COMP301
Data Structures & Algorithms [C++]

Dr. N. B. Gyan

Central University, Miotso. Ghana

What's this course about?

KindleDrip — From Your Kindle's Email Address to Using Your Credit Card

Yogev Bar-On 4 days ago · 9 min read

Or the story of how I received an 18K$ bug bounty for a critical Amazon Kindle vulnerability.

kindleDRIP

Source: https://medium.com/

Aggregates

The need to aggregate

- An **aggregate** is simply a collection of objects stored in one unit.
- In C/C++ these take different forms including **array**s, **vector**s and **structure**s.
- The **array** is the basic mechanism for storing a collection of identically-typed objects in C/C++.
- A **vector** is simply a different and more efficient ways of using arrays in C/C++.
- A different type of aggregate type is the **structure**, which stores a collection of objects that need not be of the same type.

The need to aggregate

- As a somewhat abstract example, consider the layout of an apartment building. Each floor might have a one-bedroom unit, a two-bedroom unit, a three-bedroom unit, and a laundry room. Thus each floor is stored as a structure, and the building is an array of floors.

Arrays

Arrays

Without the use of aggregates: `array1.cpp`

Arrays

Discuss:
What do you think are some of the potential challenges with `array1.cpp`?

Note the following about `array1.cpp`

1. Five variables must be declared because the numbers are to be printed in reverse order.
2. All variables are of type int—that is, of the same data type.
3. The way in which these variables are declared indicates that the variables to store these numbers all have the same name–except the last character, which is a number.
4. The data structure that lets you do all of these things in C++ is called an **array**.

Arrays

Illustrating the use of aggregates with arrays:
`array2.cpp`

Arrays

- Note that object in the collection of objects that an array denotes can be accessed by use of the **array indexing operator [ ]**.
- We say that the [ ] operator *indexes* the array, meaning that it specifies which of the objects is to be accessed.

Arrays vs. Vectors

- In C/C++ we can declare and use arrays in two basic ways. The primitive method is to use the built-in array.
- The alternative is to use a `vector`. The syntax for both methods is more or less the same; however, the `vector` is much easier and slightly safer to use than the primitive array and is preferred for most applications.
- The major philosophical difference between the two is that the `vector` behaves as a first-class type (even though it is implemented in a library), whereas the primitive array is a second-class type.

Vectors

Using the vector

- To use the standard vector, your program must include a library header file with

```
#include <vector>
```

- Just as a variable must be declared before it is used in an expression and initialized before its value is used. so must an array.
- A `vector` is declared by giving it a name, in accordance with the usual identifier rules, and by telling the compiler what type the elements are.
- A size can also be provided; if it is not, the size is zero, but `vector` will need to be resized later.

Using the vector

In C++, arrays are always indexed starting at zero. Thus the declaration

```
1 // 3 int objects: a[0],a[1] and a[2]
2 vector<int> a(3);
```

sets aside space to store three integers-namely, a[0], a[1] and a[2].

Using the vector

- The `size` of the vector can always be obtained with the size function.
- For the preceding code fragment example, `a.size()` returns 3.
- Note the syntax: The **dot** operator is used to call the vector's `size` function.

Using the vector

- The size of a vector can always be changed by calling *resize*. Thus an alternative declaration for the vector a could have been

```
1 // 0 int objects
2 vector<int> a;
3 // 3 int objects: a[0], a[1], and a[2]
4 a.resize(3)
```


Using the vector

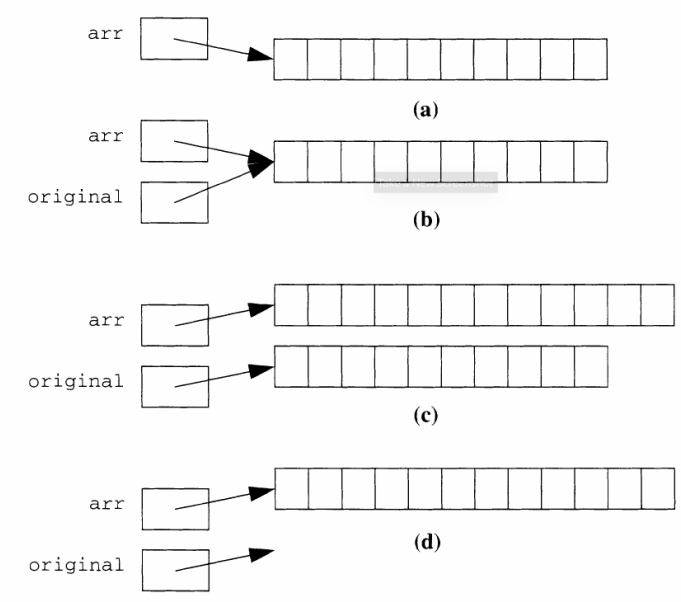
Illustrating the use of the vector:

`vector1.cpp`

Resizing a vector

- One limitation of primitive arrays is that, once they have been declared, their size can never change. Often this is a significant restriction.
- We know, however, that we can use `resize` to change the size of a vector.
- What happens is that pointers are used to give the illusion of an array that can be resized. This detail is hidden inside the implementation of vector.
- The basic idea is shown in the following figure:

Resizing a vector



Resizing a vector

From the previous picture, the following happens

1. Remember where the memory for the 10-element array is (the purpose of `original`). (b)
2. Create a new 12-element array and have `arr` use it. (c)
3. Copy the 10 elements from `original` to `arr`; the two extra elements in the new `arr` have some default value. (d)
4. Inform the system that the 10-element array can be reused as it sees fit. (d)

Resizing a vector

- A moment's thought will convince you that this is an *expensive* operation. Why?
- Because we *copy* all the elements from the originally allocated array to the newly allocated array.
- If, for instance, this array expansion is in response to reading input, expanding every time we read a few elements would be inefficient.
- Thus when array expansion is implemented, we always make it some multiplicative constant times as large.

Resizing a vector

- For instance, we might expand to make it twice as large. In this way, when we expand the array from *N* items to *2N* items, the cost of the *N* copies can be apportioned over the next *N* items that can be inserted into the array without an expansion.
- As a result, this dynamic expansion is only negligibly more expensive than starting with a fixed size, but it is much more flexible.

Resizing a vector

Illustrating the use of the vector:

`vector2.cpp`

See you next week, God willing 🙏🏾