C++ fundamentals with use cases from finance

Lecture 3: Defining your own structures in C++

Ivan Zhdankin

Arithmetics

- To do calculations in C++ we use arithmetic operators:

$$+; -; /; *$$

- There are some shortcuts of the operators:

$$+=; -=; *=; /=$$

- Increment / Decrement operators:

$$i++; ++i; i--; --i$$

- The post-increment ($i++$) return the old value and then increment i, for example:

```
int i = 1;
int j = i++;
(Output: j = 1; i = 2)

int i = 1;
int j = ++i;
(Output: j = 2; i = 2)
```

- Module % returns the reminder after dividing a/b in the example: 5 % 3
- There is no exponential operator in C++: so we can not do 2^3
- The order of arithmetics matter in similar way as we do math
- Demo: Arithmetics

Comparisons operators

- To compare in C++ we use comparisons operators:

$$>, <, >=, <=, ==, !=$$

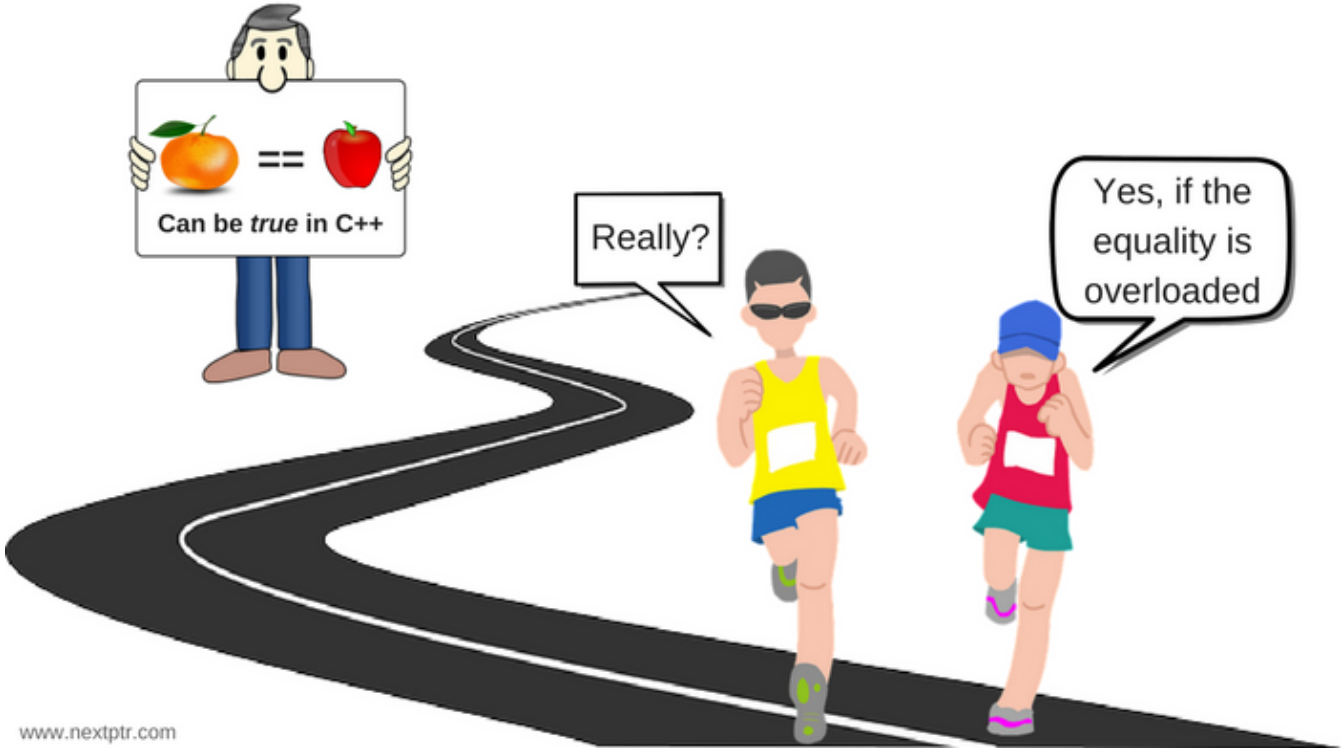
- When we want to combine two conditions we would use:

$$\&\&(and), ||(or)$$

- The not operator ! reverse the condition to an opposite
- Demo: Comparisons

Operator overloading

- Operator overloading is one of the central features of C++ which makes C++ different from other languages
- This feature makes C++ to be generic



- There are many embedded operator overloading in Standard Library so that we normally do not notice them
- "*Firstname*" + "*Lastname*" : operator + here is overloaded to be able to sum 2 strings from Standard Library
- We can overload any operators that exist in C++ and adopt them for our needs

Writing an Overload

- Suppose we are writing a class and we would like to overload the comparison operator $<$ for different *Fruits*
- There are two different ways we can do this:

$$Fruit < something$$

$$something < Fruit$$

- *Fruit* $<$ *something*:
  - We can create a member function in the class *Fruit*:

$$bool \quad MyClass :: operator < (anotherClass \quad object)$$

  - The left side has to be of *Fruit* type
  - The right side has to be of the type we are comparing against
- *something* $<$ *Fruit*:
  - We can create Free function:

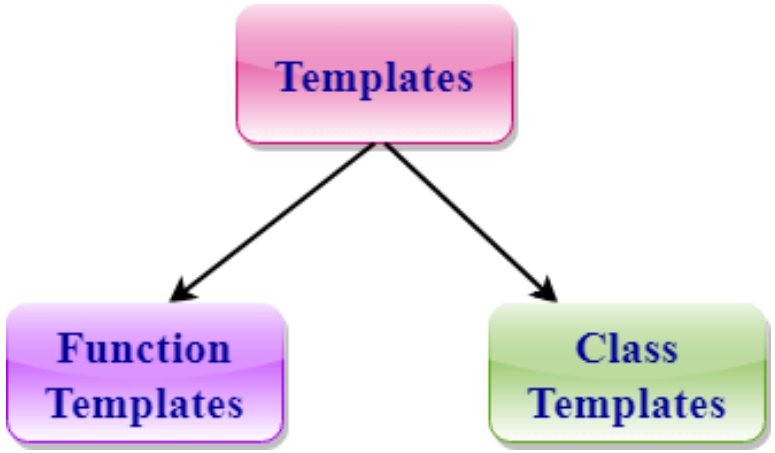
$$bool \quad operator < (anotherClass \quad object, Fruit \quad apple)$$

  - In this case we might need the free-function to be friendly for the *Fruit* class
  - Friend functions - are those that allows to access the private variables and has to be declared inside a class to indicate the friendship
- Demo: Operator Overloading

Templates in C++

- To avoid repetitions in the code we write functions in C++
- Templates are similar, they are used for generality at even higher level
- The idea for templates is that you write a function or a class once and it works for very different types
  - We can write a function that perform max, min or average for various types: numbers, fruits, strings and students
  - We can write collections which store and perform searching for different types (type safe collections). We can store transactions, bonds, trades or employees in a collection
- Templates are often rely on operator overloads
- Much of the Standard Library are based on the templates
- Templates are resolved at compile-time meaning and there is no run-time checks. This speeds up running the application at run-time

Template Functions and Classes



- We can write a template *functions* identically to the ordinary function

```
template<class T>
T max(T t1, T t2)
```

- We can also write a template *classes*

```
template<class T>
class Latest {
private:
T latest;
};
```

- Demo: Template Function and Classes

References vs Pointers

Reference

- Alias or another name for already existing variable
- An alias for variable that already exists
- Operator &
- There can NOT be *NULL* references
- Must be initialized when creating
- After initialization, it is impossible the reference to refer to another object

Pointer

- Object that stores the memory address of another value located in memory
- Holds the address of a memory location
- Operator $*$
- There can be *NULL* pointers
- Possible to initialize at anytime
- After initialization, the pointer can point to another object at any time

References and Pointers(ii)

- The following is a syntax for the references:

```
int a = 3;
int& rA = a;
```

- The following is a syntax for the pointers:
  - To initialize a pointer we have the following syntax:

```
int a = 3;
int* pA = &a;
```

  - By applying & operator to the variable *a* we return the address of its memory location
  - To get through the pointer to its target use $*$ operator:

```
*pA = 4;
```

  - To get through the pointer pointing to the object we can use $->$ operator:

```
Customer c2("James", "Bond", 7);

Customer* pC = &c2;

cout << "Customer's name is " << pC->getName() << endl;
```

- Demo: Pointers and References

Const with Indirections

- A keyword *const* indicates to developer and compiler that something can not be changed
  - If the variable is *const*, the value of it can not be changed:

```
int const ci = 3;
```

  - As a function parameter passed *by value* meaning that the parameter does not change inside the function

```
int addtwo(int const i){
int x = i + 2;
return x;}
```

  - A function parameter can be passed *by reference* meaning that we do not create local copy of the variable and work directly with the variable:

```
int addtwo(int & i){
int x = i+2;
return x;}
```

  - More commonly when the function takes a parameter *by reference* and we know that function should not change this argument we need to mention this for the compiler by using the keyword *const*:

```
int addtwo(int const & i){
int x = i+2;
return x;}
```

  - If we have a member function of the class and we know that the function will not change any of the member variables goof practise to mention this for the compiler by using the keyword *const*:

```
string getname() const{
return "First_name" + "Second_name";}
```

- Remember as a rule of thumb to use a keyword *const* as much as possible

Const with Indirections (ii)

- Remember that the *references* can not be reassigned to other objects
- A keyword *const* with a reference means that we can not change a value for the target
- We can declare the pointer to be *const*

```
int * const cpI;
```

- This means that we can not change the pointer to point to somewhere else
- Another example is when the pointer is changeable but something the pointer points to can be changed:

```
int const * cpI;
```

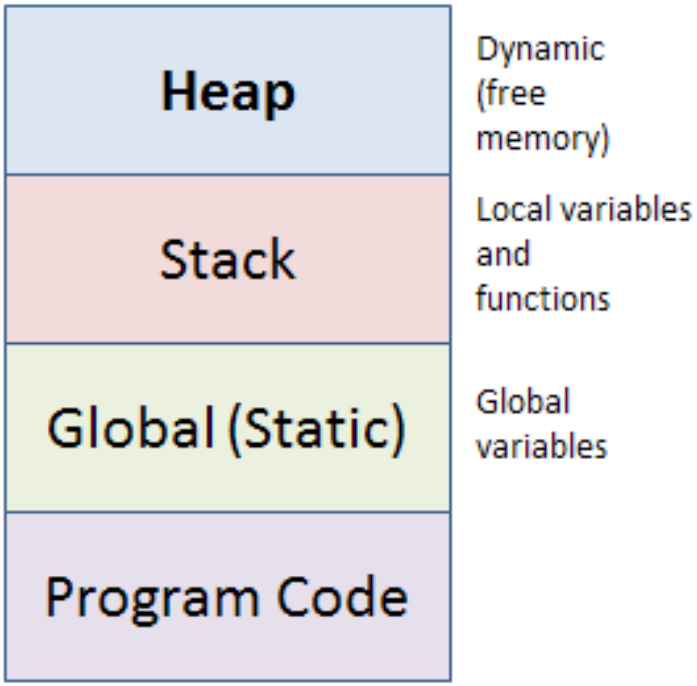
- We can not use the above *cpI* to change the value of a target
- Or we can do both at the same time: we can declare a pointer is *const* and it points at something that is *const*:

```
int const * const cpI;
```

- We can not change it to point somewhere else or use it to change the value of the target
- Demo: Const and Indirections

C++ Memory Overview

- 4 segments of the memory:



  - **Heap**: dynamically allocated variables, objects
  - **Stack**: local variables, objects and functions
  - **Global**: variables outside of stack and heap
  - **Program Code**: compiled program

Heap

- Sometimes we want the variables to live longer than the scope of the function
- We can use *Heap* for storing such variables
- Local variables are often described as being on the *stack*
- There is a *heap* data structure which is used for the *Heap*
- So we can hear that a memory is allocated on the *heap*
- Local variables are the same as *stack variables*
- To get some memory from the *Heap* we use *new* keyword
  - The keyword *new* will allocate a memory and return a pointer to the object we just created
  - The construct of the object will be run
  - When we are done with the allocated memory we clean the memory by calling the keyword *delete*
  - The memory will be released for other usages and the *destructor* of the object will be called
- Demo: Heap

Manual Memory Management

- If we created a pointer using *new*, we have to keep track on it
  - If we call *new* we have to call *delete*
- We need to think of if the copies of the pointer are created and what if we delete the pointer
- Or what if someone created a copy and then delete it
- We have to take care of memory leaks

- Manual memory management is hard and it is easy to make mistakes

Standard Library Smart Pointers

- In the Standard Library there are smart pointers that handle the dynamic memory management for us
- *unique_ptr* - unique pointer
  - We can not make a copy of the unique pointer
  - When the object goes out of scope the memory is released
- *shared_ptr* - for the cases we would like to copy
  - Reference counted - as we make copies of the pointer the total number is updated every time
  - When the object goes out of scope the total number counts back down deleting the copies until the memory is released
- *weak_ptr* - lets look at the shared pointer without changing the reference count

Interview Questions: Increments, Operator Overloading, Template

- What is the difference between: $i++$ and $++i$
- How can we define comparison between two objects (bonds for example)?
- What is template in C++? Code up a function *max* using Template.

Interview Questions: Indirections, Const, Heap

- What are the differences between references and pointers?
- If a member-function is ”const” what it means?
- What is ”Memory Leak” problem?