CS213/293 Data Structure and Algorithms 2023

Lecture 2: Containers in C++

Instructor: Ashutosh Gupta

IITB India

Compile date: 2023-08-06

What are containers?

A collection of C++ objects

- `int a[10]; //Array`
- `vector<int> b;`

Exercise 2.1

Why the use of the word 'containers'?

More container examples

- `array`
- `vector<T>`
- `set<T>`
- `map<T,T>`
- `unordered_set<T>`
- `unordered_map<T,T>`

In math, sets are unordered?

Set in C++ $\neq$ Mathematical set

Why do we need containers?

Collections are everywhere

- CPUs in a machine
- Incoming service requests
- Food items on a menu
- Shopping cart on a shopping website

Not all collections are the same

Example: using a container

Source: http://www.cplusplus.com/reference/set

```
#include <iostream>
#include <set>
int main () {
  std::set<int> s;
  for(int i=5; i>=1; i--)   // s: {50,40,30,20,10}
    s.insert(i*10);
  s.insert(20);    // no new element inserted
  s.erase(20);     // s: {50,40,30,10}

  if( s.contains(40) )
    std::cout << "s has 40!\n";

  for( int i : s )  // printing elements of a container
    std::cout << i << ’\n’;
  return 0;
}
```

Why do we need many kinds of containers?

- Expected properties and usage patterns define the container

  For example,

  - Unique elements in the collection
  - Arrival/pre-defined order among elements
  - Random access vs. sequential access
  - Only few additions(small collection) and many membership checks
  - Many additions (large collection) and a few sporadic deletes

Different containers are

efficient to use/run

in varied usage patterns

Choose a container

Exercise 2.2

Which container should we use for the following collections?

- *CPUs in a machine*
- *Incoming service requests*
- *Food items on a menu*
- *Shopping cart on a shopping website*

Some examples of containers

`set<T>`

- Unique element
- insert/erase/contains interface
- collection has implicit ordering among elements

`map<T,T>`

- Unique key-value pairs
- insert/erase interface
- collection has implicit ordering among keys
- Finding a key-value pair is not the same as accessing it
- Throws an exception if accessed using a non-existent key

Containers are abstract data types

The containers do not tell us the implementation details. They provide an interface with guarantees.

In computer science, we call the libraries abstract data types. The guarantees are called axioms of abstract data type.

Example 2.1

Axioms of abstract data type set.

- `std::set<int> s; s.contains(v) == false`
- `s.insert(v); s.contains(v) == true`
- `x = s.contains(u); s.insert(v); s.contains(u) == x`, where $u! = v$.
- `s.erase(v); s.contains(v) == false`
- `x = s.contains(u); s.erase(v); s.contains(u) == x`, where $u! = v$.

Example: map<T,T>

Source: http://www.cplusplus.com/reference/map

```
#include <iostream>
#include <string>
#include <map>
int main () {
  std::map<std::string,int> cart;
  //Set some initial values:
  cart["soap"] = 2;
  cart["salt"] = 1;
  cart.insert( std::make_pair( "pen", 10 ) );
  cart.erase("salt");
  //access elements
  std::cout << "Soap: " << cart["soap"] << "\n";
  std::cout << "Hat: " << cart["hat"] << "\n";
  std::cout << "Hat: " << cart.at("hat") << "\n";
}
```

Exercise 2.3 *What will happen at the last two calls?*

Exceptions in Containers (abstract data types)

Containers must be used under certain conditions.

Example 2.2

Read operation `cart.at("shoe")` must not be called if the cart does not value for key `"shoe"` .

Since containers cannot return an appropriate value, they throw exceptions in the situations.

Callers must be ready to catch the exceptions and respond accordingly.

Please ask in tutorial session, if you need explanations related to exceptions.

STL: container libraries with unified interfaces

Since the containers are similar

`http://www.cplusplus.com/reference`

C++ in flux

Once C++ was set in stone. Now, modern languages have made a dent!

Three major revisions in history!!

- c++98
- c++11
- c++17
- c++20 (we will use this compiler!)

Topic 2.1

Array vs. Vector

Vector

- Variable length
- Primarily stack-like access
- Allows random access
- Difficult to search
- Overhead of memory management

Array

- Fixed length
- Random access
- Difficult to search
- Low overhead

Let us create a test to compare the performance

```
#include <iostream>
#include <vector>
#include "rdtsc.h"
using namespace std; // unclear!! STOP ME!
int local_vector(size_t N) {
  vector<int> bigarray; //initially empty vector
  //Fill vector upto length N
  for(unsigned int k = 0; k<N; ++k)
    bigarray.push_back(k);
  //Find the max value in the vector
  int max = 0;
  for(unsigned int k = 0; k<N; ++k) {
    if( bigarray[k] > max )
      max = bigarray[k];
  }
  return max;
} // 3N memory operations
```

Let us create a test to compare the performance (2)

```
// call local_vector M times
int test_local_vector( size_t M, size_t N ) {
  unsigned sum = 0;
  for(unsigned int j = 0; j < M; ++j ) {
    sum = sum + local_vector( N );
  }
  return sum;
}
//In total, 3MN memory operations
```

Let us create a test to compare the performance (3)

```
// assumes the 64-bit machine
int main() {
  ClockCounter t; // counts elapsed cycles
  size_t MN = 4*32*32*32*32*16;
  size_t N = 4;
  while( N <= MN  ) {
    t.start();
    test_local_vector( MN/N , N );
    double diff = t.stop();
    //print average time for 3 memory operations
    std::cout << "N = " << N << " : "<< (diff/MN);
    N = N*32;
  }
}
```

Exercise 2.4

Write the same test for arrays.

Topic 2.2

Problem

Exercise: What is the difference between at and ..[..] accesses?

Exercise 2.5

What is the difference between "at" and "..[..]" accesses in C++ maps?

Commentary: **Solution:** The `at` function can throw an exception when the accessed element is not in range. The `[]` operator calls the default constructor of the value type and then returns the allocated object. This may be considered bad behavior because a read action causes changes in the data structure, which is undesirable. If there are many out-of-range reads to the map, the map will be filled with many dummy entries. On the other hand, throwing exceptions is not ideal from a programmer's perspective. They have to constantly add code that handles exceptions. If such a code is not added, then the exceptions may cause failure of the entire system.

Exercise: smart pointers

Exercise 2.6

C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are

- `shared_ptr`
- `unique_ptr`
- `weak_ptr`
- `auto_ptr`

Write programs that illustrate the differences among the above smart pointers.

Commentary: **Solution:** Memory leak occurs when a program allocates some memory and stops referencing it. C++ does not automatically deallocate memory when a program does not reference a part of memory. However, the language supports smart pointers. Smart pointers are classes that count references to an object. If the number of references hits zero, the memory is deallocated. `shared_ptr` allows one to allocate memory without worrying about memory leaks. `unique_ptr` is like `shared_ptr` but it does not allow a programmer to have two references to an address. `weak_ptr` allows one to refer to an object without having the reference counted. This kind of pointer is needed in case of cycles among pointers. Due to the cycle, the reference counter never goes to zero even if the program stop referencing the memory. `weak_ptr` is used to break the cycle. The program in later slides illustrates the difference between weak and shared pointers. `auto_ptr` is now a deprecated class. It was replaced by `shared_ptr` etc in C++11.

Exercise: named requirements

Exercise 2.7

Some of the containers have named requirements in their description. For example, “std::vector (for T other than bool) meets the requirements of Container, AllocatorAwareContainer (since C++11), SequenceContainer, ContiguousContainer (since C++17), and ReversibleContainer.”.

What are these? Can you describe the meaning of these? How these conditions are checked?

Topic 2.3

Extra slides: weak pointers

An illustrative example of weak pointer usage (continued)

```
#include <iostream>
#include <memory>
class Node {
public:
  Node(int value) : value(value) {std::cout << "Node " << value << " created." << std::endl; }
  // Functions to set/get the next node/weak ref to previous node/shared ref to previous node
  void setNext    ( std::shared_ptr<Node> next ) { nextNode = next;     }
  void setWeakPrev( std::shared_ptr<Node> next ) { prevWeakNode = next; }
  void setPrev    ( std::shared_ptr<Node> next ) { prevNode = next;     }
  std::shared_ptr<Node> getNext()     const      { return nextNode;            }
  std::shared_ptr<Node> getPrev()     const      { return prevNode;            }
  std::shared_ptr<Node> getWeakPrev() const      { return prevWeakNode.lock(); }
  // Function to display the value of the node
  void display() const { std::cout << "Node value: " << value << std::endl; }
private:
  int value;
  std::shared_ptr<Node> nextNode;
  std::shared_ptr<Node> prevNode;
  std::weak_ptr<Node>   prevWeakNode;
};
void print_list( std::weak_ptr<Node> current ) {
  for (int i = 0; i < 5; ++i) {
    auto current_ref = current.lock();
    if (current_ref) {
      current_ref->display();
      current = current_ref->getNext();
    } else {
      std::cout << "Next node is nullptr." << std::endl; break;
    }
  }
}
```

An example of weak pointer usage (2)

```cpp
// Creating a doubly linked list via shared_ptr/weak_ptr
std::weak_ptr<Node> shared_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  // Create a circular reference
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setPrev(node1); // shared pointer pointing to previous node is causing a reference cycle
  node3->setPrev(node2);
  return node1;
}
std::weak_ptr<Node> weak_test() {
  auto node1 = std::make_shared<Node>(1);
  auto node2 = std::make_shared<Node>(2);
  auto node3 = std::make_shared<Node>(3);
  node1->setNext(node2);
  node2->setNext(node3);
  node2->setWeakPrev(node1); // weak pointer pointing to previous node breaks cyclic reference counting
  node3->setWeakPrev(node2);
  return node1;
}
int main() {
  std::cout << "Testing shared pointer:" << std::endl;
  auto current = shared_test();
  print_list(current);
  std::cout << "Testing weak pointer:" << std::endl;
  current = weak_test();
  print_list(current);
  return 0;
}
```

End of Lecture 2