Module 13: Programming in C++

Constructors, Destructors & Object Lifetime

Intructors: Abir Das and Sourangshu Bhattacharya

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

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

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

by **Prof. Partha Pratim Das**

Module Objectives

- Understand Object Construction (Initialization)
- Understand Object Destruction (De-Initialization)
- Understand Object Lifetime

Module Outline

Program 13.01/02: Stack: Initialization

Public Data

```
#include <iostream>
using namespace std;
class Stack { public: // VULNERABLE DATA
    char data_[10]; int top_;
public:

    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s; s.top_ = -1; // Exposed initialization

    for (int i = 0; i < 5; ++i) s.push(str[i]);
    // s.top_ = 2; // RISK - CORRUPTS STACK
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
```

- Spills data structure codes into application
- `public` data reveals the *internals*
- To switch container, application needs to change
- Application may corrupt the stack!

Private Data

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public:
    void init() { top_ = -1; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s;  s.init(); // Clean initialization

    for (int i = 0; i < 5; ++i) s.push(str[i]);
    // s.top_ = 2; // Compile error - SAFE
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
```

- **No code in application, but `init()` to be called**
- `private` data protects the *internals*
- Switching container is seamless
- Application cannot corrupt the stack

Program 13.02/03: Stack: Initialization

Using init()

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public: void init() { top_ = -1; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s;  s.init(); // Clean initialization

    for (int i = 0; i < 5; ++i) s.push(str[i]);
    // s.top_ = 2; // Compile error - SAFE
    while(!s.empty()) { cout << s.top(); s.pop(); }
}
```

- `init()` **serves no visible purpose – application may forget to call**
- If application misses to call `init()`, we have a corrupt stack

Using Constructor

```
#include <iostream>
using namespace std;
class Stack { private: // PROTECTED DATA
    char data_[10]; int top_;
public: Stack() : top_(-1) { } // Initialization
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call

    for (int i = 0; i < 5; ++i) s.push(str[i]);

    while(!s.empty()) { cout << s.top(); s.pop(); }
}
```

- **Can initialization be made a part of instantiation?**
- Yes. **Constructor** is implicitly called at instantiation as set by the compiler

Program 13.04/05: Stack: Constructor

Automatic Array

```
#include <iostream>
using namespace std;
class Stack { private:
    char data_[10]; int top_; // Automatic
public: Stack(); // Constructor
    // More Stack methods
};
Stack::Stack(): // Initialization List
    top_(-1) { cout << "Stack::Stack()" << endl;
}
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call
    for (int i=0; i<5; ++i) s.push(str[i]);
    while(!s.empty()) { cout << s.top(); s.pop(); }
}
-----
Stack::Stack()
EDCBA
```

- top_ initialized to -1 in initialization list
- data_[10] initialized by default (*automatic*)
- Stack::Stack() called *automatically* when control passes Stack s; – Guarantees initialization

Dynamic Array

```
#include <iostream>
using namespace std;
class Stack { private:
    char *data_; int top_; // Dynamic
public: Stack(); // Constructor
    // More Stack methods
};
Stack::Stack(): data_(new char[10]), // Init List
    top_(-1) { cout << "Stack::Stack()" << endl;
}
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call
    for (int i=0; i<5; ++i) s.push(str[i]);
    while(!s.empty()) { cout << s.top(); s.pop(); }
}
-----
Stack::Stack()
EDCBA
```

- top_ initialized to -1 in initialization list
- data_ initialized to new char[10] in init list

Constructor: Contrasting with Member Functions

Constructor

- Is a static member function without `this` pointer – but gets the pointer to the memory where the object is constructed
- Name is same as the name of the class

```
class Stack { public: Stack(); };
```

- Has no return type - not even `void`

```
Stack::Stack(); // Not even void
```

- Does not return anything. Has no `return` statement

```
Stack::Stack(): top_(-1)
    { } // Returns implicitly
```

- Initializer list to initialize the data members

```
Stack::Stack(): // Initializer list
    data_(new char[10]), // Init data_
    top_(-1)             // Init top_
    { }
```

- Implicit call by instantiation / `operator new`

```
Stack s; // Calls Stack::Stack()
```

- May be `public` or `private`
- May have any number of parameters
- Can be overloaded

Member Function

- Has implicit `this` pointer
- Any name different from name of class

```
class Stack { public: int empty(); };
```

- Must have a return type - may be `void`

```
int Stack::empty();
```

- Must have at least one `return` statement

```
int Stack::empty() { return (top_ == -1); }
void pop()
{ --top_; } // Implicit return for void
```

- Not applicable
- Explicit call by the object

```
s.empty(); // Calls Stack::empty(&s)
```

- May be `public` or `private`
- May have any number of parameters
- Can be overloaded

Program 13.06: Complex: Parameterized Constructor


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

class Complex { private: double re_, im_;
public:
    Complex(double re, double im): // Constructor with parameters
        re_(re), im_(im)           //  Initializer List: Parameters to initialize data members
    { }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() {
        cout << "|" << re_ << "+j" << im_ << "| = ";
        cout << norm() << endl;
    }
};
int main() { Complex c(4.2, 5.3), // Complex::Complex(4.2, 5.3)
                     d(1.6, 2.9); // Complex::Complex(1.6, 2.9)

    c.print();
    d.print();
}
-----
|4.2+j5.3| = 6.7624
|1.6+j2.9| = 3.3121
```

Program 13.07: Complex: Constructor with default parameters


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

class Complex { private: double re_, im_; public:
    Complex(double re = 0.0, double im = 0.0) : // Constructor with default parameters
        re_(re), im_(im)                        //  Initializer List: Parameters to initialize data members
    { }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c1(4.2, 5.3), // Complex::Complex(4.2, 5.3) --  both parameters explicit
            c2(4.2),      // Complex::Complex(4.2, 0.0) --  second parameter default
            c3;           // Complex::Complex(0.0, 0.0) --  both parameters default

    c1.print();
    c2.print();
    c3.print();
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Program 13.08: Stack: Constructor with default parameters


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

class Stack { private: char *data_; int top_;
public: Stack(size_t = 10); // Size of data_ defaulted
    ~Stack() { delete data_[]; }
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
Stack::Stack(size_t s) : data_(new char[s]), top_(-1) // Array of size s allocated and set to data_
{ cout << "Stack created with max size = " << s << endl; }

int main() { char str[] = "ABCDE"; int len = strlen(str);
    Stack s(len); // Create a stack large enough for the problem

    for (int i = 0; i < len; ++i) s.push(str[i]);
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
-----
Stack created with max size = 5
EDCBA
```

Program 13.09: Complex: Overloaded Constructors


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

class Complex { private: double re_, im_; public:
    Complex(double re, double im): re_(re), im_(im) { } // Two parameters
    Complex(double re): re_(re), im_(0.0) { }           // One parameter
    Complex(): re_(0.0), im_(0.0) { }                   // No parameter

    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c1(4.2, 5.3), // Complex::Complex(double, double)
            c2(4.2),      // Complex::Complex(double)
            c3;           // Complex::Complex()

    c1.print();
    c2.print();
    c3.print();
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Program 13.10: Rect: Overloaded Constructors


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

class Pt { public: int x_, y_; Pt(int x, int y): x_(x), y_(y) { } }; // A Point
class Rect { Pt LT_, RB_; public:
    Rect(Pt lt, Pt rb):
        LT_(lt), RB_(rb) { }                    //  Cons 1: Points Left-Top lt and Right-Bottom rb
    Rect(Pt lt, int h, int w):
        LT_(lt), RB_(Pt(lt.x_+w, lt.y_+h)) { } //  Cons 2: Point Left-Top lt, height h & width w
    Rect(int h, int w):
        LT_(Pt(0, 0)), RB_(Pt(w, h)) { }        //  Cons 3: height h, width w & Point origin as Left-Top
    int area() { return (RB_.x_-LT_.x_) * (RB_.y_-LT_.y_); }
};
int main() { Pt p1(2, 5), p2(8, 10);
    Rect r1(p1, p2),        //  Cons 1: Rect::Rect(Pt, Pt)
         r2(p1, 5, 6),      //  Cons 2: Rect::Rect(Pt, int, int)
         r3(5, 6);          //  Cons 3: Rect::Rect(int, int)
    cout << "Area of r1 = " << r1.area() << endl;
    cout << "Area of r2 = " << r2.area() << endl;
    cout << "Area of r3 = " << r3.area() << endl;
}
-----
Area of r1 = 30
Area of r2 = 30
Area of r3 = 30
```

Program 13.11/12: Stack: Destructor

Resource Release by User

```
#include <iostream>
using namespace std;
class Stack { char *data_; int top_; // Dynamic
public: Stack(): data_(new char[10]), top_(-1)
    { cout << "Stack() called\n"; } // Constructor
    void de_init() { delete [] data_; }


    // More Stack methods
};
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call
    // Reverse string using Stack
    s.de_init();
}
-----
Stack() called
EDCBA
```

- data_ **leaks unless released within the scope of** s
- When to call de_init()? User may forget to call

Automatic Resource Release

```
#include <iostream>
using namespace std;
class Stack { char *data_; int top_; // Dynamic
public: Stack(): data_(new char[10]), top_(-1)
    { cout << "Stack() called\n"; } // Constructor
    ~Stack() { cout << "\n~Stack() called\n";
        delete [] data_; // Destructor
    }
    // More Stack methods
};
int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call
    // Reverse string using Stack

} // De-Init by automatic Stack::~Stack() call
-----
Stack() called
EDCBA
~Stack() called
```

- **Can de-initialization be a part of scope rules?**
- Yes. **Destructor** is implicitly called at end of scope

Destructor: Contrasting with Member Functions

Destructor

- Has implicit `this` pointer
- Name is `~` followed by the name of the class

```
class Stack { public:
    ~Stack();
};
```

- Has no return type - not even `void`

```
Stack::~Stack(); // Not even void
```

- Does not return anything. Has no `return` statement

```
Stack::~Stack()
    { } // Returns implicitly
```

- Implicitly called at end of scope
- May be `public` or `private`
- No parameter is allowed - unique for the class
- Cannot be overloaded

Member Function

- Has implicit `this` pointer
- Any name different from name of class

```
class Stack { public:
    int empty();
};
```

- Must have a return type - may be `void`

```
int Stack::empty();
```

- Must have at least one `return` statement

```
int Stack::empty()
{ return (top_ == -1); }
```

- Explicit call by the object

```
s.empty(); // Calls Stack::empty(&s)
```

- May be `public` or `private`
- May have any number of parameters
- Can be overloaded

Default Constructor / Destructor

- **Constructor**
  - A constructor with no parameter is called a *Default Constructor*
  - If no constructor is provided by the user, the compiler supplies a *free* default constructor
  - Compiler-provided (*free default*) constructor, understandably, cannot initialize the object to proper values. It has no code in its body
  - Default constructors (*free or user-provided*) are required to define arrays of objects
- **Destructor**
  - If no destructor is provided by the user, the compiler supplies a *free* default destructor
  - Compiler-provided (*free default*) destructor has no code in its body

Program 13.13: Complex: Default Constructor: User Defined


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

class Complex { private: double re_, im_; public:
    Complex(): re_(0.0), im_(0.0) // Default Constructor having no parameter
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Destructor
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
    void set(double re, double im) { re_ = re; im_ = im; }
};
int main() { Complex c; // Default constructor -- user provided
    c.print();       // Print initial values
    c.set(4.2, 5.3); // Set components
    c.print();       // Print values set
} // Destuctor
-----
Ctor: (0, 0)
|0+j0| = 0
|4.2+j5.3| = 6.7624
Dtor: (4.2, 5.3)
```

- User has provided a default constructor

Program 13.14: Complex: Default Constructor: Free


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

class Complex { private: double re_, im_; // private data
public: // No constructor given be user. So compiler provides a free default one
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
    void set(double re, double im) { re_ = re; im_ = im; }
};
int main() { Complex c; // Free constructor from compiler. Initialization with garbage

    c.print();       // Print initial value - garbage
    c.set(4.2, 5.3); // Set proper components
    c.print();       // Print values set
} // Free destuctor from compiler
-----
|-9.25596e+061+j-9.25596e+061| = 1.30899e+062
|4.2+j5.3| = 6.7624
```

- User has provided no constructor / destructor
- Compiler provides default (free) constructor / destructor
- Compiler-provided constructor does nothing – components have garbage values
- Compiler-provided destructor does nothing