Contents

Introducing C++ Vectors and Arrays

This is a short introduction to the standard vectors available in C++. Vectors are a powerful yet simple data structure.

Quick Reference

Vectors are good when we have an unknown sequence of similar items to store and we want to access them by their sequence numbers.

Vectors are held in a special library and can be used in a file that has

```
#include <vector>
```

at its beginning.

Declaration	vector<type> v(initial_size);
Accessors	v.empty(), v.size(), v.front(), v.back()
Mutators	v.push_back(T), v.pop_back()
Operators	v[int], v.at(int), v1=v2;, v1==v2

Key Facts

You need to remember the following facts about vectors:

---

1. *A vector is an object that contains a sequence of other objects inside it.*

2. *The objects inside must all take up the same amount of storage.*
3. They are numbered starting with 0.
   If the whole vector is called *v* then the items in it are written *v[0]*, *v[1]*, *v[2]*, ...
   The last item is *v[v.size()-1]* NOT *v[v.size()]*.
4. New items can be "pushed" onto the end of the vector.
5. The last item can be "popped" off of a vector.
6. Vectors can therefore change size.
7. We can find out the current size of a vector: v.size()
8. Vectors can be empty. If so v.empty() is true.
9. If a vector is empty then v[i] and v.pop.... crash.
10. Vectors are empty. by default, when created.
11. Vectors shouls be passed by reference whenever possible.

Details

Suppose that *T* is any type or class - say int, float, double, or the name of a class, then

```
vector<T> v;
```

declares a new and empty **vector** called *v*. Given object *v* declare like the above you can do the following things with it:

test to see if *v* is empty:

```
v.empty()
```

find how many items are in *v*:

```
v.size()
```

push *t* in *T* onto the end of *v*:

```
v.push_back(t)
```

pop the back of *v* off *v*:

```
v.pop_back()
```

Access the *i*'th item (0<=*i*<size()) without checking to see if it exists:

```
v[i]
```

Assign a copy of *v1* to *v*:

```
v = v1
```

Example

Suppose that we want to input an unknown number of numbers and then print them out forwards and then

backwards, using a vector. We will push ints onto the back of a vector called *v*. We will then print each item in v in turn. Finally we will print the vector backwards. You can download the code from [**vectors.cpp**] but here are the highlights. First we must declare the facilities we want to use

```
#include <iostream>
#include <vector>
void print( const vector<int>& ) ;//utility function outputs a vector of ints
void print_backwards( const vector<int> &);
```

Then we describe the main program:

```
int main()
{
        vector<int> v;
        int number;
        cout <<"Input some numbers and then end the input\n";
        while(cin>>number){
                v.push_back(number);
        }//while(more)
        print(v);
        print_backwards(v);
}//main
```

Finally the two procedures that print out the data:

```
void print_backwards( const vector<int> &a)
{
        for(int i=a.size()-1; i>=0; --i)
                cout << a[i] << " ";
        cout << endl;
        cout << "----------------"<<endl;
}//print_backwards
```

and

```
void print( const vector<int>& a)
{
        for(int i=0; i<a.size(); ++i)
                cout << a[i] << " ";
        cout << endl;
        cout << "----------------"<<endl;
}//print
```

Exercise

Down load (with a shift-click!) the above code from [**vectors.cpp**] , test it, and modify it to print out all the even numbered items (0,2,4,6,...) and then all the odd ones(1,3,5,...).

Arrays

The oldest example of a Container in C++ is an array. In C you had arrays and you would write code like this:

```
const int MAX = 10;

float a[MAX];

...

for(int i=0; i<MAX; ++i)

        process(a[i]);

...
```

Arrays are like vectors except that:

---

1. *They have a fixed size specified in the declaration.*
2. *They are faster than vectors.*
3. *You can not add or delete items from an array.*
4. *The syntax is simpler.*
5. *The absolutely no safety belt: if an index has wrong value, you crash.*

Strings

The designers of the C++ library made sure that the <string> library shares a lot of common features with <vector> and other sequential containers. The main differences are (1) <string>s can only hold characters, and (2)<string>s have some useful extra operations available.

Hint

It helps to draw diagrams of vectors and work out by hand, on these diagrams what an algorithm, or your program is doing. Pretending to be a computer can teach you a lot about a complex computation.

Glossary

1. **container**::=A data structure which contains a number of data items that can gain and lose items as the program runs. Examples include vectors, lists, stacks, ...

2. **increment**::**operation**=moving an iterator/pointer/index on to the next iterator/pointer/index value commonly symbolized by ++ in C-like languages.

3. **operation**::=something that can be applied to an object to extract information from it or to change its state.

4. **random_access**::=being able to do things to items in a container in an unpredictable order typically by using an integer as an index or subscript.

5. **vector**::**container**=a vector is a sequential container that is optimized to allow efficient **random_access** of items by using an index. Vectors are are given in a **contiguous** block of storage space. Vectors are useful whenever data must be stored in one order accessed in a different one. A vector is a dynamic array - an array that can grow when needed. [**Vectors**]

Errors

You must always #include <vector> if you use the word vector.

You must be sure that all indices or subscripts stay between 0 and size()-1 inclusive.

Trying to change the size of an array does *not* work.

If a program compiles, loads, crashes, and there is a subscript operator ([...]) then check to see if the subscripted item has been put into the container *before* the subscripted element is accessed. A very common error is to declare a vector with no length and to use [...] as if it created new elements. It doesn't. Use 'push_back(..)' to add new elements to a vector.

If you are unable to show that a subscript is in range then use a condition like the following to guard *thing[i]* from errors:

```
0<= i && i < thing.size()
```

If a program compiles, loads and crashes and it has a **pop_back**, function than make sure that there is an item to be "popped"! In code you can use the empty function to guard statements that contain "pop" from blowing up.

On some older compilers and libraries when you need a <string> as well as <vector> you need to

```
#include <string>
```

before including the list or vector rather in the reverse order! Older string library appear to define some special versions of vector and list operators and the older compilers can not make up its mind which to use.

If the standard <vector> is not found then you are using an older C++ compiler.

You can have vectors of vectors:

```
vector < vector <int> > matrix;
```

but you must put at least one space between the two ">" symbols. The following is *wrong*:

```
vector < vector <int>> matrix;
```

. . . . . (end of section **Errors**) <<Contents | End>>

. . . . . (end of section **Introducing C++ Vectors and Arrays**) <<Contents | End>>

Abreviations

1. **Gnu**::="Gnu's Not Unix", a long running open source project that supplies a very popular C++ compiler.
2. **KDE**::="Kommon Desktop Environment".

3. **TBA**::="To Be Announced", something I should do.
4. **TBD**::="To Be Done", something you have to do.
5. **UML**::="Unified Modeling Language", [**uml.html**] (beginner's introduction to the UML).

End