Module 14: Programming in C++

Copy Constructor and Copy Assignment Operator

Partha Pratim Das

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

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

Module Objectives

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

Module Outline

- Lifetime Examples
- Copy Constructor
  - Input Parameters
  - Call-by-Value
  - Initialization List
  - Copy with Pointers – Shallow and Deep Copy
- Copy Assignment Operator
  - Input Parameters
  - Return Type
  - Copy with Pointers – Shallow and Deep Copy
  - Self-copy

Program 14.01: 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 14.02/03: A Simple String Class

C Style

```
#include <iostream>
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);

    return 0;
}
-----
Partha: 6
```

C++ Style

```
#include <iostream>
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();
    return 0;
}
-----
ctor: (Partha: 6)
(Partha: 6)
dtor: (Partha: 6)
```

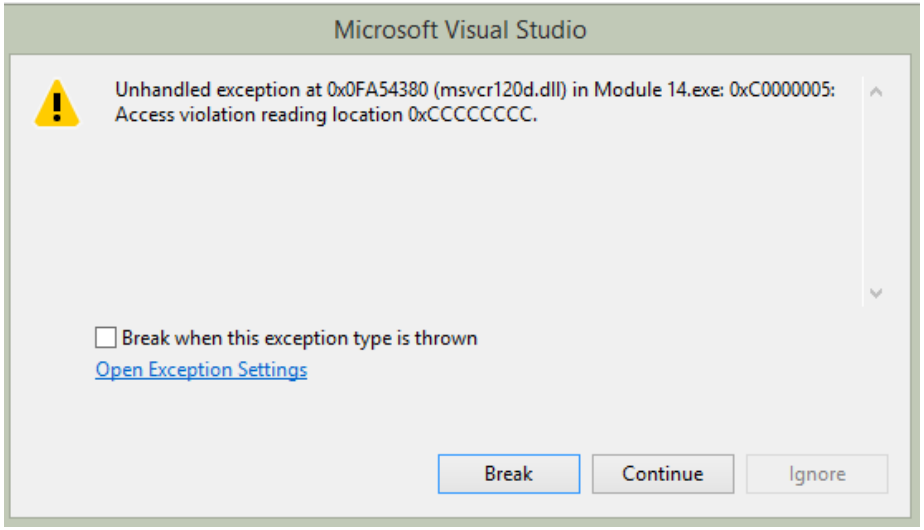
- **Note the order of initialization between `str_` and `len_`. What if we swap them?**

Program 14.04: A Simple String Class – Fails for wrong order of data members


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

class String {
    size_t len_; // Swapped members cause 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();
    return 0;
}
```

- **`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**
- **Causes the program to crash as shown in the message box**



Program 14.05: 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();
    return 0;
}
-----
ctor: 30/Jul/1961
30/Jul/1961
dtor: 30/Jul/1961
```

Program 14.06: 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_ << ")"; }
};

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 << "]"; }
};

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

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

    cout << endl;
    return 0;
}
```

```
-----
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

Program 14.07: Name & Address Classes


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

#include "String.h"
#include "Date.h"

class Name { String firstName_, lastName_;
public:
    Name(const char* fn, const char* ln) : firstName_(fn), lastName_(ln)
    { cout << "Name ctor: "; print(); cout << endl; }
    ~Name() { cout << "Name dtor: "; print(); cout << endl; }
    void print()
    { firstName_.print(); cout << " "; lastName_.print(); }
};
class Address {
    unsigned int houseNo_;
    String street_, city_, pin_;
public:
    Address(unsigned int hn, const char* sn, const char* cn, const char* pin) :
        houseNo_(hn), street_(sn), city_(cn), pin_(pin)
    { cout << "Address ctor: "; print(); cout << endl; }
    ~Address() { cout << "Address dtor: "; print(); cout << endl; }
    void print() {
        cout << houseNo_ << " ";
        street_.print(); cout << " ";
        city_.print(); cout << " ";
        pin_.print();
    }
};
```

Program 14.07: CreditCard Class


```
class CreditCard { typedef unsigned int UINT;
    char cardNumber_[17]; // 16-digit (character) card number as C-string
    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)
        { strcpy(cardNumber_, cNumber); cout << "CC ctor: "; print(); cout << endl; }
    ~CreditCard() { cout << "CC dtor: "; print(); cout << endl; }
    void print() {
        cout << cardNumber_ << " ";
        holder_.print(); cout << " ";
        addr_.print(); cout << " ";
        issueDate_.print(); cout << " ";
        expiryDate_.print(); cout << " ";
        cout << cvv_;
    }
};
int main() {
    CreditCard cc("5321711934640027", "Sharlock", "Holmes",
                  221, "Baker Street", "London", "NW1 6XE", 7, 2014, 12, 2016, 811);
    cout << endl; cc.print(); cout << endl << endl;;
    return 0;
}
```

Program 14.07: CreditCard Class: Lifetime Chart

Construction of Objects

```
        String: Sharlock
        String: Holmes
    Name: Sharlock Holmes
        String: Baker Street
        String: London
        String: NW1 6XE
    Address: 221 Baker Street London NW1 6XE
    Date: 1/Jul/2014
    Date: 1/Dec/2016
CC: 5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
```

```
typedef unsigned int UINT;
class CreditCard { char cardNumber_[17];
    Name holder_;
    Address addr_;
    Date issueDate_, expiryDate_;
    UINT cvv_; };
class Name { String firstName_, lastName_; };
class Address { unsigned int houseNo_;
    String street_, city_, pin_; };
class Date { enum Month;
    UINT date_; Month month_; UINT year_; };
```

Use of Object

```
5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
```

Destruction of Objects

```
~CC: 5321711934640027 Sharlock Holmes 221 Baker Street London NW1 6XE 1/Jul/2014 1/Dec/2016 811
    ~Date: 1/Dec/2016
    ~Date: 1/Jul/2014
    ~Address: 221 Baker Street London NW1 6XE
        ~String: NW1 6XE
        ~String: London
        ~String: Baker Street
    ~Name: Sharlock Holmes
        ~String: Holmes
        ~String: Sharlock
```

Copy Constructor

- We know:

```
Complex c1 = {4.2, 5.9}; // or 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 take 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:
    Complex(double re, double im) : re_(re), im_(im)   // Constructor
    { cout << "Complex ctor: "; print(); }
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) // Copy Constructor
    { cout << "Complex copy ctor: "; print(); }
    ~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();
    return 0;
}
-----
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)
  - *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)
  - *Call-by-value*: Make a copy (clone) of the actual parameter as a formal parameter. This needs a **Copy Constructor**
  - *Return-by-value*: Make a copy (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

    return 0;
}
-----
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
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 loop as the call to copy constructor itself needs to make copy for the Call-by-Value mechanism.

Program 14.10: Point and Rect Classes: Default, Copy and Overloaded Constructors


```
#include <iostream>
using namespace std;
class Point { int x_; int y_; public:
    Point(int x, int y) : x_(x), y_(y)                          // Constructor (Ctor)
    { cout << "Point ctor: "; print(); cout << endl; }
    Point() : x_(0), y_(0)                                      // Default Constructor (DCtor)
    { cout << "Point ctor: "; print(); cout << endl; }
    Point(const Point& p) : x_(p.x_), y_(p.y_)                  // Copy Constructor (CCtor)
    { cout << "Point cctor: "; print(); cout << endl; }
    ~Point() { cout << "Point dtor: "; print(); cout << endl; }  // Destructor (Dtor)
    void print() { cout << "(" << x_ << ", " << y_ << ")"; }
};
class Rect { Point TL_; Point BR_; public:
    Rect(int tlx, int tly, int brx, int bry):
        TL_(tlx, tly), BR_(brx, bry)                        // Ctor - Uses Ctor for Point
    { cout << "Rect ctor: "; print(); cout << endl; }
    Rect(const Point& p_tl, const Point& p_br): TL_(p_tl), BR_(p_br) // Ctor
    { cout << "Rect ctor: "; print(); cout << endl; }               // Uses CCtor for Point
    Rect(const Point& p_tl, int brx, int bry): TL_(p_tl), BR_(brx, bry) // Ctor
    { cout << "Rect ctor: "; print(); cout << endl; }                  // CCtor for Point
    Rect() { cout << "Rect ctor: "; print(); cout << endl; }               // Default Ctor
    Rect(const Rect& r): TL_(r.TL_), BR_(r.BR_)                            // Copy Ctor
    { cout << "Rect cctor: "; print(); cout << endl; }
    ~Rect() { cout << "Rect dtor: "; print(); cout << endl; }               // Dtor
    void print() { cout << "["; TL_.print(); cout << " "; BR_.print(); cout << "]"; }
};
```

- When parameter `(tlx, tly)` is set to `TL_` by `TL_(tlx, tly)`: parameterized **Ctor** of `Point` is involved
- When parameter `p_tl` is set to `TL_` by `TL_(p_tl)`: **CCtor** of `Point` is involved
- When `TL_` is set by default in **DCtor** of `Rect`: **DCtor** of `Point` is involved
- When member `r.TL_` is set to `TL_` by `TL_(r.TL_)` in **CCtor** of `Rect`: **CCtor** of `Point` is involved

Program 14.10: Rect Class: Trace of Object Lifetimes

Code	Output	Lifetime	Remarks
`int main() {`			
`  Rect r1(0, 2, 5, 7);` `  //Rect(int, int, int, int)`	Point ctor: (0, 2) Point ctor: (5, 7) Rect ctor: [(0, 2) (5, 7)]	Point r1.TL_ Point r1.BR_ Rect r1	

`  Rect r2(Point(3, 5),`
`  Point(6, 9));`
`  //Rect(Point&, Point&)`
	Point ctor: (6, 9)
	Point ctor: (3, 5)
	Point cctor: (3, 5)
	Point cctor: (6, 9)
	Rect ctor: [(3, 5) (6, 9)]
	Point dtor: (3, 5)
	Point dtor: (6, 9)
		Point t1
		Point t2
		r2.TL_ = t2
		r2.BR_ = t1
		Rect r2
		~Point t2
		~Point t1
			Second parameter
			First parameter
			Copy to r2.TL_
			Copy to r2.BR_
			
			First parameter
			Second parameter

`  Rect r3(Point(2, 2), 6, 4);` `  //Rect(Point&, int, int)`	Point ctor: (2, 2) Point cctor: (2, 2) Point ctor: (6, 4) Rect ctor: [(2, 2) (6, 4)] Point dtor: (2, 2)	Point t3 r3.TL_ = t3 Point r3.BR_ Rect r3 ~Point t3	First parameter Copy to r3.TL_ First parameter
`  Rect r4;` `  //Rect()`	Point ctor: (0, 0) Point ctor: (0, 0) Rect ctor: [(0, 0) (0, 0)]	Point r4.TL_ Point r4.BR_ Rect r4	
`  return 0;` `}`	Rect dtor: [(0, 0) (0, 0)] Point dtor: (0, 0) Point dtor: (0, 0)	~Rect r4 ~Point r4.BR_ ~Point r4.TL_	
	Rect dtor: [(2, 2) (6, 4)] Point dtor: (6, 4) Point dtor: (2, 2)	~Rect r3 ~Point r3.BR_ ~Point r3.TL_	
	Rect dtor: [(3, 5) (6, 9)] Point dtor: (6, 9) Point dtor: (3, 5)	~Rect r2 ~Point r2.BR_ ~Point r2.TL_	
	Rect dtor: [(0, 2) (5, 7)] Point dtor: (5, 7) Point dtor: (0, 2)	~Rect r1 ~Point r1.BR_ ~Point r1.TL_	

Free Copy Constructor

- If no copy constructor is provided by the user, the compiler supplies a *free* copy constructor
- Compiler-provided copy constructor, understandably, cannot initialize the object to proper values. It has no code in its body. It performs a *bit-copy*

Program 14.09: Complex: Free Copy Constructor


```
#include <iostream>
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) { 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

    return 0;
}
```

User-defined CCtor

```
ctor: |4.2+j5.3| = 6.7624
copy ctor: |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
```

Free CCtor

```
ctor: |4.2+j5.3| = 6.7624
\\ No message from free CCtor
Display: |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-copy - hence there is no message
- Correct in this case as members are of built-in type

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
    ~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();
}
int main() {
    String s = "Partha";
    s.print();
    strToUpper(s);
    s.print();
    return 0;
}
---
(Partha: 6)
strToUpper: (PARTHA: 6)
(Partha: 6)
```

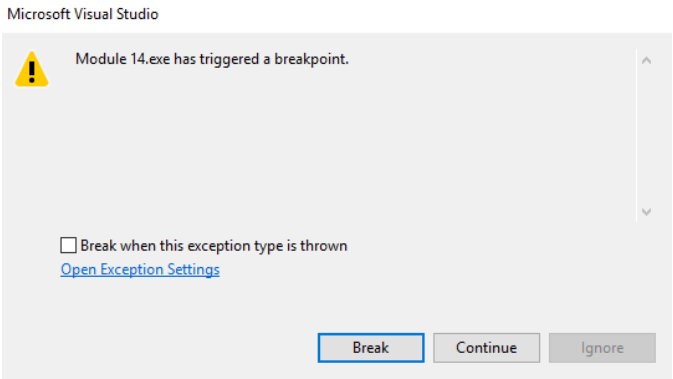
- User has provided copy constructor. So Compiler does not provide free copy constructor
- When actual parameter `s` is copied to formal parameter `a`, space is allocated for `a.str_` and then it is copied from `s.str_`. On exit from `strToUpper`, `a` is destructed and `a.str_` is deallocated. But in `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

Program 14.11: String: Free Copy Constructor


```
#include <iostream>
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
    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(); }
int main() {
    String s = "Partha";

    s.print();
    strToUpper(s);
    s.print();

    return 0;
}
```



User-defined CCtor

```
(Partha: 6)
strToUpper: (PARTHA: 6)
(Partha: 6)
```

Free CCtor

```
(Partha: 6)
strToUpper: (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 corrupted. Program crashes
- **Shallow Copy**: With bit-copy, only the pointer is copied - not the pointed object. This may be risky

Program 14.12: Complex: Copy Assignment


```
#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(); }
    Complex(const Complex& c) : re_(c.re_), im_(c.im_) { cout << "cctor: "; print(); }
    ~Complex() { cout << "dtor: "; print(); }
    Complex& operator=(const Complex& c)  // Copy Assignment Operator
    { re_ = c.re_; im_ = c.im_; cout << "copy: "; print(); return *this; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c1(4.2, 5.3), c2(7.9, 8.5); // Constructor - Complex(double, double)
    Complex c3(c2);                     // Constructor - Complex(const Complex& c)

    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
    return 0;
}
```

```
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 14.13: String: Copy Assignment


```
#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
    ~String() { free(str_); } // dtor
    String& operator=(const String& s) {
        free(str_);             // Release existing memory
        str_ = strdup(s.str_); // Perform deep copy
        len_ = s.len_;
        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();
    return 0;
}
---
(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 14.13: String: Self Copy


```
#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
    ~String() { free(str_); } // dtor
    String& operator=(const String& s) {
        free(str_);             // Release existing memory
        str_ = strdup(s.str_); // Perform deep copy
        len_ = s.len_;
        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();
    return 0;
}
---
(Football: 8)
(Cricket: 7)
(????????: 8) // Garbage is printed
```

- For self-copy `str_` and `s.str_` are the same pointers
- 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 protected**

Program 14.14: String: Self Copy – Safe


```
#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
    ~String() { free(str_); } // dtor
    String& operator=(const String& s) {
        if (this != &s) {
            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();
    return 0;
}
---
(Football: 8)
(Cricket: 7)
(Football: 8)
```

- Check for self-copy (`this != &s`)
- 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) {
        // Release resources held by *this
        // Copy members of s to members of *this
    }
    return *this;
}
```

- 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);
```

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
- **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

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655