ESS201: Programming-II
Module: C++

Jaya Sreevalsan Nair
jnair@iiitb.ac.in
International Institute of Information Technology, Bangalore

Term I: 2018-19; Lecture-04 on October 24, 2018

“Tsk, tsk,” said the Hatter, “what a mess you’ve made.”

“It is perfectly fine,” replied Alice calmly. “I will leave it for the garbage collection service to recover.”

”Don’t expect any garbage collection here. Furthermore, your polymorphic variables won’t ever be properly deleted, because you haven’t declared your destructor to be virtual.”

“My what to be what?” said Alice, starting to get worried.

“Declare your destructor. You must have a destructor. Everything that is constructed should be destroyed; it’s only natural. Furthermore, if you are ever not quite what you seem, you should declare yourself to be virtual.”

“A rule to remember!” roared the Red Queen. “Never make a mess without cleaning it up first.”

“You can ignore her,” whispered the Dormouse, picking up the tea cake Alice had just set aside, “but you shouldn’t cast away const so lightly.”

Alice began to feel that this new world she found herself in was not quite the same as the cozy sitting room she had just left.

– Timothy Budd (1999) C++ for Java Programmers. p. v

1 Constructors

Notes on C++ constructors.

1.1 Order of Construction

The order of construction followed is first for global objects, and then local objects – in the order of construction within `main` (i.e. in the main function itself or other functions). e.g.,

```
#include <iostream>
#include <string>

class Test { public: Test(std::string const &name) ; };

Test::Test(string const &name) {
std::cout << "Test: " << name << std::endl; }
void func() { Test functest("func") ; return ; }

Test globaltest("global")

int main() { Test l1("main1") ; func() ; Test l2("main2") ; }
```

Generated output:

```
Test: global
Test: main1
Test: func
Test: main2
```

1.2 Member Initialization

In the class `Test`, say the data members are given as:

```
class Test{
   private:
      std::string _name } ;
```

Then the constructor could be:

```
Test::Test(std::string const &name) { _name = name }
```

This would mean an assignment of the data member using the parameters to the constructor. This implies that the object `name` must exist prior to the actual construction, and in the body of the constructor `_name` changes value from an initialized value to an assigned value. The shortcomings to the assignment in constructor are:

- This is an inefficient way of constructing an instance of the class.
- This does not work if the data member is of `const`. Say the data member `_dateOfBirth` of a class `Person` is provided only once and hence is of const type.

Hence, a better option would be to perform an initialization using the input parameters in the constructor function. C++ syntax allows member initializers as a list of constructor specifications between a colon following the parameter list of the constructor and the definition of the constructor. e.g.

```
Test::Test(std::string const &name):_name(name) {}
```

2 Miscellaneous Notes on C++ Functions

1. *Question: The public member functions can access the private data members of object which is an input parameter to the function, but belonging to the same class. (True/False)?*

   ```
   class ComplexNumber {
     public:
       ComplexNumber add(ComplexNumber &c) ;
     private:
       float _real, _imag ;
     ...
   } ;

   ComplexNumber ComplexNumber::add(ComplexNumber &c) {
      ComplexNumber sum(_real+c._real, _imag+c._imag) ;
      return sum ;
   }
   ```

   The answer to the question is Yes. The member function `add` can access the private data members of the object `c`. This is because access control rules are sorted out during compile time and the C++ compiler checks only class-level access rules, and not object-level. Hence, the access control rules are applicable only at the class-level and not at the object-level.

2. *Question: Returning reference to a local variable of a function, even for a class member function, is a bad idea. (True/False)?*

   ```
   ComplexNumber& ComplexNumber::add1(ComplexNumber &c) {
   ComplexNumber sum(_real+c._real, _imag+c._imag) ;
        ComplexNumber &result = sum ;
        return result ;
   }

   ComplexNumber& ComplexNumber::add2(ComplexNumber &c( {
       ComplexNumber *sum = new ComplexNumber(_real+c._real,
       _imag+c._imag) ;
      return sum ;
   }
   ```

   The answer to the question is Yes. It is illegal to access the stack memory allocation done during the lifetime of a function (even for a member function) beyond the scope. There could be instances where returning an object may be more expensive in comparison to returning a reference. If one is compelled to return a reference, use pointers to allocate memory in the heap, and then use the reference to that. Thus, if the requirement stays, use an implementation such as `add2` instead of `add1`.

   Nonetheless, for a simple C++ code, the illegal access may work and interestingly, even outside of the lifetime of the function, one may be able to access the memory. That is because the memory on the stack has not been made inaccessible, which does not imply that the memory access is legal.

   Interestingly, for multiple calls to the function add1, if the function of the same object `c1` is called using another object `c2` as its input parameter, i.e. multiple calls of `c1.add1(c2)` will return the same memory location. However, multiple calls of `c1.add2(c2)` will return different memory locations each time, owing to creation of new object. The behavior of `add1` is seen because it keeps accessing the same stack memory which may remain a valid memory location until it is used elsewhere.

That said, excluding the access to stack memory outside of the function, returning references is sometimes desirable. e.g. Reference returns are expected for *operator overloading*, which is a powerful feature in C++.

3 Operator Overloading

The right and left shift operators (>> and <<) are widely used for IO-stream operations. It then becomes desirable that the IO-stream operations seamlessly works in implementations of new classes. This is facilitated in C++ using its feature called operator overloading, where operators can have multiple definitions.

3.1 Assignment Operator

Assignment operator (=) is widely used in programming, explicitly and implicitly. There is a default implementation in C++ which allows the assignment operation.

```
class Person {
  public:
    Person() ;
    Person(char const *name, char const *address) ;
    ~Person() ;

  private:
    char *_name, *_address ;
} ;

void copyPerson(Person const &person) {
  Person tmp ;
  tmp = person ;
}
```

What happens in the internals of the assignment operator is a *byte-by-byte* assignment or copy from one data member to another. Thus, in `copyPerson`, when a reference to an object `person` is expected, a local variable `tmp` is defined with initialized data members. Then, the object referenced by `person` is copied byte-by-byte to `tmp`, for a number of bytes equivalent to the `sizeof(Person)`.

This is dangerous in this case, as the local variable is in the stack memory and will run into the danger of getting deleted outside of the function. When the local variable, `tmp`, gets deleted, the destructor will be called. A well-written destructor would then `delete` the private members of `tmp`. That is fine for all data members, except in the case of pointers, which point to dynamically allocated memory on the heap. This particular case is dangerous because the pointers really belong to the object `person`. Thus, overall, the desirable assignment operator is the one which allocates separate memory and then makes a physical copy of all the data members in the new memory allocations, as shown in Figure 1.

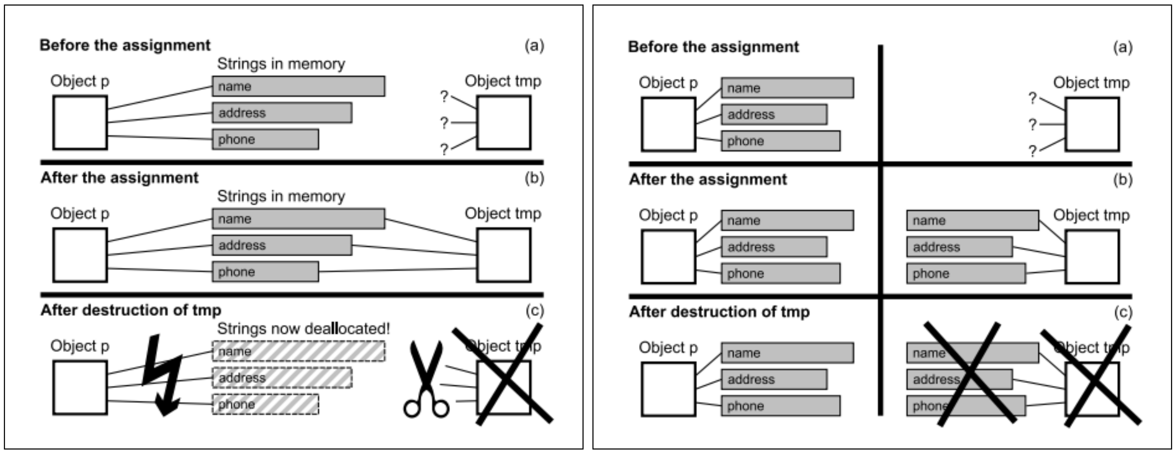

Figure 1: Assignment operator in C++ for copying private data and public interface functions of the class `Person`, using: (left) default byte-by-byte assignment and (right) desirable deep-copy.
(Image courtesy: C++ Annotations).

Thus, a better implementation, as shown in Figure 1, is:

```
void Person::assign(Person const &person) {
  delete [] _name ; // delete old private data
  delete [] _address ;
  _name = new char*(person._name) ; // something in these lines
  _address = new char*(person._address) ; // -do-
  return ;
}

void copyPerson(Person const &person) {
  Person tmp ;
  tmp.assign(person) ;
  return;
}
```

While this is the same in implementation of `assign`, this implementation will additionally take care of all instances when the operator = is used.

Referencing to self in an assignment operator: If one would like a sequential assignment, such as `a=b=c`, the assignment operator will need to return a reference to itself. This is because the sequential assignment works right-to-left, i.e. the expression `b=c` is evaluated first, and its result in turn is assigned to `a`.

```
Person& Person :: operator=(Person const &person) {
  // Put in the same implementation using in the assign function
  delete [] _name ; delete [] _address ;
  _name = new char*(person._name) ; // something in these lines
  _address = new char*(person._address) ; // -do-
  return (*this) ;
}
```

We elaborate on the special feature of C++ of the `this` pointer in Section 4.

3.2 When and Where to Use Operator Overloading?

Even though `C++` allows operator overloading, this feature should *not* be used *indiscriminately.*

Operator overloading may be considered *appropriate* in the following cases:

1. where an operator has a definite default action, but this default action has undesirable side effects in a given situation (e.g. case of the class `Person` we just looked at.)

2. where the operator is commonly applied and its redefinition is intuitively understood, even for new classes. e.g. in the class `std::string`, assigning one string object to another provides the destination string with a copy of the contents of the source string. Similarly, for the class `gridcell` in your assignment, the *intuitive* copy action would mean copying both the `unsigned char` array for `neighborhood`, as well as the `unsigned char` value of the `state` of the cell.

4 `this` Pointer

When an object (instantiation) of a class calls the member function, how does the function "know" the object itself? `C++` defines a keyword, this, to access the object. `this` is a pointer variable containing the address or the memory location of the object whose member function was called. `this` pointer is implicitly declared in all member functions, irrespective of the access control rules, i.e. public, protected, private. It is a `const` pointer to the object of the concerned class.

5 Example Codes and Practicals

We shall use a few examples to articulate various aspects of C++ programming.

1. Overloading assignment, index, and binary operators, e.g. +=: `overload_index_binary.cpp`

2. Header file separation and overloading IO operators: `header_files/`

3. Demo of `fluid`