Module 14: Programming in C++

Copy Constructor and Copy Assignment Operator

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- More on Object Lifetime
- Understand Copy Construction
- Understand Copy Assignment Operator
- Understand Shallow and Deep Copy

Module Outline

Program 14.01/02: Order of Initialization: Order of Data Members

```
#include <iostream>
using namespace std;
int init_m1(int m) { // Func. to init m1_
    cout << "Init m1_: " << m << endl;
    return m;
}
int init_m2(int m) { // Func. to init m2_
    cout << "Init m2_: " << m << endl;
    return m;
}
class X { int m1_; // Initialize 1st
          int m2_; // Initialize 2nd
public: X(int m1, int m2) :
        m1_(init_m1(m1)),  // Called 1st
        m2_(init_m2(m2))   // Called 2nd
        { cout << "Ctor: " << endl; }
    ~X() { cout << "Dtor: " << endl; } };
int main() { X a(2, 3); return 0; }
-----
Init m1_: 2
Init m2_: 3
Ctor:
Dtor:
```

```
#include <iostream>
using namespace std;
int init_m1(int m) { // Func. to init m1_
    cout << "Init m1_: " << m << endl;
    return m;
}
int init_m2(int m) { // Func. to init m2_
    cout << "Init m2_: " << m << endl;
    return m;
}
class X { int m2_; // Order of data members swapped
          int m1_;
public: X(int m1, int m2) :
        m1_(init_m1(m1)),  // Called 2nd
        m2_(init_m2(m2))   // Called 1st
        { cout << "Ctor: " << endl; }
    ~X() { cout << "Dtor: " << endl; } };
int main() { X a(2, 3); return 0; }
-----
Init m2_: 3
Init m1_: 2
Ctor:
Dtor:
```

- *Order of initialization does not depend on the order in the initialization list. It depends on the order of data members in the definition*

Program 14.03/04: A Simple String Class

Module 14

Instructors: Abir Das and Jibesh Patra

Obj. Lifetime
String
Date
Rect
Copy Constructor
Call by Value
Signature
Free Copy & Pitfall
Assignment Op.
Copy Objects
Self-Copy
Signature
Free Assignment
Comparison
Class as Type
Module Summary

C Style

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
struct String { char *str_;  // Container
                size_t len_; // Length
};
void print(const String& s) {
    cout << s.str_ << ": "
         << s.len_ << endl;
}
int main() { String s;

    // Init data members
    s.str_ = strdup("Partha");
    s.len_ = strlen(s.str_);
    print(s);
    free(s.str);
}
-----
Partha: 6
```

C++ Style

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { char *str_;  // Container
               size_t len_; // Length
public: String(char *s) : str_(strdup(s)), // Uses malloc()
                          len_(strlen(str_))
    { cout << "ctor: "; print(); }
    ~String() { cout << "dtor: "; print();
        free(str_); // To match malloc() in strdup()
    }
    void print() { cout << "(" << str_ << ": "
                        << len_ << ")" << endl; }
    size_t len() { return len_; }
};
int main() { String s = "Partha"; // Ctor called
    s.print();
}
-----
ctor: (Partha: 6)
(Partha: 6)
dtor: (Partha: 6)
```

- *Note the order of initialization between `str_` and `len_`. What if we swap them?*

Program 14.05: A Simple String Class: Fails for wrong order of data members


```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class String {
    size_t len_; // Swapped members cause garbage to be printed or program crash (unhandled exception)
    char *str_;
public:
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { cout << "ctor: "; print(); }
    ~String() { cout << "dtor: "; print(); free(str_); }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s = "Partha";
    s.print();
}
----- // May produce garbage or crash
ctor: (Partha: 20)
(Partha: 20) // Garbage
dtor: (Partha: 20)
```

- **`len_` precedes `str_` in list of data members**
- **`len_(strlen(str_))` is executed before `str_(strdup(s))`**
- **When `strlen(str_)` is called `str_` is still uninitialized**
- **May causes the program to crash**

Practice: Program 14.06: A Simple Date Class

```
#include <iostream>
using namespace std;

char monthNames[][4]={ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayNames[][10] ={ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
class Date {
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
    typedef unsigned int UINT;
    UINT date_; Month month_; UINT year_;
public:
    Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y) { cout << "ctor: "; print(); }
    ~Date() { cout << "dtor: "; print(); }
    void print() { cout << date_ << "/" << monthNames[month_ - 1] << "/" << year_ << endl; }
    bool validDate() { /* Check validity */ return true; } // Not implemented
    Day day() { /* Compute day from date using time.h */ return Mon; } // Not implemented
};
int main() {
    Date d(30, 7, 1961);
    d.print();
}
-----
ctor: 30/Jul/1961
30/Jul/1961
dtor: 30/Jul/1961
```

Practice: Program 14.07: Point and Rect Classes: Lifetime of Data Members or Embedded Objects

```
#include <iostream>
using namespace std;
class Point { int x_; int y_; public:
    Point(int x, int y):
        x_(x), y_(y)
    { cout << "Point ctor: ";
      print(); cout << endl; }
    ~Point() { cout << "Point dtor: ";
               print(); cout << endl; }
    void print() { cout << "(" << x_ << ", "
                   << y_ << ")"; }
};

int main() {
    Rect r (0, 2, 5, 7);

    cout << endl; r.print(); cout << endl;

    cout << endl;
}
```

```
class Rect { Point TL_; Point BR_; public:
    Rect(int tlx, int tly, int brx, int bry):
        TL_(tlx, tly), BR_(brx, bry)
    { cout << "Rect ctor: ";
      print(); cout << endl; }
    ~Rect() { cout << "Rect dtor: ";
              print(); cout << endl; }
    void print() { cout << "["; TL_.print();
           cout << " "; BR_.print(); cout << "]"; }
};
-----
Point ctor: (0, 2)
Point ctor: (5, 7)
Rect ctor: [(0, 2) (5, 7)]

[(0, 2) (5, 7)]

Rect dtor: [(0, 2) (5, 7)]
Point dtor: (5, 7)
Point dtor: (0, 2)
```

- Attempt is to construct a Rect object
- That, in turn, needs constructions of `Point` data members (or embedded objects) – `TL_` and `BR_` respectively
- Destruction, initiated at the end of scope of destructor's body, naturally follows a reverse order

Copy Constructor

- We know:

  `Complex c1(4.2, 5.9);`

  invokes

  `Constructor Complex::Complex(double, double);`

- Which constructor is invoked for?

  `Complex c2(c1);`

  Or for?

  `Complex c2 = c1;`

- It is the **Copy Constructor** that takes an object of the same type and constructs a copy:

  `Complex::Complex(const Complex &);`

Program 14.08: Complex: Copy Constructor


```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    // Constructor
    Complex(double re, double im):
        re_(re), im_(im)
    { cout << "Complex ctor: "; print(); }
    // Copy Constructor
    Complex(const Complex& c):
        re_(c.re_), im_(c.im_)
    { cout << "Complex copy ctor: "; print(); }
    // Destructor
    ~Complex()
    { cout << "Complex dtor: "; print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c1(4.2, 5.3), // Constructor - Complex(double, double)
            c2(c1),       // Copy Constructor - Complex(const Complex&)
            c3 = c2;      // Copy Constructor - Complex(const Complex&)

    c1.print(); c2.print(); c3.print();
}
```

```
-----
Complex ctor: |4.2+j5.3| = 6.7624      // Ctor: c1
Complex copy ctor: |4.2+j5.3| = 6.7624 // CCtor: c2 of c1
Complex copy ctor: |4.2+j5.3| = 6.7624 // CCtor: c3 of c2
|4.2+j5.3| = 6.7624                    // c1
|4.2+j5.3| = 6.7624                    // c2
|4.2+j5.3| = 6.7624                    // c3
Complex dtor: |4.2+j5.3| = 6.7624      // Dtor: c3
Complex dtor: |4.2+j5.3| = 6.7624      // Dtor: c2
Complex dtor: |4.2+j5.3| = 6.7624      // Dtor: c1
```

Why do we need Copy Constructor?

- Consider the **function call mechanisms** in C++:
  - *Call-by-reference*: Set a reference to the actual parameter as a formal parameter. Both the formal parameter and the actual parameter share the same location (object). *No copy is needed*
  - *Return-by-reference*: Set a reference to the computed value as a return value. Both the computed value and the return value share the same location (object). *No copy is needed*
  - *Call-by-value*: Make a *copy* or *clone* of the actual parameter as a formal parameter. This needs a **Copy Constructor**
  - *Return-by-value*: Make a *copy* or *clone* of the computed value as a return value. This needs a **Copy Constructor**
- **Copy Constructor** is needed for *initializing the data members* of a UDT from an existing value

Program 14.09: Complex: Call by value

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im): re_(re), im_(im)   // Constructor
    { cout << "ctor: "; print(); }
    Complex(const Complex& c): re_(c.re_), im_(c.im_) // Copy Constructor
    { cout << "copy ctor: "; print(); }
    ~Complex() { cout << "dtor: "; print(); }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
void Display(Complex c_param) { // Call by value
    cout << "Display: "; c_param.print();
}
int main() { Complex c(4.2, 5.3);    // Constructor - Complex(double, double)

    Display(c); // Copy Constructor called to copy c to c_param
}
-----
ctor: |4.2+j5.3| = 6.7624          // Ctor of c in main()
copy ctor: |4.2+j5.3| = 6.7624     // Ctor c_param as copy of c, call Display()
Display: |4.2+j5.3| = 6.7624       // c_param
dtor: |4.2+j5.3| = 6.7624          // Dtor c_param on exit from Display()
dtor: |4.2+j5.3| = 6.7624          // Dtor of c on exit from main()
```

Signature of Copy Constructors

- Signature of a *Copy Constructor* can be one of:

```
MyClass(const MyClass& other);          // Common
                                        // Source cannot be changed
MyClass(MyClass& other);                // Occasional
                                        // Source needs to change. Like in smart pointers
MyClass(volatile const MyClass& other); // Rare
MyClass(volatile MyClass& other);       // Rare
```

- None of the following are copy constructors, though they can copy:

```
MyClass(MyClass* other);
MyClass(const MyClass* other);
```

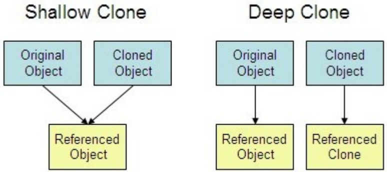
- *Why the parameter to a copy constructor must be passed as Call-by-Reference?*

```
MyClass(MyClass other);
```

The above is an infinite recursion of copy calls as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism

Free Copy Constructor

- If no copy constructor is provided by the user, the compiler supplies a *free* one
- *Free* copy constructor cannot initialize the object to proper values. It performs *Shallow Copy*
- **Shallow Copy** aka *bit-wise copy*, *field-by-field copy*, *field-for-field copy*, or *field copy*
  - An object is created by simply *copying the data of all variables* of the original object
  - Works well if *none of the variables of the object are defined in heap / free store*
  - For dynamically created variables, the *copied object refers to the same memory location*
  - Creates *ambiguity* (changing one changes the copy) and *run-time errors* (dangling pointer)
- **Deep Copy** or its variants *Lazy Copy* and *Copy-on-Write*
  - An object is created by copying data of all variables except the ones on heap
  - Allocates similar memory resources with the same value to the object
  - **Need to explicitly define the copy constructor and assign dynamic memory as required**
  - **Required to dynamically allocate memory to the variables in the other constructors**



Pitfalls of Bit-wise Copy: Shallow Copy

- Consider a class:

```
class A { int i_;      // Non-pointer data member
          int* p_;     // Pointer data member
public:
    A(int i, int j) : i_(i), p_(new int(j)) { } // Init. with pointer to dynamically created object
    ~A() { cout << "Destruct " << this << ": ";                                 // Object identity
        cout << "i_ = " << i_ << " p_ = " << p_ << " *p = " << *p_ << endl; // Object state
        delete p_;                                                                // Release resource
    }
};
```

- As no copy constructor is provided, the implicit copy constructor does a bit-wise copy. So when an A object is copied, p_ is copied and continues to point to the same dynamic int:

```
int main() { A a1(2, 3); A a2(a1); // Construct a2 as a copy of a1. Done by bit-wise copy
    cout << "&a1 = " << &a1 << " &a2 = " << &a2 << endl;
}
```

- The output is wrong, as a1.p_ = a2.p_ points to the same int location. Once a2 is destructed, a2.p_ is released, and a1.p_ becomes dangling. The program may print garbage or crash:

```
&a1 = 008FF838 &a2 = 008FF828                          // Identities of objects
Destruct 008FF828: i_ = 2 p_ = 00C15440 *p = 3         // Dtor of a2. Note that a2.p_ = a1.p_
Destruct 008FF838: i_ = 2 p_ = 00C15440 *p = -17891602 // Dtor of a1. a1.p_=a2.p_ points to garbage
```

- The bit-wise copy of members is known as **Shallow Copy**

Pitfalls of Bit-wise Copy: Deep Copy

- Now suppose we provide a user-defined copy constructor:

```
class A { int i_;       // Non-pointer data member
          int* p_;      // Pointer data member
public:
    A(int i, int j) : i_(i), p_(new int(j)) { } // Init. with pointer to dynamically created object
    A(const A& a) : i_(a.i_),        // Copy Constructor
        p_(new int(*a.p_)) { }       // Allocation done and value copied - Deep Copy
    ~A() { cout << "Destruct " << this << ": ";                               // Object identity
        cout << "i_ = " << i_ << " p_ = " << p_ << " *p = " << *p_ << endl; // Object state
        delete p_;                                                            // Release resource
    }
};
```

- The output now is correct, as $a1.p\_ \neq a2.p\_$ points to the different int locations with the values $*a1.p\_ = *a2.p\_$ properly copied:

```
&a1 = 00B8F9E0 &a2 = 00B8F9D0                          // Identities of objects
Destruct 00B8F9D0: i_ = 2 p_ = 00C95480 *p = 3  // Dtor of a2. a2.p_ is different from a1.p_
Destruct 00B8F9E0: i_ = 2 p_ = 00C95440 *p = 3  // Dtor of a1. Works correctly!
```

- This is known as **Deep Copy** where every member is copied properly. Note that:
  - In every class, provide copy constructor to adopt to deep copy which is always safe
  - Naturally, shallow copy is cheaper than deep copy.

Practice: Program 14.10: Complex: Free Copy Constructor

Module 14

Instructors: Abir Das and Jibesh Patra

Obj. Lifetime
String
Date
Rect

Copy Constructor
Call by Value
Signature
Free Copy & Pitfall

Assignment Op.
Copy Objects
Self-Copy
Signature
Free Assignment

Comparison

Class as Type

Module Summary

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) { cout << "ctor: "; print(); } // Ctor
 // Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout<<"copy ctor: "; print(); } // CCtor: Free only
    ~Complex() { cout << "dtor: "; print(); }                                          // Dtor
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
void Display(Complex c_param) { cout << "Display: "; c_param.print(); }
int main() { Complex c(4.2, 5.3);  // Constructor - Complex(double, double)
    Display(c);                     // Free Copy Constructor called to copy c to c_param
}
```

User-defined CCtor	Free CCtor
`ctor: |4.2+j5.3| = 6.7624`	`ctor: |4.2+j5.3| = 6.7624`
`copy ctor: |4.2+j5.3| = 6.7624`	`No message from free CCtor`
`Display: |4.2+j5.3| = 6.7624`	`Display: |4.2+j5.3| = 6.7624`
`dtor: |4.2+j5.3| = 6.7624`	`dtor: |4.2+j5.3| = 6.7624`
`dtor: |4.2+j5.3| = 6.7624`	`dtor: |4.2+j5.3| = 6.7624`

- User has provided *no copy constructor*
- Compiler provides *free copy constructor*
- Compiler-provided copy constructor *performs bit-wise copy* - hence there is no message
- *Correct in this case* as members are of built-in type and there is no dynamically allocated data

Practice: Program 14.11: String: User-defined Copy Constructor

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }          // Ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor: User provided
    ~String() { free(str_); }                                          // Dtor
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
void strToUpper(String a) { // Make the string uppercase
    for (int i = 0; i < a.len_; ++i) { a.str_[i] = toupper(a.str_[i]); }
    cout << "strToUpper: "; a.print();
} // a.~String() is invoked releasing a.str_. s.str_ remains intact
int main() { String s = "Partha"; s.print(); strToUpper(s); s.print(); }
---
(Partha: 6)
strToUpper: (PARTHA: 6)
(Partha: 6)
```

- User has *provided copy constructor*. So Compiler *does not provide free copy constructor*
- When actual parameter `s` is copied to formal parameter `a`, space is allocated for `a.str_` and then it is copied from `s.str_`. On exit from `strToUpper`, `a` is destructed and `a.str_` is deallocated. But in `main`, `s` remains intact and access to `s.str_` is valid.
- **Deep Copy**: While copying the object, the pointed object is copied in a fresh allocation. *This is safe*

Practice: Program 14.12: String: Free Copy Constructor


```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }              // Ctor
    // String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor: Free only
    ~String() { free(str_); }                                              // Dtor
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
void strToUpper(String a) { // Make the string uppercase
    for (int i = 0; i < a.len_; ++i) { a.str_[i] = toupper(a.str_[i]); } cout<<"strToUpper: "; a.print();
} // a.~String() is invoked releasing a.str_ and invalidating s.str_ = a.str_
int main() { String s = "Partha"; s.print(); strToUpper(s); s.print(); } // Last print fails
```

User-defined CCtor	Free CCtor
`(Partha: 6)`	`(Partha: 6)`
`strToUpper: (PARTHA: 6)`	`strToUpper: (PARTHA: 6)`
`(Partha: 6)`	`(????????????????????????????: 6)`

- User has provided *no copy constructor*. Compiler provides *free copy constructor*
- Free copy constructor performs *bit-copy* - hence no allocation is done for `str_` when actual parameter `s` is copied to formal parameter `a`. `s.str_` is merely copied to `a.str_` and both continue to point to the same memory. On exit from `strToUpper`, `a` is destructed and `a.str_` is deallocated. Hence in `main` access to `s.str_` is dangling. Program prints garbage and / or crashes
- **Shallow Copy**: With bit-copy, only the pointer is copied - not the pointed object. *This is risky*

Copy Assignment Operator

- We can copy an existing object to another existing object as

```
Complex c1 = (4.2, 5.9), c2(5.1, 6.3);

c2 = c1;    // c1 becomes { 4.2, 5.9 }
```

  This is like normal assignment of built-in types and overwrites the old value with the new value

- It is the **Copy Assignment** that takes an object of the same type and overwrites into an existing one, and returns that object:

```
Complex::Complex& operator= (const Complex &);
```

Program 14.13: Complex: Copy Assignment


```cpp
#include <iostream>
#include <cmath>
using namespace std;
class Complex { double re_, im_; public:
    Complex(double re, double im) : re_(re), im_(im) { cout << "ctor: "; print(); }     // Ctor
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout << "cctor: "; print(); } // CCtor
    ~Complex() { cout << "dtor: "; print(); }                                           // Dtor
    Complex& operator=(const Complex& c)  // Copy Assignment Operator
    { re_ = c.re_; im_ = c.im_; cout << "copy: "; print(); return *this; } // Return *this for chaining
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; } }; // Class Complex
int main() { Complex c1(4.2, 5.3), c2(7.9, 8.5); Complex c3(c2); // c3 Copy Constructed from c2
    c1.print(); c2.print(); c3.print();
    c2 = c1; c2.print();                                 // Copy Assignment Operator
    c1 = c2 = c3; c1.print(); c2.print(); c3.print(); // Copy Assignment Chain
}
```

```
ctor: |4.2+j5.3| = 6.7624   // c1 - ctor
ctor: |7.9+j8.5| = 11.6043  // c2 - ctor
cctor: |7.9+j8.5| = 11.6043 // c3 - ctor
|4.2+j5.3| = 6.7624         // c1
|7.9+j8.5| = 11.6043        // c2
|7.9+j8.5| = 11.6043        // c3
copy: |4.2+j5.3| = 6.7624   // c2 <- c1
|4.2+j5.3| = 6.7624         // c2
```

```
copy: |7.9+j8.5| = 11.6043  // c2 <- c3
copy: |7.9+j8.5| = 11.6043  // c1 <- c2
|7.9+j8.5| = 11.6043        // c1
|7.9+j8.5| = 11.6043        // c2
|7.9+j8.5| = 11.6043        // c3
dtor: |7.9+j8.5| = 11.6043  // c3 - dtor
dtor: |7.9+j8.5| = 11.6043  // c2 - dtor
dtor: |7.9+j8.5| = 11.6043  // c1 - dtor
```

- Copy assignment operator should *return the object to make chain assignments possible*

Program 14.14: String: Copy Assignment

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }        // Ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
    ~String() { free(str_); }                                         // Dtor
    String& operator=(const String& s) {                              // Copy Assignment Operator
        free(str_);            // Release existing memory
        str_ = strdup(s.str_); // Perform deep copy
        len_ = s.len_;         // Copy data member of built-in type
        return *this;          // Return object for chain assignment
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s2 = s1; s2.print(); }
---
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- In copy assignment operator, `str_ = s.str_` should not be done for two reasons:
  1) Resource held by `str_` will *leak*
  2) *Shallow copy* will result with its related issues
- What happens if a self-copy `s1 = s1` is done?

Program 14.15: String: Self Copy

```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }        // Ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
    ~String() { free(str_); }                                        // Dtor
    String& operator=(const String& s) {                             // Copy Assignment Operator
        free(str_);            // Release existing memory
        str_ = strdup(s.str_); // Perform deep copy
        len_ = s.len_;         // Copy data member of built-in type
        return *this;          // Return object for chain assignment
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s1 = s1; s1.print(); }
---
(Football: 8)
(Cricket: 7)
(????????: 8) // Garbage is printed. May crash too
```

- For self-copy

- Hence, `free(str_)` first releases the memory, and then `strdup(s.str_)` tries to copy from released memory
- **This may crash or produce garbage values**
- **Self-copy must be detected and guarded**

Program 14.16: String: Self Copy: Safe


```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;
class String { public: char *str_; size_t len_;
    String(char *s) : str_(strdup(s)), len_(strlen(str_)) { }        // Ctor
    String(const String& s) : str_(strdup(s.str_)), len_(s.len_) { } // CCtor
    ~String() { free(str_); }                                         // Dtor
    String& operator=(const String& s) {                             // Copy Assignment Operator
        if (this != &s) { // Check if the source and destination are same
            free(str_);
            str_ = strdup(s.str_);
            len_ = s.len_;
        }
        return *this;
    }
    void print() { cout << "(" << str_ << ": " << len_ << ")" << endl; }
};
int main() { String s1 = "Football", s2 = "Cricket"; s1.print(); s2.print(); s1 = s1; s1.print(); }
---
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for se

- In case of self-copy, do nothing

Signature and Body of Copy Assignment Operator

- For class `MyClass`, typical copy assignment operator will be:

```
MyClass& operator=(const MyClass& s) {
    if (this != &s) { // Check if the source and destination are same
                      // Release resources held by *this
                      // Copy members of s to members of *this
    }
    return *this;     // Return object for chain assignment
}
```

- Signature of a *Copy Assignment Operator* can be one of:

```
MyClass& operator=(const MyClass& rhs); // Common. No change in Source
MyClass& operator=(MyClass& rhs);       // Occasional. Change in Source
```

- The following *Copy Assignment Operator*s are occasionally used:

```
MyClass& operator=(MyClass rhs);
const MyClass& operator=(const MyClass& rhs);
const MyClass& operator=(MyClass& rhs);
const MyClass& operator=(MyClass rhs);
MyClass operator=(const MyClass& rhs);
MyClass operator=(MyClass& rhs);
MyClass operator=(MyClass rhs);
```

Free Assignment Operator

- If no copy assignment operator is provided/overloaded by the user, the compiler supplies a *free* one
- *Free* copy assignment operator cannot copy the object with proper values. It performs *Shallow Copy*
- In every class, provide copy assignment operator to adopt to deep copy which is always safe

Comparison of Copy Constructor and Copy Assignment Operator

Copy Constructor	Copy Assignment Operator
• An overloaded constructor	• An operator overloading
• Initializes a new object with an existing object	• Assigns the value of one existing object to another existing object
• Used when a new object is created with some existing object	• Used when we want to assign existing object to another object
• Needed to support call-by-value and return-by-value	
• Newly created object use new memory location	• Memory location of destination object is reused with pointer variables being released and reallocated
	• Care is needed for self-copy
• If not defined in the class, the compiler provides one with bitwise copy	• If not overloaded, the compiler provides one with bitwise copy

Class as a Data-type

- We add the copy construction and assignment to a class being a composite data type in C++

```
// declare i to be of int type
int i;

// initialise i
int i = 5;
int j = i;
int k(j);

// print i
cout << i;

// add two ints
int i = 5, j = 6;
i+j;

// copy value of i to j
int i = 5, j;
j = i;
```

```
// declare c to be of Complex type
Complex c;

// initialise the real and imaginary components of c
Complex c = (4, 5); // Ctor
Complex c1 = c;     // CCtor
Complex c2(c1);     // CCtor

// print the real and imaginary components of c
cout << c.re << c.im;
OR c.print(); // Method Complex::print() defined for printing
OR cout << c; // operator<<() overloaded for printing

// add two Complex objects
Complex c1 = (4, 5), c2 = (4, 6);
c1.add(c2); // Method Complex::add() defined to add
OR c1+c2; // operator+() overloaded to add

// copy value of one Complex object to another
Complex c1 = (4, 5), c2 = (4, 6);
c2 = c1; // c2.re <- c1.re and c2.im <- c1.im by copy assignment
```

Module Summary

- **Copy Constructors**
  - A new object is created
  - The new object is initialized with the value of data members of another object
- **Copy Assignment Operator**
  - An object is already existing (and initialized)
  - The members of the existing object are replaced by values of data members of another object
  - Care is needed for self-copy
- **Deep and Shallow Copy for Pointer Members**
  - Deep copy allocates new space for the contents and copies the pointed data
  - Shallow copy merely copies the pointer value – hence, the new copy and the original pointer continue to point to the same data

Module 15: Programming in C++

Const-ness

Instructors: Abir Das and Jibesh Patra

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{abir, jibesh}@cse.iitkgp.ac.in

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand const-ness of objects in C++
- Understand the use of const-ness in class design

Module Outline

Constant Objects

- Like objects of built-in type, objects of user-defined types can also be made constant
- If an object is constant, none of its data members can be changed
- The type of the `this` pointer of a constant object of class, say, `MyClass` is:

```
// const Pointer to const Object
const MyClass * const this;
```

  instead of

```
// const Pointer to non-const Object
MyClass * const this;
```

  as for a non-constant object of the same class
- A constant object cannot invoke normal methods of the class as these methods can change the object

Program 15.01: Non-Constant Objects

```
#include <iostream>
using namespace std;
class MyClass { int myPriMember_;
public: int myPubMember_;
    MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) { }
    int getMember() { return myPriMember_; }
    void setMember(int i) { myPriMember_ = i; }
    void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};
int main() { MyClass myObj(0, 1);             // Non-constant object

    cout << myObj.getMember() << endl;
    myObj.setMember(2);
    myObj.myPubMember_ = 3;
    myObj.print();
}
---
0
2, 3
```

- It is okay to invoke methods for non-constant object **myObj**
- It is okay to make changes in non-constant object **myObj** by method (**setMember()**)
- It is okay to make changes in non-constant object **myObj** directly (**myPubMember_**)

Program 15.02: Constant Objects


```
#include <iostream>
using namespace std;

class MyClass { int myPriMember_; public: int myPubMember_;
    MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) { }
    int getMember() { return myPriMember_; }
    void setMember(int i) { myPriMember_ = i; }
    void print() { cout << myPriMember_ << ", " << myPubMember_ << endl; }
};
int main() { const MyClass myConstObj(5, 6); // Constant object

    cout << myConstObj.getMember() << endl; // Error 1
    myConstObj.setMember(7);                // Error 2
    myConstObj.myPubMember_ = 8;            // Error 3
    myConstObj.print();                     // Error 4
}
```

- It is not allowed to invoke methods or make changes in constant object **myConstObj**
- Error (1, 2 & 4) on method invocation typically is:
  cannot convert 'this' pointer from 'const MyClass' to 'MyClass &'
- Error (3) on member update typically is:
  'myConstObj' : you cannot assign to a variable that is const
- With `const`, **this** pointer is **const MyClass * const** while the methods expects **MyClass * const**
- Consequently, we cannot print the data member of the class (even without changing it)
- Fortunately, constant objects can invoke (select) methods if they are **constant member functions**

Constant Member Function

- To declare a constant member function, we use the keyword `const` between the function header and the body. Like:

```
void print() const { cout << myMember_ << endl; }
```

- A constant member function expects a `this` pointer as:

```
const MyClass * const this;
```

  and hence can be invoked by constant objects
- In a constant member function no data member can be changed. Hence,

```
void setMember(int i) const
{ myMember_ = i; } // data member cannot be changed
```

  gives an error
- Interesting, *non-constant objects* can invoke *constant member functions* (by casting – we discuss later) and, of course, *non-constant member functions*
- *Constant objects*, however, can **only** invoke *constant member functions*
- **All member functions that do not need to change an object must be declared as constant member functions**

Program 15.03: Constant Member Functions

```
#include <iostream>
using namespace std;
class MyClass { int myPriMember_; public: int myPubMember_;
    MyClass(int mPri, int mPub) : myPriMember_(mPri), myPubMember_(mPub) { }
    int getMember() const { return myPriMember_; }                    // const Member Func.
    void setMember(int i) { myPriMember_ = i; }                       // non-const Member Func.
    void print() const { cout << myPriMember_ << ", " << myPubMember_ << endl; } // const Member Func.
};
int main() { MyClass myObj(0, 1);   // non-const object
    const MyClass myConstObj(5, 6); // const object
    // non-const object can invoke all member functions and update data members
    cout << myObj.getMember() << endl;
    myObj.setMember(2);
    myObj.myPubMember_ = 3;
    myObj.print();
    // const object cannot allow any change
    cout << myConstObj.getMember() << endl;
    // myConstObj.setMember(7);      // Cannot invoke non-const member functions
    // myConstObj.myPubMember_ = 8; // Cannot update data member
    myConstObj.print();
}
```

Output

```
0
2, 3
5
5, 6
```

- Now **myConstObj** can invoke **getMember()** and **print()**, but cannot invoke **setMember()**
- Naturally **myConstObj** cannot update **myPubMember_**
- **myObj** can invoke all of **getMember()**, **print()**, and **setMember()**

Constant Data members

- Often we need part of an object, that is, one or more data members to be constant (non-changeable after construction) while the rest of the data members should be changeable. For example:
  - For an **Employee**: `employee ID` and `DoB` should be *non-changeable* while `designation`, `address`, `salary` etc. should be *changeable*
  - For a **Student**: `roll number` and `DoB` should be *non-changeable* while `year of study`, `address`, `gpa` etc. should be *changeable*
  - For a **Credit Card**[1]: `card number` and `name of holder` should be *non-changeable* while `date of issue`, `date of expiry`, `address`, `cvv number` etc. should be *changeable*
- We do this by making the *non-changeable* data members as constant by putting the `const` keyword before the declaration of the member in the class
- **A constant data member cannot be changed even in a non-constant object**
- **A constant data member must be initialized on the initialization list**

[1] May not hold for a card that changes number on re-issue

Program 15.04: Constant Data Member

```
#include <iostream>
using namespace std;
class MyClass { const int cPriMem_; /* const data member */ int priMem_; public:
    const int cPubMem_; /* const data member */ int pubMem_;
    MyClass(int cPri, int ncPri, int cPub, int ncPub) :
        cPriMem_(cPri), priMem_(ncPri), cPubMem_(cPub), pubMem_(ncPub) { }
    int getcPri() { return cPriMem_;  }
    void setcPri(int i) { cPriMem_ = i; } // Error 1: Assignment to const data member
    int getPri() { return priMem_; }
    void setPri(int i) { priMem_ = i; }
};
int main() { MyClass myObj(1, 2, 3, 4);

    cout << myObj.getcPri() << endl; myObj.setcPri(6);
    cout << myObj.getPri() << endl; myObj.setPri(6);

    cout << myObj.cPubMem_ << endl;
    myObj.cPubMem_ = 3;                   // Error 2: Assignment to const data member

    cout << myObj.pubMem_ << endl; myObj.pubMem_ = 3;
}
```

- It is not allowed to make changes to constant data members in **myObj**
- Error 1: **l-value specifies const object**
- Error 2: 'myObj' : you cannot assign to a variable that is const

Credit Card Example

We now illustrate constant data members with a complete example of `CreditCard` class with the following supporting classes:

- `String` class
- `Date` class
- `Name` class
- `Address` class

Program 15.05: String Class: String.h


```
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

class String { char *str_; size_t len_;
public:
    String(const char *s) : str_(strdup(s)), len_(strlen(str_))           // Ctor
    { cout << "String ctor: "; print(); cout << endl; }
    String(const String& s) : str_(strdup(s.str_)), len_(strlen(str_))     // CCtor
    { cout << "String cctor: "; print(); cout << endl; }
    String& operator=(const String& s) {
        if (this != &s) {
            free(str_);
            str_ = strdup(s.str_);
            len_ = s.len_;
        }
        return *this;
    }
    ~String() { cout << "String dtor: "; print(); cout << endl; free(str_); } // Dtor
    void print() const { cout << str_; }
};
```

- Copy Constructor and Copy Assignment Operator added
- `print()` made a constant member function

Program 15.05: Date Class: Date.h


```
#include <iostream>
using namespace std;

char monthNames[][4]={ "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
char dayNames[][10]={ "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
class Date {
    enum Month { Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
    enum Day { Mon, Tue, Wed, Thr, Fri, Sat, Sun };
    typedef unsigned int UINT;
    UINT date_; Month month_; UINT year_;
public:
    Date(UINT d, UINT m, UINT y) : date_(d), month_((Month)m), year_(y)
    { cout << "Date ctor: "; print(); cout << endl; }
    Date(const Date& d) : date_(d.date_), month_(d.month_), year_(d.year_)
    { cout << "Date cctor: "; print(); cout << endl; }
    Date& operator=(const Date& d) { date_ = d.date_; month_ = d.month_; year_ = d.year_; return *this; }
    ~Date() { cout << "Date dtor: "; print(); cout << endl; }
    void print() const { cout << date_ << "/" << monthNames[month_ - 1] << "/" << year_; }
    bool validDate() const { /* Check validity */ return true; }             // Not Implemented
    Day day() const { /* Compute day from date using time.h */ return Mon; } // Not Implemented
};
```

- Copy Constructor and Copy Assignment Operator added
- `print()`, `validDate()`, and `day()` made constant member functions

Program 15.05: Name Class: Name.h

```
#include <iostream>
using namespace std;

#include "String.h"

class Name { String firstName_, lastName_;
public:
    Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)    // Uses Ctor of String class
    { cout << "Name ctor: "; print(); cout << endl; }
    Name(const Name& n) : firstName_(n.firstName_), lastName_(n.firstName_) // Uses CCtor of String class
    { cout << "Name cctor: "; print(); cout << endl; }
    Name& operator=(const Name& n) {
        firstName_ = n.firstName_; // Uses operator=() of String class
        lastName_ = n.lastName_;   // Uses operator=() of String class
        return *this;
    }
    ~Name() { cout << "Name dtor: "; print(); cout << endl; } // Uses Dtor of String class
    void print() const // Uses print() of String class
    { firstName_.print(); cout << " "; lastName_.print(); }
};
```

- Copy Constructor and Copy Assignment Operator added
- print() made a constant member function

Program 15.05: Address Class: Address.h

Module 15

Instructors: Abir Das and Jibesh Patra


```
#include <iostream>
using namespace std;
#include "String.h"

class Address { unsigned int houseNo_; String street_, city_, pin_;
public:
    Address(unsigned int hn, const char* sn, const char* cn, const char* pin): // Uses Ctor of String class
        houseNo_(hn), street_(sn), city_(cn), pin_(pin)
    { cout << "Address ctor: "; print(); cout << endl; }
    Address(const Address& a): // Uses CCtor of String class
        houseNo_(a.houseNo_), street_(a.street_), city_(a.city_), pin_(a.pin_)
    { cout << "Address cctor: "; print(); cout << endl; }
    Address& operator=(const Address& a) { // Uses operator=() of String class
        houseNo_ = a.houseNo_; street_ = a.street_; city_ = a.city_; pin_ = a.pin_; return *this; }
    ~Address() { cout << "Address dtor: "; print(); cout << endl; } // Uses Dtor of String class
    void print() const { // Uses print() of String class
        cout << houseNo_ << " "; street_.print(); cout << " ";
        city_.print(); cout << " "; pin_.print();
    }
};
```

- Copy Constructor and Copy Assignment Operator added
- print() made a constant member function

Program 15.05: Credit Card Class: CreditCard.h


```
#include <iostream>
using namespace std;
#include "Date.h"
#include "Name.h"
#include "Address.h"
class CreditCard { typedef unsigned int UINT; char *cardNumber_;
    Name holder_; Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public: CreditCard(const char* cNumber, const char* fn, const char* ln, unsigned int hn, const char* sn,
    const char* cn, const char* pin, UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear,
    UINT cvv): holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
    expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv) // Uses Ctor's of Date, Name, Address
    { cardNumber_ = new char[strlen(cNumber) + 1]; strcpy(cardNumber_, cNumber);
          cout << "CC ctor: "; print(); cout << endl; }
    // Uses Dtor's of Date, Name, Address
    ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; delete[] cardNumber_; }
    void setHolder(const Name& h)      { holder_ = h; }      // Change holder name
    void setAddress(const Address& a) { addr_ = a; }        // Change address
    void setIssueDate(const Date& d)  { issueDate_ = d; }  // Change issue date
    void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
    void setCVV(UINT v)                { cvv_ = v; }         // Change cvv number
    void print() const { cout<<cardNumber_<<" "; holder_.print(); cout<<" "; addr_.print();
        cout<<" "; issueDate_.print(); cout<<" "; expiryDate_.print(); cout<<" "; cout<<cvv_; }
};
```

- Set methods added
- print() made a constant member function

Program 15.05: Credit Card Class Application

```
#include <iostream>
using namespace std;
#include "CreditCard.h"

int main() { CreditCard cc("5321711934640027", "Sherlock", "Holmes",
                  221, "Baker Street", "London", "NW1 6XE", 7, 2014, 6, 2016, 811);
    cout << endl; cc.print(); cout << endl << endl;;

    cc.setHolder(Name("David", "Cameron"));
    cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
    cc.setIssueDate(Date(1, 7, 2017));
    cc.setExpiryDate(Date(1, 6, 2019));
    cc.setCVV(127);
    cout << endl; cc.print(); cout << endl << endl;;
}
// Construction of Data Members & Object
5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Jun/2016 811

// Construction & Destruction of temporary objects
5321711934640027 David Cameron 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127

// Destruction of Data Members & Object
```

- We could change address, issue date, expiry date, and cvv. This is fine
- **We could change the name of the holder! This should not be allowed**

Program 15.06: Credit Card Class: Constant data members

```
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
    char *cardNumber_;
    const Name holder_;          // Holder name cannot be changed after construction
    Address addr_;  Date issueDate_, expiryDate_; UINT cvv_;
public: CreditCard(...) : ... { ... } ~CreditCard() { ... }

    void setHolder(const Name& h)     { holder_ = h; }      // Change holder name
    // error C2678: binary '=' : no operator found which takes a left-hand operand
    // of type 'const Name' (or there is no acceptable conversion)

    void setAddress(const Address& a) { addr_ = a; }        // Change address
    void setIssueDate(const Date& d)  { issueDate_ = d; }  // Change issue date
    void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
    void setCVV(UINT v)               { cvv_ = v; }         // Change cvv number

    void print() { ... }
};
```

- We prefix `Name holder_` with `const`. Now the holder name cannot be changed after construction
- **In `setHolder()`, we get a compilation error for `holder_ = h;` in an attempt to change `holder_`**
- With `const` prefix `Name holder_` becomes constant – unchangeable

Program 15.06: Credit Card Class: Clean

```
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
    char *cardNumber_;
    const Name holder_;         // Holder name cannot be changed after construction
    Address addr_;
    Date issueDate_, expiryDate_; UINT cvv_;
public:
    CreditCard(...) : ... { ... }
    ~CreditCard() { ... }

    void setAddress(const Address& a)  addr_ = a;         // Change address
    void setIssueDate(const Date& d)   issueDate_ = d;    // Change issue date
    void setExpiryDate(const Date& d)  expiryDate_ = d;   // Change expiry date
    void setCVV(UINT v)                cvv_ = v;          // Change cvv number

    void print() { ... }
};
```

- Method `setHolder()` removed

Program 15.06: Credit Card Class Application: Revised

```
#include <iostream>
using namespace std;
#include "CreditCard.h"
int main() {
    CreditCard cc("5321711934640027", "Sherlock", "Holmes",
                  221, "Baker Street", "London", "NW1 6XE", 7, 2014, 6, 2016, 811);
    cout << endl; cc.print(); cout << endl << endl;;

//    cc.setHolder(Name("David", "Cameron"));
    cc.setAddress(Address(10, "Downing Street", "London", "SW1A 2AA"));
    cc.setIssueDate(Date(1, 7, 2017));
    cc.setExpiryDate(Date(1, 6, 2019));
    cc.setCVV(127);
    cout << endl; cc.print(); cout << endl << endl;;
}
// Construction of Data Members & Object
5321711934640027 Sherlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Jun/2016 811

// Construction & Destruction of temporary objects
5321711934640027 Sherlock Holmes 10 Downing Street London SW1A 2AA 1/Jul/2017 1/Jun/2019 127

// Destruction of Data Members & Object
```

- Now holder_ cannot be changed. So we are safe
- **However, it is still possible to replace or edit the card number. This, too, should be disallowed**

Program 15.07: Credit Card Class: cardNumber_ Issue

```
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;

class CreditCard { typedef unsigned int UINT;
    char *cardNumber_;          // Card number is editable as well as replaceable
    const Name holder_;         // Holder name cannot be changed after construction
    Address addr_;
    Date issueDate_, expiryDate_; UINT cvv_;
public:
    CreditCard(...) : ... { ... }
    ~CreditCard() { ... }

    void setAddress(const Address& a) { addr_ = a; }        // Change address
    void setIssueDate(const Date& d)  { issueDate_ = d; }   // Change issue date
    void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
    void setCVV(UINT v)               { cvv_ = v; }         // Change cvv number

    void print() { ... }
};
```

- It is still possible to replace or edit the card number
- To make the cardNumber_ *non-replaceable*, we need to make this *constant pointer*
- Further, to make it *non-editable* we need to make cardNumber_ point to a *constant string*
- Hence, we change `char *cardNumber_` to `const char * const cardNumber_`

Program 15.07: Credit Card Class: cardNumber_ Issue


```
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;
class CreditCard {
    typedef unsigned int UINT;
    const char * const cardNumber_; // Card number cannot be changed after construction
    const Name holder_;             // Holder name cannot be changed after construction
    Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public: CreditCard(const char* cNumber, const char* fn, const char* ln,
        unsigned int hn, const char* sn, const char* cn, const char* pin,
        UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
        holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
        expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv) {
        cardNumber_ = new char[strlen(cNumber) + 1]; // ERROR: No assignment to const pointer
        strcpy(cardNumber_, cNumber);                // ERROR: No copy to const C-string
        cout << "CC ctor: "; print(); cout << endl;
    }
    ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; delete[] cardNumber_; }

    // Set methods and print method skipped ...
};
```

- cardNumber_ is now a *constant pointer to a constant string*
- With this the allocation for the C-string fails in the body as constant pointer cannot be assigned
- Further, copy of C-string (strcpy()) fails as copy of constant C-string is not allowed
- **We need to move these codes to the initialization list**

Program 15.07: Credit Card Class: cardNumber_ Issue: Resolved


```
// Include <iostream>, "String.h", "Date.h", "Name.h", "Address.h"
using namespace std;
class CreditCard { typedef unsigned int UINT;
    const char * const cardNumber_; // Card number cannot be changed after construction
    const Name holder_;             // Holder name cannot be changed after construction
    Address addr_; Date issueDate_, expiryDate_; UINT cvv_;
public: CreditCard(const char* cNumber, const char* fn, const char* ln,
        unsigned int hn, const char* sn, const char* cn, const char* pin,
        UINT issueMonth, UINT issueYear, UINT expiryMonth, UINT expiryYear, UINT cvv) :
        cardNumber_(strcpy(new char[strlen(cNumber)+1], cNumber)),
        holder_(fn, ln), addr_(hn, sn, cn, pin), issueDate_(1, issueMonth, issueYear),
        expiryDate_(1, expiryMonth, expiryYear), cvv_(cvv)
    { cout << "CC ctor: "; print(); cout << endl; }
    ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; delete[] cardNumber_; }
    void setAddress(const Address& a) { addr_ = a; }        // Change address
    void setIssueDate(const Date& d)  { issueDate_ = d; }   // Change issue date
    void setExpiryDate(const Date& d) { expiryDate_ = d; } // Change expiry date
    void setCVV(UINT v)               { cvv_ = v; }         // Change cvv number
    void print() const { cout<<cardNumber_<<" "; holder_.print(); cout<<" "; addr_.print();
        cout<<" "; issueDate_.print(); cout<<" "; expiryDate_.print(); cout<<" "; cout<<cvv_; }
};
```

- Note the initialization of cardNumber_ in initialization list
- **All constant data members must be initialized in initialization list**

mutable Members

mutable Data Members

- While a *constant* data member is *not changeable* even in a *non-constant object*, a **mutable** data member is *changeable* in a *constant object*
- `mutable` is provided to model *Logical (Semantic) const-ness* against the default *Bit-wise (Syntactic) const-ness* of C++
- Note that:
    - `mutable` is applicable only to data members and not to variables
    - Reference data members cannot be declared `mutable`
    - Static data members cannot be declared `mutable`
    - `const` data members cannot be declared `mutable`
- If a data member is declared `mutable`, then it is legal to assign a value to it from a `const` member function

Program 15.08: mutable Data Members

```
#include <iostream>
using namespace std;
class MyClass {
    int mem_;
    mutable int mutableMem_;
public:
    MyClass(int m, int mm) : mem_(m), mutableMem_(mm) { }
    int getMem() const { return mem_; }
    void setMem(int i) { mem_ = i; }
    int getMutableMem() const { return mutableMem_; }
    void setMutableMem(int i) const { mutableMem_ = i; } // Okay to change mutable
};
int main() { const MyClass myConstObj(1, 2);

    cout << myConstObj.getMem() << endl;
    // myConstObj.setMem(3);                // Error to invoke

    cout << myConstObj.getMutableMem() << endl;
    myConstObj.setMutableMem(4);
}
```

- **setMutableMem()** is a constant member function so that constant **myConstObj** can invoke it
- **setMutableMem()** can still set **mutableMem_** because **mutableMem_** is **mutable**
- In contrast, **myConstObj** cannot invoke **setMem()** and hence **mem_** cannot be changed

Logical vis-a-vis Bit-wise Const-ness

- `const` in C++, models *bit-wise* constant. Once an object is declared `const`, no part (actually, *no bit*) of it can be changed after construction (and initialization)
- However, while programming we often need an object to be *logically* constant. That is, the concept represented by the object should be constant; but if its representation need more data members for computation and modeling, these have no reason to be constant.
- `mutable` allows such surrogate data members to be changeable in a (bit-wise) constant object to model logically const objects
- To use `mutable` we shall look for:
  - A logically constant concept
  - A need for data members outside the representation of the concept; but are needed for computation

Program 15.09: When to use mutable Data Members?

- Typically, when a class represents a constant concept, and
- It computes a value first time and caches the result for future use

```
// Source: http://www.highprogrammer.com/alan/rants/mutable.html
#include <iostream>
using namespace std;
class MathObject {                        // Constant concept of PI
    mutable bool piCached_;               // Needed for computation
    mutable double pi_;                   // Needed for computation
public:
    MathObject() : piCached_(false) { } // Not available at construction
    double pi() const {                   // Can access PI only through this method
        if (!piCached_) {                 // An insanely slow way to calculate pi
            pi_ = 4;
            for (long step = 3; step < 1000000000; step += 4) {
                pi_ += ((-4.0 / (double)step) + (4.0 / ((double)step + 2)));
            }
            piCached_ = true;             // Now computed and cached
        }
        return pi_;
    }
};
int main() { const MathObject mo; cout << mo.pi() << endl; /* Access PI */ }
```

- Here a MathObject is logically constant; but we use mutable members for computation

Program 15.10: When *not* to use mutable Data Members?

- mutable should be **rarely used** – only when it is really needed. A bad example follows:

Improper Design (mutable)

```
class Employee { string _name, _id;
    mutable double _salary;
public: Employee(string name = "No Name",
        string id = "000-00-0000",
        double salary = 0): _name(name), _id(id)
    { _salary = salary; }
    string getName() const;
    void setName(string name);
    string getid() const;
    void setid(string id);
    double getSalary() const;
    void setSalary(double salary);
    void promote(double salary) const
    { _salary = salary; }
};
---
const Employee john("JOHN","007",5000.0);
// ...
john.promote(20000.0);
```

Proper Design (const)

```
class Employee { const string _name, _id;
    double _salary;
public: Employee(string name = "No Name",
        string id = "000-00-0000",
        double salary = 0): _name(name), _id(id)
    { _salary = salary; }
    string getName() const;
    // void setName(string name); // _name is const
    string getid() const;
    // void setid(string id);      // _id is const
    double getSalary() const;
    void setSalary(double salary);
    void promote(double salary)
    { _salary = salary; }
};
---
Employee john("JOHN","007",5000.0);
// ...
john.promote(20000.0);
```

- Employee is not logically constant. If it is, then _salary should also be const
- Design on right makes that explicit