C++ Report

Arda Elibol

1 C++ Fundamentals: Data Types, Memory Model, and Basic Data Structures

1. Simple Variables

```
My Variables
----------------------------------------------
int:     15                 [Size: 4]
float:   0.515              [Size: 4]
double:  0.151515           [Size: 8]
char:    a                  [Size: 1]
bool:    1                  [Size: 1]
----------------------------------------------

Variable Limits
----------------------------------------------
int:                        -2147483648, 2147483647
float:                      1.17549e-38, 3.40282e+38
double:                     2.22507e-308, 1.79769e+308
char:                       �,
bool:                       0, 1
----------------------------------------------

Behaviors
----------------------------------------------
08 bit int                  [�, ]
16 bit int                  [-32768, 32767]
32 bit int                  [-2147483648, 2147483647]
Unsigned Plain int          [0, 4294967295]

float and double with precision 2:
0.51                        0.15

float and double with precision 5:
0.51500                     0.15152

bool: 1 // Prints 1 for true value.

bool: 0 // Prints 0 for false value.

char: a // char for the ASCII: 97
----------------------------------------------
```

2. Variables and Storage Duration

- **auto:** auto keyword deducts the variable type as the type of the assigned value.

  auto x = 10;
  As *10* is an integer, *x* is also deducted as an integer.

  Auto is mostly useful while dealing with complicated variable types:
  for (const auto& p: path)
  Here, for example, the path is a vector of **pair<float, float>**. Using **auto** simplyfies the code

- **const:** const keyword makes the variable immutable to changes and read-only. This protects the values from accidental changes.
- **constexpr:** In addition to const, constexpr keyword makes the value of the variable known at the compile-time. This saves memory.

- **lvalue and rvalue:** lvalue is an object with a location and a name. rvalue is a non-persistent value.

  int y;
  y = 5;
  y is the lvalue, while *5* is the rvalue. The value *5* is assigned to the object with name *y* and adress *&y.*

  10 = y;
  causes an error, since 10 doesn't have an adress.

3. Intro to Pointers

The memory adress is the place where an object stays. A pointer, points to the adress of the object and accesses the value via it's adress. The value can be obtained by dereferencing the pointer.

```
"x:" << x << endl
"&x:" << ptr << endl
"x:" << *ptr << endl;
```

```
x:   5
&x:  0x7ffd29f8acec
x:   5
```

Value of *x*
Adress of *x*
Value of *x* by dereferencing

If a pointer doesn't have an initializer, it points to a random point at the memory. As such a situation is risky (like corrupting a data), assigning nullptr to the pointer is the safe way to keep it when it doesn't have an object to point.

```
int* ptr = new int[5];
cout << ptr << endl;
ptr = nullptr;
cout << ptr << endl;
```

```
0x5f9fa94fceb0
0
```

The pointer points to a random adress
The pointer points to nothing

If a pointer gets destroyed without destroying the object, the object causes memory leak. In order to prevent memory leak, there are three smart pointers in modern C++.

- **unique_ptr:** The pointer takes the ownership of the object and whenever gets deleted, the object gets deleted with it.
- **shared_ptr:** A bunch of pointers take the ownership of the object and whenever all of them gets deleted, the object gets deleted with them.

- **weak_ptr:** If there are objects pointing each other, a cycle appears between objects keeping alive each other. weak_ptr breaks the cycle by making an object behave like a spectator.

4. Dynamic Memory Allocation

At most of the cases, the memory that will be used is unknow. Thus, dynamic memory allocation applications are common in real-world systems.

```
int* ptr = new int;         // Allocate memory for one int
delete ptr;                 // Deallocate the memory

int* ptr = new int[x];      // Allocate memory for x ints
delete[] ptr;               // Deallocate the memory
```

When memory is allocated manually and be forgotten to be deallocated, memory leaks occur. To avoid memory leaks, manual memory allocation should be used at minimum. Some structures (e.g., *vector*) call new and delete by themselves, avoiding memory leaks.

5. Introduction to Basic Data Structures

- **Array:** It is used to store previously-known amount of data.
  + Array works performance-friendy because of fixed size.
  - That array is not flexible makes it hard to deal with unknown elements.
- **Linked List:** Each element keeps a pointer to another element. Last element points to nullptr, which shows the list has ended.
  + It is easier to add/pop elements.
  - There is no direct access to the elements.
- **Stack and Queue:** Elements can only be added from the back. There is no random access. Stack obeys "Last In, First Out", while Queue obeys "First In, First Out" principle.
  + There is no iteration and random access in them, so accidental changes are prevented.
  - Looping is not possible unles *.pop()* is used.

6. C++ STL Containers

```
Pointers and Dynamic Memory Allocation
-----------------------------------------
Enter an integer:         5
x:                        5
Size of x:                4
Adress of x:              0x7ffc0312cf04
Size of the adress:       8
x by pointer:             5
Adress of x by pointer:   0x7ffc0312cf04
Elements of the array:    1 2 3 4 5
Elements of the vector:   1 2 3 4
New x:                    10
-----------------------------------------
```

```
List, Stack and Queue
-----------------------------------------
First of the list:        6
New First of the list:    1
First:                    1
Second:                   5

Top of the stack:         5
New Top of the stack:     4

Front of the queue:       1
Back of the queue:        5
New Front of the queue:   2
-----------------------------------------
```