Module 13: Programming in C++

Constructors, Destructors & Object Lifetime

Sourangshu Bhattacharya

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

sourangshu@cse.iitkgp.ac.in

Slides taken from NPTEL course on Programming in C++

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

- Constructors
  - Parameterized
  - Overloaded
- Destructor
- Default Constructor
- Object Lifetime
  - Automatic
  - Array
  - Dynamic

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();
    }
    return 0;
}
```

- 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();
    }
    return 0;
}
```

- **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();
    }
    return 0;
}
```

- **`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();
    }
    return 0;
}
```

- **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() called" << 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();
    }
    return 0;
}
-----
Stack::Stack() called
EDCBA
```

- top_ initialized to -1 in initialization list
- data_[10] initialized by default (automatic)

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
                top_(-1) {           // List
    cout << "Stack::Stack() called" << 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();
    }
    return 0;
}
-----
Stack::Stack() called
EDCBA
```

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

- Stack::Stack() called automatically when control passes Stack s; – Guarantees initialization

Constructor: Contrasting with Member Functions

Constructor	Member Function
Is a member function with `this` pointer	Has implicit `this` pointer

Name is same as the name of the class
`class Stack { public:`
`    Stack();`
`};`
	Any name different from name of class
	`class Stack { public:`
	`    int empty();`
	`};`

Has no return type `    Stack::Stack(); // Not even void`	Must have a return type `int Stack::empty();`

No `return`; hence has no `return` statement
`Stack::Stack(): top_(-1)`
`    { } // Returns implicitly`
	Must have at least one `return` statement
	`int Stack::empty()`
	`{ return (top_ == -1); }`
	`void pop()`
	`{ --top_; } // Implicit return`

Initializer list to initialize the data members
`Stack::Stack(): // Initializer list`
`    data_(new char[10]), // Init data_`
`    top_(-1)              // Init top_`
`    { }`
	Not applicable

Implicit call by instantiation / operator `new` `Stack s; // Calls Stack::Stack()`	Explicit call by the object `s.empty(); // Calls Stack::empty(&s)`
May have any number of parameters	May have any number of parameters
Can be overloaded	Can be overloaded

Program 13.06: Complex: Parameterized Constructor

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

class Complex { private: double re_, im_;
public:
    Complex(double re, double im): // Ctor w/ params
        re_(re), im_(im)           // Params used to initialize
    {}
    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();

    return 0;
}
-----
|4.2+j5.3| = 6.7624
|1.6+j2.9| = 3.3121
```

Program 13.07: Complex: Constructor with default parameters

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

class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0) : // Ctor w/ default params
        re_(re), im_(im)                        // Params used to initialize
    {}
    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();

    return 0;
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Program 13.08: Stack: Constructor with default parameters

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

class Stack { private: char *data_; int top_;
public:
    Stack(size_t = 10); // Size of data_ defaulted

    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]), // Array of size s allocated
                         top_(-1)
{ cout << "Stack created with max size = " << s << endl; }

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

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

Program 13.09: Complex: Overloaded Constructors

```
#include <iostream>
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(4.2, 5.3)
            c2(4.2),      // Complex::Complex(4.2)
            c3;           // Complex::Complex()

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

    return 0;
}
-----
|4.2+j5.3| = 6.7624
|4.2+j0| = 4.2
|0+j0| = 0
```

Program 13.10/11: Stack: Destructor

Automatic Array

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




int main() { char str[10] = "ABCDE";
    Stack s; // Init by Stack::Stack() call

    // Reverse string using Stack
    s.de_init();
    return 0;
}
-----
Stack::Stack() called
EDCBA
```

- **Dynamically allocated `data_` leaks unless released before program loses scope of `s`**
- Application may forget to call `de_init()`; Also, when should `de_init()` be called?

Dynamic Array

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

    // Reverse string using Stack

    return 0;
} // De-Init by Stack::~Stack() call
-----
Stack::Stack() called
EDCBA
Stack::~Stack() called
```

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

Destructor: Contrasting with Member Functions

Destructor

- Is a member function with `this` pointer
- Name is ~ followed by the name of the class

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

- Has no return type

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

- No return; hence has no `return` statement

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

- Implicitly called at end of scope or by `operator delete`. May be called explicitly by the object (rare)

```
{
    Stack s;
    // ...
} // Calls Stack::~Stack(&s)
```

- 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

```
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 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 (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 (default) destructor has no code in its body

Program 13.12: Complex: Default Constructor

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

class Complex {
private: double re_, im_; // private data
public:
    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

    return 0;
} // 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

Program 13.13: Complex: Default Constructor

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

class Complex { private: double re_, im_;
public:
    Complex(): re_(0.0), im_(0.0) // Default Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
    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

    return 0;
} // Destuctor
-----
Ctor: (0, 0)
|0+j0| = 0
|4.2+j5.3| = 6.7624
Dtor: (4.2, 5.3)
```

- User has provided a default constructor

Object Lifetime: When is an Object ready? How long can it be used?

Application

```
void MyFunc() // E1: Allocation of c on Stack
{
        ...
    Complex c; // E2: Ctor called
        ...


    c.norm(); // E5: Use
        ...




    return; // E7: Dtor called
} // E9: De-Allocation of c from Stack
```

Class Code

```
Complex::Complex(double re = 0.0, // Ctor
                 double im = 0.0):
    re_(re), im_(im) // E3: Initialization
{ // E4: Object Lifetime STARTS
    cout << "Ctor:" << endl;
}

double Complex::norm() // E6
{ return sqrt(re_*re_ + im_*im_); }


Complex::~Complex() // Dtor
{
    cout << "Dtor:" << endl;
} // E8: Object Lifetime ENDS
```

Event Sequence and Object Lifetime

E1	MyFunc called. Stackframe allocated. c is a part of Stackframe
E2	Control to pass `Complex c`. Ctor `Complex::Complex(&c)` called with the address of c on the frame
E3	Control on Initializer list of Complex::Complex(). Data members initialized (constructed)
E4	Object Lifetime STARTS for c. Control reaches the start of the body of Ctor. Ctor executes
E5	Control at `c.norm()`. `Complex::norm(&c)` called. Object is being used
E6	`Complex::norm()` executes
E7	Control to pass `return`. Dtor `Complex::~Complex(&c)` called
E8	Dtor executes. Control reaches the end of the body of Dtor. Object Lifetime ENDS for c
E9	return executes. Stackframe including c de-allocated. Control returns to caller

Object Lifetime

- **Execution Stages**
  - Memory Allocation and Binding
  - Constructor Call and Execution
  - Object Use
  - Destructor Call and Execution
  - Memory De-Allocation and De-Binding
- **Object Lifetime**
  - Starts with execution of Constructor Body
    - Must *follow* Memory Allocation
    - As soon as Initialization ends and control enters Constructor Body
  - Ends with execution of Destructor Body
    - As soon as control leaves Destructor Body
    - Must *precede* Memory De-allocation
  - For Objects of *Built-in / Pre-Defined Types*
    - No Explicit Constructor / Destructor
    - Lifetime spans from object definition to end of scope

Program 13.14: Complex: Object Lifetime: Automatic

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }

    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }

    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c(4.2, 5.3), d(2.4); // Complex::Complex() called -- c, then d -- objects ready

    c.print();                   // Using objects
    d.print();

    return 0;
}                                // Scope over, objects no more available.
                                 // Complex::~Complex() called -- d then c
                                 // Note the reverse order!
-----
Ctor: (4.2, 5.3)
Ctor: (2.4, 0)
|4.2+j5.3| = 6.7624
|2.4+j0| = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

Program 13.15: Complex: Object Lifetime: Automatic: Array of Objects

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0) : re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }

    void opComplex(double i) { re_ += i; im_ += i; } // Some operation with Complex

    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() {
    Complex c[3]; // Default ctor Complex::Complex() called thrice -- c[0], c[1], c[2]

    for (int i = 0; i < 3; ++i) { c[i].opComplex(i); c[i].print(); } // Use array
    return 0;
}   // Scope over. Complex::~Complex() called thrice -- c[2], c[1], c[0] -- reverse order
-----
Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (0, 0)
|0+j0| = 0
|1+j1| = 1.41421
|2+j2| = 2.82843
Dtor: (2, 2)
Dtor: (1, 1)
Dtor: (0, 0)
```

Program 13.16: Complex: Object Lifetime: Static

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

class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};

Complex c(4.2, 5.3); // Static (global) object
                     // Constructed before main starts
                     // Destructed after main ends
int main() {
    cout << "main() Starts" << endl;
    Complex d(2.4);  // Ctor for d

    c.print(); // Use static object
    d.print(); // Use local object

    return 0;
} // Dtor for d

// Dtor for c
```

```
----- OUTPUT -----
Ctor: (4.2, 5.3)
main() Starts
Ctor: (2.4, 0)
|4.2+j5.3| = 6.7624
|2.4+j0| = 2.4
Dtor: (2.4, 0)
Dtor: (4.2, 5.3)
```

Program 13.17: Complex: Object Lifetime: Dynamic

```
#include <iostream>
using namespace std;
class Complex { private: double re_, im_;
public:
    Complex(double re = 0.0, double im = 0.0): re_(re), im_(im) // Ctor
    { cout << "Ctor: (" << re_ << ", " << im_ << ")" << endl; }
    ~Complex() // Dtor
    { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; }
    double norm() { return sqrt(re_*re_ + im_*im_); }
    void print() { cout << "|" << re_ << "+j" << im_ << "| = " << norm() << endl; }
};
int main() { unsigned char buf[100];           // Buffer for placement of objects
    Complex* pc = new Complex(4.2, 5.3);       // operator new: allocates memory, calls Ctor
    Complex* pd = new Complex[2];              // operator new []: allocates memory,
                                               //     calls default Ctor twice
    Complex* pe = new (buf) Complex(2.6, 3.9); // operator placement new: only calls Ctor
                                               //     no allocation of memory, uses buf
    // Use objects
    pc->print();
    pd[0].print(); pd[1].print();
    pe->print();

    // Release of objects - can be done in any order
    delete pc;      // delete: calls Dtor, release memory
    delete [] pd;   // delete[]: calls 2 Dtor’s, release mem
    pe->~Complex(); // No delete: explicit call to Dtor
                    // Use with extreme care
    return 0;
}
```

```
----- OUTPUT -----
Ctor: (4.2, 5.3)
Ctor: (0, 0)
Ctor: (0, 0)
Ctor: (2.6, 3.9)
|4.2+j5.3| = 6.7624
|0+j0| = 0
|0+j0| = 0
|2.6+j3.9| = 4.68722
Dtor: (4.2, 5.3)
Dtor: (0, 0)
Dtor: (0, 0)
Dtor: (2.6, 3.9)
```

Module Summary

- Objects are initialized by Constructors
- Constructors can be Parameterized and can be Overloaded
- Default Constructor does not take any parameter. It is necessary for defining arrays of objects
- Objects are cleaned-up by Destructors. Destructor for a class is unique
- Compiler provides *free* Default Constructor and Destructor, if not provides by the program
- Objects have a well-defined lifetime spanning from execution of the beginning of the body of a constructor to the execution till the end of the body of the destructor
- Memory for an object must be available before its construction and can be released only after its destruction