Programming in Modern C++

Module M11: Classes and Objects

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

Weekly Recap

- Revisited cv-qualifiers - `const` & `volatile` and compared macors with `inline` functions
- Introduced Reference variable or Alias in C++ and compared Call-by-reference with Call-by-value & Return-by-reference with Return-by-value
- Studied the differences between References and Pointers
- Introduced Default parameter and Function overloading for Static Polymorphism
- Studied Overload Resolution with Default parameters and Function Overloading
- Understood the differences between Operators & Functions and introduced Operator Overloading with examples
- Understood Operator Overloading Rules and Restrictions
- Did a roundup of Memory management in C and in C++
- Introduced allocation (`new`) / de-allocation (`delete`) operators in C++, their overloading and mixing with C styles

Module Objectives

- Understand the concept of classes and objects in C++

Module Outline

Classes

Classes

Classes

- A class is an implementation of a **type**. It is the only way to implement **User-defined Data Type (UDT)**
- A class contains *data members* / *attributes*
- A class has *operations* / *member functions* / *methods*
- A class defines a **namespace**
- Thus, classes offer **data abstraction** / **encapsulation** of **Object Oriented Programming**
- Classes are similar to structures that aggregate data logically
- A class is defined by `class` keyword
- Classes provide *access specifiers* for members to enforce **data hiding** that separates **implementation** from **interface**
    - `private` — accessible inside the definition of the class
    - `public` — accessible everywhere
- A class is a **blue print** for its instances (objects)

Objects

Objects

Objects

- An *object* of a class is an *instance* created according to its **blue print**. Objects can be automatically, statically, or dynamically created
- A object comprises *data members* that specify its *state*
- A object supports *member functions* that specify its *behavior*
- Data members of an object can be accessed by "." (dot) operator on the object
- Member functions are invoked by "." (dot) operator on the object
- An implicit `this` pointer holds the address of an object. This serves the *identity* of the object in C++
- `this` pointer is implicitly passed to methods

Data Members

Data Members

Program 11.01/02: Complex Numbers: Attributes

C Program

```
// File Name:Complex_object.c
#include <stdio.h>


typedef struct Complex { // struct
    double re, im;         // Data members
} Complex;
int main() {
    // Variable c declared, initialized
    Complex c = { 4.2, 5.3 };
    printf("%lf %lf", c.re, c.im); // Use by dot
}
-----
4.2 5.3
```

- `struct` is a keyword in C for *data aggregation*
- `struct Complex` is defined as *composite data type* containing two `double` (`re`, `im`) data members
- `struct Complex` is a derived data type used to create Complex type variable `c`
- Data members are accessed using '.' operator
- `struct` *only aggregates*

C++ Program

```
// File Name:Complex_object_c++.cpp
#include <iostream>
using namespace std;

class Complex { public: // class
    double re, im;        // Data members
};
int main() {
    // Object c declared, initialized
    Complex c = { 4.2, 5.3 };
    cout << c.re << " " << c.im; // Use by dot
}
-----
4.2 5.3
```

- `class` is a new keyword in C+ for *data aggregation*
- `class Complex` is defined as *composite data type* containing two `double` (`re`, `im`) data members
- `class Complex` is **User-defined Data Type (UDT)** used to create Complex type object `c`
- Data members are accessed using '.' operator.
- `class` *aggregates* and *helps build a UDT*

Program 11.03/04: Points and Rectangles: Attributes

C Program

```
// File Name:Rectangle_object.c
#include <stdio.h>


typedef struct { // struct Point
    int x; int y;
} Point;
typedef struct { // Rect uses Point
    Point TL;    // Top-Left. Member of UDT
    Point BR;    // Bottom-Right. Member of UDT
} Rect;
int main() { Rect r = { { 0, 2 }, { 5, 7 } };
    // r.TL <-- { 0, 2 }; r.BR <-- { 5, 7 }
    // r.TL.x <-- 0; r.TL.y <-- 2
    // Members of Structure r accessed
    printf("[(%d %d) (%d %d)]",
        r.TL.x, r.TL.y, r.BR.x, r.BR.y);
}
-----
[(0 2) (5 7)]
```

C++ Program

```
// File Name:Rectangle_object_c++.cpp
#include <iostream>
using namespace std;

class Point { public: // class Point
    int x; int y;     // Data members
};
class Rect { public: // Rect uses Point
    Point TL;        // Top-Left. Member of UDT
    Point BR;        // Bottom-Right. Member of UDT
};
int main() { Rect r = { { 0, 2 }, { 5, 7 } };
    // r.TL <-- { 0, 2 }; r.BR <-- { 5, 7 }
    // r.TL.x <-- 0; r.TL.y <-- 2
    // Rectangle Object r accessed
    cout << "[(" << r.TL.x << " " << r.TL.y <<
        ") (" << r.BR.x << " " << r.BR.y << ")]";
}
-----
[(0 2) (5 7)]
```

- Data members of user-defined data types

Program 11.05/06: Stacks: Attributes

C Program

```
// File Name:Stack_object.c
#include <stdio.h>


typedef struct Stack { // struct Stack
    char data[100]; // Container for elements
    int top;        // Top of stack marker
} Stack;

// Codes for push(), pop(), top(), empty()

int main() {
    // Variable s declared
    Stack s;
    s.top = -1;

    // Using stack for solving problems
}
```

C++ Program

```
// File Name:Stack_object_c++.cpp
#include <iostream>
using namespace std;

class Stack { public: // class Stack
    char data[100]; // Container for elements
    int top;        // Top of stack marker
};

// Codes for push(), pop(), top(), empty()

int main() {
    // Object s declared
    Stack s;
    s.top = -1;

    // Using stack for solving problems
}
```

- Data members of mixed data types

Member Functions

Member Functions

Program 11.07/08: Complex Numbers: Member Functions

C Program

```
// File Name:Complex_func.c
#include <stdio.h>
#include <math.h>

// Type as alias
typedef struct Complex { double re, im; } Complex;
// Norm of Complex Number - global fn.
double norm(Complex c) { // Parameter explicit
    return sqrt(c.re*c.re + c.im*c.im); }
// Print number with Norm - global fn.
void print(Complex c) { // Parameter explicit
    printf("|%lf+j%lf| = ", c.re, c.im);
    printf("%lf", norm(c)); // Call global
}

int main() { Complex c = { 4.2, 5.3 };
    print(c); // Call global fn. with c as param
}
-----
|4.200000+j5.300000| = 6.762396
```

- Access functions are *global*

C++ Program

```
// File Name:Complex_func_c++.cpp
#include <iostream>
#include <cmath>
using namespace std;
// Type as UDT
class Complex { public: double re, im;
    // Norm of Complex Number - method
    double norm() { // Parameter implicit
        return sqrt(re*re + im*im); }
    // Print number with Norm - method
    void print() { // Parameter implicit
        cout << "|"<< re<< "+j"<< im<< "| = ";
        cout << norm(); // Call method
    }
}; // End of class Complex
int main() { Complex c = { 4.2, 5.3 };
    c.print(); // Invoke method print of c
}
-----
|4.2+j5.3| = 6.7624
```

- Access functions are *members*

Program 11.09/10: Rectangles: Member Functions

Using `struct`

```
#include <iostream>
#include <cmath>
using namespace std;
typedef struct { int x; int y; } Point;
typedef struct {
    Point TL; // Top-Left
    Point BR; // Bottom-Right
} Rect;
// Global function
void computeArea(Rect r) { // Parameter explicit
    cout << abs(r.TL.x - r.BR.x) *
            abs(r.BR.y - r.TL.y);
}

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

    computeArea(r); // Global fn. call
}
-----
25
```

- Access functions are *global*

Using `class`

```
#include <iostream>
#include <cmath>
using namespace std;
class Point { public: int x; int y; };
class Rect { public:
    Point TL; // Top-Left
    Point BR; // Bottom-Right

    // Method
    void computeArea() { // Parameter implicit
        cout << abs(TL.x - BR.x) *
                abs(BR.y - TL.y);
    }
};
int main() { Rect r = { { 0, 2 }, { 5, 7 } };

    r.computeArea(); // Method invocation
}
-----
25
```

- Access functions are *members*

Program 11.11/12: Stacks: Member Functions

Using `struct`

```
#include <iostream>
using namespace std;
typedef struct Stack { char data_[100]; int top_;
} Stack;
// Global functions
bool empty(const Stack& s) { return (s.top_ == -1); }
char top(const Stack& s) { return  s.data_[s.top_]; }
void push(Stack& s, char x) { s.data_[++(s.top_)] = x; }
void pop(Stack& s) { --(s.top_); }

int main() { Stack s; s.top_ = -1;
    char str[10] = "ABCDE"; int i;
    for (i = 0; i < 5; ++i) push(s, str[i]);
    cout << "Reversed String: ";
    while (!empty(s)) {
        cout << top(s); pop(s);
    }
}
-----
Reversed String: EDCBA
```

- Access functions are *global*

Using `class`

```
#include <iostream>
using namespace std;
class Stack { public:
    char data_[100]; int top_;
    // Member functions
    bool empty() { return (top_ == -1); }
    char top() { return data_[top_]; }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
};
int main() { Stack s; s.top_ = -1;
    char str[10] = "ABCDE"; int i;
    for (i = 0; i < 5; ++i) s.push(str[i]);
    cout << "Reversed String: ";
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
}
-----
Reversed String: EDCBA
```

- Access functions are *members*

`this` Pointer

`this` **Pointer**

Program 11.13: this Pointer

- An *implicit* `this` pointer holds the address of an object
- `this` pointer serves as the **identity** of the object in C++
- Type of `this` pointer for a `class X` object: `X * const this;`
- `this` pointer is accessible *only in member functions*

```
#include <iostream>
using namespace std;
class X { public: int m1, m2;
    void f(int k1, int k2) {                // Sample member function
        m1 = k1;                            // Implicit access without this pointer
        this->m2 = k2;                      // Explicit access with this pointer
        cout << "Id   = " << this << endl; // Identity (address) of the object
    }
};
int main() { X a;
    a.f(2, 3);
    cout << "Addr = " << &a << endl;        // Address (identity) of the object
    cout << "a.m1 = " << a.m1 << "  a.m2 = " << a.m2 << endl;
    return 0;
}
-----
Id   = 0024F918
Addr = 0024F918
a.m1 = 2  a.m2 = 3
```

this Pointer

- `this` pointer is implicitly passed to methods

In Source Code	In Binary Code
• `class X { void f(int, int); ... }` • `X a; a.f(2, 3);`	• `void X::f(X * const this, int, int);` • `X::f(&a, 2, 3); // &a = this`

- Use of `this` pointer
  - Distinguish member from non-member

```
class X { public: int m1, m2;
    void f(int k1, int k2) {
        m1 = k1;       // this->m1 (member) is valid; this->k1 is invalid
        this->m2 = k2; // m2 (member) is valid; this->k2 is invalid
    }
};
```

  - Explicit Use

```
// Link the object
class DoublyLinkedNode { public: DoublyLinkedNode *prev, *next; int data;
    void append(DoublyLinkedNode *x) { next = x; x->prev = this; }
}
---
// Return the object
Complex& inc() { ++re; ++im; return *this; }
```

State of an Object

State of an Object

State of an Object: Complex

- The *state of an object* is determined by the *combined value of all its data members*

```
class Complex { public:
    double re_, im_;  // ordered tuple of data members decide the state at any time

    double get_re { return re_; }        // Read re_
    void set_re(double re) { re_ = re; } // Write re_
    double get_im { return im_; }        // Read im_
    void set_im(double im) { im_ = im; } // Write im_
};

Complex c = { 4.2, 5.3 };
// STATE 1 of c = { 4.2, 5.3 } // Denotes a tuple / sequence
```

- A method may change the state:

```
Complex c = { 4.2, 5.3 };
// STATE 1 of c = { 4.2, 5.3 }

c.set_re(6.4);
// STATE 2 of c = { 6.4, 5.3 }

c.get_re();
// STATE 2 of c = { 6.4, 5.3 } // No change of state

c.set_im(7.8);
// STATE 3 of c = { 6.4, 7.8 }
```

State of an Object: Rectangle

```
// Data members of Rect class: Point TL; Point BR; // Point class type object
// Data members of Point class: int x; int y;

Rectangle r = { { 0, 5 }, { 5, 0 } };  // Initialization
// STATE 1 of r = { { 0, 5 }, { 5, 0 } }
{ r.TL.x = 0; r.TL.y = 5; r.BR.x = 5; r.BR.y = 0 }

r.TL.y = 9;
// STATE 2 of r = { { 0, 9 }, { 5, 0 } }

r.computeArea();
// STATE 2 of r = { { 0, 9 }, { 5, 0 } } // No change in state

Point p = { 3, 4 };
r.BR = p;
// STATE 3 of r = { { 0, 9 }, { 3, 4 } }
```

State of an Object: Stack

```
// Data members of Stack class: char data[5] and int top;

Stack s;
// STATE 1 of s = {{?, ?, ?, ?, ?}, ?} // No data member is initialized

s.top_ = -1;
// STATE 2 of s = {{?, ?, ?, ?, ?}, -1}

s.push(’b’);
// STATE 3 of s = {{’b’, ?, ?, ?, ?}, 0}

s.push(’a’);
// STATE 4 of s = {{’b’, ’a’, ?, ?, ?}, 1}

s.empty();
// STATE 4 of s = {{’b’, ’a’, ?, ?, ?}, 1} // No change of state

s.push(’t’);
// STATE 5 of s = {{’b’, ’a’, ’t’, ?, ?}, 2}

s.top();
// STATE 5 of s = {{’b’, ’a’, ’t’, ?, ?}, 2} // No change of state

s.pops();
// STATE 6 of s = {{’b’, ’a’, ’t’, ?, ?}, 1}
```

Module Summary

- Class

```
class Complex { public:
    double re_, im_;

    double norm() { // Norm of Complex Number
        return sqrt(re_ * re_ + im_ * im_);
    }
};
```

- Attributes

```
Complex::re_, Complex::im_
```

- Member Functions

```
double Complex::norm();
```

- Object

```
Complex c = {2.6, 3.9};
```

- Access

```
c.re_ = 4.6;
cout << c.im_;
cout << c.norm();
```

- this Pointer

```
double Complex::norm() { cout << this; return ... }
```

- State of Object

```
Rectangle r = { { 0, 5 }, { 5, 0 } }; // STATE 1 r = { { 0, 5 }, { 5, 0 } }
r.TL.y = 9;                            // STATE 2 r = { { 0, 9 }, { 5, 0 } }
r.computeArea();                       // STATE 2  r = { { 0, 9 }, { 5, 0 } }
Point p = { 3, 4 }; r.BR = p;          // STATE 3  r = { { 0, 9 }, { 3, 4 } }
```

Programming in Modern C++

Module M12: Access Specifiers

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

- Class

```
class Complex { public:
    double re_, im_;

    double norm() { // Norm of Complex Number
        return sqrt(re_ * re_ + im_ * im_);
    }
};
```

- Attributes

```
Complex::re_, Complex::im_
```

- Member Functions

```
double Complex::norm();
```

- Object

```
Complex c = {2.6, 3.9};
```

- Access

```
c.re_ = 4.6;
cout << c.im_;
cout << c.norm();
```

- this Pointer

```
double Complex::norm() { cout << this; return ... }
```

- State of Object

```
Rectangle r = { { 0, 5 }, { 5, 0 } }; // STATE 1 r = { { 0, 5 }, { 5, 0 } }
r.TL.y = 9;                           // STATE 2 r = { { 0, 9 }, { 5, 0 } }
r.computeArea();                      // STATE 2  r = { { 0, 9 }, { 5, 0 } }
Point p = { 3, 4 }; r.BR = p;         // STATE 3  r = { { 0, 9 }, { 3, 4 } }
```

Module Objectives

- Understand access specifiers in C++ classes to control the visibility of members
- Learn to design with Information Hiding

Module Outline

Access Specifiers

Access Specifiers

Access Specifiers

- Classes provide **access specifiers** for members (data as well as function) to enforce **data hiding** that separates *implementation* from *interface*
  - `private` — accessible inside the definition of the class
    - member functions of the same class
  - `public` — accessible everywhere
    - member functions of the same class
    - member function of a different class
    - global functions
- The keywords `public` and `private` are the *Access Specifiers*
- Unless specified, the access of the members of a class is considered `private`
- A class may have multiple access specifier. The effect of one continues till the next is encountered

Program 12.01/02: Complex Number: Access Specification

Public data, Public method

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { public: double re, im;
public:
    double norm() { return sqrt(re*re + im*im); }
};
void print(const Complex& t) { // Global fn.
    cout << t.re << "+j" << t.im << endl;


}
int main() { Complex c = { 4.2, 5.3 }; // Okay


    print(c);
    cout << c.norm();
}
```

- public data can be accessed by any function
- norm (method) can access (re, im)
- print (global) can access (re, im)
- main (global) can access (re, im) & initialize

Private data, Public method

```
#include <iostream>
#include <cmath>
using namespace std;
class Complex { private: double re, im;
public:
    double norm() { return sqrt(re*re + im*im); }
};
void print(const Complex& t) { // Global fn.
    cout << t.re << "+j" << t.im << endl;
    // Complex::re / Complex::im: cannot access
    // private member declared in class 'Complex'
}
int main() { Complex c = { 4.2, 5.3 }; // Error
    // 'initializing': cannot convert from
    // 'initializer-list' to 'Complex'
    print(c);
    cout << c.norm();
}
```

- private data can be accessed *only* by methods
- norm (method) can access (re, im)
- print (global) cannot access (re, im)
- main (global) cannot access (re, im) to initialize

Information Hiding

Information Hiding

Information Hiding

- The `private` part of a class (*attributes* and *member functions*) forms its *implementation* because the class alone should be concerned with it and have the right to change it
- The `public` part of a class (*attributes* and *member functions*) constitutes its *interface* which is available to all others for using the class
- Customarily, we put all *attributes* in `private` part and the *member functions* in `public` part. This ensures:
  - The **state** of an object can be changed only through one of its *member functions* (with the knowledge of the class)
  - The **behavior** of an object is accessible to others through the *member functions*
- This is known as **Information Hiding**

Information Hiding

- For the sake of efficiency in design, we at times, put *attributes* in `public` and / or *member functions* in `private`. In such cases:
    - The `public` *attributes should not* decide the *state* of an object, and
    - The `private` *member functions* cannot be part of the *behavior* of an object

We illustrate information hiding through two implementations of a stack

Information Hiding: Stack Example

Information Hiding: Stack Example

Program 12.03/04: Stack: Implementations using public data

Using dynamic array

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Stack { public: char *data_; 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() { Stack s; char str[10] = "ABCDE";
    s.data_ = new char[100]; // Exposed Allocation
    s.top_ = - 1;            // Exposed Init

    for(int i = 0; i < 5; ++i) s.push(str[i]);
    // Outputs: EDCBA -- Reversed string
    while(!s.empty()) { cout << s.top(); s.pop(); }
    delete [] s.data_;     // Exposed De-Allocation
}
```

Using vector

```
#include <iostream>
#include <vector>
using namespace std;
class Stack { public: vector<char> data_; 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() { Stack s; char str[10] = "ABCDE";
    s.data_.resize(100); // Exposed Sizing
    s.top_ = -1;         // Exposed Init

    for(int i = 0; i < 5; ++i) s.push(str[i]);
    // Outputs: EDCBA -- Reversed string
    while(!s.empty()) { cout << s.top(); s.pop(); }

}
```

- `public` data reveals the *internals of the stack* (no information hiding)
- Spills data structure codes (`Exposed Init / De-Init`) into the application (`main`)
- To switch from array to vector or vice-versa the application needs to change

Program 12.03/04: Stack: Implementations using public data
Risky

Using dynamic array

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Stack { public: char *data_; 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() { Stack s; char str[10] = "ABCDE";
    s.data_ = new char[100]; // Exposed Allocation
    s.top_ = - 1;            // Exposed Init

    for(int i=0; i<5; ++i) s.push(str[i]);
    s.top_ = 2; // STACK GETS INCONSISTENT
    // Outputs: CBA -- WRONG!!!
    while (!s.empty()) { cout << s.top(); s.pop(); }
    delete [] s.data_;       // Exposed De-Init
}
```

Using vector

```
#include <iostream>
#include <vector>
using namespace std;
class Stack { public: vector<char> data_; 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() { Stack s; char str[10] = "ABCDE";
    s.data_.resize(100); // Exposed Sizing
    s.top_ = -1;         // Exposed Init

    for(int i=0; i<5; ++i) s.push(str[i]);
    s.top_ = 2; // STACK GETS INCONSISTENT
    // Outputs: CBA -- WRONG!!!
    while (!s.empty()) { cout << s.top(); s.pop(); }

}
```

- **Application may intentionally or inadvertently tamper the value of `top_` – this corrupts the stack!**
- `s.top_ = 2;` **destroys consistency of the stack and causes wrong output**

Program 12.05/06: Stack: Implementations using private data

Safe

Using dynamic array

```
#include <iostream>

using namespace std;
class Stack { private: char *data_; int top_;
public: // Initialization and De-Initialization
    Stack(): data_(new char[100]), top_(-1) { }
    ~Stack() { delete[] data_; }
    // Stack LIFO Member Functions
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { Stack s; char str[10] = "ABCDE";
    for (int i=0; i<5; ++i) s.push(str[i]);
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
```

Using vector

```
#include <iostream>
#include <vector>
using namespace std;
class Stack { private: vector<char> data_; int top_;
public: // Initialization and De-Initialization
    Stack(): top_(-1) { data_.resize(100); }
    ~Stack() { };
    // Stack LIFO Member Functions
    int empty() { return (top_ == -1); }
    void push(char x) { data_[++top_] = x; }
    void pop() { --top_; }
    char top() { return data_[top_]; }
};
int main() { Stack s; char str[10] = "ABCDE";
    for (int i=0; i<5; ++i) s.push(str[i]);
    while (!s.empty()) { cout << s.top(); s.pop(); }
}
```

- `private` data hides the *internals* of the stack (information hiding)
- Data structure codes *contained within itself* with *initialization* and de-initialization
- To switch from array to vector or vice-versa the application needs **no change**
- **Application cannot tamper stack – any direct access to `top_` or `data_` is compilation error!**

Program 12.07: Interface and Implementation

Interface

```
// File: Stack.h -- Interface
class Stack { private: // Implementation
    char *data_; int top_;
public: // Interface
    Stack();
    ~Stack();
    int empty();
    void push(char x);
    void pop();
    char top();
};
```

Implementation

```
// File: Stack.cpp -- Implementation
#include "Stack.h"

Stack::Stack(): data_(new char[100]), top_(-1) { }
Stack::~Stack() { delete[] data_; }
int Stack::empty() { return (top_ == -1); }
void Stack::push(char x) { data_[++top_] = x; }
void Stack::pop() { --top_; }
char Stack::top() { return data_[top_]; }
```

Application

```
#include <iostream>
using namespace std;
#include "Stack.h"
int main() {
    Stack s; char str[10] = "ABCDE";
    for (int i = 0; i < 5; ++i) s.push(str[i]);
    while (!s.empty()) {
        cout << s.top(); s.pop();
    }
}
```

Get-Set Idiom

Get-Set Idiom

Get–Set Methods: Idiom for fine-grained Access Control

- We put *attributes* in `private` and the *methods* in `public` to restrict the access to data
- `public` methods to *read* (`get`) and / or *write* (`set`) data members provide fine-grained control

```
class MyClass { // private
    int readWrite_; // Like re_, im_ in Complex -- common aggregated members

    int readOnly_;  // Like DateOfBirth, Emp_ID, RollNo -- should not need a change

    int writeOnly_; // Like Password -- reset if forgotten

    int invisible_; // Like top_, data_ in Stack -- keeps internal state

    public:
    // get and set methods both to read as well as write readWrite_ member
    int getReadWrite() { return readWrite_; }
    void setReadWrite(int v) { readWrite_ = v; }

    // Only get method to read readOnly_ member - no way to write it
    int getReadOnly() { return readOnly_; }

    // Only set method to write writeOnly_ member - no way to read it
    void setWriteOnly(int v) { writeOnly_ = v; }

    // No method accessing invisible_ member directly - no way to read or write it
}
```

Get, Set Methods

- Get, Set methods of a class are the interface defined for accessing and using the private data members. The implementation details of the data members are hidden.
- Not all data members are allowed to be updated or read, hence based on the requirement of the interface, data members can be read only, write only, read and write both or not visible at all.
- Let get and set be two variables of `bool` type which signifies presence of get and set methods respectively. In the below table, T denotes true (that is, method is present) and F denotes False (that is, method is absent)

Variables	**get**	**set**
Non Visible	F	F
Read Only	T	F
Write Only	F	T
Read - Write	T	T

Program 12.08: Get - Set Methods: Employee Class

Get-Set Methods

```
// File Name:Employee_c++.cpp:
#include <iostream>
#include <string>
using namespace std;

class Employee { private:
    string name;         // read and write: get_name() and set_name() defined
    string address;      // write only: set_addr() defined. No get method
    double sal_fixed;    // read only: get_sal_fixed()defined. No set method
    double sal_variable; // not visible: No get-set method

    public: Employee() { sal_fixed = 1200; sal_variable = 10; } // Initialize
    string get_name() { return name; }
    void set_name(string name) { this->name = name; }
    void set_addr(string address) { this->address = address; }
    double get_sal_fixed() { return sal_fixed; }
    // sal_variable (not visible) used in computation method salary()
    double salary() { return sal_fixed + sal_variable; }
};
int main() {
    Employee e1; e1.set_name("Ram"); e1.set_addr("Kolkata");
    cout << e1.get_name() << endl;  cout << e1.get_sal_fixed() << endl <<  e1.salary() << endl;
}
```

Encapsulation

Encapsulation

Encapsulation

- classes wrap data and functions acting on the data together as a single data structure. This is **Aggregation**
- The important feature introduced here is that members of a class has a **access specifier**, which defines their visibility outside the class
- This helps in *hiding information* about the implementation details of data members and methods
  - If properly designed, any change in the *implementation*, should not affect the *interface* provided to the users
  - Also hiding the implementation details, prevents unwanted modifications to the data members.
- This concept is known as **Encapsulation** which is provided by classes in C++.

Class as a Data-type

Class as a Data-type

Class as a Data-type

- We can conclude now that class is a composite data type in C++ which has similar behaviour to built in data types. We explain below with the Complex class (representing complex number) as an example

`// declare i to be of int type` `int i;`	`// declare c to be of Complex type` `Complex c;`
`// initialise i` `int i = 5;`	`// initialise the real and imaginary components of c` `Complex c = { 4, 5 };`

`// print i`
`cout << i;`
	`// 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 ints`
`int i = 5, j = 6;`
`i+j;`
	`// 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`

Module Summary

- 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

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

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
```

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_[10]` initialized by default (*automatic*)

- `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 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

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

- 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 or by `operator delete`. May be called explicitly by the object (rare)

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

- 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

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

Object Lifetime

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)
```

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

Programming in Modern C++

Module M14: Copy Constructor and Copy Assignment Operator

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

- 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

Module Objectives

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

Module Outline

Object Lifetime Examples

Object Lifetime Examples

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

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

Practice: Program 14.08: Name & Address Classes

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

#include "String.h" // Containing class String from slide 14.7
#include "Date.h"

class Name { String firstName_, lastName_;
public: Name(char* fn, 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, char* sn, char* cn, 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();
    }
};
```

Practice: Program 14.08: 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(char* cNumber, char* fn, char* ln, unsigned int hn, char* sn, char* cn, 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;;
}
```

Practice: Program 14.08: CreditCard Class: Lifetime Chart

```
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_; };
```

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
```

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

Copy Constructor

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 14.09: 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 14.10: 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

Program 14.11: Point and Rect Classes: Embedded Objects 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) { cout << "Point ctor: "; print(); cout << endl; }         // Ctor
    Point(): x_(0), y_(0) { cout << "Point ctor: "; print(); cout << endl; }                     // DCtor
    Point(const Point& p): x_(p.x_), y_(p.y_) { cout << "Point cctor: "; print(); cout << endl; } // CCtor
    ~Point() { cout << "Point dtor: "; print(); cout << endl; }                                  // Dtor
    void print() { cout << "(" << x_ << ", " << y_ << ")"; } };  // Class Point
class Rect { Point TL_; Point BR_; public:
    Rect(int tlx, int tly, int brx, int bry): TL_(tlx, tly), BR_(brx, bry) // Ctor of Rect: 4 coords
    { cout << "Rect ctor: "; print(); cout << endl; }                       // Uses Ctor for Point
    Rect(const Point& p_tl, const Point& p_br): TL_(p_tl), BR_(p_br)       // Ctor of Rect: 2 Points
    { 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 of Rect: Point + 2 coords
    { cout << "Rect ctor: "; print(); cout << endl; }                       // Uses CCtor for Point
    Rect() { cout << "Rect ctor: "; print(); cout << endl; }                // DCtor of Rect: // DCtor Point
    Rect(const Rect& r): TL_(r.TL_), BR_(r.BR_)                             // CCtor of Rect
    { cout << "Rect cctor: "; print(); cout << endl; }                      // Uses CCtor for Point
    ~Rect() { cout << "Rect dtor: "; print(); cout << endl; }               // Dtor
    void print() { cout << "["; TL_.print(); cout << " "; BR_.print(); cout << "]"; } }; // Class Rect
```

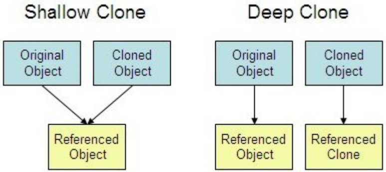
- When parameter `(tlx, tly)` is set to `TL_` by `TL_(tlx, tly)`: parameterized **Ctor** of `Point` is invoked
- When parameter `p_tl` is set to `TL_` by `TL_(p_tl)`: **CCtor** of `Point` is invoked
- When `TL_` is set by default in **DCtor** of `Rect`: **DCtor** of `Point` is invoked
- When member `r.TL_`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

Practice: Program 14.11: Rect Class: Trace of Object Lifetimes

Code	Output	Lifetime	Remarks
`int main() {`			
`Rect r1(0, 2, 5, 7);`	Point ctor: (0, 2)	Point r1.TL_	
`//Rect(int, int, int, int)`	Point ctor: (5, 7)	Point r1.BR_	
	Rect ctor: [(0, 2) (5, 7)]	Rect r1	
`Rect r2(Point(3, 5),`	Point ctor: (6, 9)	Point t1	Second parameter
`Point(6, 9));`	Point ctor: (3, 5)	Point t2	First parameter
`//Rect(Point&, Point&)`	Point cctor: (3, 5)	r2.TL_ = t2	Copy to `r2.TL_`
	Point cctor: (6, 9)	r2.BR_ = t1	Copy to `r2.BR_`
	Rect ctor: [(3, 5) (6, 9)]	Rect r2	
	Point dtor: (3, 5)	~Point t2	First parameter
	Point dtor: (6, 9)	~Point t1	Second parameter
`Rect r3(Point(2, 2), 6, 4);`	Point ctor: (2, 2)	Point t3	First parameter
`//Rect(Point&, int, int)`	Point cctor: (2, 2)	r3.TL_ = t3	Copy to `r3.TL_`
	Point ctor: (6, 4)	Point r3.BR_	
	Rect ctor: [(2, 2) (6, 4)]	Rect r3	
	Point dtor: (2, 2)	~Point t3	First parameter
`Rect r4;`	Point ctor: (0, 0)	Point r4.TL_	
`//Rect()`	Point ctor: (0, 0)	Point r4.BR_	
	Rect ctor: [(0, 0) (0, 0)]	Rect r4	
`return 0;`	Rect dtor: [(0, 0) (0, 0)]	~Rect r4	
`}`	Point dtor: (0, 0)	~Point r4.BR_	
	Point dtor: (0, 0)	~Point r4.TL_	
	Rect dtor: [(2, 2) (6, 4)]	~Rect r3	
	Point dtor: (6, 4)	~Point r3.BR_	
	Point dtor: (2, 2)	~Point r3.TL_	
	Rect dtor: [(3, 5) (6, 9)]	~Rect r2	
	Point dtor: (6, 9)	~Point r2.BR_	
	Point dtor: (3, 5)	~Point r2.TL_	
	Rect dtor: [(0, 2) (5, 7)]	~Rect r1	
	Point dtor: (5, 7)	~Point r1.BR_	
	Point dtor: (0, 2)	~Point r1.TL_	

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. So some languages support variants as *Lazy Copy* or *Copy-on-Write* for efficiency

Practice: Program 14.12: Complex: Free Copy Constructor

```
#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

```
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-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 14.13: 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 14.14: 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

```
(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 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

Copy Assignment Operator

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 14.15: 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(); }     // 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 14.16: 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 14.17: 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 14.18: 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)
```

- In case of self-copy, do nothing

- Check for se

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

Comparison of Copy Constructor and Copy Assignment Operator

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

Class as a Data-type

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

Programming in Modern C++

Module M15: Const-ness

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

- **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 Objectives

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

Module Outline

Constant Objects

Constant Objects

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 objects cannot invoke normal methods of the class lest these methods 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 Functions

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

Constant Data Members

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`

```
#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` 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

Module Summary

- Studied const-ness in C++
- In C++, there are three forms of const-ness
  - **Constant Objects**
    - No change is allowed after construction
    - Cannot invoke normal member functions
  - **Constant Member Functions**
    - Can be invoked by constant (as well as non-constant) objects
    - Cannot make changes to the object
  - **Constant Data Members**
    - No change is allowed after construction
    - Must be initialized in the initialization list
- Further, learnt how to model *logical const-ness* over *bit-wise const-ness* by proper use of `mutable` members

Programming in Modern C++

Tutorial T03: How to build a C/C++ program?: Part 3: `make` Utility

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

Tutorial Recap

- Understood the overall build process for a C/C++ project with specific reference to the build pipeline of GCC
- Understood the differences and relationships between source and header files
- Understood how CPP can be harnessed to manage code during build
- Understood the management of C/C++ dialects and C/C++ Standard Libraries

Tutorial Objective

- Building a software project is a laborious, error-prone, and time consuming process. So it calls for automation by scripting
- `make`, primarily from GNU, is the most popular free and open source dependency-tracking builder tool that all software developers need to know

Tutorial Outline

`make` Utility

Source: Accessed 15-Sep-21
GNU make Manual
GNU Make
A Simple Makefile Tutorial

make Utility: Example Build

- Consider a tiny project comprising three files – **main.c**, **hello.c**, and **hello.h** in a folder

main.c

```
#include "hello.h"

int main() {
    // call a function in
    // another file
    myHello();
    return 0;
}
```

hello.c

```
#include <stdio.h>
#include "hello.h"

void myHello(void) {
    printf("Hello World!\n");
    return;
}
```

hello.h

```
// example include file
#ifndef __HEADER_H
#define __HEADER_H

void myHello(void);

#endif // __HEADER_H
```

- We build this by executing the command in the current folder (`-I.`):

```
gcc -o hello hello.c main.c -I. // Generates hello.o & main.o and removes at the end
```

which actually expands to:

```
gcc -c hello.c -I. // Compile and Generate hello.o
gcc -c main.c -I.  // Compile and Generate main.o
gcc -o hello hello.o main.o -I. // Link and Generate hello
rm -f hello.o main.o // Is it really necessary? . hello.o & main.o may be retained
```

Why we need `make` Utility?

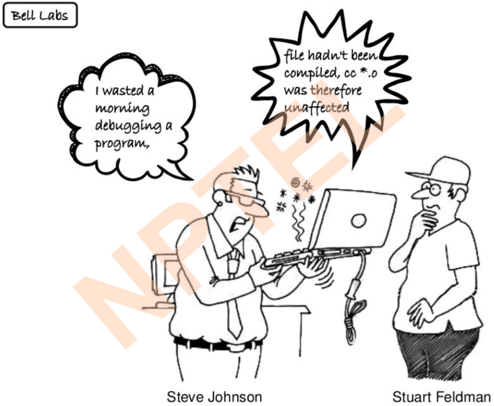
- This manual process of build would be difficult in any practical software project due to:
  - **[Volume]** Projects have *hundreds of folders, source and header files*. They need *hundreds of commands*. It is *time-taking* to type the commands and is *error-prone*
  - **[Workload]** Build needs to be *repeated several times a day* with code changes in some file/s
  - **[Depenency]** Often with the change in one file, all translation units do not need to be re-compiled (assuming that we do not remove `.o` files). For example:
    - ▷ If we change only `hello.c`, we do not need to execute

      ```
      gcc -c main.c -I.  // main.o is already correct
      ```
    - ▷ If we change only `main.c`, we do not need to execute

      ```
      gcc -c hello.c -I.  // hello.o is already correct
      ```
    - ▷ However, if we change `hello.h`, we need to execute all

    There are *dependencies* in build that can be exploited to optimize the build effort
  - **[Diversity]** Finally, we may need to use different build tools for different files, different build flags, different folder structure etc.
- This calls for *automation by scripting*. **GNU** `Make` **is a tool which controls the generation of executables and other non-source files of a program from the program's source files**

Why we need `make` Utility?: What happened in Bell Labs

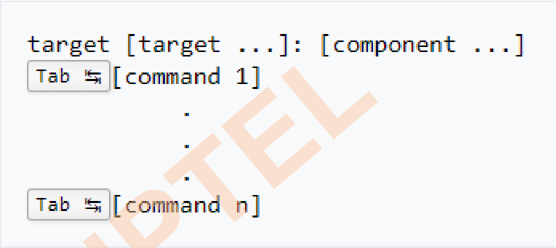


Broadlinux | Linux of Things

Stuart Feldman created make in April 1976 at Bell Labs

Makefile Martial Arts - Chapter 1. The morning of creation

Anatomy of a makefile



```
target [target ...]: [component ...]
Tab ⇆[command 1]
        .
        .
        .
Tab ⇆[command n]
```

Make (software), Wikipedia

Anatomy of a makefile

`makefile`: Anatomy

- A simple make file would be like (`makefile_1.txt`):

```
hello: hello.c main.c
    gcc -o hello hello.c main.c -I.
```

  Write these lines in a text file named `makefile` or `Makefile` and run `make` command and it will execute the command:

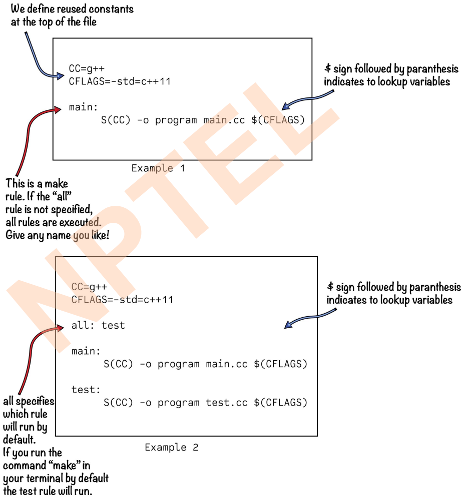
```
$ make
gcc -o hello hello.c main.c -I.
```

- Make file comprises a number of Rules. Every rule has a target to build, a colon separator (`:`), zero or more files on which the target depends on and the commands to build on the next line

```
hello: hello.c main.c # Rule 1
    gcc -o hello hello.c main.c -I.
```

- Note:
  - **There must be a tab at the beginning of any command**. Spaces will not work!
  - If any of the file in the dependency (hello.c or main.c) change since the last time `make` was done (target `hello` was built), the rule will fire and the command (gcc) will execute. This is decided by the last update timestamp of the files
  - Hash (#) starts a comment that continues till the end of the line

makefile: Anatomy



We define reused constants at the top of the file

```
CC=g++
CFLAGS=-std=c++11

main:
     S(CC) -o program main.cc $(CFLAGS)
```

Example 1

$ sign followed by paranthesis indicates to lookup variables

This is a make rule. If the "all" rule is not specified, all rules are executed. Give any name you like!

```
CC=g++
CFLAGS=-std=c++11

all: test

main:
     S(CC) -o program main.cc $(CFLAGS)

test:
     S(CC) -o program test.cc $(CFLAGS)
```

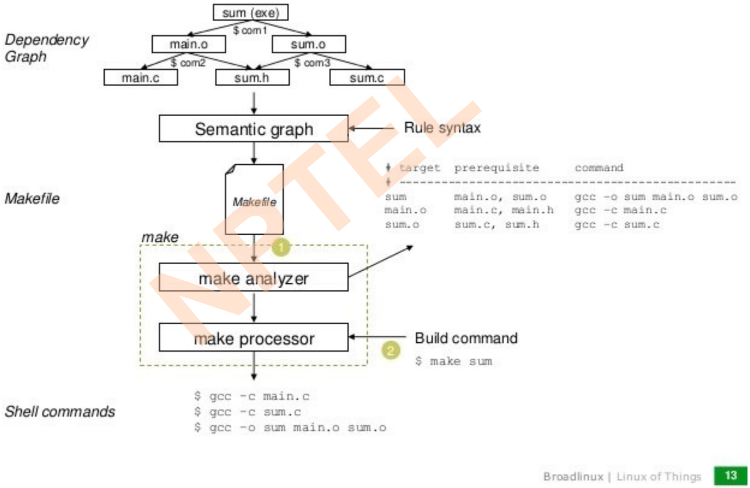
Example 2

$ sign followed by paranthesis indicates to lookup variables

all specifies which rule will run by default. If you run the command "make" in your terminal by default the test rule will run.

How to Write a Makefile with Ease

`makefile`: Architecture



```
# target   prerequisite      command
#-------------------------------------------------------
sum        main.o, sum.o     gcc -o sum main.o sum.o
main.o     main.c, main.h    gcc -c main.c
sum.o      sum.c, sum.h      gcc -c sum.c
```

```
$ make sum
```

```
$ gcc -c main.c
$ gcc -c sum.c
$ gcc -o sum main.o sum.o
```

Makefile Martial Arts - Chapter 1. The morning of creation

makefile: Simple and Recursive Variables

- We can make the make file smarter (makefile_2.txt) using variables:

```
# CC is a simple variable, gcc is its value
CC := gcc
# CFLAGS is a recursive variable, -I. is its value
CFLAGS = -I.

hello: hello.c main.c        # Rule 1
    $(CC) -o hello hello.c main.c $CFLAGS
# $(CC) is value gcc of CC, $CFLAGS is value -I. of CFLAGS, Variables can be expanded by $(.) or ${.}
```

- If there are several commands, to change gcc to g++, we just need to change one line CC=g++.
- There are two types of variables in make (Chapter 3. Variables and Macros, O'Reilly)
  - Simply expanded variables (defined by := operator) and *evaluated as soon as encountered*
  - Recursively expanded variables (by =) and *lazily evaluated, may be defined after use*

Simply Expanded	**Recurively Expanded**
`MAKE_DEPEND := $(CC) -M`	`MAKE_DEPEND = $(CC) -M`
`...`	`...`
`# Some time later`	`# Some time later`
`CC = gcc`	`CC = gcc`
`$(MAKE_DEPEND)$ expands to:`	
`<space>-M`	`gcc -M`

makefile: Dependency

- We are still missing the dependency on the include (header) files. If `hello.h` changes, the above `Rule 1` will not detect the need for a re-build. So we improve further (`makefile_3.txt`):

```
CC=gcc
CFLAGS=-I.
#Set of header files on which .c depends
DEPS = hello.h

# Rule 1: Applies to all files ending in the .o suffix
# The .o file depends upon the .c version of the file and the .h files in the DEPS macro
# To generate the .o file, make needs to compile the .c file using the CC macro
# The -c flag says to generate the object file
# The -o $@ says to put the output of the compilation in the file named on the LHS of :
# The $< is the first item in the dependencies list
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

hello: hello.o main.o        # Rule 2: Link .o files
    $(CC) -o hello hello.o main.o -I.
```

makefile: Dependency

- We can further simplify on the object files (makefile_4.txt):

```
CC=gcc
CFLAGS=-I.
DEPS = hello.h
#Set of object files on which executable depends
OBJ = hello.o main.o

# Rule 1: Applies to all files ending in the .o suffix
%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

# Rule 2: Linking step, applies to the executable depending on the file in OBJ macro
# The -o $@ says to put the output of the linking in the file named on the LHS of :
# The $^ is the files named on the RHS of :
hello: $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)
```

`makefile`: Code Organization

- Finally, let us introduce a source organization that is typical of a large project where under a project `Home` folder, we have the following folders:
  - `Home`: The make file and the following folders:
    - ▷ `bin`: The executable of the project. For example `hello` / `hello.exe`
    - ▷ `inc`: The include / header (`.h`) files of the project. For example `hello.h`
    - ▷ `lib`: The local library files (`.a`) of the project
    - ▷ `obj`: The object files (`.o`) of the project. For example `hello.o` & `main.o`
    - ▷ `src`: The source files (`.c`/`.cpp`) of the project. For example `hello.c` & `main.c`

makefile: Code Tree

```
Home // Project Home
  |
  ---- bin // Application binary
  |      |
  |      ---- hello.exe
  |
  ---- inc // Headers files to be included in application
  |      |
  |      ---- hello.h
  |
  ---- lib // Library files to be linked to application. Check Tutorial Static and Dynamic Library
  |
  ---- obj // Object files
  |      |
  |      ---- hello.o
  |      ---- main.o
  |
  ---- src // Source files
  |
  |      ---- hello.c
  |      ---- main.c
  |
  ---- makefile // Makefile
```

makefile: Code Organization

- To handle this hierarchy, we modify as (`makefile_5.txt`):

```
CC = gcc
# Folders
BDIR = bin
IDIR = inc
LDIR = lib
ODIR = obj
SDIR = src
# Flags
CFLAGS = -I$(IDIR)
# Macros
_DEPS = hello.h # Add header files here
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))
_SRC = hello.c main.c # Add source files here
SRC = $(patsubst %,$(SDIR)/%,$(_SRC))
_OBJ = hello.o main.o # Add source files here
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))
# Rule 1: Object files
$(ODIR)/%.o: $(SDIR)/%.c $(DEPS); $(CC) -c -o $@ $< $(CFLAGS) -I.
#Rule 2: Binary File  Set binary file here
$(BDIR)/hello: $(OBJ); $(CC) -o $@ $^ $(CFLAGS)
# Rule 3: Remove generated files. .PHONY rule keeps make from doing something with a file named clean
.PHONY: clean
clean: ; del $(ODIR)\*.o $(BDIR)\*.exe
#         rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

`make` Command

`make` **Command**

make Command: Options and Features

- The format of `make` command is:

```
make [ -f makefile ] [ options ] ... [ targets ] ...
```

  `make` executes commands in the `makefile` to update one or more target `names`
  - With no `-f`, `make` looks for `makefile`, and `Makefile`. To use other files do:

```
$ make -f makefile_1.txt                // Using makefile_1.txt
gcc -o hello hello.c main.c -I.
```

  - `make` updates a target if its prerequisite files are dated. Starting empty obj & bin folders:

```
$ make -f makefile_5.txt obj/hello.o    // Build hello.o, place in obj
gcc -c -o obj/hello.o src/hello.c -Iinc -I.

$ make -f makefile_5.txt obj/main.o     // Build main.o, place in obj
gcc -c -o obj/main.o src/main.c -Iinc -I.

$ make -f makefile_5.txt bin/hello      // Build hello.exe linking .o files and place in bin
gcc -o bin/hello obj/hello.o obj/main.o -Iinc

$ make -f makefile_5.txt  clean         // Remove non-text files generated - obj/*.o & bin/*.exe
del obj\*.o bin\*.exe

$ make -f makefile_5.txt                // By default targets bin/hello and builds all
gcc -c -o obj/hello.o src/hello.c -Iinc -I.
gcc -c -o obj/main.o src/main.c -Iinc -I.
gcc -o bin/hello obj/hello.o obj/main.o -Iinc
```

make Command: Options and Features

- More `make` options / features:
  - To change to directory `dir` before reading the makefiles, use `-C dir`
  - To print debugging information in addition to normal processing, use `-d`
  - To specify a directory `dir` to search for included makefiles, use `-I dir`
  - To print the version of the `make` program, use `-v`. We are using

    ```
    GNU Make 3.81
    Copyright (C) 2006  Free Software Foundation, Inc.
    This is free software; see the source for copying conditions.
    There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

    This program built for i386-pc-mingw32
    ```
  - `make` can be recursive - one make file may include a command to `make` another
  - Multiline: The backslash ("\") character gives us the ability to use multiple lines when the commands are too long

    ```
    some_file:
        echo This line is too long, so \
            it is broken up into multiple lines
    ```
  - Comments: Lines starting with `#` are used for comments
  - Macros: Besides simple and recursive variables, `make` also supports macros

`make` Utility: Capabilities and Derivatives

- `make` was created by Stuart Feldman in April 1976 at Bell Labs and included in Unix since PWB/UNIX 1.0. He received the 2003 ACM Software System Award for `make`
- `make` is one of the most popular build utilities having the following major **Capabilities**:
  - `make` enables the end user to *build and install a package* without knowing the details of how that is done (it is in the makefile supplied by the user)
  - `make` *figures out automatically which files it needs to update*, based on which source files have changed and *automatically determines the proper order for updating files*
  - `make` is *not limited to any particular language* - C, C++, Java, and so on.
  - `make` is *not limited to building a package* - can control installation/uninstallation etc.
- `make` has several **Derivative** and is available on all OS platforms:
  - GNU Make (all types of Unix): Used to build many software systems, including:
    - ▷ GCC, the Linux kernel, Apache OpenOffice, LibreOffice, and Mozilla Firefox
  - Make for Windows, GnuWin32 (We are using here)
  - Microsoft nmake, a command-line tool, part of Visual Studio
  - Kati is Google's replacement of GNU Make, used in Android OS builds. It translates the makefile into Ninja (used for Chrome) for faster incremental builds

All links Accessed on 21-Sep-21

Tutorial Summary

- Learnt `make`, the most popular free and open source dependency-tracking builder tool, with its anatomy, architecture and options through a series of examples