Introduction to Programming

Pointers and the STL Library

April 16, 2020

Memory, Pointers & Reference

What is Memory?

- Memory $\neq$ Storage
- Evolution in perspective:
  - Were data is stored.

- How we think of memory **drastically** affects how we approach it.

What is Memory?

- Memory $\neq$ Storage
- Evolution in perspective:
  - Were data is stored.
  - A physical component that allows us to store data.

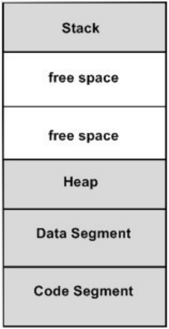
- How we think of memory **drastically** affects how we approach it.

What is Memory?

- Memory $\neq$ Storage
- Evolution in perspective:
  - Were data is stored.
  - A physical component that allows us to store data.
  - Is an abstraction that allows us to manage the physical components in a way we can understand.
- How we think of memory **drastically** affects how we approach it.

Accessing Memory

- The smallest unit of memory we can access is called a *byte*.
- Memory can be seen as a **HUGE** array of bytes, each with an *address*.
- Each program separates memory into different areas, each with its own purpose and *permissions*.
- For now we only work on the *Stack* and *Heap*.



Motivation

```cpp
void add_2(int x){
    x += 2;
}

int main(){
    int x = 5;
    add_2(x);
    cout<<x<<endl;
    return 0;
}
```

- What would this code print?

Motivation

```cpp
void add_2(int x){
    x += 2;
}

int main(){
    int x = 5;
    add_2(x);
    cout<<x<<endl;
    return 0;
}
```

- What would this code print?
- Does the `add_2` function work?

Motivation

```
void add_2(int x){
    x += 2;
}

int main(){
    int x = 5;
    add_2(x);
    cout<<x<<endl;
    return 0;
}
```

- What would this code print?
- Does the `add_2` function work?
- What can we do to make it work?

Pointers

- A pointer is a data type that allow us to store the address of another variable.
- This way we don't keep track of the value of a variable, but *where it is located in memory*.
- By knowing a variables location rather than value, we can access it from **anywhere** in the program.

Pointers - How to use them

When Declaring:

```
<datatype> *<name> = &<variable>;
int *ptr = &var
```

When Using:

*<name> ← The value of in the location in memory

<name> ← The location of the variable itself

Pointers - What would this print?

a)

```
int a = 5;
int *p = a;
cout<<*p<<endl;
```

b)

```
int a = 5;
int *p = a;
cout<<p<<endl;
```

c)

```
int *p = 5;
cout<<*p<<endl;
```

Pointers - What would this print?

a)

```
int a = 5;
int *p = a;
cout<<*p<<endl;
```

5

b)

```
int a = 5;
int *p = a;
cout<<p<<endl;
```

c)

```
int *p = 5;
cout<<*p<<endl;
```

Pointers - What would this print?

a)

```
int a = 5;
int *p = a;
cout<<*p<<endl;
```

```
5
```

b)

```
int a = 5;
int *p = a;
cout<<p<<endl;
```

```
0x0F032010
```

c)

```
int *p = 5;
cout<<*p<<endl;
```

Pointers - What would this print?

a)

```
int a = 5;
int *p = a;
cout<<*p<<endl;
```

5

b)

```
int a = 5;
int *p = a;
cout<<p<<endl;
```

0x0F032010

c)

```
int *p = 5;
cout<<*p<<endl;
```

ERROR

Referencing

- In `C++` we can also create *reference variables*.
- This will essentially give a new name to an already existing variable.
- `int &r = a` ← Anything that happens to `r` will happen to `a`.
- We can use it in *functions* and solve the reference problem.

```
void add_2(int &x){
    x += 2;
}
```

```
void add_2(int *x){
    *x += 2;
}
```

In Review

- Each varaible is located in memory and has an address.
- Pointers allow us to directly interact with memory addresses.
- Referencing allows us to give new names to variables.

`int *p = &a;` ← Creates pointer `p` referencing `a`.

`*p = 5;` ← Updates the value in that location in memory.

`p = &b;` ← Updates the position in memory `p` references.

`int &r = a;` ← `a` can now be called as `r`.

The STL Library

Data Structures - Challenge

We want to write a program that stores a sequence of numbers and allows the following three operations

- Add an element to the front
- Add an element to the back
- Pop the element in the front

How would you do this?

Data Structures

How we store and process data will affect significantly the performance and efficiency of out program.

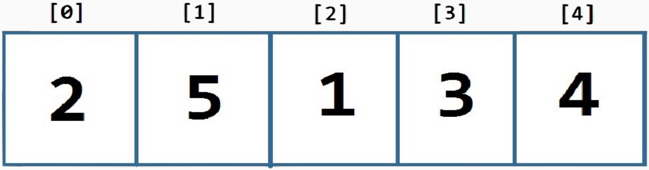
- A data structure is a way of organizing and processing data.
- It is like a blueprint that tells us how we should store and connect values, how these values should relate and the functions that we can apply to the data.
- Just like algorithms, data structures are an abstract concepts that we implement in our computer programs.

What is STL?

- **S**tandard **T**emplate **L**ibrary
- It is the name given to all the native C++ libraries.
- It works with any kind of variables.
- It is mainly made up of:
  - Containers
  - Algorithms
  - Functions
  - Iterators

Vector

- A vector is a structure of dynamic size that stores data sequentially, assigning an index to each element.
- It is basically a more powerful implementation of the native `C++` *array*.



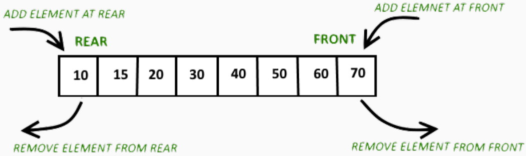
[0]	[1]	[2]	[3]	[4]
2	5	1	3	4

Common Functions

- `size` : Returns size of vector
- `clear` : Erase all elements inside the vector
- `push_back` : Insert an element to the back
- `pop_back` : Erase the last element
- `insert` : Insert element in any position
- `erase` : Erase an element in any position
- `empty` : Returns `true` if the vector is empty
- `front` : Access first element
- `back` : Access last element

Deque

- A *deque* is the `C++` implementation of a *list*.
- It is similar to a vector but some of its *methods* have different *complexities*.
- It has two additional functions:
  - `push_front` : Insert an element at the beginning.
  - `pop_front` : Erase the first element.



Deque - Example

```
int main(){
    deque<int> myDeque = {1,2,3,4,5};
    myDeque.push_front(6);
    myDeque.push_front(0);
    myDeque.push_front(7);
    myDeque.pop_front();
    for(int i = 0 ; i < myDeque.size() ; i++){
        cout << myDeque[i] << ' ';
    }
    return 0;
}
```

Output

Deque - Example

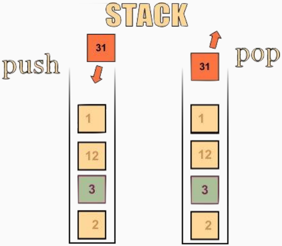
```
int main(){
    deque<int> myDeque = {1,2,3,4,5};
    myDeque.push_front(6);
    myDeque.push_front(0);
    myDeque.push_front(7);
    myDeque.pop_front();
    for(int i = 0 ; i < myDeque.size() ; i++){
        cout << myDeque[i] << ' ';
    }
    return 0;
}
```

Output

```
0 6 1 2 3 4 5
```

Stack

- A stack is a data structure that in which we can only access to the last element added.
- It can be seen as a restricted deque.
- It has the following methods:
  - `empty` : Is the stack empty?
  - `size` : Returns the stacks size.
  - `top` : Returns the top element.
  - `push` : Inserts new element.
  - `pop` : Erase the element on top.



Stack - Example

```
int main(){
    stack<int> myStack;
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);
    myStack.pop();
    while(!myStack.empty()){
        cout << myStack.top() << ’ ’;
        myStack.pop();
    }
    return 0;
}
```

Output

Stack - Example

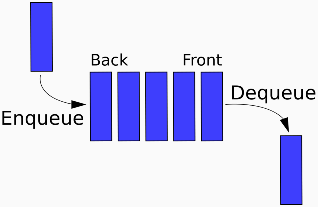
```
int main(){
    stack<int> myStack;
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);
    myStack.pop();
    while(!myStack.empty()){
        cout << myStack.top() << ’ ’;
        myStack.pop();
    }
    return 0;
}
```

Output

```
2 1
```

Queue

- A queue is a data structure in which we can only access the oldest element.
- It can be seen as a restricted deque.
- It has the same methods as *stack*, with the only differences being in `push` and `pop`.



Queue - Example

```
int main(){
    queue<int> myQueue;
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);
    myQueue.pop();
    while(!myQueue.empty()){
        cout << myQueue.front() << ’ ’;
        myQueue.pop();
    }
    return 0;
}
```

Output

Queue - Example

```cpp
int main(){
    queue<int> myQueue;
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);
    myQueue.pop();
    while(!myQueue.empty()){
        cout << myQueue.front() << ’ ’;
        myQueue.pop();
    }
    return 0;
}
```

Output

```
2 3
```

Map

- A data structure that associates a key to a value.
- It is like a vector, but the key can be any kind of datatype and have no particular order.
- The key and the value can have different datatypes.
- Declaration: `map<key type, value type>`

Map - Example

```
int main(){
    map<int,string> myMap;
    myMap.insert({1,"World"});
    myMap[0] = "Hello";
    myMap[2] = "Lorem Ipsum";
    cout << myMap.size() << "\n";
    for(auto i:myMap){
        cout << i.first << ’ ’ << i.second << "\n";
    }
    return 0;
}
```

Output

Map - Example

```cpp
int main(){
    map<int,string> myMap;
    myMap.insert({1,"World"});
    myMap[0] = "Hello";
    myMap[2] = "Lorem Ipsum";
    cout << myMap.size() << "\n";
    for(auto i:myMap){
        cout << i.first << ' ' << i.second << "\n";
    }
    return 0;
}
```

Output

```
0 Hello
1 World
2 Lorem Ipsum
The map is empty
```

Other Structures

- priority_queue
- set
- unordered_set
- multiset
- unordered_map
- multimap
- pair
- tuple

Algorithms - `std::sort`

- The `sort` algorithm allows us to sort containers very quickly.
- It is usally used on *vectors* and *arrays*
- `sort(start address, end address, comparator)`
- There are very few cases in which we should implement our own sort.
- Sort works over the container, it doesn't return a copy but actually swaps elements.
- **Ex.** `sort(vec.begin(), vec.end());`