COMP2113 Programming Technologies / ENGG1340 Computer Programming II

Module 9: STL - Containers

Objectives:
- Understand Standard Template Library (STL)
- Understand three basic STLs (vector, list, and map)

1 A phone book example

Assume that we want a phone book application with the following two functions.
- `Add name, number` : Add a record to the phone book
- `Search name` : Return the phone number of a person

One quick idea is to define a `Record` class.

```
class Record {
public:
        string name;
        int number;
};
int main() {
        Record phone_book[ ? ];
}
```

```
// Treat the class as struct as follow
struct Record {
        string name;
        int number;
};
```

***Note**: "class" is a user-defined datatype like "struct." The difference between a class and a struct in C++ is that struct has members public by default and class has members private by default. You can see we used the keyword `public` to define public members so the members could be accessed outside the class. Besides, a "class" can have its own functions. You will learn more about `class` in the "COMP2396 Object-Oriented Programming and Java" course in the future. For now, you can just consider "class" as an in-place substitution of "struct" with the use of the keyword "public."*

However, what should be the size of `phone_book`? Can I implement a **dynamic array** which grows depending on the amount of content?

In this course, we have learned the linked-list data structure that can serve this purpose. We can hence implement a linked list for our phone book application. However, other people also like using dynamic arrays. Can I reuse their existing code? Notice that reusing their code does not mean copy and paste. The better way to reuse existing code is by using the corresponding libraries.

Many libraries are provided by C++, e.g., `<iostream>`, `<string>` and `<fstream>`, etc. An important library with the linked-list data structure is the **Standard Template Library (STL)**. This library also contains other functionalities like sorting and searching. This chapter gives a brief discussion on using this library.

STL contains efficient implementations of many useful classes and functions. Those classes and functions are grouped into three categories:
- **Containers** : Classes for storing data, e.g., linked lists and sets
- **Iterators** : Classes for accessing items in containers
- **Algorithms** : Common algorithms, e.g., sorting, searching

2 STL containers

We start with STL containers. Roughly speaking, an STL container is a class that allows you to insert items, retrieve items and remove items. Containers are very general that they can be used for items of different types. Different containers support different functionalities and have different performance guarantees.

2.1 Vectors

The simplest container is "vector". A vector is a linear array of items. It supports four main functions.

- `void push_back( item )`: Add an item to the end of the vector
- `int size()` : Return the number of items in the vector
- `item operator[int index]`: Access the index-th item in the vector
- `void pop_back()` : Remove the last item in the vector

See below for an example of using a vector to implement the phone book application.

```
#include <iostream>
#include <string>
#include <vector>   //header file for vector
using namespace std;
int main() {
	vector<string> names;  //create an empty vector of string
	vector<int> numbers;

	string command, name; int number;
	while (cin >> command >> name) {
		if (command == "Add") {
			cin >> number;
			names.push_back(name);  //add a string to the end
			numbers.push_back(number);
		}
		else {
			//command == "Search"
			for (int i = 0; i < names.size(); i++)
				if (names[i] == name)    //access the i-th string
					cout << numbers[i];
		}
	}
}
```

Notice that to use the vector class, we need to include the header file `<vector>`. Then, to create an empty vector that can store strings, we use the following interesting syntax.

```
vector<string> names;
```

Vector is a **template class**. The word "template" means that this class can store item of different types. The word `<string>` tells the compiler that we now want a vector that stores `strings`. It is called a **template parameter** of the template class. Similarly, the line `vector<int> numbers` creates a vector that can store integers.

```
vector<int> numbers;
```

A newly created vector has size 0, because it contains nothing initially. We can insert more strings to the vector by the line

```
names.push_back(name);
```

The string name is added to the end of the vector. Hence, if there are 3 strings in the vector already, the new string will be inserted as the 4th string.

To access the i[th] string in a vector, we simply use [ ]. Hence, `names[i]` is the i[th] string in the vector names. The for-loop below simply checks each string in the vector and see if one of the entries in the vector matches the input name.

```
for (int i = 0; i < names.size(); i++)
	if (names[i] == name)	//access the i-th string
		cout << numbers[i];
```

Notice that `numbers` is a vector of `int`. The syntax for creating and using a vector of `int` is identical to that for a vector of `string`. This makes using vector very convenient.

Vectors of user-defined classes

Our previous program uses a vector of `string` to store the names and a vector of `int` to store the phone numbers. It may be more intuitive to define a `Record` class and a phone book is simply a vector of `Records`. It can be done easily as follows.

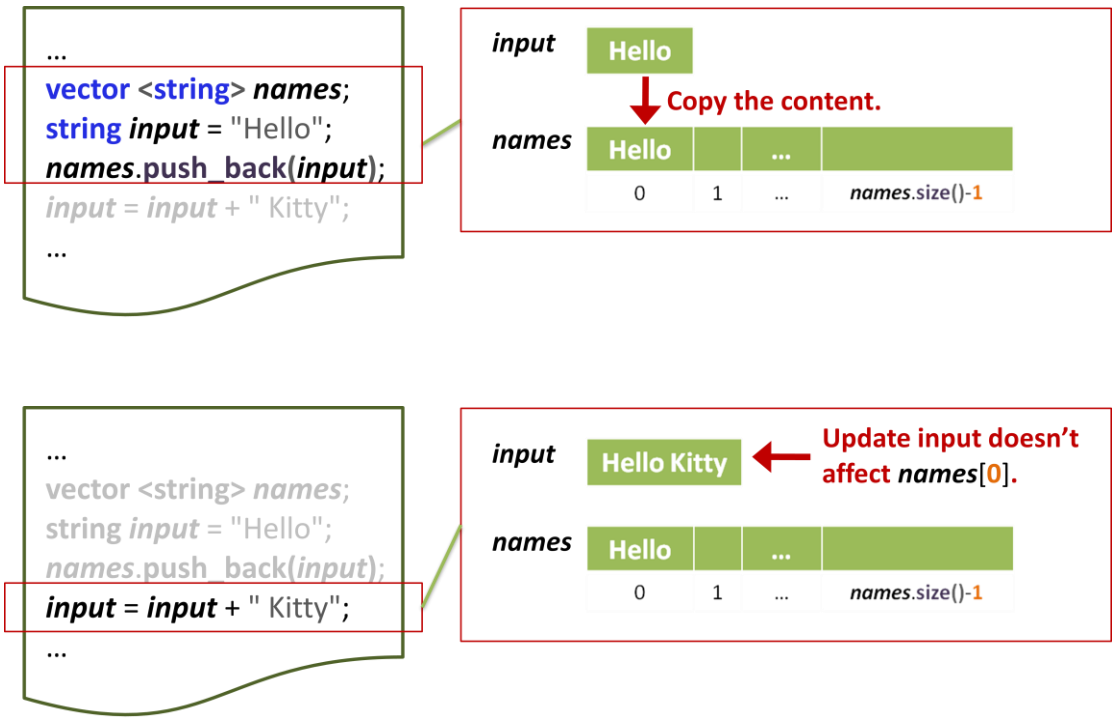
```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Record {
public:
	string name;
	int number;
};

int main() {
	vector<Record> book;
	string command; Record r;
	while (cin >> command >> r.name) {
		if (command == "Add") {
			cin >> r.number;
			book.push_back(r);	// insert records to the end
		}
		else {
			for (int i = 0; i < book.size(); i++) {
				if (book[i].name == r.name) {	//access the records
					cout << book[i].number;
				}
			}
		}
	}
}
```

Note that we create a vector of `Record` by the following statement.

```
vector<Record> book;
```

Then we can insert `Records` to the end by `push_back()` and access Records by [ ] just like what we did for vectors of `string` and `int`.

Notice that when we `push_back( )` a `Record r` to the vector, a copy of `r` is created inside the `vector`. Hence, any change in `r` afterwards does not affect the `Record` inside the `vector`.



Performance guarantees of `vector`

The table below gives a summary on the functionalities of vectors. Notice that `T` is a template parameter, which can be of any type like `string`, `int` or `Record`.

`vector<T> v;`	Parameter	Effect
`v.push_back(T t)`	An item t with type T	insert t to end, increase size by 1
`v[i]`	integer i	access the object at position i
`v.size()`	none	return number of items in v
`v.pop_back()`	none	delete item at end, decrease size by 1
=	v1 = v2	set v1.size() = v2.size(); perform v1[i] = v2[i] for each i = 0, 1, ..., v2.size() – 1

STL guarantees that `push_back()` takes O(1) time on average. The operator [ ], `size()` and `pop_back()` all take O(1) time (in the worst case). Hence, `vector` is very efficient for applications that need dynamic arrays.

Vector also supports other operations like inserting to or deleting from a certain position. But they take O(n) time on average.

***Note**: The Big-O notation, i.e. O(), denotes the computational complexity of an algorithm or the space complexity of a data structure. The knowledge of Big-O notation won't be covered by COMP2113 / ENGG1340 and was not covered by any of its prerequisites, so it won't be tested at all. But for the sake of completeness of our materials, we decided to also mention the Big-O of different STL data structures and their corresponding operations.*

If you are interested in knowing more about the Big-O notation, you might want to go through this short tutorial: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

2.2 Lists

The second STL container we will learn is `list`, which stores items in the well-known doubly linked-list format. It supports the following functions.

- `void push_back( item ):` Insert an item to the back
- `void pop_back():` Remove the item at the back
- `void push_front( item ):` Insert an item to the front
- `void pop_front():` Remove the item at the front
- `front():` Access the first item
- `back():` Access the last item
- `int size():` Return the number of items in the list

Note that `list` provides functions to insert, access and remove the first and last items. However, it does not support efficient access to the i^{th} items. Below is the same phone book application implemented using the STL `list`.

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
int main() {
	list<string> names;	//create a list of strings
	list<int> numbers;
	string command, name; int number;
	while (cin >> command >> name) {
		if (command == "Add") {
			cin >> number;
			names.push_back(name);  //insert a string to the end
			numbers.push_back(number);
		}
		else {
			for (int i = 0; i < names.size(); i++) {
				if (names.front() == name)
					cout << numbers.front();  //access first string
				else {
					names.push_back(names.front());
					names.pop_front();
					numbers.push_back(numbers.front());
					numbers.pop_front();
				}
			}
		}
	}
}
```

Similar to vector, we create a list of strings by using the template class name list with the template parameter `<string>`, as follows.

```
list<string> names;
```

Then we can insert strings to the list by the `push_back` function.

To search for a name, notice that we always compare the query with the first name in the list. If the name does not match, we move the name and the phone number to the back. The same operation is performed `size()` times. If no match is found after `size()` times, we have checked the whole linked-list and we know that name does not exist inside the phone book.

List of user-defined classes

Similar to vectors, we can have a list of user-defined classes. In the example below, we create a list of `Record` as the phone book.

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
class Record {
public:
	string name;
	int number;
};
int main() {
	list<Record> book;   //create a list of Records
	string command; Record r;
	while (cin >> command >> r.name) {
		if (command == "Add") {
			cin >> r.number;
			book.push_back(r);
		}
		else {
			for (int i = 0; i < book.size(); i++) {
				if (book.front().name == r.name) {
					cout << book.front().number;
					break;
				}
				else {
					book.push_back(book.front());
					book.pop_front();
				}
			}
		}
	}
}
```

Similar to vectors, when we `push_back()` a `Record r` into a list, a copy of `r` is created and changes to `r` will not affect the entry inside the list.

Performance guarantees of `list`

The table below gives a summary on the functionalities of lists. Notice that `T` is a template parameter, which can be any type like `string`, `int` or `Record`.

`list<T> l;`	Effect
`l.push_back( T t)`	Insert an item to the back
`l.pop_back()`	Remove the item at the back
`l.front()`	Access the first item
`l.back()`	Access the last item
`l.push_front(T t)`	Insert an item to the front
`l.pop_front()`	Remove the item at the front
`l.size()`	Get the number of items in the list
=	Assign one list to another. Same as vector

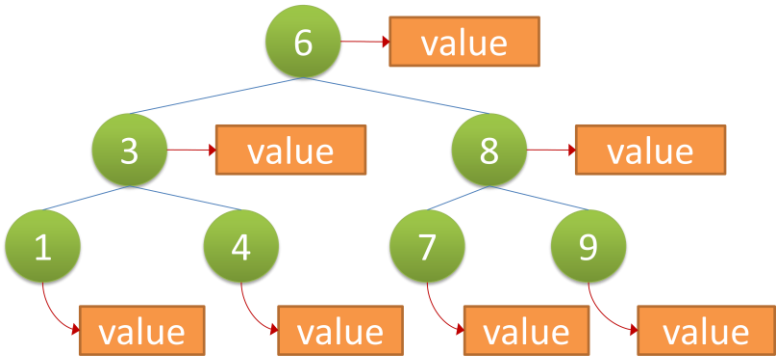
STL guarantees that the first 7 operations above takes O(1) time. Hence, it is very efficient to access the first and the last item. However, the operator [ ] is missing. Hence, accessing the i^{th} item is quite inefficient. It takes O(i) time by traversing from the first item until the i^{th} item.

2.3 Maps

When we use vectors or lists to implement the phone book application, adding a record take O(1) time and searching for a name takes O(n) time, where n is the size of the phone book. If we need to perform many search operations, we would like a container that supports more efficient searching. The STL `map` is a container that supports more efficient searching.

Yes! The STL map is a **container** that organizes the items in a **balanced binary search tree**, which supports more efficient searching **O(log *n*)**.

What is a balanced binary search tree?



- Note that we are not going to teach balanced binary search tree in this course, we are just going to use it.
- Nevertheless, let's have a brief review on the characteristics of balance binary search tree.
  - Binary tree – Each node in the tree has at most two child nodes (can be one child node or two child nodes).
  - Search tree – Keys on the left sub-tree must be smaller than the keys on the right sub-tree.
  - Balanced tree – The tree is kept balance to avoid a tall tree with nodes skewed to one side. A skewed tree will affect the searching performance.
- Searching is pretty fast when compared with **array** and **linked-list**.
  When searching for a particular key, we do not need to scan through all the items.
  For example, searching for key 7 we need to do 2 comparisons only.
  - Compare with the root, since 7 is larger than 6, we go to the right path.
  - Compare with node 8, since 7 is smaller than 8, we go to the left path.
  - Found key 7.

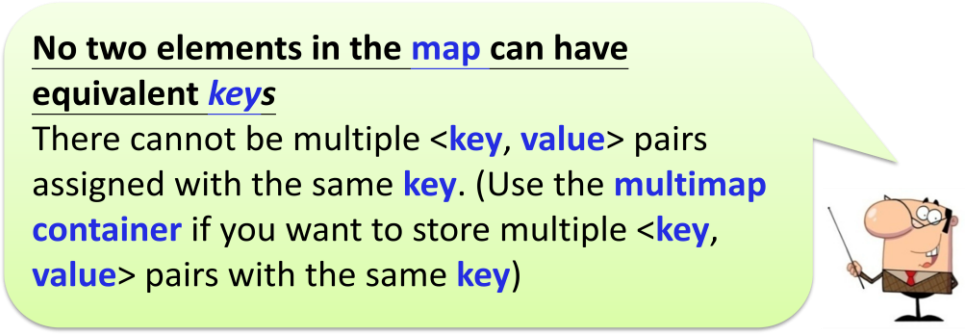
A `map` stores a set of <key,value> pairs. It supports the following functions.

- `map<K, V> m`: This creates a map `m` such that each pair in `m` has key of type `K` and value of type `V`. For example, `map<string,int> m` create a map `m` where each pair in `m` has a `string` as the key and an `int` as the value.

- `map[key]`: Given any key, the [ ] operator access the corresponding value. If no pair with such key exists in the map, `map[key]` will create a pair with this key and set the default value for the pair. For example, assume there is no pair in `m` with key `"Alan"`, the statement below is still valid.

```
cout << m["Alan"] << endl;
```

  In particular, a default value of `int` will be created for `map["Alan"]`, which is an arbitrary integer.
- `int count(key)`: Given any key, this function returns the number of pairs in the map with this key. Hence, it returns 1 if such key exists; it returns 0 otherwise. It is usually used to check if a key exists before accessing the value through `m[key]`.
- `int size()`: Return the number of pairs in the map.



The program below shows a very simple implementation of the phone book application. Notice that we check whether a name exists before printing out the corresponding phone number.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main() {
	map<string, int> book;
	string command, name; int number;
	while (cin >> command >> name) {
		if (command == "Add") {
			cin >> number;
			book[name] = number;
		}
		else {
			if (book.count(name) > 0) {
				cout << book[name];
			}
			else { cout << "Name not found" << endl; }
		}
	}
}
```

Note that we use `book[name]` both for inserting a <key, value> pair as well as accessing the value of the pair. The function `book[name]` takes O(log n) time, where n is the size of the map. Hence, inserting a <key, value> pair is slightly slower than that of `vector` or `list`, but searching for a value is significantly faster.

Map of user-defined classes

We have seen that vector and list can store objects of user-defined classes without any modification. However, the following program tries to use objects of `Record` as keys and a compilation error is reported.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
class Record {
public:
	string name;
	int number;
};
int main() {
	map<Record, int> book;
	string command; Record r;
	while (cin >> command >> r.name) {
		if (command == "Add") {
			cin >> r.number;
			book[r] = r.number;
		}
		else {
			if (book.count[r] > 0) {
				cout << book[r];
			}
			else { cout << "Name not found" << endl; }
		}
	}
}
```

The reason is as follows. To support efficient searching using the keys, `map` stores the <key, value> pairs in ascending order of the keys. Hence, it requires the keys to have a total order (i.e. a totally ordered set or a linearly ordered set). In particular, it requires that the less-than operator < is defined for the keys.

For user-defined classes, the operator < is not defined. Hence a compilation error occurs. To solve the problem, add the following operator overloading before the `main()` function.

```
bool operator<(const Record& a, const Record& b) {
	return a.name < b.name;
}
```

- **Two input parameters**
  - Reminded that the operator<() function for the `Record` objects has two input parameters as "<" is a binary operator. i.e., the two input parameters are the left- and right-hand sides of the "<" operator, respectively.
- **The type of input parameters**
  - We are adding the logic of comparing two `Record` objects (i.e., when a<b is used for `Record` objects a and b), therefore the first and second input parameters are of `Record` type.
- **The "&" sign**
  - The input parameters are pass by reference instead of pass by copy because we want the function to be more efficient.

- **Keyword const**
  - The input parameters are pass by constant reference because we will not alter the content of a and b in the function.
- **Return type - bool**
  - The relational operator returns either true or false, so the return type is of type bool.
- **Why "return a.name<b.name"?**
  - We can compare two string variables by using the "<" operator (i.e., the relational operator function operator<() is defined for string variables by default).
  - It simply gives an order for the string values, e.g., "abc" is ordered ahead of "abd", so "abc" < "abd" returns true.

Performance guarantees of `map`

The table below gives a summary on the functionalities of `maps`. Notice that `T1` and `T2` are template parameters, which can be any type like `string`, `int` or `Record`.

`map<T1, T2> m;`	Parameter	Effect
`m[T1 t]`	A key t of type `T1`	Access the item with key `t`; create if not exist
`m.count(T1 t)`	A key t of type `T1`	Return 0 if no item has key `t`; 1 otherwise
`m.size()`		same as vector and list
`m1 = m2`		For each `key` in `m2`, performs `m1[key] = m2[key]`

STL guarantees that the first two operations take O(log n) time, where n is the size of the map. The third operation takes O(1) time. The last operation equals the total creation time of the keys and values.

2.4 Comparison of vector, list, map

The table below summarizes the performance guarantees of vector, list and map. Each container has its strength and weakness depending on the application requirement. E.g., vector is more efficient when we need random access of items, while map is more efficient in searching. A symbol / means it is not directly supported.

Operation	`vector`	list	map
Insert at back	O(1)	O(1)	/
Remove at back	O(1)	O(1)	/
Insert at front	/	O(1)	/
Remove at front	/	O(1)	/
Insert at sorted order	/	/	O(log n)
Search	O(n)	O(n)	O(log n)
Find size	O(1)	O(1)	O(1)

***Note**: The Big-O notation, i.e. O(), denotes the computational complexity of an algorithm or the space complexity of a data structure. The knowledge of Big-O notation won't be covered by COMP2113 / ENGG1340 and was not covered by any of its prerequisites, so it won't be tested at all. But for the sake of completeness of our materials, we decided to also mention the Big-O of different STL data structures and their corresponding operations.*

If you are interested in knowing more about the Big-O notation, you might want to go through this short tutorial: https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

3 Further readings

We will cover STL iterators and algorithms in the next part of this Module.

For more information about STL containers, see Chapter 15 of "C++ How to Program, Ninth Edition Problem".

A reference of STL containers is available at
http://www.cplusplus.com/reference/stl/