Programming in Modern C++

Module M13: Constructors, Destructors & Object Lifetime

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Access Specifiers help to control visibility of data members and methods of a class
- The private access specifier can be used to hide information about the implementation details of the data members and methods
- Get, Set methods are defined to provide an interface to use and access the data members

Module Objectives

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

Module Outline

Constructor

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	Member Function
• Is a static member function without `this` pointer – but gets the pointer to the memory where the object is constructed	• 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 - not even `void` `Stack::Stack(); // Not even void`	• Must have a return type - may be `void` `int Stack::empty();`

• Does not return anything. 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 for void`

• 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 be `public` or `private`	• May be `public` or `private`
• 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>
#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
```

Destructor

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	Member Function
• Has implicit `this` pointer	• Has implicit `this` pointer

• Name is ~ followed by 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 - not even `void` `Stack::~Stack(); // Not even void`	• Must have a return type - may be `void` `int Stack::empty();`

• Does not return anything. Has no `return` statement
`Stack::~Stack()`
`  { } // Returns implicitly`
	• Must have at least one `return` statement
	`int Stack::empty()`
	`{ return (top_ == -1); }`

• Implicitly called at end of scope or by `operator delete`. May be called explicitly by the object (rare)
`{`
`  Stack s;`
`  // ...`
`} // Calls Stack::~Stack(&s) implicitly`
	• Explicit call by the object
	`s.empty(); // Calls Stack::empty(&s)`

• May be `public` or `private`	• May be `public` or `private`
• No parameter is allowed - unique for the class	• May have any number of parameters
• Cannot be overloaded	• Can be overloaded

Default Constructor

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

Object Lifetime

Sources:

- Storage class specifiers, cppreference.org

Object Lifetime

- In OOP, the **object lifetime** (or **life cycle**) of an object is the time between an *object's creation* and its *destruction*
- Rules for object lifetime vary significantly:
  - *Between languages*
  - in some cases *between implementations* of a given language, and
  - lifetime of a particular object may vary *from one run of the program to another*
- **Context C++**: *Object Llifetime* coincides with **Variable Lifetime** (the extent of a variable when in a program's execution the variable has a *meaningful* value) of a variable with that object as value (both for static variables and automatic variables). However, in general, object lifetime may not be tied to the lifetime of any one variable
- **Context Java / Python**: In OO languages that use *garbage collection (GC)*, objects are allocated on the heap
  - object lifetime is not determined by the lifetime of a given variable
  - the value of a variable holding an object actually corresponds to a reference to the object, not the object itself, and
  - destruction of the variable just destroys the reference, not the underlying object

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: Constructor called
        ...

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

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

Class Code

```
Complex::Complex(double re = 0.0, // Constructor
                 double im = 0.0):
    re_(re), im_(im) //  E3: Initialization
{ //  E4: Object Lifetime STARTS with initialization
    cout << "Ctor:" << endl;
}
double Complex::norm() //  E6 norm executes
{ return sqrt(re_*re_ + im_*im_); }
Complex::~Complex() { cout << "Dtor:" << endl;
} //  E8: Object Lifetime ENDS with destructor
```

Event Sequence and Object Lifetime

E1	`MyFunc` called. Stackframe allocated. `c` is a part of Stackframe
E2	Control to pass to `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 Constructor. Constructor executes
E5	Control at `c.norm()`. `Complex::norm(&c)` called. Object is being used
E6	`Complex::norm()` executes
E7	Control to pass `return` in `MyFunc`. Desturctor `Complex::~Complex(&c)` called
E8	Destructor executes. Control reaches the end of the body of Destructor. 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 13.15: Complex: Object Lifetime: Automatic


```
#include <iostream>
#include <cmath>
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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor

    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();
} // Scope over, objects no more available. Complex::~Complex() called -- d then c in 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 13.16: Complex: Object Lifetime: Automatic: Array of Objects


```
#include <iostream>
#include <cmath>
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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor
    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
} // Scope over. Complex::~Complex() called thrice -- c[2], c[1], c[0] in the 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 13.17: Complex: Object Lifetime: Static


```
#include <iostream>
#include <cmath>
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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor

    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 c
                     // 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
} // 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 13.18: Complex: Object Lifetime: Dynamic


```
#include <iostream>
#include <cmath>
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() { cout << "Dtor: (" << re_ << ", " << im_ << ")" << endl; } // Dtor
    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);      // new: allocates memory, calls Ctor
    Complex* pd = new Complex[2];             // new []: allocates memory
                                              // calls default Ctor twice
    Complex* pe = new (buf) Complex(2.6, 3.9); // placement new: only calls Ctor
                                              // No alloc. 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 memory
    pe->~Complex(); // No delete: explicit call to Dtor. Use with extreme care
}
```

```
----- 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)
```

Storage Class Specifiers

Post-Recording

- The storage class specifiers are a part of the decl-specifier-seq of a name's declaration syntax
- Together with the scope of the name, they control two independent properties of the name: its *storage duration* (or *Lifetime*) and its *linkage*
  - `auto` or no specifier: Automatic storage duration. Deprecated in **C++11** and used for a difference semantics (**Module 46**)
  - `register`: Automatic storage duration that hints to the compiler to place the object in the processor's register. Deprecated in **C++17**
  - `static`: Static (or thread storage as discussed in **Module 59**) duration and internal linkage (or external linkage for static class members not in an anonymous namespace)
  - `extern`: Static (or thread storage as discussed in **Module 59**) duration and external linkage
  - `thread_local`: Thread storage duration in concurrency support since **C++11** (**Module 59**)
  - `mutable`: Related to `const` / `volatile` does not affect storage duration or linkage
- Only one storage class specifier may appear in a declaration except that `thread_local` may be combined with `static` or with `extern` since **C++11** (**Module 59**)

Module Summary

- Objects are initialized by Constructors that can be Parameterized and / or Overloaded
- Default Constructor does not take any parameter – necessary for 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