Computer Science

Dr. Y. Liow (February 5, 2022)

Contents

Chapter 105

Containers and arrays

File: chap.tex

File: containers.tex

105.1 Containers

From your programming experience, you see that your programming languages provide some basic types (`int`, `double`, `bool`, etc.) with operations/operators and functions. You then build a software using these types. On the way to building your software system, you will find that you usually have to build new types (such as structures) and functions to be used by your software. Some times, you will find that after some thought, certain functions can be placed inside structures, i.e., you combine data and functions, to get classes.

For instance if you want to develop an RSA crypto system, you will need to perform computations on integers with arbitrary number of digits. Well ... it cannot be arbitrary since you are limited by your computer system resources: you would need to work with integers with hundreds of digits. C/C++ does not allow you to do that. To model integers with arbitrary number of digits, you would probably use heap-based arrays, i.e., you would probably want to use pointers to `int` arrays in the free store. Since you expect to perform the usual arithmetic operations on such integers, you would probably want to develop a class called, say, `LongInt`.

Each `LongInt` object is essentially a container of digits. (Well ... technically speaking it contains a pointer that points to a container – i.e., array – of digits in the heap. But you get the point.)

In your arithmetic operators (addition, subtraction, blah-blah-blah) you see that you frequently need to *travel through* the container, i.e., you will see something like this frequently in your code:

```
for (int i = 0; i < n; i++)
{
    ... do something with p[i] ...
}
```

if `p` is the pointer in your class and `n` is the number of digits `p` points to.

The technical CS term is that you have to *traverse* the container.

Since the heap-based array is linear, i.e., the layout of the data in the container is along a straight line, you see at least two different ways to travel through (traverse) the container. It's either

```
for (int i = 0; i < n; i++) // forward
{
    ... do something with p[i] ...
}
```

or

```
for (int i = n - 1; i >= 0; i--) // reverse
{
    ... do something with p[i] ...
}
```

So in this case, there are two obvious traversals.

You can also fantasy about something like this:

```
for (int i = 0; i < n; i += 2) // forward, even index
{
    ... do something with p[i] ...
}
```

Anyhow, you see that your `LongInt` class will need to have a container and you need to traverse the container.

Other things you want to do with a container is to put things into the container, remove things from the container, search for a value in the container, etc.

If the above idea of a container applies only to your `LongInt` class, then there's no reason to make a big deal out of it and create a whole theory or body of knowledge of such thingies. But it turns out that containers of different shapes (i.e. not necessarily along a straight line) is actually very common in nature (i.e., in computations).

When I say that containers are common in nature I really mean it. Think of your bookshelves. As a whole, it's a container of books. When you need a book, you have a search for it and then remove it from your shelves. Of course you probably want to add books to your shelves too.

In the same way, a database is a container of data. For scientific purposes, it could be a database of genomic data; for business applications, it could be a database of customer, product, and sales transaction data. You also want to add data, remove data, and search for data in databases.

In a computer game, you might have a container of all spaceships attacking you. In a fantasy role playing game, you might have an inventory list (weapons and what-nots) which is a container.

So I'm not lying nor exaggerating when I say that the world is full of containers.

By the way, although beginning algorithms courses focus on the algorithms operating on containers, you should know that the study of algorithms is broader than that.

105.2 Key and satellite data

In the case of an array, such an array of integers representing the digits of a long integer, the index provides a search "key". For instance "the third digit of the long integer represented by `x`" (i.e., the hundreds) means `x[2]`. In this case "third" means "2" which leads us to `x[2]`.

In many applications, the key can be different (i.e., not an index value). For instance suppose I have array of students:

```
#include <iostream>
#include <string>

class Student
{
public:
    Student()
    {}

    Student(const std::string & id,
            const std::string & firstname,
            const std::string & lastname,
    : id_(id),
      firstname_(firstname),
      lastname_(lastname)
    {}

private:
    std::string id_;
    std::string firstname_;
    std::string lastname_;
};

int main()
{
    const int CAPACITY = 1000;
    Student student[CAPACITY];
    int n = 0;

    // Set student[0], student[1], student[2], ...,
    // student[94].
    // Set n to 95
    // Altogether we have 95 students in the system.

    return 0;
}
```

In this case, search for a particular student might depends on supplying the system with a student id. The value of the student id (for each object in the array) is stored in instance variable `id_`.

In this case student id is the key. A field that uniquely identifies a value in a container of values is called **key**. The other data which is not used to uniquely identify a particular entry in the array is called **satellite** data. In the above, the first name and last name for each entry in the array is satellite data of that entry. The value of firstname and lastname does uniquely identify a student the array since it's possible to have two students named John Doe. Right?

key

satellite

105.3 Operations on containers

What are some of the common operations you want to have on a container, such as for instance an array of students where the key is the student's id and the satellite data include the student's first name, last name, etc.?

- You want to add a value (example: a student object) to a container (example: a container of students).
- You want to delete a value from the container.
- You want to search for a student in the container.

In cases where the container has a sense of ordering of values, you might want to do this:

- You want to find the k–th value in the container.

In the case of an array, there *is* a concept of ordering. You have the concept of the "third student", the "105–th student", etc. If you prefer to call students the "zeroth" student, the "first" student, the "second" student, the "third" student, etc then the "third" student in the container is `student[3]`. Of course what's meant by "third" depends on how you order the values. In the case where the order is

```
student[0], student[1], student[2], ..., student[n - 1]
```

the k–th value is `student[k]`. However if I prefer to view the values like this (i.e., in reverse direction):

```
student[n - 1], student[n - 2], student[n - 3], ..., student[0]
```

then the k value means `student[n - k]`.

There are many, many, many other possible operations on the container, depending on the "structure" of the contains.

For instance if the container is a graph, i.e., think of this as a bunch of dots and lines where data is stored at the dots. Then one operation is "find the shortest path from one dot to another."

File: static-unsorted-array.tex

105.4 Static unsorted array

I will say "static array" to mean arrays with memory allocated in the frame, i.e. not in the heap, because the amount of memory used for such things are fixed while your array is in scope.

Here's an example:

```
const int N = 1000;
int x[N];
```

If `T` is a type (`int`, `double`, `bool`, structure, class, ...), you do the same thing:

```
const int N = 1000;
T x[N];
```

The size of the array (see `N`) is usually a constant although some C/C++ compilers allow you to use a variable. But in any case, after the array is requested, you cannot change the size of your `x`:

```
const int N = 1000;
T x[N];
...
gimme_more(x, 1000000); // NOPE!!!
```

This is probably the simplest container. Here are some of the things you can do with your array.

If you want to simulate a container containing different number of items in the container, you have to include a variable that records the number of things already placed in the container:

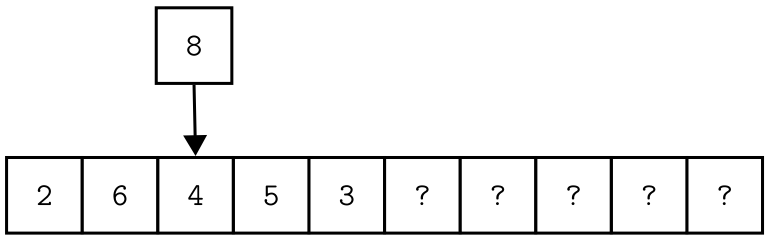
```
const int N = 1000;
T x[N];
int n = 0; // number of things in x, i.e., only x[0],...,
           // x[n-1] are considered values in the
           // container. The values x[n],...,x[N-1]
           // should be considered extra unused space.
```

When you package that into a class, it would look like this:

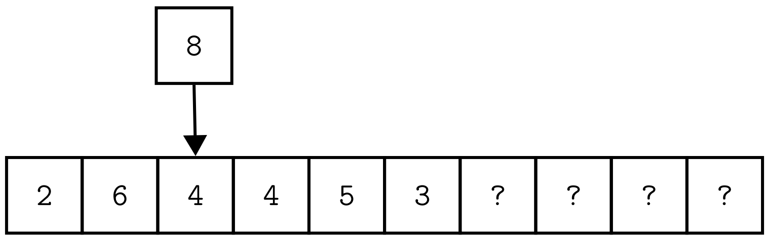
```
template < typename T >
class Array
{
private:
    const int N = 1000;
    T x[N];
    int n;
};
```

105.4.1 Insert

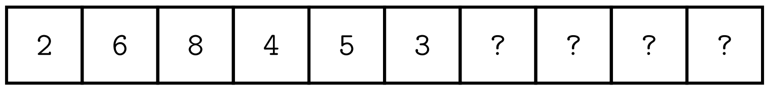
If you need to insert a value at index position `i`, then you have to move the values at positions `i`, ..., `n - 1` to positions `i + 1`, ..., `n` to make an "empty space" for the new value. If `i` is 0, that requires moving lots of values. If `i` is `n - 1`, then only one value has to be moved. If `i` is `n`, the value is to be placed just one step past the last value in the array – you don't have to move anything. Here's the picture of insert at index 2 where `n` is 5 and the capacity (size) if 10:



The values from index values `2` to `4` are moved (i.e., copied) to the right by one step:



and then `8` (the new value) is placed at index `2`:



Of course after the above done, you have to increment `n`, the count of number of things in your array.

```
for (int j = n - 1; j >= i; j--)
{
    x[j + 1] = x[j];
}
x[i] = newvalue;
n++;
```

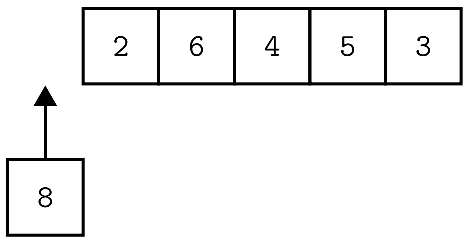
The time taken is

$$A + B(n - i)$$

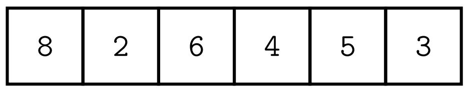
for constants A and B.

Let me call the first value of the array the **head**. If a value is to be inserted at index 0, I will say that I'm "inserting at the head" (to be precise, I'm inserting before the current head – the new value becomes the head of the new array). Pictorially, this is what happens: I have an array of values $2, 6, 4, 5, 3$ and I want to insert an 8 at the head:

head

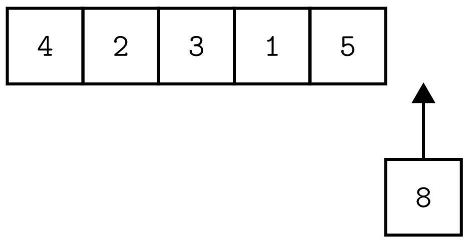


After doing that I get this:

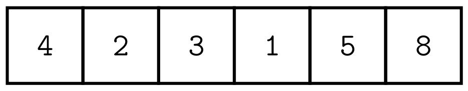


Let me call the last value of the array the **tail**, i.e. at index $n - 1$. If a value is to be inserted at index $n - 1$, I will say that I'm "inserting a tail" (to be precise, I mean inserting after the current tail – it becomes the tail of the new array). Pictorially, this is what happens: I have an array of values $4, 2, 3, 1, 5$ and I want to insert an 8 at the head:

tail



After that I get



Inserting at the head takes the most time. Big-O-wise, insert at the head has a runtime of

$$O(n)$$

Inserting at the tail is the fastest since the runtime is

$$O(1)$$

Averaging over all possible $i = 0, 1, ..., n-1$ and $i = n$ (the case where you're inserting just beyond the array) the average runtime is

$$\begin{aligned}
&\frac{1}{n+1}\left((A + Bn) + (A + B(n-1)) + \cdots + (A + B) + A\right) \\
&= \frac{1}{n+1}\left(A(n+1) + B(n + \cdots + 1)\right) \\
&= \frac{1}{n+1}\left(A(n+1) + B\frac{n(n+1)}{2}\right) \\
&= A + \frac{B}{2}n \\
&= \frac{B}{2}n + A
\end{aligned}$$

I've used the formula:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

OK ... now we know that the average time taken to insert is $O(n)$. [Don't recall the big-O notation and all that algorithmic analysis goodies? Don't panic. Just check the relevant notes/chapters.] Of course I'm assuming in the above that all index values are equally likely.

Here are runtime complexities for insert:

- Insert head: $O(n)$
- Insert tail: $O(1)$
- Average insert: $O(n)$

105.4.2 Delete

If you want to remove a value, say it's at index position `i`, you have to move the value at `i + 1`, ..., `n - 1` to the "left" by one position to index positions `i`, ..., `n - 2`. After that's done, you have to decrement your `n`. The average and worse runtime is again $O(n)$. The best case runtime is $O(1)$.

Here are the runtime complexities for delete:

- Delete head: $O(n)$
- Delete tail: $O(1)$
- Average delete: $O(n)$

105.4.3 Search

What about search? In the worse case, you have to scan the whole array. Which means that the time taken is $O(n)$. The average case is also $O(n)$. The best case runtime is $O(1)$.

Now suppose we sort the values in `x[0]`,..., `x[n - 1]` (say in ascending order). Depending on the sorting algorithm used (see relevant notes) the best time for sorting is $O(n \lg n)$. The benefit of having a sorted array is that search time using binary search takes

$$O(\lg n)$$

(see chapter on algorithmic analysis) which is better than $O(n)$. If you want to ensure that the array is always sorted, when you insert a value in the array, you (of course) don't have to specify where to insert the value but make sure that the value is inserted so that the list remains sorted. How much time do you need? Well ... first you do a binary search. If it's found, you can just insert at that place. You would have a duplicate value - if that's allowed in your list. If it's not found, a -1 is returned. Now you call a version of the binary search that returns the index of where a new value should go instead of -1. (This modified binary search is not difficult to write.) You performed two binary searches. So the time taken is $O(2 \lg n) = O(\lg n)$. Then you perform the insert, taking $O(n)$ time. So all in all, insert also takes $O(n)$ time to insert. Of course instead of performing two binary searches, you can just do a linear search and perform the insert. All in all, you would still get $O(n)$ for runtime.

For delete, say you specify the value to delete and not the index. In this case, you again need to search for the index where the value occurs (taking $O(\lg n)$ time using binary search).

Unsorted array	Best	Average	Worse
Insert	$O(1)$	$O(n)$	$O(n)$
Delete	$O(1)$	$O(n)$	$O(n)$
Search	$O(1)$	$O(n)$	$O(n)$

Note that in many cases the best case runtime is not that useful. The average and worse is more important. When you build a system, you want to focus on the worse case scenario, right? The best case scenario probably does not occur frequently and it would be overly optimistic to focus only on that.

But what if we're in a situation where the best case always occurs? Maybe the program you're trying to build requires only insert and delete at one end of the array. In the case of the (unsorted) array, the best case for insert and delete is pretty fast if you insert and delete at the tail end of the array. Remember that!!! (See a later chapter on stacks, queues, etc. implemented using arrays.)

There's one thing to note about the array: You can very quickly get to the i-th value in the array. The time taken is the same (pretty much) regardless of the value of i. You will see later that there are containers where going to the i–th value of the container is pretty costly. So this is a plus point for arrays.

105.4.4 Capacity issue

The *BAD* thing about the static array (and you absolutely have to remember this) is that the maximum size (or capacity if you like) of the container is fixed and cannot be changed during runtime. If I declare an array with `N` = 1000 values and I keep adding things to this array, ultimately my `n` will reach 1000 and I won't be able to add more stuff into the array. You might think ... "Well ... I'm going to start with 100000 then!" True, true, true. But what if 1000000000 things were intended to be put into the array? Another thing is this: What if there's a scenario where the number of things needed is actually 100? You would be wasting lots of memory. It's just difficult to write flexible software to handle different situations if you're using static arrays. So the static'ness of the memory usage of static arrays is a very serious disadvantage.

Exercise 105.4.1. Write a template `Array` class with the following (public! ... right?) methods:

```
void insert(int index, T key);
void delete(int index);
int  search(T key); // returns index or -1 (if not found)
T    operator[](int index) const;
T &  operator[](int index);
```

Allow duplicate values in an `Array` object. Also, write a template function to print the contents of your object:

```
template < class T >
std::ostream & operator<<(std::ostream & cout,
                          const Array &);
```

If the `Array< int >` object contains 1, 2, 3, then printing it will give you

```
[1, 2, 3]
```

Set the maximum capacity of your `Array` objects to 1000000. Remember my advice in class: Always write a non-template version first!!! Collect timing data for the following scenarios for `Array< int >` objects:

- What is the time taken to insert a value when n = 100000, 200000, 300000, 400000, ..., 1000000?
- What is the time taken to delete the value at a given index position when n = 100000, 200000, 300000, 400000, ..., 1000000?
- What is the time taken to search for a value when n = 100000, 200000,

300000, 400000, ..., 1000000?

- What is the time taken to access a value at an index position when $n = 100000, 200000, 300000, 400000, ..., 1000000$?

Make a plot. For your experiments, make sure your array contains distinct values. [NOTE: See below for timing programs or sections of code.] □

Exercise 105.4.2. Write methods to return the size of the array, boolean functions to tell you is the array is empty or full (i.e., `n` has reached the maximum capacity.) Also, throw some exceptions objects when performing invalid deletes or inserts.

We can abstract the situation and think of an `Array<T>` object as containing `T` values laid out on a straight line. The supporting methods are

```
list.insert(i, v): insert v into list at index position i
list.delete(v)   : remove one value v from list
list.delete(i)   : remove the value at index position i
list.search(v)   : return the index where v occurs (left-
                   to-right) or return -1 if not found
list[i]          : return the value at index i
list[i] = v      : change value at index i to v
list.size()      : return the size of list
list.is_empty()  : return true iff list is empty
list.is_full()   : return true iff list is full
```

Note that when you describe a container and the operations available on the container (or using OO-speak ... what the container can do) without referring to the implementation (or even the programming language used), you're describing an **abstract data type (ADT)**.

abstract data type (ADT)

Technically speaking, the above description uses OO syntax and there are language out there which are non-OO. We would have to be even more free-form to call out description the description of an ADT. However there's no standardization ADT description just like there's no standardization of pseudocode language (!!!)

If I have to describe the above using a non-OO language say a procedural/functional language I would have to write:

```
insert(list, i, v): insert v into list at index position i
delete(list, v)   : remove one value v from list
delete(list, i)   : remove the value at index position i
search(list, v)   : return the index where v occurs (left-
                    to-right) or return -1 if not found
get(list, i)      : return the value at index i
set(list, i, v)   : change value at index i to v
size(list)        : return the size of list
is_empty(list)    : return true iff list is empty
is_full(list)     : return true iff list is full
```

Unfortunately there are others which feel the above is not free-form enough.

Bah!

I'm not going to argue with the language lawyers if one should describe ADT using a pseudo-OO language or pseudo-procedural/functional language. This is how I'm going to describe our unordered list ADT ... and that's that:

```
ADT: Unordered list
list.insert(i, v): insert v into list at index position i
list.delete(v)   : remove one value v from list
list.delete(i)   : remove the value at index i from list
list.search(v)   : return the index where v occurs (left-
                   to-right) or return -1 if not found
list[i]          : return the value at index i
list[i] = v      : change value at index i to v
list.size()      : return the size of list
list.is_empty()  : return true iff list is empty
list.is_full()   : return true iff list is full
```

In general when you have a container where the value is abstractly laid out in a straight line and you have operations to insert, delete, etc on the list, you have the ADT called a **list**. Specifically, a list is either empty or among the values in the list there is one that is the head of the list and there is one that is the tail. (The head can be the tail.) Associated to every value in the list that is not a tail, there is a next value.

list

If the values in the list is not ordered in any way we say that it is an **unordered list**.

unordered list

Our Array<T> is an implementation of the unordered list. The language used is of course C++ and the implementation uses the C/C++ static array.

If the list is ordered, we say that the ADT is an **ordered list**. Here's the ADT. In the next section, I'll talk about the sorted array which is an example of a sorted list.

ordered list

```
ADT: Ordered list
list.insert(v)    : insert v into list
list.delete(v)    : remove one value v from list
list.delete(i)    : remove the value at index i from list
list.search(v)    : return the index where v occurs (left-
                    to-right) or return -1 if not found
list[i]           : return the value at index i
list[i] = v       : change value at index i to v
list.size()       : return the size of list
list.is_empty()   : return true iff list is empty
list.is_full()    : return true iff list is full
```

Note that the insert method for a sorted list does not require an index value since the list will insert v in the right place since the list is ordered.

The SortedArray<T> is an implementation of the order list. The language used is again C++ and the static array is again used.

`File: static-sorted-array.tex`

105.5 Static sorted array

What if your static array is always sorted?

105.5.1 Insert

If you have to insert a value at an index position i, the runtime is still the same as the case of a static unsorted array. (Think about it.)

105.5.2 Delete

If you have to delete a value at an index position i, the runtime is still the same as the case of a static unsorted array. (Right?)

105.5.3 Search

In this case, since the array is sorted, instead of scanning the array left-to-right, you would use binary search. The worse runtime (in fact also the average) is

$$O(\lg n)$$

Of course if the value you're looking for is exactly in the middle of the array (in the sense that the first probe of binary search hits that value), then the runtime is

$$O(1)$$

Sorted array	Best	Average	Worse
Insert	$O(1)$	$O(n)$	$O(n)$
Delete	$O(1)$	$O(n)$	$O(n)$
Search	$O(1)$	$O(\lg n)$	$O(\lg n)$

Exercise 105.5.1. Write a template `SortedArray` class with the following (public!) methods:

```
void insert(T key);
void delete(T key);
int  search(T key); // returns index or -1 (if not found)
T    operator[](int index) const;
T &  operator[](int index);
```

Do not allow duplicate values in the `SortedArray` object. During an `insert`, if the value to be inserted is already into the object, throw a `DuplicateValue` exception object. Set the maximum capacity of your `Array` objects to 1000000. Collect timing data for the following scenarios for `SortedArray< int >` objects:

- What is the time taken to insert a value when n = 100000, 200000, 300000, 400000, ..., 1000000?
- What is the time taken to delete the value at a given index position when n = 100000, 200000, 300000, 400000, ..., 1000000?
- What is the time taken to search for a value when n = 100000, 200000, 300000, 400000, ..., 1000000?
- What is the time taken to access a value at an index position when n = 100000, 200000, 300000, 400000, ..., 1000000?

Make a plot. For your experiments, make sure your array contains distinct values. □

Exercise 105.5.2. For the case of sorted and unsorted *integer* array classes, i.e., not template classes, note that a sorted integer array is an integer array. Try writing `IntArray` class and `SortedIntArray` classes using inheritance. □

File: dynamic-array.tex

105.6 Dynamic array

By a dynamic array, I mean an array in the heap or free store. Recall that a dynamic array can have variable size:

```
T * p;
...
p = new T[size]; // size is a not-necessarily const variable
...
delete [] p;
...
```

All the runtimes are the same except when the array needs to be reallocated. For instance let's think about the insert tail. In the case of static array, if the array is not full, then the runtime is $O(1)$. If it's full, you abort mission – either you thrown an exception right away or you do nothing to the array. What about the dynamic array?

If the dynamic array is not full, then the runtime is again $O(1)$. If it's full, you request for a large array. Then you need to copy your values over to the new array and perform insert tail and (don't forget!) deallocate the memory used by the smaller array. So in this case, the runtime is $O(n)$, not $O(1)$. (We're ignoring the time taken for the memory request.) Of course the memory allocation for the new array can fail – but let's ignore that.

What about delete? If the delete is delete tail? Then the runtime is $O(1)$ just like the case of static array. However if you want to make sure your dynamic array is not wasting too much memory, then you want to use a smaller array when the length of the array is less than 1/3 of the capacity. You allocate an array of size say $2n$, copy the values over to the new array, deallocate the old array. In that case, you again have a runtime of $O(n)$.

File: index-values.tex

105.7 Accessing values in the container: index values

Note that in the case of an array, you have the concept of an index value of a particular value in the array. The index is of course the "position" or placement of that value. If the array is `x`, and `i` contains a valid index value, then `x[i]` arrives at a value in `x`. You can (and *should*) think of an index value as a mechanism for referring to or reaching a value in your array.

In the case of an array,

- You want to add a value to a container *at an index position.* This case potentially overwrite a value. So for instance in the case of an array, values at that index position up to the last index value of the array is moved to the right by one index position.
- You want to delete a value from the container *at an index position.*
- You want to search for a student in the container, *returning the index position if it's found.* If the student is not found, a special index value is returned to indicate failure.

Again, the index represents a position. In the case of the search, if a student is found and the index position is returned, the index value can then be used to locate the student (so as to modify satellite data, print the satellite data, etc.)

You'll see very soon that a more uniform way to access a value in a container is through a pointer. And to make the pointer more flexible, we wrap the pointer in an object. In fact sometimes, depending on what you want to do to the value the pointer is pointing do, this object can contain more than just a single pointer. Such an object (that contains a pointer – and maybe more – to access a value in a container) is called an iterator. But first let's compare the index variable and the pointer ...

File: pointers.tex

105.8 Accessing values in a container: pointers

You usually want to traverse a container, i.e., "travel through" a container. In the case of an array, you can traverse the container by running through the index values of the array. (Like I said in the previous section, think of the index as a mechanism for accessing a value in the array.) For instance:

```
for (int i = 0; i < n; ++i)
{
    std::cout << student[i] << std::endl;
}
```

Note that when you're doing a search, you are in fact doing a traversal. When doing a search in an array, when the target key value is found, an index value is returned (with -1 indicating search failure). Then index value is then used to access the value in the array for some work (such as printing, modification, etc.)

In any case, an index value can be used to access a value in an array.

Note that there's another way to locate a particular student in the `student` array: by using pointers. So instead of

```
...

int find(Student student[], std::string id)
{
     ...
}

int main()
{
    ...
    std::string id;
    std::cin >> id;
    int index = find(student, id);
    std::cout << student[index] << std::endl;
    ...
}
```

we can do this:

```
...

Student * find(Student student[], std::string id)
{
     ...
}

int main()
{
    ...
    std::string id;
    std::cin >> id;
    Student * p = find(student, id);
    std::cout << (*p) << std::endl;
    ...
}
```

Compare the following traversal (using index values):

```
for (int i = 0; i < n; ++i)
{
    std::cout << student[i] << std::endl;
}
```

with this (using pointers):

```
for (Student * p = &student[0]; p != &student[n]; ++p)
{
    std::cout << (*p) << std::endl;
}
```

Note that `&student[n]` is the address *just past* the *last* address occupied by the last student value `&student[n - 1]`. Think about the above very carefully. Drawing a picture of the computer's memory helps.

So what exactly is the difference between the two?!? And which is better? Note that the index version looks like this

```
...
    std::cout << student[i] << std::endl;
...
```

and it uses `student` and `i` whereas the pointer version

```
...
    std::cout << (*p) << std::endl;
...
```

uses `p`. Without going into assembly/machine code, it should already be clear that the index version is slower since it requires memory access to more variables (if nothing else), i.e., two variables `student` and `i` compared against one variable `p` for the pointer version. In fact that *is* the case at the assembly code level: compilers actually translate array traversal by index into array traversal by pointer.

Another reason why the pointer version is better is because, if you think about the meaning of `student[i]`, you see that you start at the memory location of `student[0]` and go to the memory address of the i–th value in order to get to the i–value in `student`. In the case of the array, this can be done rather quickly. You will see later that there are other containers where the computation to get to the i–th value from the first value is very slow. For some containers, it's a lot easier to compute the location of the i–th value from the $(i-1)$–st value. In other words the method of using index values is not appropriate in some cases. I'll give you specific examples later. For now just remember that index values might not make sense for some containers.

Note that the pointer method requires us to know the beginning address of the container and the "end of address" of the container or rather the address that is just outside the address space occupied by the container's values. Nonetheless I'm going to call this "outside end of address" the end address.

```
File:  iterators.tex
```

105.9 Accessing values in a container: iterators

In general, lots of C++ STL containers come with features to access values in the containers using iterators (which are more or less pointers). For instance in the case of `std::vector` class (see CISS245), you can do the following to traverse a vector of integers `v`:

```
for (typename std::vector< int >::iterator p = v.begin();
     p != v.end(); ++p)
{
     ... do something with (*p) ...
}
```

In the above

`std::vector< int >::iterator`

is a class: it's a class inside the `std::vector` class. (Yes you can put a class inside a class.) So you would expect the `std::vector` class to have this shape:

```
template < typename T >
class vector
{
public:
    class iterator
    {
    };
};
```

Objects of this class, `iterator`, are iterators to for instance `std::vector< int >` objects (and remember that iterators are like pointers).

`v.begin()` will "essentially" return a pointer to `v[0]`. OK ... it's not a pointer ... it's an iterator containing a pointer that points to `v[0]`. `v.end()` return a pointer value of `&v[n]` where `n` is `v.size()`, which means that the pointer value is not the address of `v[0]`, `v[1]`, ..., `v[n - 1]`. Again, `v.end()` does not return a pointer value, but rather an iterator that contains a pointer that points to `&v[n]`. This iterator `p` will "point" to a value in `v`. This means that `*p` is a value of `v`.

C++ STL has a container called the `std::list`. If `x` is a list (i.e., STL list) of integers, then you can do this:

```
for (typename std::list< int >::iterator p = x.begin();
     p != x.end(); ++p)
{
     ... do something with (*p) ...
}
```

A list is a kind of container that I will talk about soon: it's a doubly-linked list. There's another C++ STL contains called the `std::set`. If `x` is a set of integers, then you can do this:

```
for (typename std::set< int >::iterator p = x.begin();
     p != x.end(); ++p)
{
     ... do something with (*p) ...
}
```

As you can see, the code to run through the values in the above three containers look almost exactly the same.

Get it?

By the way, as I said above, the code

```
... typename std::vector< int >::iterator p ...
```

tells you that *inside* the class `std::vector< int >`, there's a class called `iterator`. In other words the `std::vector` class looks like this:

```
template < typename T >
class vector
{
public:
    class iterator
    {
    };
private:
};
```

Now you might ask ... what's this `typename` keyword thing for:

```
... typename std::vector< int >::iterator p ...
```

The `typename` tells your compiler that `std::vector< int >::iterator` is a type. Why must you do this? Because remember that from CISS245, when C++ sees `[class]::[something]`, it could mean that `[something]` is a static instance member. The default is static instance member. The `typename` tells your C++ compiler that it's a class, not a static member.

Note that for the case of `std::vector`, you do have `operator[]` so that you can do this:

```
for (size_t i = 0; i < v.size(); ++i)
{
     ... v[i] ...
}
```

i.e., you can run through a vector container using index values instead of iterators.

The problem is that `std::list` does *not* have `operator[]`. So the index method of traversal does *not* apply to STL lists. And why doesn't STL list have `operator[]`? Because the list container has a certain structure that makes this operator, i.e. `operator[]`, very slow if this was ever implemented. In other words, you shouldn't ever want or think about getting the k–th value of a list using `operator[]`. That's why it's not included in the list library.

However arrays can go to the k–th value extremely fast. That's why arrays (and `std::vector` objects) have `operator[]`. (This will be explained in CISS360.)

Wrapping a pointer inside an object (an iterator) is more flexible simply because there are times when the "move pointer to the next object in the container" is very complex and needs extra data. For instance suppose for some bizarre reason you have a vector `v` of 10 values and you want to access the values in this order:

```
v[1], v[2], v[2], v[3], v[3], v[3], v[4], v[4], ...
```

i.e., you want to process `v[i]` `i` times. If you do this frequently, you might want to create an iterator class that does that so that doing `++p` moves the iterator like the above. The iterator object must then remember the index value and the number of times it has processes the value at the index value. Right? Get it?? If you put all that information into the iterator object, it will simplify your (crazy) algorithm.

Exercise 105.9.1. Implement your own `vector` class (see CISS245 assignment) and include an `iterator` class inside you `vector` class and should support dereferencing operator. (See for instance the `IntPointer` class example in the CISS245 notes.) Your `vector` class should includes a `begin()` and an `end()` method. □

File: constant-and-non-constant-iterator.tex

105.10 C++ STL: iterators and constant iterators

Implementing an iterator class is easy: refer to CISS245 if you need help. In particular, look at the notes on the `IntPointer` class. Specifically, your personal vector class should look like this:

```
template < typename T >
class vector
{
public:
    class iterator
    {
    private:
        T * q_; // points to a T value in the array that p_
                // points to
    };
private:
    T * p_; // points to an array in the free store
}
```

In the case of

```
for (typename std::vector< int >::iterator p = v.begin();
     p != v.end(); ++p)
{
     ... do something with (*p) ...
}
```

The iterator `p` can modify the value that it points to:

```
for (std::vector< int >::iterator p = v.begin();
     p != v.end(); ++p)
{
     (*p) = 0;
}
```

In case you do *not* want that to happen, or that you cannot do this (for instance `v` is a constant and therefore you cannot change the values in `v`), you do this:

```
for (typename std::vector< int >::const_iterator p = v.begin();
     p != v.end(); ++p)
{
     ... only read access to (*p), not write access ...
}
```

i.e., `p` is a **constant iterator**.

constant iterator

Exercise 105.10.1. Rewrite the following using iterators:

```
template< typename T >
std::ostream & operator<<(std::ostream & cout,
                          const std::vector< T > & v)
{
    std::string delim = "";
    cout << '{';
    for (size_t i = 0; i < v.size(); ++i)
    {
        cout << delim << v[i];
        delim = ", ";
    }
    cout << '}';
    return cout
}
```

Test it. □

Exercise 105.10.2. Add a `const_iterator` class inside your `vector` class. Note that if `p` is a constant iterator to (say) a C++ STL vector object, then `*p` refers to a value in that vector object. `*p` has read access but not write acceess, i.e.,:

```
int a = *p; // OK
*p = 42;    // BAD!!!
```

This means that in your constant iterator class inside your vector class you have

```
template < typename T >
class vector
{
    class const_iterator
    {
    public:
        T operator*() const
        { ... }
    };
};
```

It's true that the actual value in the vector is not changed since you're returning a copy of that value. But this might be misleading to the person using your constant iterator class (including yourself if you're not careful). It's better to do this:

```
template < typename T >
class vector
{
    class const_iterator
    {
    public:
        const T & operator*() const
        { ... }
    };
};
```

Why? (You ought to know.) □

You *should* write a complete template vector class that has supporting iterators (constant and nonconstant).

File: vector-methods-with-iterators.tex

105.11 C++ STL: `std::vector` operations/methods using iterators

Here are some uses of iterators. Here's the code:

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm> // for std::find and std::sort

template< typename T >
std::ostream & operator<<(std::ostream & cout,
                          const std::vector< T > & x)
{
    std::string delim = "";
    std::cout << '{';
    for (typename std::vector< T >::const_iterator p = x.begin();
         p != x.end(); ++p)
    {
        std::cout << delim << (*p);
        delim = ", ";
    }
    std::cout << '}';
    return std::cout;
}

int main()
{
    std::vector< int > v = {13, 4, 5, 10, 57, 23, 52, 12, 7};
    std::cout << "v: " << v << '\n';

    typename std::vector< int >::iterator p = v.begin();
    std::cout << (*p) << ' ';
    ++p;
    std::cout << (*p) << ' ';
    p++;
    std::cout << (*p) << ' ';
    p += 2;
    std::cout << (*p) << ' ';
    p -= 3;
    std::cout << (*p) << '\n';
    p = v.begin() + 3;
    v.insert(p, -1);                          // insert value
    std::cout << "v: " << v << '\n';

    std::vector< int > u = {-1, -2, -3, -4, -5, -6};
    p = v.begin() + 1;
    v.insert(p, u.begin() + 1, u.end() - 2);  // insert range
    std::cout << "v: " << v << '\n';

    p = v.begin();
    v.erase(p + 5);                           // delete value
    std::cout << "v: " << v << '\n';

    p = v.begin();
    v.erase(p, p + 3);                        // delete range
    std::cout << "v: " << v << '\n';

    std::sort(v.begin(), v.end() - 2);        // sort range
    std::cout << "v: " << v << '\n';

    p = std::find(v.begin(), v.end(), 10);
    if (p != v.end())
    {
        std::cout << "10 found\n";
    }
    std::cout << "v: " << v << '\n';

    std::cout << std::vector< int >{2, 3, 5, 7, 11, 13} << '\n';

    return 0;
}
```

Here's the output:

```
[student@localhost containers] g++ main.cpp; ./a.out
v: {13, 4, 5, 10, 57, 23, 52, 12, 7}
13 4 5 57 4
v: {13, 4, 5, -1, 10, 57, 23, 52, 12, 7}
v: {13, -2, -3, -4, 4, 5, -1, 10, 57, 23, 52, 12, 7}
v: {13, -2, -3, -4, 4, -1, 10, 57, 23, 52, 12, 7}
v: {-4, 4, -1, 10, 57, 23, 52, 12, 7}
v: {-4, -1, 4, 10, 23, 52, 57, 12, 7}
10 found
```

```
v: {-4, -1, 4, 10, 23, 52, 57, 12, 7}
{2, 3, 5, 7, 11, 13}
```

In the above example, note that the initializer list for `std::vector` initialization (i.e., constructor)

```
std::vector< int > u = {-1, -2, -3, -4, -5, -6};
std::cout << std::vector< int >{2, 3, 5, 7, 11, 13} << '\n';
```

is available in g++ (as per the C++11 standard) of our current fedora virtual machine. However it might not be available in your Microsoft Visual Studio yet.

```
File: type-deduction.tex
```

105.12 Automatic type deduction

[NOTE: Cross reference CISS362 notes.]

Later versions of g++ (C++11 and later) support automatic type deductions and range-based for-loops. Here's an easy example on automatic type deduction:

```
#include <iostream>
#include <set>

int main()
{
    auto i = 42;
    auto x = 3.14;
    auto j = i;
    auto & k = i;
    std::cout << i << ' ' << x << ' ' << j << ' ' << k << '\n';
    i = -1;
    std::cout << i << ' ' << x << ' ' << j << ' ' << k << '\n';

    return 0;
}
```

```
[student@localhost containers] g++ main.cpp; ./a.out
42 3.14 42 42
-1 3.14 42 -1
```

In this case g++ figured out that `i` should have type `int` and `x` should be type `double`.

(To understand how type deductions work, see CISS445 where I'll talk about type inferencing for the OCAML language.)

Just because you can use `auto`, it does *not* mean you can forget about what is the actual type you want. Using something blindly is dangerous. Frequently going back to explicit code is helpful. You should learn the concept of C++ type deduction and ranged-based for-loop. You might want to use it to quickly write code for assignments. But once you are done with the assignment, you should replace the type deduction code and range-based for-loops with the explicit version so that you don't forget what is the C++ code generated. Of course you can use it in your personal projects.

File: range-based-for-loops.tex

105.13 Range-based loops

Here's an example of range-based for-loop where `x` is a `std::vector< int >` object:

```
for (int i: x)
{
    std::cout << i << '\n';
}
```

This is frequently used together with automatic type deduction:

```
for (auto i: x)
{
    std::cout << i << '\n';
}
```

Here's an example:

```
#include <iostream>
#include <string>
#include <algorithm>
#include <vector>

template< typename T >
std::ostream & operator<<(std::ostream & cout,
                          const std::vector< T > & x)
{
    std::string delim = "";
    cout << '{';
    for (auto i: x)
    {
        cout << delim << i;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    int w[] = {1,2,3};
    for (int i: w)
    {
        std::cout << i << ' ';
    }
    std::vector< int > v = {2, 3, 5, 7, 11, 13};
    std::cout << v << '\n';

    auto p = v.begin();
    ++p;
    std::cout << (*p) << '\n';
    auto q = v.end();
    --q;
    v.erase(p, q);
    std::cout << v << '\n';

    return 0;
}
```

```
[student@localhost containers] g++ main.cpp; ./a.out
1 2 3 {2, 3, 5, 7, 11, 13}
3
{2, 13}
```

If you want to modify a value in a container, you should use this in the declaration of `v`:

```
for (auto & v: x)
{
    v = 0;
}
```

so that `v` is a reference to a value in `x`. Of course you cannot change the values of `x` if `x` is constant. So in the above I'm assuming `x` is not constant. That's because for

```
for (auto v: x)
{
    v = 0;
}
```

a separate local copy of a value in `x` is created for `v` and changing `v` to `0` will not change the corresponding value in `x`. And of course it's slow especially if the values in container `x` are complex and time consuming to construct. There are some cases where you have to do *this* instead

```
for (auto && v: x)
{
    v = 0;
}
```

if you want to change the values in `x`. So I suggest you use this `&&` method anyway if you want to change the values in `x`.

Usually you should make `v` a reference to a value in `x` and not make copies all the time. If x is constant and your loop does not change the values in x, you can do this when the values in x are complex (example: objects):

```
for (const auto & v: x)
{
    ...
}
```

Note that if your x is constant and you do

```
for (auto & v: x)
{
    ...
}
```

the type deduction will be smart enough to make the v a constant reference. There are many other variations.

In summary if you do not want to change the values in x and the values are simple and you don't mind copying the values of x to v, do this:

```
for (auto v: x)
{
    ...
}
```

If you do not want to change the values in x and you want v to reference values in x, you should do this:

```
for (const auto & v: x)
{
    ...
}
```

(the `const` is redundant if x is constant). If x is not constant and you want to change the values in x, you can do

```
for (auto & v: x)
{
    ...
}
```

or

```
for (auto && v: x)
{
    ...
}
```

where the second version is probably better.

The following is an example involving `std::vector`:

```
#include <iostream>
```

```
#include <string>
#include <vector>

template< typename T >
std::ostream & operator<<(std::ostream & cout,
                          const std::vector< T > & x)
{
    std::string delim = "";
    cout << '{';
    for (auto & v: x)
    {
        cout << delim << v;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    std::vector< int > v = {2, 3, 5, 7, 11, 13};
    std::cout << v << '\n';

    for (auto x: v)
    {
        x = 0;
    }
    std::cout << v << '\n';

    for (auto & x: v)
    {
        x = 1;
    }
    std::cout << v << '\n';

    for (auto && x: v)
    {
        x = 2;
    }
    std::cout << v << '\n';

    return 0;
}
```

```
[student@localhost containers] g++ main.cpp; ./a.out
{2, 3, 5, 7, 11, 13}
{2, 3, 5, 7, 11, 13}
{1, 1, 1, 1, 1, 1}
{2, 2, 2, 2, 2, 2}
```

And again these are not the only possibilities. And if the compiler does not deduce the right type or your code does not allow a suitable type deduction, I suggest you go back to basics and type in the actual type you really want.

In some STL containers, the values cannot be changed. For instance the values in a `std::set` object cannot be changed – they are immutable. Of course you know that values in a `std::vector` can be changed if it's not constant.

Again be aware of the above syntax. Try out the above examples. But avoid using it in assignments until you really know your iterators without using shortcuts like `auto` and range-based for-loops.

File: products-and-tuples.tex

105.14 C++ STL: `std::pair` and `std::tuple`

C++ provides two STL classes for tuples: `std::pair` for 2-tuples and `std::tuple` for general k–tuples.

Each `std::pair` object is made up of two values which can be of different types. If `x` is a `std::pair` object, then the two values are `x.first` and `x.second`.

Each `std::tuple` object is made up of two or more values which can be of different types. The number of value is fixed. If `x` is a `std::tuple` object, then the values are `std::get<0>(x)`, `std::get<1>(x)`, `std::get<2>(x)`, etc.

Run the following example code:

```
#include <iostream>
#include <set>
#include <utility> // for std::pair
#include <tuple>

template< typename S, typename T >
std::ostream & operator<<(std::ostream & cout,
                          const std::pair< S, T > & x)
{
    cout << '(' << x.first << ", " << x.second << ')';
    return cout;
}

template< typename S, typename T, typename U >
std::ostream & operator<<(std::ostream & cout,
                          const std::tuple< S, T, U > & x)
{
    cout << '('
         << std::get<0>(x) << ", "
         << std::get<1>(x) << ", "
         << std::get<2>(x) << ')';
    return cout;
}

int main()
{
    std::pair< int, double > u = {2, 3.14159};
    std::cout << u << '\n';
    u.first = 1;
    u.second = 2.718281;
    std::cout << u << '\n';

    std::tuple< int, double, char > v = {-2, 0.01, 'A'};
    std::cout << v << '\n';
    std::get<0>(v) = -1;
    std::get<1>(v) = -4.2;
    std::get<2>(v) = 'B';
    std::cout << v << '\n';

    return 0;
}
```

```
[student@localhost containers] g++ main.cpp; ./a.out
(2, 3.14159)
(1, 2.71828)
(-2, 0.01, A)
(-1, -4.2, B)
```

Of course a `std::pair` is just a special case of `std::tuple`.

It's obvious that the length of the tuple in the above example is fixed at 3. If you want to have a tuple of arbitrary length you should not use `std::tuple`. If the values in your tuple have the same type but you want the length to be arbitrary, then you should use a `std::vector`. Of course a `std::vector` is homogeneous: the type of the values must be the same.

File: typedefs.tex

105.15 Initializer lists and typedefs

Besides the fact that automatic type deductions and range-based for-loops makes container code easier to write, the initializer lists and typedefs help too. You have already seen typedefs from CISS240, CISS245. The initialize list refers to the array initializer from CISS240:

```
int x[] = {2, 3, 5};
```

Note that the original notation for (array) initializer list can only be used during declaration.

In the newer C++, the initializer list notation can be used when you work with C++ STL containers.

Run and study the following:

```
#include <iostream>
#include <vector>

typedef std::pair< double, double > twodouble;
typedef std::vector< twodouble > twodoubles;

std::ostream & operator<<(std::ostream & cout, const twodouble & x)
{
    cout << '(' << x.first << ", " << x.second << ')';
    return cout;
}

std::ostream & operator<<(std::ostream & cout, const twodoubles & v)
{
    cout << '{';
    std::string delim = "";
    for (auto & x: v)
    {
        cout << delim << x;
        delim = ", ";
    }
    cout << '}';
    return cout;
}

int main()
{
    twodoubles v = {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}};
    std::cout << v << '\n';

    v = {{1.1, 2.2}};
    std::cout << v << '\n';

    std::cout << (twodoubles {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}}) << '\n';

    return 0;
}
```

has output

```
[student@localhost containers] g++ main.cpp; ./a.out
{(0, 0.1), (0.2, 0.3), (0.4, 0.5)}
{(1.1, 2.2)}
{(0, 0.1), (0.2, 0.3), (0.4, 0.5)}
```

Clearly the typedefs help. But note also the benefit of using initializer lists. For

```
twodoubles v = {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}};
```

without using typedefs, it would become

```
std::vector< std::pair< double, double > > v
    = {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}};
```

Without the inner initializer list, it becomes

```
std::vector< std::pair< double, double > > v
    = {std::pair< double, double >(0.0, 0.1),
       std::pair< double, double >(0.2, 0.3),
       std::pair< double, double >(0.4, 0.5)};
```

i.e., initializer lists have to be replaced by explicit constructor calls. And without the outer initializer list it becomes

```
std::vector< std::pair< double, double > > v;
v.push_back(std::pair< double, double >(0.0, 0.1));
v.push_back(std::pair< double, double >(0.2, 0.3)),
v.push_back(std::pair< double, double >(0.4, 0.5));
```

Before the introduction of initializer lists for STL template classes (in C++11), it was very tedious to initialize an STL container. You can first initialize it to an empty container and then you put values into the container one value at a time (like the above). You can also called the constructor with the beginning pointer and ending pointer to an array of values, for instance:

```
int x[] = {2, 3, 5, 7};
std::vector< int > v(x, x + 4);
```

But you do have to create an array of the values and the unnecessary name of `x`. Both cases are tedious.

With the intiializer lists for STL template classes, you can do

```
std::vector< int > v = {2, 3, 5, 7};
```

or

```
std::vector< int > v {2, 3, 5, 7};
```

Note that the initializer notation can be used with the assignment operator

```
std::vector< int > v;
v = {2, 3, 5, 7};
```

By the way in the above, just like the case for `std::vector`

```
twodoubles v = {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}};
```

can also be written this way:

```
twodoubles v {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}};
```

Finally the initializer notation can be used in a constructor call to create an object without declaring a name:

```
std::cout << (twodoubles {{0.0, 0.1}, {0.2, 0.3}, {0.4, 0.5}}) << '\n';
```

In the same way you can do

```
std::cout << (std::vector< int > {2, 3, 5}) << '\n';
```

Index