Homework 3 (CS3100)

Anurag Mittal
Dept. of Computer Sci. & Engg.,
IIT Madras.

Submitted by **Ameet Deshpande (CS15B001)**.

September 24, 2017

1. **Should constructors in base classes be made virtual? What about destructors?**

Solution: Constructors in base classes should not be defined to be virtual as it will create problems. Infact, constructors in C++ are not allowed to be virtual. Inline is the only keyword that is allowed for constructors.
The point of virtual functions is that the functions can be overridden in the derived classes. If no overriding happens, the same base class function is used. When a function is called using a pointer of base class type, if the pointer was initialized using a derived class type, the function defined in the derived class is called. This is the functionality that the keyword *virtual* provides.
In most languages, the object's type is determined at compile time. These languages are called statically typed languages. C++ is one such language. In this scenario, it makes no sense to have virtual constructors because you already know at compile time which constructor to call, i.e., the one with which it was defined. Put in other words, for a language that is statically typed, it makes no sense to create an object polymorphically. The compiler thus flags an error when virtual is added for a constructor. Whenever a *new* or an equivalent function is being used, the compiler already knows which constructor to call and this, after being decided at compile time, is not changed at runtime. Thus virtual constructors have no meaning for statically typed languages.
However, for destructors it is very crucial to have virtual functions. The main reason is to avoid memory leaks, as this following code shows.

```
1  #include <bits/stdc++.h>
2
3  using namespace std;
4
5  class base {
6    public:
7      base()
8      { cout<<"Constructing base \n"; }
9      ~base()
10     { cout<<"Destructing base \n"; }
11 };
12
```

```
13  class derived: public base {
14    public:
15      derived()
16      { cout<<"Constructing derived \n"; }
17      ~derived()
18      { cout<<"Destructing derived \n"; }
19  };
20
21  int main()
22  {
23    base *b = new derived();
24    delete b;
25  }
```

The output of the code is as follows.

```
1  Constructing base
2  Constructing derived
3  Destructing base
```

This means that, at destruction time, only the base constructor is being called. But at construction time derived class' constructor is also being called. This thus creates a memory leak because, if there were any fields in the derived class which were not present in the base class, they would be constructed, but not destructed. This problem can thus be solved by declaring the destructor as a virtual function. In this case, at run time the derived class' destructor is matched, which calls the base constructor on its own. The output after changing it to virtual is as follows.

```
1  Constructing base
2  Constructing derived
3  Destructing derived
4  Destructing base
```

Note that constructors are actually the way *vptr* (virtual pointer) is made to point to the correct *vtable* (virtual table) entry. It thus does not make sense to resolve constructors using the vtable as it has not even been resolved correctly.
In conclusion, virtual constructors are mostly meaningless and cannot be implemented using virtual table and pointers and virtual destructor is very useful to avoid memory leaks.

2. **Write a generic Stack class and then derive StackInt and StackFloat from it. State the advantages and disadvantages of this approach vs. the template approach.**

Solution:

In this case, both the stackint and stackfloat classes will have similar pop and empty functions. However, the top functions returns different data types for both the classes. We thus cannot use virtual functions for the same. We need to define them separately. The following is the code for inheritance.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  class Stack{
5  public:
6      class Node{
7      public:
8          int data;
9          Node *next;
10     };
11
12     Node *head;
13
14     void push(int);
15     void push(float);
16
17     virtual void pop(){       // Observe that this function will be the same
18         Node *temp = head;
19         head = head->next;
20         delete temp;
21     }
22
23     virtual bool empty(){       // Observe that this function will be the same
24         if (head){
25             return true;
26         }
27         else{
28             return false;
29         }
30     }
31
32     Stack(){
33         head = NULL;
34     }
35 };
36
37 class StackInt : public Stack{
38 public:
39     class Node{
40     public:
41         int data;
42         Node *next;
43     };
44
45     Node *head;
```

```
46
47      int top(){
48          if(head == NULL){
49              return -1;
50          }
51          else{
52              return head->data;
53          }
54      }
55
56      void push(int data){
57          Node *temp = new Node;
58          temp->data = data;
59          temp->next = head;
60          head=temp;
61      }
62  };
63
64  class StackFloat : public Stack{
65  public:
66      class Node{
67      public:
68          float data;
69          Node *next;
70      };
71
72      Node *head;
73
74      float top(){
75          if(head == NULL){
76              return -1;
77          }
78          else{
79              return head->data;
80          }
81      }
82
83      void push(float data){
84          Node *temp = new Node;
85          temp->data = data;
86          temp->next = head;
87          head=temp;
88      }
89  };
90
```

```
91
92  int main(){
93      StackInt a;
94      StackInt b;
95  }
```

The code uses the code reuse feature very well by having to define only the push and top functions. The pop and empty functions are written only once. This code can thus be extended to any other classes byt just writing the same two functions. But I personally feel that, because of the reuse of the exact same code, templates will work better here. Following is the code for template. An explanation about why this would work better is mention after the code.

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   template < class T>
5   class Stack{
6   public:
7       class Node{
8       public:
9           T data;
10          Node *next;
11      };
12
13      Node *head;
14
15      void push(T data){
16          Node *temp = new Node;
17          temp->data = data;
18          temp->next = head;
19          head=temp;
20      }
21
22      void pop(){          // Observe that this function will be the same
23          Node *temp = head;
24          head = head->next;
25          delete temp;
26      }
27
28      bool empty(){        // Observe that this function will be the same
29          if (head)
30              return true;
31          else
32              return false;
33      }
34
```

```
35      T top(){
36          if (head == NULL)
37              return -1;
38          else
39              return head->data;
40      }
41
42      Stack(){
43          head = NULL;
44      }
45  };
46
47  int main(){
48      Stack<int> a;
49      Stack<float> b;
50  }
```

In just 50 lines of code, Stack objects of any class type can be formed.

- The reason why templates work better in this case is that the code that is run for all the different object types is exactly the same. There is no code change that is being made from int to float. At the same time, the compiler will be able to optimize based on the class type.
- Inheritance is more useful when we want to add extra functionalities compared to the already existing functionalities of the base class. In that scenario, we can inherit the class and then add only those features which are extra. Since the extra features are the same in all the cases, inheritance is not the best option here.
- Another problem with inheritance in this case is that virtual function cannot be declared for the top function because the return types are different for different types but the function arguments are the same (void). Function overriding will not work in this case.

3. **List the different uses of "const" in C++ with examples.**

Solution: const is a keyword in C++ used to define something as a constant in the program. But there are several ways to do the same and there are several ways you can use the keyword.

- The first and most common one is const values. They are defined to make sure that the value of the variable is not changed throughout scope of the variable. The following program will thus give a compile time error because we are trying to modify the value of a const integer. const can thus provide a good alternative to #define macros to define constants to be used in the program. As should be obvious, we need to provide the value of const at initialization time.

```
1   const int i = 5;
2   int x = i*5; // Accessing is fine
3   i = 10;    // Error
```

```
4       i++;    // Error
```

- const can also be used with pointer declarations. There are a few cases here, pointer to a constant variable and a constant pointer and others. They are all explaned in the code below. We can use the spiral rule to read what the pointer declarations mean, which essentially warrants us to read the declaration backwards.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      int x = 3;
7      const int y = 3;
8      int *p1;                  // Pointer to integer
9      int const * p2;           // Pointer to const integer
10     int * const p3 = &x;      // const integer pointer
11     int const * const p4 = &y; // const integer pointer to const int
12 }
```

- Another use of const is to have constant member functions. When a function is declared as const, it is not allowed to modify the object which has called it. So that object is essentially non-mutatable or read only for the function. It is actually a recommended practice to have as many functions as const as this leaves lesser scope for unnoticed error where the programmer might have made changes to the data though the function doesn't warrant the same. As an obvious extension, a const function can make calls only to other const functions. An interesting thing to note is that, if an object is declared const, it can be called only by const functions, thus enforcing the programmer to explicitly mention which functions are const and which are not. The following code highlights all the main functionalities.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  class Point{
5      int x;
6      int y;
7
8  public:
9      Point(int x1, int y1){
10         x = x1; y = y1;
11     }
12
13     int get_x() const{
14         return x;
15     }
16
```

```
17      int get_y() const{
18          return y;
19      }
20  } ;
21
22
23  int main(){
24      const Point center(3,4) ;
25      cout<<"Values are: "<<center.get_x()<<" : "<<center.get_y()<<endl;
26  }
```

- const can be used as a way to implement the "Copy on Write" mechanism. This is an optimization in which, if the compiler knows that some call or set of statements are taking in a value from an object, but are not modifying it, we might as well pass the pointer to the object itself rather than creating extra memory. This way, we can end up saving memory. An additional mutable pointer is passed only when there is a write happening. const thus becomes a way to let the compiler decide when and when not to copy. This can be done in the code snippet below.

```
1       struct string_new {
2       char * getData() { return myData; }      // Incase variable can be changed
3       char const* getData() const { return mData; }   // If variable is not changed
4   };
```

  To motivate the same, the following code suggests how the compiler implements COW. When a const is initialized with another const, as we know that the values will not be changed, we can have them point to the same memory address. This can be seen in the code below.

```
1   int main() {
2       string const a = "ParadigmsOfProgramming";
3       string const b = a;
4       cout << (void*)&a[0] << ", " << (void*)&b[0];
5   }
```

  Interestingly, both the addresses are the same because of the reason explained before.

- const can be used as a smart way to pass values to functions by reference. In many a cases, passing by value becomes very expensive. Passing by reference is used in those cases. But as a programming practice it is a bad idea because the function might change the values of the object and it will get reflected in the calling scope. To avoid this, functions can be called by passing a const reference. This way, it is a reference and it wont be modified as well. The following code snippet shows the same. It can be seen that if the size of Object is very big, like in most real life examples, passing this const reference has instant gains.

```
1   Object_ RandomFunction( const Object_& a, const Object_& b )
2   {
3       return a+b;      // Say '+' is an overloaded operator
4   }
```

- Copy constructors usually make copies of the passed object. It is however necessary to make sure that the object is not modified. This can be viewed as a subcase of the previous point. Since we do not want to modify the object, we can pass it by reference.

```
1  class Point {
2  public:
3      int x;
4      int y;
5      Point(const Point &other)
6      {
7          x = other.x;
8          y = other.y;
9      }
10 }
```

- Though the list was not exhaustive, it encompasses many ways in which const is used.

4. **Write a max function to find the maximum of two inputs. State the advantages and disadvantages of using a macro vs. an inline function for the same.**

Solution: The following code illustrates the three types of performing the max that are required in the question. #define macro, inline function and a normal function.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  #define max1(a,b) (((a)>(b))?(a):(b))
5
6  inline int max2(int a, int b)
7  {
8      if (a > b)
9          return a;
10     else
11         return b;
12 }
13
14 int normal_max(int a, int b)
15 {
16     if (a > b)
17         return a;
18     else
19         return b;
20 }
21
22 int main()
```

```
23  {
24      int x,y;
25      cin>>x>>y;
26      cout<<"Using Macro Definition: "<<max1(x,y)<<endl;
27      cout<<"Using Inline Function: "<<max2(x,y)<<endl;
28      cout<<"Using Normal Function: "<<normal_max(x,y)<<endl;
29  }
```

- A first point which is common to both is that overhead for function call is lesser. Procedure calls are implicitly implemented by pushing and popping on the stack. By having an inline function or a macro, all those are avoided. But the trade-off is that the code/executable size becomes bigger.
- Macro expressions can result in side effects as they are copied whenever the call is seen. To avoid exactly this, on line number 4 of the code, there is extensive use of brackets. An example to show the problem is, #define mod 1000+7. Say we do mod*7. We expect 1007*7 as the answer. But however, because of the precedence, we and up getting 1000+(7*7) as the answer. In inline function, the function is exactly the same as any other functions except the code is injected before runtime.
- The freedom of expression is far less in Macros than in Inline functions. Inline functions allow almost all functionalities of Normal functions. But conditional statements like `if` cannot be used in macros **directly**. This is because Macros are just textual replacement. We can't expect them to return values and assign to variables. Though there are workarounds by using ternary operators, this makes the code very unreadable. Macros are supposed to be used to make the code readable. They are thus only used normally to define constants, *small* expressions and other similar things. But note that we can also have macro expressions which mimic function calls. For example, #define PRINT(X) printf("%d",X);.
- Macro expressions can in general be expanded recursively. That is, one macro expression can have another macro expression in it and all the macros will be expanded. This is not necessarily the case with inline functions. It might not inline the function calls present in the body. One problem for inline functions could be recursive calls. While in principle a compiler can convert a recursive function into an iterative function and inline it, it is not always done. Some compiler may complain about inlining recursive functions. Compilers may also not inline functions which ave loops. As said before, inlining is just a request and the compiler has full authority to decided if it actually wants to inline or not.
- Inline functions are exactly similar to normal functions for most purposes. The type checking and other rules enforced on normal functions are also enforced on inline functions. Macros are however expanded first by preprocessor and then the compilation is done. Thus, the number of places inline can be used is restrictive (only places where normal functions can be used). Macros are more flexible. But however typechecking in inline functions will be more rigorous and it is easier to catch errors. In macros, no typechecking is enforced and any object can be passed as an argument. Consider the following code where the program will compile but the programmer may have no clue why the wrong answer is being outputted. The output of the following code will be 10 which is actually the wrong answer.

```
#include <bits/stdc++.h>
using namespace std;

#define max1(a, b)   ((a < b) ? b : a)

int main( void)
{
    cout <<"Maximum of 10 and 20 is "<<max1("20", "10")<<endl;
    return 0;
}
```

- One point that cannot be ignored is that, there may be cases when the request to inline a function may be ignores if the compiler feels that the function has a lot of lines and it would make the file very big if all the functions are expanded. But with Macros, the programmer has full control over the fact that Macros **will** be expanded for sure. Inline can thus be viewed as a equest to the compiler and macros as a command.

5. **Why should the operator "=" return a reference to *this?**

Solution:

- Like any other operator, = can also be regarded as a fucntion. It thus assigns the rvalue to lvalue. There are several languages like C++ which allow chaining of assignment statement. There will this be expressions like the following.

```
    A a,b,c;
      .....
    a = b = c;
```

Here we see that there are essentially two assignment statements. b = c is executed, and then a reference to this of b is returned, which is then assigned to a. The following code shows how the operator can be overloaded to achieve the same.

```
    class A{
    public:
    A& operator =(A& a);
    };

    A& A::operator =(A& a){
        if (this == &a){
            return *this;
        }
        return *this;
    }

```

```
13      int main(){}
```

- The previous code block illustrated how a reference can be returned. If we know for sure that there will be no chain assignment statements, we can make the return type of the function to be void by redefining the overloaded = operator. It is however advised not to change the default behavior.
- One other way to change the function is to make it return by value.

```
1  class A{
2  public:
3  A operator =(A& a);
4  };
5
6  A A::operator =(A& a){
7       if (this == &a){
8            return *this;
9       }
10      return *this;
11 }
12
13 int main(){}
```

But this approach faces the problem that a copy of the object will be created each time the value is returned. Thus a chain assignment will necessitate that the constructor is called twice. This will result in unnecessary overheads. When it is known that assignment statements will not change the rvalue, it makes more sense to pass the reference so that overheads are saved.

- Another use is ease in programming. An assignment statement statement can be present in an expression as it can return the value. It is thus not required to have two separate statements. As discussed in class, the value of assignment can be retained in the register and thus the program on the whole will run faster. The following is an example of the same.

```
1  #include <bits/stdc++.h>
2  using namespace std;
3
4  int main()
5  {
6      bool flag=false;
7      if(flag=true){
8          cout<<"Using \"=\" function"<<endl;
9      }
10 }
```

6. **State the advantages and disadvantages of using a default parameter vs. overloaded functions.**

Solution:

The following are advantages and disadvantages of default arguments and constructors.

- Default parameters allow users call functions with a lesser number of arguments than required. The default values for parameters can be defined at the time of programming and when a function is called without value for a certain parameter which has a default value, that value is inserted. Note, however, at a function signature remains the same throughout all calls. It's just that, one or more of the arguments could be missing. This is especially useful when there is an argument which is called with the same value again and again. For example, while calculating the amount of interest for a user in a bank, the interest rate usually stays constant for a decently long period of time it would thus make sense to have a default value for the interest rate which makes it easier to call the function. On the other hand, constructors allow all kinds of different signatures. It allows different functions to have the same name and their differentiated based on their signature which includes the types of their arguments and the number of their arguments. So in our running example of calculating interest, there would be two functions of the same name and one of them would have the interest rate argument missing. It will have that value hardcoded inside its body.

- As can be immediately seen from the definition in the previous point, default parameters may be very useful when the functionality remains the same even if parameters value is missing. In the interest rate case, the function is doing the same thing even when the interest rate argument was missing. However, if we wanted to have different functionality is based on the signature this method will not work. For example, if the interest rate was not provided we might want to Prompt the bank manager to enter a suitable interest rate. Is the functionality is different in this case you can have two functions with different signatures and the compiler will decide which function to call based on if the value is missing or not. This is thus one case where function overloading could be more beneficial.

- Function overloading is a very general concept. We can have two functions doing very different things based on what the arguments are and at the same time have two different functions operating on different kinds of data types but doing the same thing. A classic example of this is overloading the operator "+". This operator doing very different things based on what data type is operating on. Images of integers it will return the addition of the two integers. In the case of Strings it might return the concatenation of two strings. Overloading thus provides a very user-friendly way to use functions as a user will not have to worry about what the function name is depending on what data type is. Note that function binding is done statically.

- It is allowed only for the rightmost arguments in the function to have default values. This is done to ensure there is no ambiguity for the compiler. This limits the flexibility of using default values. But one case where this can be more beneficial than function overloading is, say there is a function like the following.

```
1 int calculate_something(int a, int b, int c=1, int d=2, int e=3, int f=4){
2
3 }
```

To achieve the same in function overloading, we will have to write 5 different functions depending on if f is absent, or both f and e are absent, and so on. And writing the same code again and again is not an elegant way to design your software.

7. **Is it a good practice to return a reference to internal data members via functions? Explain.**

Solution: It is in general considered a bad practice to return reference to internal data members. Here are a few reasons highlighting the same.

- First one is that it violates the principle of Data abstraction and Encapsulation. When we return a reference to a data member, we are leaking an implementation detail. We lose the power of manipulating the internals however we wish and provide only the functionality. For example, if the functionality was to maintain a few records of students in some school, we could have maintained the internal structure however we wanted, provided the functions were "fast" enough and the program was producing the right answer. But returning the reference now breaks that as we need to leak an implementation detail in terms of a reference. This leads to the next point.
- It is harder to maintain the **Data Invariant** that was talked about in class. When we return a reference, there are functions out of the scope of the class that have access to the members. They can manipulate it however they will. A function out of the class may manipulate some fields of the object and it would have no clue anything happened. This might lead to serious problems. As is obvious, the data invariant is not maintained anymore. For example, if there is a field called $seats$ which models the number of seats in an office. Suppose the invariant is that it has to be greater than 10 to be able to accommodate enough people. This can be easily done by having an if condition like, $\texttt{if}(seats < 10)$ after every manipulation to the object. This ensures that the value is checked wherever it is being manipulated. But when a reference is returned, if a function, using that reference, changes the value of $seats$ to 5, there is no check performed, and the object does not satisfy the data invariant anymore.
- Like with other references, lifetime issues are created. Say a reference to some field in object A is returned and an object B is using that reference. If the object A goes out of scope, or gets destroyed or relocated, there is no way for B to know that anything like that has happened. This is the *dangling reference* problem. Though returning $\texttt{const}$ references may get rid of problem number 2, they will still have the dangling reference problem and the first problem mentioned.
- If the programmer is aware that a reference is being passed and is careful about the invariants, references can be used for convenience, but they are usually to be avoided.

8. **Explain how and when implicit type casts are done and used by C++.**

Solution: Type casting is a very useful feature and is one place where the compiler is just a little loose about type checking. Following points explain the same in some detail.

- Assigning a variable with a value of different compatible data type. Compatible implies that the compiler should be able to cast one of the values into another value.

```
1 double val(3);
2 val = 6;
```

- The same functionality can be used if we want to pass values to a function. If say the function argument expected is *long long* and you pass an *int*. This is very common in programming. The compiler type casts it implicitly. Infact the same thing is done for objects as well. If the argument expected is of base class type and the argument passed is of derived class type, type casting is performed implicitly.

```
1 void useless_func(long l){
2 }
3
4 useless_func(3);   // Type cast
```

- Another obvious place where it is used is in expressions. The following code is self-explanatory.

```
1     double division = 4.0 / 3; // Convert 3 to (double)3
```

- Implicit casts are basically performed when the programmer does not explicitly say how to convert a variable. All the above examples are those of implicit type conversions. There are usually two categories.
  - Numeric Promotion: The variable type is converted a bigger version of the same thing. For example int to long long or float to double.
  - Numeric Conversions: When a variable is converted to a different data type. That is, there is no such hierarchical relationship between the two data types. For example, converting an int to a double. These kind of conversions may result in loss of data. For example assigning an `int` to a `char`, though legal, may lead to unexpected results because of overflows.

  At execution time, the variable which is to be type casted is replaced with a temporary instance of the correct data type. For example, if an integer a is being implicitly typecasted to float, at *runtime*, the value stored in the integer are interpreted and converted to float using standard conversion rules. This is how C++ internally implements it.