;
  cout << "back  = " << vec_i.back()  << endl;
  cout << endl;

  vec_i.pop_back();                // *** remove element from vector ***

  // print updated values
  cout << "size  = " << vec_i.size()  << endl;
  cout << "front = " << vec_i.front() << endl;
  cout << "back  = " << vec_i.back()  << endl;
  cout << endl;

  // print values
  for (int i = 0; i < vec_i.size(); ++i) {
    cout << i << "    " <<  vec_i[i] << endl;
  }
  return 0;
}
```

- **Data elements are added and removed from the end of a vector.**
  1. In my experience, adding extra data elements to a vector is common.
  2. Programming with vectors makes heavy use of `push_back()`.
  3. However, removing data elements from a vector is rare.
  4. **In my career, I have very rarely used `pop_back()`.**

14 Vectors: resizing a vector

- Suppose a vector has already been populated to a certain size.
- However, we wish to resize it to a new size.
- This is accomplished via the **resize()** function.
    1. The new size can be *smaller* than the current size of the vector: `resize()` can grow or shrink the length of a vector.
    2. If the new size is smaller, then some data in the vector is lost (goes out of scope).
    3. If the new size is larger, we can supply an initialization value for the new data elements. (If we do not, the default value is zero.)
- Here is a working C++ program demonstrating the use of `resize()`.
- Note that `resize()` is called twice, once to increase the size and latet to reduce it.
- **See next page(s).**

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

int main()
{
  vector<int> vec_i;

  vec_i.push_back(5);
  vec_i.push_back(7);
  vec_i.push_back(6);
  vec_i.push_back(12);

  cout << "size  = " << vec_i.size()  << endl;
  cout << endl;

  for (int i = 0; i < vec_i.size(); ++i) {
    cout << i << "    " <<  vec_i[i] << endl;
  }
  cout << endl;

  vec_i.resize(10, 4);           // *** reinitialize to larger size ***

  // print updated values
  cout << "size  = " << vec_i.size()  << endl;
  cout << endl;

  for (int i = 0; i < vec_i.size(); ++i) {
    cout << i << "    " <<  vec_i[i] << endl;
  }

  vec_i.resize(3, 9);           // *** reinitialize to smaller size ***

  // print updated values
  cout << "size  = " << vec_i.size()  << endl;
  cout << endl;

  for (int i = 0; i < vec_i.size(); ++i) {
    cout << i << "    " <<  vec_i[i] << endl;
  }

  return 0;
}
```

15 Vectors: summary

- For now, we can think of a vector as a generalized array.
- The data type is specified in angle brackets: `vector<int>`, `vector<double>`, etc.
- We can declare a vector as follows:
  (i) without specifying an initial length (default = 0),
  (ii) specifying an initial length,
  (iii) specifying an initial length and an initial value for the elements.
- The syntax to get/set the values of elements is the same as for a C++ array.
- We must populate a vector before we can get/set the element values.
- Attempts to get/set values which are out of bounds results in a run-time error.
- To populate a vector, we add new elements to the end of the vector, via `push_back()`.
- Calling `push_back()` increases the length of the vector by 1.
- We can remove elements from the end of a vector, via `pop_back()`.
- Calling `pop_back()` decreases the length of the vector by 1.
- The length of a vector is obtained by calling `size()`.
- We clear a vector (and reset its size to zero) by calling `clear()`.
- The functions `front()` and `back()` return the values of the first and last elements.
- We can resize the length of a vector at any time during program execution by calling `resize()`.
- If the resized length is shorter, some data in the original vector is lost (goes out of scope).