Module 20

Instructors: Abir Das and Jibesh Patra

Module 20: Programming in C++

Namespaces

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 `namespace` as a free scoping mechanism to organize code better

Module Outline

namespace Fundamental

- A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it
- It is used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries
- namespace provides a class-like modularization without class-like semantics
- Obliviates the use of File Level Scoping of C (file) static
- A good resource for File Level Scoping in C Something Linky

Program 20.01: namespace Fundamental

- Example:

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

namespace MyNameSpace {
    int myData;                                                    // Variable in namespace
    void myFunction() { cout << "MyNameSpace myFunction" << endl; } // Function in namespace
    class MyClass { int data;                                      // Class in namespace
    public:
        MyClass(int d) : data(d) { }
        void display() { cout << "MyClass data = " << data << endl; }
    };
}
int main() {
    MyNameSpace::myData = 10;      // Variable name qualified by namespace name
    cout << "MyNameSpace::myData = " << MyNameSpace::myData << endl;

    MyNameSpace::myFunction();     // Function name qualified by namespace name

    MyNameSpace::MyClass obj(25); // Class name qualified by namespace name
    obj.display();
}
```

- A name in a namespace is prefixed by the name of it
- Beyond scope resolution, all namespace items are treated as global

Scenario 1: Redefining a Library Function Program 20.02

- cstdlib has a function `int abs(int n);` that returns the absolute value of parameter n
- You need a special `int abs(int n);` function that returns the absolute value of parameter n if n is between -128 and 127. Otherwise, it returns 0
- **Once you add your abs, you cannot use the abs from library! It is hidden and gone!**
- **namespace comes to your rescue**

Name-hiding: abs()

```cpp
#include <iostream>
#include <cstdlib>

int abs(int n) {
    if (n < -128) return 0;
    if (n > 127) return 0;
    if (n < 0) return -n;
    return n;
}

int main() { std::cout << abs(-203) << " "
         << abs(-6) << " "
         << abs(77) << " "
         << abs(179) << std::endl;
    // Output: 0 6 77 0
}
```

namespace: abs()

```cpp
#include <iostream>
#include <cstdlib>
namespace myNS {
    int abs(int n) {
        if (n < -128) return 0;
        if (n > 127) return 0;
        if (n < 0) return -n;
        return n;
    }
}
int main() { std::cout << myNS::abs(-203) << " "
        << myNS::abs(-6) << " "
        << myNS::abs(77) << " "
        << myNS::abs(179) << std::endl;
    // Output: 0 6 77 0
    std::cout << abs(-203) << " " << abs(-6) << " "
        << abs(77) << " " << abs(179) << std::endl;
    // Output: 203 6 77 179
}
```

Scenario 2: Students' Record Application: The Setting Program 20.03

- An organization is developing an application to process students records
- `class St` for Students and `class StReg` for list of Students are:

```
#include <iostream>
#include <cstring>
using namespace std;
class St { public: // A Student
    typedef enum GENDER { male = 0, female };
    St(char *n, GENDER g) : name(strcpy(new char[strlen(n) + 1], n)), gender(g) { }
    void setRoll(int r) { roll = r; }      // Set roll while adding the student
    GENDER getGender() { return gender; } // Get the gender for processing
    friend ostream& operator<< (ostream& os, const St& s) { // Print a record
        cout << ((s.gender == St::male) ? "Male " : "Female ")
             << s.name << " " << s.roll << endl;
        return os;
    }
private: char *name; GENDER gender; // name and gender provided for the student
         int roll;                  // roll is assigned by the system
};
class StReg {      // Students' Register
    St **rec; /* List of students */ int nStudents; // Number of student
public: StReg(int size) : rec(new St*[size]), nStudents(0) { }
    void add(St* s) { rec[nStudents] = s; s->setRoll(++nStudents); }
    St *getStudent(int r) { return (r == nStudents + 1) ? 0 : rec[r - 1]; }
};
```

- The classes are included in a header file `Students.h`

Scenario 2: Students' Record Application: Team at Work Program 20.03

- Two engineers – Sabita and Niloy – are assigned to develop processing applications for male and female students respectively. Both are given the Students.h file
- The lead Purnima of Sabita and Niloy has the responsibility to integrate what they produce and prepare a single application for both male and female students. The engineers produce:

Processing for males by Sabita

```
////////////// App1.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
void ProcSt() { cout << "MALE STUDENTS: " << endl;
    int r = 1; St *s;
    while (s = reg->getStudent(r++))
        if (s->getGender() == St::male) cout << *s;
    cout << endl << endl;
    return;
}
////////////// Main.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"
StReg *reg = new StReg(1000);
int main()
{ St s("Ravi", St::male); reg->add(&s); ProcSt(); }
```

Processing for females by Niloy

```
////////////// App2.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"
extern StReg *reg;
void ProcSt() { cout << "FEMALE STUDENTS: " << endl;
    int r = 1; St *s;
    while (s = reg->getStudent(r++))
        if (s->getGender() == St::female) cout << *s;
    cout << endl << endl;
    return;
}
////////////// Main.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"
StReg *reg = new StReg(1000);
int main()
{ St s("Rhea", St::female); reg->add(&s); ProcSt(); }
```

Scenario 2: Students' Record Application: Integration Nightmare: Program 20.03

- To integrate, Purnima prepares the following `main()` in her `Main.cpp` where she intends to call the processing functions for males (as prepared by Sabita) and for females (as prepared by Niloy) one after the other:

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

void ProcSt(); // Function from App1.cpp by Sabita
void ProcSt(); // Function from App2.cpp by Niloy

StReg *reg = new StReg(1000);

int main() {
    St s1("Rhea", St::female); reg->add(&s1);
    St s2("Ravi", St::male); reg->add(&s2);

    ProcSt();  // Function from App1.cpp by Sabita
    ProcSt();  // Function from App2.cpp by Niloy
}
```

- **But the integration failed due to name clashes**
- **Both use the same signature `void ProcSt();` for their respective processing function. Actually, they have several functions, classes, and variables in their respective development with the same name and with same / different purposes**
- **How does Purnima perform the integration without major changes in the codes? – namespace**

Scenario 2: Students' Record Application: Wrap in namespace Program 20.03

- Introduce two namespaces – App1 for Sabita and App2 for Niloy
- Wrap the respective codes:

Processing for males by Sabita

```
////////////// App1.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"

extern StReg *reg;

namespace App1 {
    void ProcSt() {
        cout << "MALE STUDENTS: " << endl;
        int r = 1;
        St *s;

        while (s = reg->getStudent(r++))
        if (s->getGender() == St::male)
                cout << *s;

        cout << endl << endl;
        return;
    }
};
```

Processing for females by Niloy

```
////////////// App2.cpp //////////////
#include <iostream>
using namespace std;
#include "Students.h"

extern StReg *reg;

namespace App2 {
    void ProcSt() {
        cout << "FEMALE STUDENTS: " << endl;
        int r = 1;
        St *s;

        while (s = reg->getStudent(r++))
        if (s->getGender() == St::female)
                cout << *s;

        cout << endl << endl;
        return;
    }
};
```

Scenario 2: Students' Record Application: A Good Night's Sleep Program 20.03

- Now the integration gets smooth:

```
using namespace std;

#include "Students.h"

namespace App1 { void ProcSt(); } // App1.cpp by Sabita

namespace App2 { void ProcSt(); } // App2.cpp by Niloy

StReg *reg = new StReg(1000);

int main() {
    St s1("Ravi", St::female); reg->add(&s1);
    St s2("Rhea", St::male); reg->add(&s2);

    App1::ProcSt(); // App1.cpp by Sabita
    App2::ProcSt(); // App2.cpp by Niloy

    return 0;
}
```

- **Clashing names are made distinguishable by distinct** `names`

Program 20.04: Nested namespace

- A namespace may be nested in another namespace

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

int data = 0;               // Global name ::

namespace name1 {
    int data = 1;           // In namespace name1
    namespace name2 {
        int data = 2;       // In nested namespace name1::name2
    }
}

int main() {
    cout << data << endl;               // 0
    cout << name1::data << endl;        // 1
    cout << name1::name2::data << endl; // 2

    return 0;
}
```

Program 20.05: Using using namespace and using for shortcut

- Using using namespace we can avoid lengthy prefixes

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

namespace name1 {
    int v11 = 1;
    int v12 = 2;
}
namespace name2 {
    int v21 = 3;
    int v22 = 4;
}

using namespace name1;  // All symbols of namespace name1 will be available
using name2::v21;       // Only v21 symbol of namespace name2 will be available

int main() {
    cout << v11 << endl;        // name1::v11
    cout << name1::v12 << endl; // name1::v12
    cout << v21 << endl;        // name2::v21
    cout << name2::v21 << endl; // name2::v21
    cout << v22 << endl;        // Treated as undefined
}
```

Program 20.06: Global namespace

- using or using namespace hides some of the names

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

int data = 0;       // Global Data

namespace name1 {
    int data = 1;   // namespace Data
}

int main() {
    using name1::data;

    cout << data << endl;          // 1 // name1::data -- Hides global data
    cout << name1::data << endl; // 1
    cout << ::data << endl;        // 0 // ::data -- global data
}
```

- Items in Global namespace may be accessed by scope resolution operator (::)

Program 20.07: std Namespace

- Entire C++ Standard Library is put in its own namespace, called std

Without using using std

```
#include <iostream>


int main() {
    int num;
    std::cout << "Enter a value: " ;
    std::cin >> num;
    std::cout << "value is: " ;
    std::cout << num ;
}
```

- Here, cout, cin are explicitly qualified by their namespace. So, to write to standard output, we specify std::cout; to read from standard input, we use std::cin
- It is useful if a few library is to be used; no need to add entire std library to the global namespace

With using using std

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

int main() {
    int num;
    cout << "Enter a value: " ;
    cin >> num;
    cout << "value is: " ;
    cout << num ;
}
```

- By the statement using namespace std; std namespace is brought into the current namespace, which gives us direct access to the names of the functions and classes defined within the library without having to qualify each one with std::
- When several libraries are to be used it is a convenient method

Program 20.08: namespaces are Open

- namespace are open: New Declarations can be added

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

namespace open // First definition
{ int x = 30; }

namespace open // Additions to the last definition
{ int y = 40; }

int main() {
    using namespace open; // Both x and y would be available

    x = y = 20;
    cout << x << "  " << y ;
}
```

Output: 20 20

namespace vis-a-vis class

namespace	class
• Every namespace is not a class	• Every class defines a namespace
• A namespace can be reopened and more declaration added to it	• A class cannot be reopened
• No instance of a namespace can be created	• A class has multiple instances
• using-declarations can be used to short-cut namespace qualification	• No using-like declaration for a class
• A namespace may be unnamed	• An unnamed class is not allowed

Lexical Scope

- The scope of a name binding – an association of a name to an entity, such as a variable – is the part of a computer program where the binding is valid: where the name can be used to refer to the entity
- C++ supports a variety of scopes:
  - Expression Scope – restricted to one expression, mostly used by compiler
  - Block Scope – create local context
  - Function Scope – create local context associated with a function
  - Class Scope – context for data members and member functions
  - Namespace Scope – grouping of symbols for code organization
  - File Scope – limit symbols to a single file
  - Global Scope – outer-most, singleton scope containing the whole program

Lexical Scope

- Scopes may be named or Unnamed
  - Named Scope – Option to refer to the scope from outside
    - Class Scope – class name
    - Namespace Scope – namespace name or unnamed
    - Global Scope – "::"
  - Unnamed Scope
    - Expression Scope
    - Block Scope
    - Function Scope
    - File Scope
- Scopes may or may not be nested
  - Scopes that may be nested
    - Block Scope
    - Class Scope
    - Namespace Scope
  - Scopes that cannot be nested
    - Expression Scope
    - Function Scope – may contain Class Scopes
    - File Scope – will contain several other scopes
    - Global Scope – will contain several other scopes

Module Summary

- Understood `namespace` as a scoping tool in c++
- Analyzed typical scenarios that `namespace` helps to address
- Studied several features of `namespace`
- Understood how `namespace` is placed in respect of different lexical scopes of C++

Module 21: Programming in C++

Inheritence (Part 1)

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 ISA Relationship in OOAD and understand how hierarchy can be created in C++ with Inheritance

Module Outline

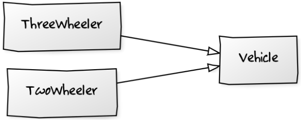
1. ISA Relationship
2. Inheritance in C++
   - Phones
   - Semantics
3. Module Summary

ISA Relationship

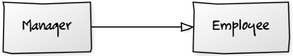
- We often find one object is a *specialization* / *generalization* of another
- OOAD models this using **ISA** relationship
- C++ models **ISA** relationship by *Inheritance* of classes

ISA Relationship

- Rose **ISA** Flower
  - Rose has the properties of Flower – like fragrance, having petals etc.
  - Rose has some additional properties – like rosy fragrance
  - Rose is a *specialization* of Flower
  - Flower is a *generalization* of Rose
- Red Rose **ISA** Rose
  - Red Rose has the properties of Rose – like rosy fragrance etc.
  - Red Rose has some additional properties – like it is red
  - Red Rose is a *specialization* of Rose
  - Rose is a *generalization* of Red Rose



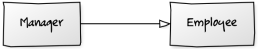
- TwoWheeler **ISA** Vehicle; ThreeWheeler **ISA** Vehicle



- Manager **ISA** Employee

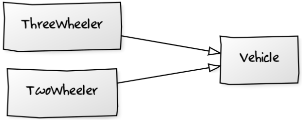
Inheritance in C++: Hierarchy

- Manager **ISA** Employee **[Single Inheritance]**




```
class Employee;                   // Base Class = Employee
class Manager: public Employee; // Derived Class = Manager; Base Class = Employee
```

- TwoWheeler **ISA** Vehicle; ThreeWheeler **ISA** Vehicle **[Hybrid Inheritance]**




```
class Vehicle;                          // Base Class = Vehicle -- Root
class TwoWheeler: public Vehicle;       // Derived Class = TwoWheeler; Base Class = Vehicle
class ThreeWheeler: public Vehicle;     // Derived Class = ThreeWheeler; Base Class = Vehicle
```

- Red Rose **ISA** Rose **ISA** Flower **[Multi-Level Inheritance]**


```
class Flower;                 // Base Class = Flower -- Root
class Rose: public Flower;    // Derived Class = Rose; Base Class = Flower
class RedRose: public Rose;   // Derived Class = RedRose; Base Class = Rose
```

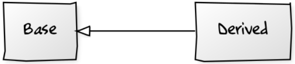
Inheritance in C++: Phones

- **Landline Phone**
  - Call: By dial / keyboard
  - Answer
  - Caller ID (with special attached device)
- **Mobile Phone**
  - Call: By keyboard – shows number
    - By Number
    - By Name
  - Answer
  - Caller ID
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
- **Smart Phone**
  - Call: By touchscreen – shows number & photo
    - By Number
    - By Name
  - Answer
  - Caller ID
  - Redial
  - Set Ring Tone
  - Add Contact
    - Number
    - Name
    - Photo

- There exists a substantial overlap between the functionality of the phones
- A mobile phone is more capable than a land line phone and can perform (almost) all its functions
- A smart phone is more capable than a mobile phone and can perform (almost) all its functions
- **These phones belong to a Specialization / Generalization Hierarchy**

Inheritance in C++: Semantics

- Derived **ISA** Base




```
class Base;                    // Base Class = Base
class Derived: public Base;    // Derived Class = Derived
```

- Use keyword `public` after class name to denote inheritance
- Name of the Base class follow the keyword

Public inheritance means "is-a." Everything that applies to base classes must also apply to derived classes, because every derived class object is a base class object

– *Scott Meyers in Item 32, Effective C++ (3rd. Edition)*

Inheritance in C++: Semantics

- Derived **ISA** Base
- **Data Members**
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- **Member Functions**
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* protected members of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

Module Summary

- Understood Hierarchy or ISA Relationship in OOAD
- Introduced the Semantics of Inheritance in C++

Module 22: Programming in C++

Inheritence (Part 2): Override and Overload

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 how inheritance impacts data members and member functions
- Introduce overriding of member function and its interactions with overloading

Module Outline

Inheritance in C++: Semantics

- Derived **ISA** Base
- **Data Members**
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- **Member Functions**
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* protected members of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

Data Members

- Derived **ISA** Base
- **Data Members**
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- **Object Layout**
  - Derived class *layout* contains an instance of the Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Base class instance and Derived class members

Object Layout

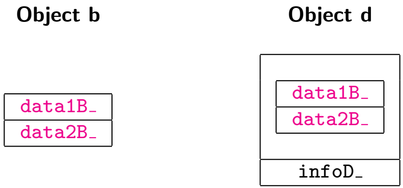
```
class B { // Base Class
    int data1B_;
public:
    int data2B_;
    // ...
};

class D: public B { // Derived Class
    // Inherits B::data1B_
    // Inherits B::data2B_
    int infoD_; // Adds D::infoD_
public:
    // ...
};

B b; // Base Class Object

D d; // Derived Class Object
```

Object Layout



- d *cannot access* data1B_ even though is a part of d!
- d *can access* data2B_

Member Functions

- Derived **ISA** Base
- **Member Functions**
  - Derived class *inherits* all member functions of Base class
    - **Note**: Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- **Static Member Functions**
  - Derived class *does not inherit* the static member functions of Base class
- **Friend Functions**
  - Derived class *does not inherit* the friend functions of Base class

Overrides and Overloads

Inheritance

```
class B { public: // Base Class
    void f(int i);
    void g(int i);
};
class D: public B { public: // Derived Class
    // Inherits B::f(int)


    // Inherits B::g(int)

};
B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

Override & Overload

```
class B { public: // Base Class
    void f(int);
    void g(int i);
};
class D: public B { public: // Derived Class
    // Inherits B::f(int)
    void f(int);       // Overrides B::f(int)
    void f(string&);  // Overloads B::f(int)
    // Inherits B::g(int)
    void h(int i);     // Adds D::h(int)
};
B b;
D d;

b.f(1);       // Calls B::f(int)
b.g(2);       // Calls B::g(int)

d.f(3);       // Calls D::f(int)
d.g(4);       // Calls B::g(int)

d.f("red");  // Calls D::f(string&)
d.h(5);       // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string&) overloads B::f(int)

Comparison of Overloading vis-a-vis Overriding

Basis	Function Overloading	Function Overriding
Name of Function	• All overloads have the same function name	• All overrides have the same function name
Signature	• Function signatures must be different	• Function signatures are same
Type of Function	• Can be global, friend, static or non-static member function	• Must be a non-static member function - non-virtual or virtual
Inheritance	• Can happen with or without inheritance	• Happens only with inheritance
Polymorphism	• Static (Compile time)	• Static (Compile time) or Dynamic (Runtime)
Scope	• Overloaded functions are in the same scope	• Functions are in different scopes (base clase and derived class)
Purpose	• To have multiple functions with same name that act differently depending on parameters	• To perform additional or different tasks than the base class function
Constructor	• Constructors can be overloaded	• Constructors cannot be overridden
Destructor	• The destructor cannot be overloaded	• The destructor cannot be overridden
Usage	• Can be overloaded multiple times	• Can be overridden once in the derived class

Module Summary

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading

Module 23

Instructors: Abir Das and Jibesh Patra

Module 23: Programming in C++

Inheritence (Part 3): Constructors, Destructors & Object Lifetime

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 Recap

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading

Module Objectives

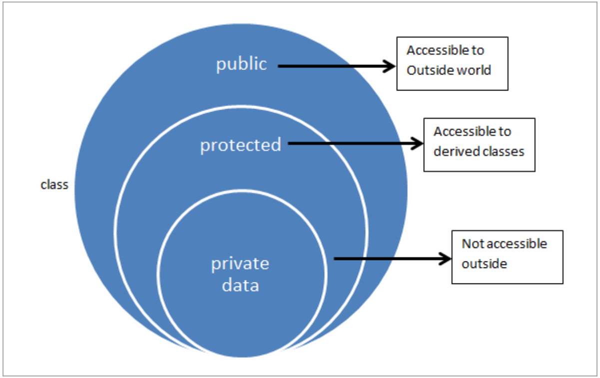
- Understand `protected` access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy

Module Outline

Inheritance in C++: Semantics

- Derived **ISA** Base
- **Data Members**
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- **Member Functions**
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* protected members of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

`protected` Access



`protected` **Access**

Access Members of Base: `protected` Access

- Derived **ISA** Base
- **Access Specification**
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* public members of Base class
- **protected Access Specification**
  - A new protected access specification is introduced for Base class
  - Derived class *can access* protected members of Base class
  - No other class or global function *can access* protected members of Base class
  - A protected member in Base class is like public in Derived class
  - A protected member in Base class is like private in other classes or global functions

protected Access

private Access

```
class B {
private:   // Inaccessible to child
           // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout << data_ << endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", "; // Inaccessible
        cout << info_ <<endl;
    }
};
B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to all

b.Print();
d.Print();
```

- D::Print() *cannot access* B::data_ as it is private

protected Access

```
class B {
protected: // Accessible to child
           // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_ << ", "; // Accessible
        cout << info_ << endl;
    }
};
B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to others

b.Print();
d.Print();
```

- D::Print() *can access* B::data_ as it is protected

Why do we need `protected` access?

- **Handling Encapsulation**: Encapsulation, the first principle of OOAD, can be enforced in a single class by `private` and `public` access specifiers
    - `private` hides the state (data) of the object and `public` allows the service (method / interface) to be exposed
    - We fine-grain this by `get/set` paradigm to achieve effective information hiding
    - Further `friend` provides a way to sneak through encapsulation for *easy yet safe coding*
- **Encapsulation-Inheritance Conflict**: The above approach to Encapsulation conflicts with Inheritance, the second principle of OOAD
  What should be the access specification for data members of a Base class?
    - If they are `public`, the encapsulation is lost for the base class objects
    - If they are `private`, even the derived class methods cannot access them
    - So the derived class object contains the base class data members but cannot access them
      Notably, the state of the derived class object depends on the state of its base class part
    - The `get/set` paradigm does not work as it is clumsy and creates an encapsulation hole like `public` if used for all data members
- **Solution**: The `protected` access specifier provides a neat solution by making `protected` base class members available to the derived class while being hidden from the rest of the world
- **Caveat**: `protected` specifier still does not solve all situations and we need to use `friend` to provide a way to sneak through encapsulation as the next example illustrates

Streaming

Streaming in B

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) { os << "D Object: ";
    //    os << d.data_ << ' ' << d.info_ << endl;
    //    return os;
    //}
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a B object
```

```
B Object: 0
B Object: 1
```

- d printed as a B object; info_ missing

Streaming in B & D

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) { os << "D Object: ";
        os << d.data_ << ' ' << d.info_ << endl;
        return os;
    }
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a D object
```

```
B Object: 0
D Object: 1 2
```

- d printed as a D object as expected

Constructor and Destructor

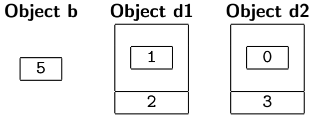
- Derived **ISA** Base
- **Constructor-Destructor**
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - Derived class *cannot override* or *overload* a Constructor or the Destructor of Base class
- **Construction-Destruction**
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

Constructor and Destructor

```
class B { protected: int data_; public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_; public:
    D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i)              // ctor-2: Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};

B b(5);
D d1(1, 2);     // ctor-1: Explicit construction of Base
D d2(3);        // ctor-2: Default construction of Base
```

Object Layout



Object Lifetime

Module 23

Instructors: Abir Das and Jibesh Patra

Objectives & Outlines

Inheritance in C++

protected Access

Streaming

Constructor & Destructor

Object Lifetime

Module Summary

```
class B { protected: int data_; public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_; public:
    D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i)              // ctor-2: Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};
B b;
D d1(1, 2);    // ctor-1: Explicit construction of Base
D d2(3);       // ctor-2: Default construction of Base
```

Construction O/P

```
B::B(int): 0          // Object b
B::B(int): 1          // Object d1
D::D(int, int): 1, 2 // Object d1
B::B(int): 0          // Object d2
D::D(int): 0, 3       // Object d2
```

Destruction O/P

```
D::~D(): 0, 3      // Object d2
B::~B(): 0         // Object d2
D::~D(): 1, 2      // Object d1
B::~B(): 1         // Object d1
B::~B(): 0         // Object b
```

- **First construct base class object, then derived class object**
- **First destruct derived class object, then base class object**

Module Summary

- Understood the need and use of `protected` Access specifier
- Discussed the Construction and Destruction process of class hierarchy and related Object Lifetime