Module 10: Programming in C++

Dynamic Memory Management

Instructors: Abir Das and Jibesh Patra

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

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

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

by **Prof. Partha Pratim Das**

Module Objectives

- Understand the dynamic memory management in C++

Module Outline

1. Dynamic Memory Management in C
   - `malloc` & `free`
2. Dynamic Memory Management in C++
   - `new` and `delete` operator
   - Dynamic memory allocation for Array
   - Placement `new`
   - Restrictions
3. Operator Overloading for Allocation and De-allocation
4. Module Summary

Program 10.01/02: malloc() & free(): C & C++

C Program

```c
#include <stdio.h>
#include <stdlib.h>


int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    printf("%d", *p);  // Prints: 5

    free(p);
}
```

C++ Program

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
```

- Dynamic memory management functions in `stdlib.h` header for C (`cstdlib` header for C++)
- `malloc()` allocates the memory on heap or free store
- `sizeof(int)` needs to be provided
- Pointer to allocated memory returned as `void*` – needs cast to `int*`
- Allocated memory is released by `free()` from heap or free store
- `calloc()` and `realloc()` also available in both languages

Program 10.02/03: operator new & delete: Dynamic memory management in C++

- C++ introduces operators `new` and `delete` to dynamically allocate and de-allocate memory:

Functions `malloc()` & `free()`

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;
    cout << *p;  // Prints: 5

    free(p);
}
```

- Function `malloc()` for allocation on heap
- `sizeof(int)` needs to be provided
- Allocated memory returned as `void*`
- *Casting* to `int*` *needed*
- *Cannot be initialized*
- Function `free()` for de-allocation from heap
- Library feature – header `cstdlib` needed

`operator new` & `operator delete`

```
#include <iostream>

using namespace std;

int main() {
    int *p = new int(5);

    cout << *p;  // Prints: 5

    delete p;
}
```

- `operator new` for allocation on heap
- *No size* specification needed, *type suffices*
- Allocated memory returned as `int*`
- *No casting needed*
- *Can be initialized*
- `operator delete` for de-allocation from heap
- Core language feature – no header needed

Program 10.02/04: Functions: operator new() & operator delete()

- C++ also allows `operator new()` and `operator delete()` functions to dynamically allocate and de-allocate memory:

Functions `malloc()` & `free()`

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
```

- Function `malloc()` for allocation on heap
- Function `free()` for de-allocation from heap

Functions `operator new()` & `operator delete()`

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

int main() {
    int *p = (int *)operator new(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    operator delete(p);
}
```

- Function `operator new()` for allocation on heap
- Function `operator delete()` for de-allocation from heap

There is a major difference between `operator new` and function `operator new()`. We explore this angle later

Program 10.05/06: `new[]` & `delete[]`: Dynamically managed Arrays in C++

Functions `malloc()` & `free()`

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

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "      ";

    free(a);
}
-----
a[0] = 10      a[1] = 20      a[2] = 30
```

- Allocation by `malloc()` on heap
- # of elements implicit in size passed to `malloc()`
- Release by `free()` from heap

`operator new[]` & `operator delete[]`

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


int main() {
    int *a = new int[3];
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "      ";

    delete [] a;
}
-----
a[0] = 10      a[1] = 20      a[2] = 30
```

- Allocation by `operator new[]` (**different from** `operator new`) on heap
- # of elements explicitly passed to `operator new[]`
- Release by `operator delete[]` (**different from** `operator delete`) from heap

Program 10.07: Operator new(): Placement new in C++


```
#include <iostream>
using namespace std;
int main() { unsigned char buf[sizeof(int)* 2]; // Byte buffer on stack
    // placement new in buffer buf
    int *pInt = new (buf) int (3);
    int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *)(buf + 0);           // *pInt in buf[0] to buf[sizeof(int)-1]
    int *qBuf = (int *)(buf + sizeof(int)); // *qInt in buf[sizeof(int)] to buf[2*sizeof(int)-1]
    cout << "Buf Addr  Int Addr" << pBuf << "  " << pInt << endl << qBuf << "  " << qInt << endl;
    cout << "1st Int  2nd Int" << endl << *pBuf << "         " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl << rInt << "   " << *rInt << endl;
    delete rInt;            // delete integer from heap
    // No delete for placement new
}
-----
Buf Addr  Int Addr
001BFC50  001BFC50
001BFC54  001BFC54
1st Int  2nd Int
3         5
Heap Addr  3rd Int
003799B8   7
```

- Placement operator new takes a buffer address to place objects
- These are not dynamically allocated on heap – may be allocated on stack or heap or static, wherever the buffer is located
- Allocations by Placement operator new must not be deleted

Mixing Allocators and De-allocators of C and C++

- Allocation and De-allocation must correctly match.
  - Do not free the space created by `new` using `free()`
  - And do not use `delete` if memory is allocated through `malloc()`

  These may result in memory corruption

Allocator	De-allocator
`malloc()`	`free()`
`operator new`	`operator delete`
`operator new[]`	`operator delete[]`
`operator new()`	No delete

- Passing `NULL` pointer to `delete` operator is secure
- Prefer to use only `new` and `delete` in a C++ program
- The `new` operator allocates exact amount of memory from Heap or Free Store
- `new` returns the given pointer type – no need to typecast
- `new`, `new[ ]` and `delete`, `delete[]` have separate semantics

Program 10.08: Overloading operator new and operator delete


```
#include <iostream>
#include <stdlib.h>
using namespace std;

void* operator new(size_t n) { // Definition of Operator new
    cout << "Overloaded new" << endl;
    void *ptr = malloc(n);     // Memory allocated to ptr. Can be done by function operator new()
    return ptr;
}
void operator delete(void *p) { // Definition of operator delete
    cout << "Overloaded delete" << endl;
    free(p);                    // Allocated memory released. Can be done by function operator delete()
}
int main() { int *p = new int; // Calling overloaded operator new
    *p = 30;          // Assign value to the location
    cout << "The value is :" << *p << endl;
    delete p;         // Calling overloaded operator delete
}
-----
Overloaded new
The value is :  30
Overloaded delete
```

- operator new overloaded
- The first parameter of overloaded operator new must be size_t
- The return type of overloaded operator new must be void*
- The first parameter of overloaded operator delete must be void*
- The return type of overloaded operator delete must be void
- More parameters may be used for overloading
- operator delete should not be overloaded (usually) with extra parameters

Program 10.09: Overloading operator new[] and operator delete[]

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

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}
void operator delete[] (void *ss) {
    operator delete(ss);
}
int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (unsigned int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
}
-----
p = 19421992
##########
```

- operator new[] overloaded with initialization
- The first parameter of overloaded operator new[] must be size_t
- The return type of overloaded operator new[] must be void*
- Multiple parameters may be used for overloading
- operator delete [] should not be overloaded (usually) with extra parameters

Module Summary

- Introduced `new` and `delete` for dynamic memory management in C++
- Understood the difference between `new`, `new[]` and `delete`, `delete[]`
- Compared memory management in C with C++
- Explored the overloading of `new`, `new[]` and `delete`, `delete[]` operators

Module 11: Programming in C++

Classes and Objects

Instructors: Abir Das and Jibesh Patra

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

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

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

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

1. Classes
2. Objects
3. Data Members
   - Complex
   - Rectangle
   - Stack
4. Member Functions
   - Complex
   - Rectangle
   - Stack
5. `this` Pointer
6. State of an Object
   - Rectangle
   - Stack
7. Module Summary

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

- 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

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
- 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
- Data members are accessed using '.' operator.
- `class` *aggregates* and *helps build a User-defined Data Type (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 are of user-defined data types

Program 11.05/06: Stacks: Attributes

C Program

```c
// 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

```cpp
// 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

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*

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: Rectangle

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

```
// 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.pop();
// 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 } }
```

Module 12: Programming in C++

Access Specifiers

Instructors: Abir Das and Jibesh Patra

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

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

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

by **Prof. Partha Pratim Das**

Module Objectives

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

Module Outline

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

- 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

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 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

- 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

- 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;

// initialise i
int i = 5;

// print i
cout << i;



// add two ints
int i = 5, j = 6;
i+j;
```

```
// declare c to be of Complex type
Complex c;

// initialise the real and imaginary components of c
Complex c = { 4, 5 };

// 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
```

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