Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers
Examples

Information Hiding

Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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

1. Access Specifiers
   - Access Specifiers: Examples
2. Information Hiding
3. Information Hiding: Stack Example
   - Stack (public)
     - Risky
   - Stack (private)
     - Safe
   - Interface and Implementation
4. Get-Set Idiom
5. Encapsulation
6. Class as a Data-type
7. Module Summary

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers

Examples

Information Hiding

Stack Example

Stack (public)

Risky

Stack (private)

Safe

Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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	Private data, Public method

<pre>#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();
}</pre>
	<pre>#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();
	}</pre>

• `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 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

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers
Examples

Information Hiding

Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers

Examples

Information Hiding

Stack Example

Stack (public)

Risky

Stack (private)

Safe

Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

- 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

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers
Examples

Information Hiding

Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

Information Hiding: Stack Example

Program 12.03/04: Stack: Implementations using public data

Using dynamic array	Using vector

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

Module M12

Partha Pratim Das

Objectives & Outlines
Access Specifiers
Examples
Information Hiding
Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation
Get-Set Idiom
Encapsulation
Class as a Data-type
Module Summary

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();
    }
}
```

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers

Examples

Information Hiding

Stack Example

Stack (public)

Risky

Stack (private)

Safe

Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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;
}
```

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers

Examples

Information Hiding

Stack Example

Stack (public)

Risky

Stack (private)

Safe

Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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++.

Module M12

Partha Pratim Das

Objectives & Outlines

Access Specifiers
Examples

Information Hiding

Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation

Get-Set Idiom

Encapsulation

Class as a Data-type

Module Summary

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 M12
Partha Pratim Das
Objectives & Outlines
Access Specifiers
Examples
Information Hiding
Stack Example
Stack (public)
Risky
Stack (private)
Safe
Interface and Implementation
Get-Set Idiom
Encapsulation
Class as a Data-type
Module Summary

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