

INHERITANCE

Inheritance in Object-oriented design (OOD) represents the "**is-a**" ("**kind-of**") relationship.

A "*Kind of*" or "*is a*" Relationship:

We know that desktop PCs, laptops, tablets, and servers are (kinds of) computers.

- All of them have some common properties, e.g., they have CPUs and memories.
- They also have some common abilities, e.g., running programs and storing data.

We can say "laptop **is a** computer" and "tablet is **a kind of** computer".

- Besides the common properties, they also have unique features.

  For example, a server has a magnetic disk and can process big data, a tablet has a touch-on screen, a smartphone can make phone calls, etc.

Other examples:

- Undergraduate students, master's students, and Ph.D. students are all students.

  They have **common attributes and abilities** (behavior, responsibility).
- The dean of the faculty **is a** professor.

  They have all the properties and abilities of a professor. Besides, the dean has additional administrative duties.

INHERITANCE (contd)

Generalization – Specialization:

- With the help of inheritance, we can create more “special” types (classes) of general types (classes).
- Special classes may have more members (data and methods) than general classes.
  - For example, the computer is a general type. All computers contain a CPU and memory.
  - A tablet is a special type of computer. In addition to CPU and memory, it contains a touch-on screen.
  - A server can run programs like all other computers. In addition, it can process big data.

Other Examples:

- Employee ← worker ← manager: A worker is an employee; a manager is a worker.
- Vehicle ← air vehicle ← helicopter: The vehicle is general, and the helicopter is special.
- Professor ← Dean: A dean is a professor; they can teach and research like a regular professor.

  In addition, they administrate faculty affairs.

Modification During Specialization:

- In OOP, **inheritance** enables the **modification** and **extension** of a class without changing its code.

  When we create more “special” types (classes) of general types, we can add new properties and abilities (new members) to the more specialized classes.

  In addition, we can also **modify some features** of the general type if necessary.
- The code of the existing class, called the ***base (parent) class***, is not modified.
- However, the new class, called the ***derived (child) class***, can
  - **use** all the features of the old one,
  - **add** new features (attributes and methods),
  - **modify** some features of the base class.

  For example, a manager is a worker.

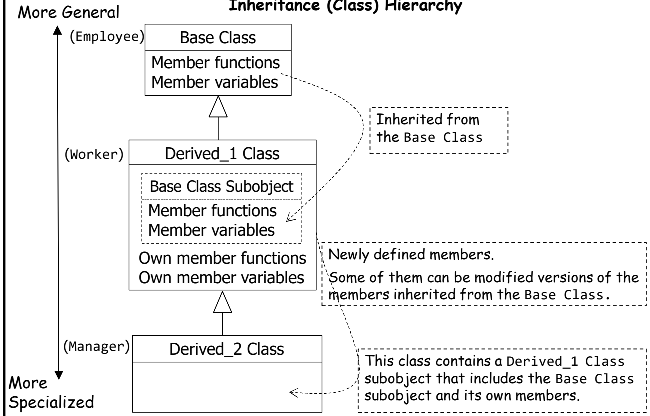
  Managers have all the workers' properties and some additional features.

  Workers have a procedure to calculate their salaries.

  Managers also have a procedure for salary calculation, but it may differ from the workers' procedure.

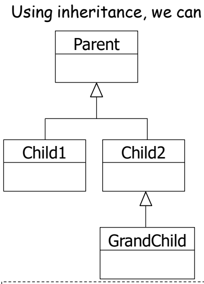
  The Manager type should modify the procedure derived from the general type Worker.

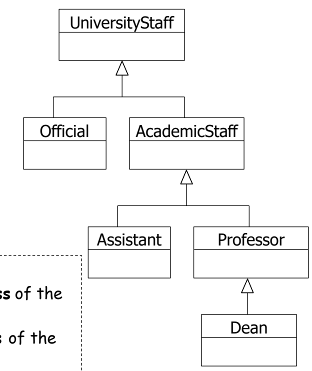
Inheritance (Class) Hierarchy



Inheritance (Class) Hierarchy (contd)

Using inheritance, we can create various class (type) hierarchies:





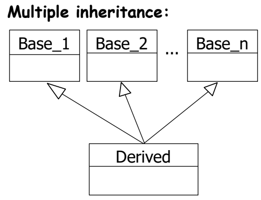
Terminology:

The **base (parent)** class is the **superclass** of the derived (child) class.

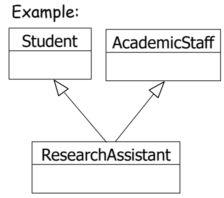
The **derived (child)** class is the **subclass** of the base (parent) class.

Inheritance (Class) Hierarchy (contd)

Multiple inheritance:



Example:



- A research assistant is a student and an academic staff.
- A research assistant has all the features of a student and an academic staff.

- C++ supports multiple inheritance.
- Some programming languages, like Java and C#, use a different mechanism (interfaces) instead of multiple inheritance.
- Multiple inheritance can introduce ambiguity and complexity in the code. It must be used judiciously and only when necessary (slide 7.55).

Aggregation, Composition: *has a* relation **vs.** **Inheritance:** *is a* relation

Although the objects of the derived class contain a subobject of the base class, this is not a composition (not has-a relationship).

Remember, **composition** in OOP models the real-world situation in which objects are composed (or part) of other objects.

For example, the triangle is composed of three points.

A triangle has points. A triangle is not a kind of point."

On the other hand, **inheritance** in OOP mirrors the concept that we call *generalization - specialization* in the real world.

When we model a company's officials, workers, managers, and researchers, we know that these are all specific types of a more general concept employee.

Every kind of employee has specific features: name, age, ID number, etc.

However, in addition to these general features, a researcher has a project they work on.

We can say, "The researcher is an employee"; we **cannot** say, "The researcher has an employee".

These relationships also have different effects in terms of programming.

We will cover these differences in the following slides.

Inheritance in C++

The simplest example of inheritance requires two classes: **a base class (parent class, superclass)** and **a derived class (child class, subclass)**.

The base class does not need any special syntax. On the other hand, the derived class must indicate that it is derived from the base class.

Example:

Assume that we need points with colors and related functions.

This is a specialized version of the `Point` class we already defined.

We do not need to define a new `ColoredPoint` class from scratch.

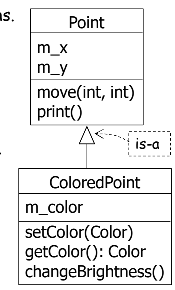
We can **reuse** the existing class `Point` and derive the new `ColoredPoint` class from it by adding only the new features.

ColoredPoint is a Point. is-a

Explained in 7.18

```
// Derived Class
class ColoredPoint : public Point {
    :          // Additional features
};
```

UML:



Example: `ColoredPoint` is a `Point`.

The existing base class `Point` does not have any special syntax.
Another programmer might have written it, or it may be a class from the library.

```
class Point {                    // Base Class (parent)
public:
  Point() = default;             // Default Constructor
  // Getters and setters
      :
  bool move(int, int);           // A method to move points
private:
  int m_x{MIN_x}, m_y{MIN_y};  // x and y coordinates
};
```

+ Inherited (added)

```
class ColoredPoint : public Point { // Derived Class (child)
public:
  ColoredPoint (Color);    // Constructor of the colored point
  Color getColor() const;  // Getter
  void setColor(Color);    // Setter
private:
  Color m_color;           // Color of the point
};
```

Additional features

Example: `ColoredPoint` is a `Point` (contd)

```
// Enumeration to define colors
enum class Color {Blue, Purple, Green, Red};

int main()
{
  ColoredPoint col_point1{ Color::Green };     // A green point
  col_point1.move(10, 20);    // move function is inherited from base Point
  col_point1.print();         // print function is inherited from base Point
  col_point1.setColor(Color::Blue); // New member function setColor
  if (col_point1.getColor() == Color::Blue) std::print("Color is Blue");
  else std::print("Color is not Blue");
```

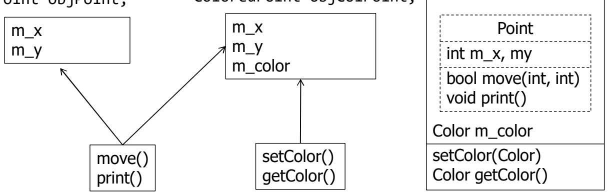
The objects of `ColoredPoint`, e.g., `col_point1`, can access `public` methods inherited from `Point` (e.g., `move` and `print`) and newly defined public methods of `ColoredPoint` (e.g., `getColor`).

Example: `ColoredPoint` is a `Point` (contd)

Objects in Memory:

An object of `Point`:
`Point objPoint;`

An object of `ColoredPoint`:
`ColoredPoint objColPoint;`



See Example e07_1a.cpp

Operator functions are also inherited

Example:

Assume that base class `Point` overloads the greater-than operator > to compare the distance of a point from zero (`0,0`) with a double literal.

```
bool Point::operator>(double in_distance) const {
  return sqrt(m_x * m_x + m_y * m_y) > in_distance;
}
```

This function is inherited by the derived class `ColoredPoint` and can be used by its objects.

```
int main()
{
   ColoredPoint col_point1{ Color::Green }; // A green point
   if(col_point1 > 50) ... ;
   else ... ;
    :
}
```

The operator function inherited from `Point` is used for `ColoredPoint`.

See Example e07_1b.cpp

Access Control

Remember: The `private` access specifier determines that members are totally private to the class; they cannot be accessed outside of the class.

- Private members of the `Base` class **cannot be accessed directly from the `Derived` class** that inherits them.

  For example, `m_x` and `m_y` are `private` members of the `Point` class.

  Private variables are inherited by the derived class `ColoredPoint`, but the methods of `ColoredPoint` cannot access `m_x` and `m_y` directly.

  ```
  void ColoredPoint::wrtX(int in_x) { m_x = in_x; } // Error! Private
  ```
- The derived class may access them only through the `public` interface of the base class, e.g., setters or the `move` function provided by the creator of the `Point` class.

  ```
  void ColoredPoint::wrtX(int in_x) { setX(in_x); }    // OK. Public
  ```
- The creator of the derived class (e.g., `ColoredPoint`) is a client programmer (user) of the base class (e.g., `Point`).
- Remember the data-hiding principle. It allows you to preserve the integrity of an object's state.

  It prevents accidental changes in the attributes of objects (see slide 3.14).

Access Control (contd)

Protected Members:

When we want to use inheritance, in addition to the `public` and `private` access specifiers for base class members, we can declare members as **`protected`**.

Without inheritance, the `protected` keyword has the same effect as the `private`.

Protected members cannot be accessed outside the class except for functions specified as `friend` functions.

If there is an inheritance, member functions of a derived class can access **`public`** and **`protected`** members of the base class but not **`private`** members.

Objects of a derived class can access only `public` members of the base class.

Access Specifier in Base	Accessible from Own Class	Accessible from Derived Class	Accessible from Objects (Outside Class)
`public`	yes	yes	yes
`protected`	yes	yes	no
`private`	yes	no	no

Protected Members (contd):

Example:

The base class `Point` has an ID as a **protected** data member.

```
class Point {
public:                                    All functions (also non-members) can access
  :
protected:                                 Members of the base and derived class
  string m_ID{}; // Protected member       can access
private:                                   Only the members of the Point can access
  int m_x{}, m_y{};
};
```

```
// Member function of the Derived Class ColoredPoint
// Colored Point access the protected member of the Base directly
void ColoredPoint::setAll(int in_x, int in_y, const string& in_ID,
                          Color in_color) {
   setX(in_x);       // calls the public method of the Base (Point)
   setY(in_y);       // calls the public method of the Base (Point)
// m_x = in_x;       // Error! m_x is private in Point
   m_ID = in_ID;     // OK. It can access the protected member directly
   m_color = in_color; // Its own member
}
```

See Example e07_2.cpp

Protected vs. Private Members

Remember the **data hiding** principle (see slide 3.14).

Public data is open to modification by any function anywhere in the program and should almost always be avoided.

Potential problems may be caused by protected members:

- Protected member variables have many of the same disadvantages as public ones.
- Anyone can derive one class from another and thus gain access to the base class's protected data.
- Extra code added to getter and setter functions in the base class to control access becomes useless because derived classes can bypass it.
- When the derived classes directly manipulate the member variables of a base class, changing the internal implementation of the base would also require changing all the derived classes.

When to use them:

In applications where speed is important, such as real-time systems, function calls to access private members are time-consuming.

In such systems, data may be defined as `protected` to allow derived classes to access data directly and faster.

Protected vs. Private Members (contd)

It is safer and more reliable if derived classes cannot access base class data directly.

Member variables of a class should always be `private` unless there is a good reason not to do so.

If code outside of the class requires access to member variables, add `public` or `protected` getter and/or setter **methods** to your class.

Example: The problem caused by `protected` members

- If the `m_x` and `m_y` members of the `Point` class are specified as `protected`, the limit checks in the setters, and the move function becomes useless.
- Methods of the derived class `ColoredPoint` can modify the coordinates of a point object directly and move it beyond the allowed limits.

See Example e07_3.cpp

```
// Colored Point access the coordinates directly
void ColoredPoint::setAll(int in_x, int in_y, ...) {
  m_x = in_x;        // It can access the protected member directly
  m_y = in_y;        // It can access the protected member directly
}
 colored_point1.setAll(-100, -500);   // moves beyond the limits
```


Base Class Access Specification

When we derive a new class from a base class, we provide an access specifier for the base class.

Example:

Base class specifier is public

```
class ColoredPoint : public Point {
};
```

- There are three possibilities for the base class access specifier:

  `public`, `protected`, or `private`.
- The base class access specifier does not affect how the derived class accesses the members of the base.
- It affects the access status of the inherited members in the derived class for the users (objects or subclasses) of that class.

  For example, if the base class specifier is `public`, the access status of the inherited members remains unchanged.

  Thus, inherited `public` members are `public`, and the objects of the derived class can access them.

  In the example `e07_1.cpp`, the objects of the `ColoredPoint` class can call the `public` methods of the `Point` class.

  ```
  col_point1.move(10, 20);  // move is public in Point and ColoredPoint
  ```

Base Class Access Specification

Public inheritance (or sometimes *public derivation*):

- The access status of the inherited members remains unchanged.
- Inherited `public` members are `public`, and inherited `protected` members are `protected` in a derived class.

Protected inheritance (*protected derivation*):

- Both `public` and `protected` members of a base class are inherited as `protected` members.
- They can be accessed if they are inherited in another derived (grandchild) class.
- The objects of the derived class cannot access them.

Private inheritance (*private derivation*):

- When the base class specifier is `private`, inherited `public` and `protected` members become `private` in the derived class.
- They are still accessible by member functions of the derived class but cannot be accessed if they are inherited in another derived (grandchild) class.
- The objects of the derived class cannot access them either.

Base Class Access Specification (contd)

`class Derived : public Base`

public Base Class	Derived Class
public members	public
protected members	protected
private members	inherited but not accessible

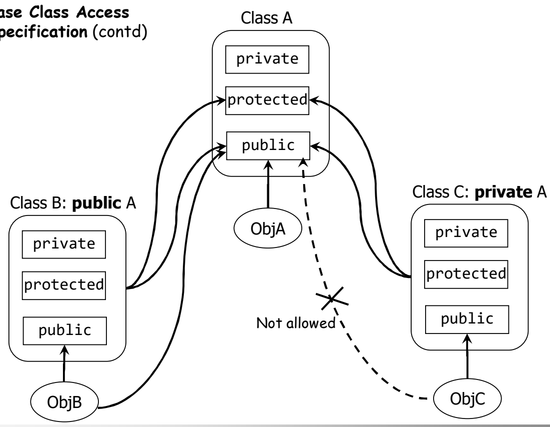
`class Derived : protected Base`

protected Base Class	Derived Class
public members	protected
protected members	protected
private members	inherited but not accessible

`class Derived : private Base`

private Base Class	Derived Class
public members	private
protected members	private
private members	inherited but not accessible

Base Class Access Specification (contd)



Redefining Access Specifications:

- When you inherit privately, all the **public** members of the base class become **private** for the users of the derived class.
- After a `private` derivation, the creator of the derived class can make `public` members of the base class visible again by writing their names (no arguments or return values) along with the **using** keyword into the **public:** section of the derived class.

Example:

```
class Point {                // Base Class (parent)
public:
  bool move(int, int);
  void print() const;
   :
};
class ColoredPoint : private Point { // Private inheritance
public:
  using Point::print;   // print() of Point is public again
   :
};
```

```
ColoredPoint cp;
cp.move(10, 20);   // Error! move is private
cp.print();        // OK. Print is public again
```


Example: Private inheritance

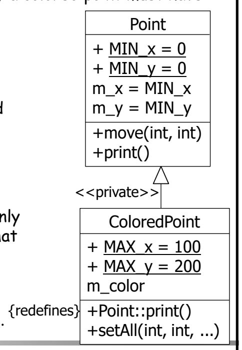
Problem:

- Assume that the `Point` class supports only lower limits, `MIN_x` and `MIN_y`.
- According to the requirements, the coordinates of a colored point must have lower and upper limits.

Solution:

- The creator of the `ColoredPoint` class must **privately** inherit members of the `Point` class (specifically the setters and the `move` method) and add upper limits.

  So, the users (objects) of the `ColoredPoint` class cannot call the `move` function or setters inherited from `Point`, which only checks the lower limits.
- Now, the objects of the `ColoredPoint` class can only call `public` methods provided by the creator of that class, e.g., `setAll()` that checks the upper limits.
- Moreover, the creator of the derived class can redefine the access specification of the `print` method to make it visible again for the class users.



Example: Private Inheritance The `ColoredPoint` class has lower and upper limits

```
class ColoredPoint : private Point {      //Private inheritance
public:
  void setAll(int, int, const string&,  Color);
  using Point::print;  // print() of Point is public again
     // Upper Limits of x and y coordinates (new attributes)
  static inline const int MAX_x{100}; // MAX_x = 100
  static inline const int MAX_y{200}; // MAX_y = 200
private:
  Color m_color;       // Color of the point
};
// The derived class checks the upper limit values
void ColoredPoint::setAll(int in_x, int in_y,...){
   if (in_x <= MAX_x) setX(in_x);  // setX of Point checks the lower limits
   if (in_y <= MAX_y) setY(in_y);
     :
}
```

In this example, the `Point` class checks the lower limits, while the `ColoredPoint` checks the upper ones.

There are clearly defined responsibilities for each class (***separation of concerns***).

Example: The `ColoredPoint` class has lower and upper limits (contd)

```
int main()
{
  ColoredPoint colored_point1{ Color::Green };   // A green point
        // X = 200 is not accepted due to the upper limit
  colored_point1.setAll(200, 200, "Colored Point1", Color::Red);

       // X and Y coordinates are not accepted due to the lower limit
  colored_point1.setAll(-10, -20, "Colored Point1", Color::Red);

  colored_point1.print(); // OK print function of Point is public again

  colored_point1.move(200, 200); // Error! move() from Point is private
  colored_point1.setX(200);      // Error! setX() from Point is private
    :
}
```

See Example e07_4a.cpp

Redefining Access Specifications (contd):

- After a `public` derivation, the creator of the derived class can make the selected `public` members of the base class `private` (or `protected`).
- You cannot loosen the rules set by the class creator; you can only tighten them. So, you cannot make `private` members of the base class `public` or `protected`.

```
class ColoredPoint : public Point {   // Public inheritance
   :
private:
  using Point::move;   // Non-constant method are private
  using Point::setX;
  using Point::setY;
   :
};
int main(){
ColoredPoint colored_point1{ Color::Green };      // A green point
colored_point1.setX(200); // Error! setX function in ColoredPoint is private
colored_point1.move(200,200); // Error! move in ColoredPoint is private
colored_point1.Point::move(200, 200);  // OK! Using the base name explicitly
```

See Example e07_4b.cpp

Under `public` inheritance, the `move` in `Point` is still `public`.

Summary of Access Specification

```
class Base {
public:
  :
protected:
  :
private:
  :
};
```

These determine if the clients of the `Base` (objects and directly derived classes) can access the members of the `Base`.

`public`: Objects of `Base` and methods of `Derived1` can access

`protected`: Methods of `Derived1` can access, not the `Base` objects

`private`: Only the members of the `Base` can access it.

```
class Derived1: public/protected/private Base {
  :
};
```

These determine if the clients of the `Derived1` (objects and directly derived classes) can access the <u>members inherited from the `Base`</u>.

`public`: Objects of `Derived1` can access `public` members inherited from the `Base`.

The methods of `Derived2` can access `public` and `protected` members inherited from the `Base`.

`private`: Only the methods of the `Derived1` can access `public` and `protected` members inherited from the `Base`.

```
class Derived2: public/... Derived1 {
  :
};
```

```
int main(){
Base base_Object;
Derived1 derived1_Object;
Derived2 derived2_Object;
```


Redefining the Members of the Base (Name Hiding)

Some base class members (data or function) may not be suitable for the derived class. These members should be redefined in the derived class.

Example: The `Point` class has a print function that prints the properties of the points on the screen.

However, this function is not sufficient for the class `ColoredPoint` because colored points (specialized points) have more properties (e.g., color) to be printed.

So, the print function must be redefined in the `ColoredPoint` class.

```
class Point {
public:
  void print() const;        // prints coordinates on the screen
   :
};

class ColoredPoint : public Point {
public:
  void print() const;      // redefines the print function
   :                       // this function prints the color as well
};
```

`ColoredPoint` contains two `print()` functions with the same signature but different bodies.

Example (contd): Redefining the `print` function of the `Point` class

The `print()` function of the `ColoredPoint` class hides the `print()` function of the `Point` class.

Now, the `ColoredPoint` class has two `print()` functions. The base class members with the same name can be accessed using the scope resolution operator (`::`).

```
// ColoredPoint redefines the print function of Point
// This function prints the color as well
void ColoredPoint::print() const
{
   Point::print();  // calls print inherited from Point to print x and y
   ...              // Additional code for printing the color
}

int main()
{
  ColoredPoint col_point1{ Color::Green }; // A green point
  col_point1.print();        // print function of the ColoredPoint
  col_point1.Point::print(); // print function inherited from Point
```

If the base class access specifier is `public`

See Example e07_5.cpp

Preventing derived objects from accessing redefined members of the base:

- When the access specifier of the base class is `public`, i.e., `class Derived:`**`public`**` Base`, the objects of `Derived` can still access the redefined `public` members of the `Base` using the scope resolution operator `::` .

  For example, in e07_5.cpp, the object `col_point1` of the `ColoredPoint` class can also access the `print()` function of the `Point` class.

  `col_point1.Point::print(); // calls the redefined method of the Base`
- However, this is not preferable because the author of the derived redefines the members of the base when they are not appropriate for the derived objects.
- We can inherit redefined members privately to prevent derived objects from accessing them.

Example:

Redefining the `move` function of the `Point` class under a `private` inheritance:

- In example `e07_4a.cpp`, according to the requirements, the coordinates of colored points have lower and upper limits.
- Since the base class `Point` has only lower limits, the author of the `ColoredPoint` class must privately inherit members of the `Point` class (specifically the setters and the move method) and add upper limits and related methods to check them.

Example (contd):
Redefining the `move` function of the `Point` class under a `private` inheritance

- Since the access specifier of the base class `Point` is `private` now, the users (objects) of the `ColoredPoint` class cannot call the `move` function or setters inherited from `Point` that check only the lower limits.
- The author will redefine the `move` function to check both the lower and upper limits.

```
class ColoredPoint : private Point {  // Private inheritance
public:
  bool move(int, int);  // move of Point is redefined
  void print() const;   // print of Point is redefined
   :
};
```

See Example e07_6.cpp

```
int main() {
  ColoredPoint colored_point1{ Color::Green };// A green point
  colored_point1.move(200, 2000);           // move of ColoredPoint
  colored_point1.print();                   // print of ColoredPoint
  colored_point1.Point::move(200, 200);// Error! Point is private base
  colored_point1.setX(100);            // Error! Point is private base
  colored_point1.Point::print();       // Error! Point is private base
```


During redefinition, the parameters of the `Base` methods can be changed:
Example:

```
class Base {                  // Base Class
public:
  void method() const;        // Method of Base
protected:
  int m_data1 {1};           // protected integer data member of Base
private:
  int m_data2 {2};           // private integer data member of Base
};

class Derived : public Base {  // Derived Class
public:
  void method(int) const;      // Method of Base is redefined
private:
  std::string m_data1 { "ABC" };  // data members can be also redefined
  int m_data2 {3};               // private data member of Base is redefined
};
```

The `Derived` class has two methods: `void method()` and `void method(int)`.

It has four data members: `int m_data1`, `string m_data1`, `int m_data2` inherited from `Base`, and `int m_data2`.

Example (contd): Name Hiding

```
// A method of Derived
void Derived::method(int in_i) const {
  std::print("m_data1 of Derived = {}", m_data1);   // m_data of Derived
  std::print("m_data1 of Base =  {}", Base::m_data1; // OK. protected in Base
  std::print("m_data2 of Base =  {}", Base::m_data2; // Error! private
  Base::method();      // OK. method() of Base is public
}
```

Since `m_data2` of `Base` is private, methods of `Derived` **cannot** access `Base::m_data2`.

```
int main() {
  Derived derived_object;          // An object of Derived
  derived_object.method(2);        // method(int) of Derived
//derived_object.method();         // Error! Redefined, hidden
  derived_object.Base::method();   // OK. method() of Base is public
}
```

Since the `Derived` class redefines (hides) the `method()` of the `Base`, its objects cannot access the method of the `Base` directly (implicitly).
If the method in the `Base` is `public`, the objects can still access the redefined method using the name `Base`.

Overloading and Name Hiding in C++:

Overloading:

Remember, overloading occurs when two or more methods of the same class or multiple nonmember methods in the same namespace have the same name but different parameters (Slide 2.38).

Since the overloaded functions have different signatures, the compiler treats them as distinct functions, so there is no uncertainty when we call them.

- Methods of the same class or nonmember methods in the same namespace
- Functions have the same names but different input parameters.

Name Hiding:

Name hiding occurs when a derived class redefines the methods of the base class.

The methods may have the same or different parameters, but they will have different bodies.

- Only with inheritance
- Functions have the same names. The parameters can be the same or different.

Overriding:

Overriding of functions during inheritance helps us to achieve **polymorphism**, which we will cover in Chapter 8.

See Example e07_7.cpp

Constructors and Destructors in Inheritance

Default Constructor:

If the `Base` class contains a default constructor, the `Derived` constructor calls it automatically if another constructor is not invoked in the initialization list.

In this chapter's previous examples, the base class `Point` had a default constructor, i.e., `Point() = default`.

Since the constructor of the derived class, `ColoredPoint` calls this default constructor; we can compile and run these programs.

```
ColoredPoint::ColoredPoint(Color in_color): m_color{in_color}
{ }
```

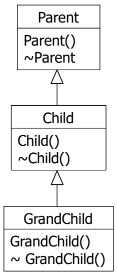
Since a base constructor with parameters such as `Point(10,20)` is not invoked in the initialization list, the default constructor of the `Point` is called implicitly.

The order of construction:

- Firstly, the subobject inherited from the `Base` is constructed.
- Then the remaining part of the `Derived` object is initialized.
- Since a derived class's object has a base class's object inside it, the base object must be created before the rest of the object.
- If that base class is derived from another class, the same applies.

Destructor:

- You never need to make explicit destructor calls because there is only one destructor for any class, and it does not take any arguments.
- The compiler ensures that all destructors are called, which means all destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root.



When the derived object goes out of scope, the destructors are called in reverse order, i.e., the derived object is destroyed first, then the subobject inherited from the `Base`.

Example:

```
int main()
{
  GrandChild grandchild_object;
  cout << "Program terminates";
  return 0;
}
```

The Output:

```
Parent constructor
Child constructor
GrandChild constructor
Program terminates
GrandChild destructor
Child destructor
Parent destructor
```

See Example e07_8.cpp

Constructors with parameters:

- If the `Base` class contains constructors with parameters instead of a default constructor, the `Derived` class **must have a constructor** that calls one of the `Base` class's constructors in its initialization list.

Example:

- In this example, we assume that the base class `Point` has only one constructor with two integer parameters and **no default constructor**:

```
Class Point{
   Point(int, int);   // Constructor to initialize x and y coordinates
```

- The constructors of the derived class `ColoredPoint` **must call** this constructor in the initialization list.

```
ColoredPoint::ColoredPoint(int in_x, int in_y, Color in_color)
              : Point{in_x, in_y}, m_color{in_color}
{ }
```

See Example e07_9a.cpp

- Since the `Point` class does not contain a default constructor, the following code **will not compile**.

```
ColoredPoint::ColoredPoint(Color in_color): m_color{in_color}
{ }
```

Tries to call the default constructor of the `Point`. Error!

Constructors with parameters (contd):

- If the `Base` class contains multiple constructors, the author of the `Derived` class can call one of them in the initialization list of the derived constructors.
- The constructors with parameters are not invoked automatically like the default constructor.
- The author of the `Derived` class must decide which base constructor to invoke and supply it with the necessary arguments.

Example:

The base class `Point` has three constructors, i.e., a default constructor and two constructors with parameters:

```
Class Point{
   Point();           // Default constructor
   Point(int);        // Constructor assigns same value to x and y
   Point(int, int);   // Constructor to initialize x and y coordinates
```

The constructors of the derived class `ColoredPoint` can call any of these constructors in the initialization list.

Example (contd):

```
Class Point{
   Point();           // Default constructor
   Point(int);        // Constructor assigns same value to x and y
   Point(int, int);   // Constructor to initialize x and y coordinates
     :
};

ColoredPoint::ColoredPoint(int in_x, int in_y, Color in_color)
              : Point{in_x, in_y}, m_color{in_color}
{ }

ColoredPoint::ColoredPoint(Color in_color): Point{1}, m_color{in_color}
{ }

ColoredPoint::ColoredPoint()
{ }
```

See Example e07_9b.cpp

Constructors and destructors in the inheritance with composition

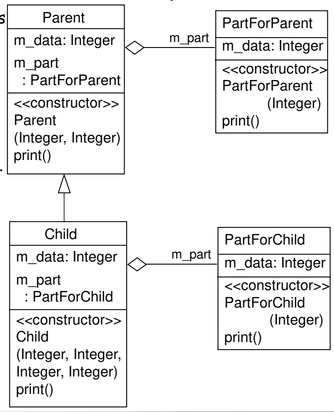
In OOD, "is-a" and "has-a" relationships can occur together.

Remember: In composition, the objects' lifecycles are tied (6.13).

In the design on the right, the `Child` class contains a `PartForChild` object and is derived from the Parent class, which includes a `PartForParent` object.

- The `Child's` constructor must first initialize the subobject inherited from the `Parent`, then the part object, and finally its data member.
- The constructor of the `Parent` must first initialize the part object and then its data member.
- The constructors of the parts must initialize their data members.

In this example, a `Child` object contains four integers (`m_data`).



Constructors and destructors in the inheritance with composition

Default Constructors:

Example 1:

In this example, all classes have default constructors.

- We do not need to call constructors explicitly.
- Default constructors are called, and objects are initialized in the proper order automatically.

See Example e07_10a.cpp

The order of construction:

```
PartForParent
Parent
PartForChild
Child
```

The order of destruction:

```
Child
PartForChild
Parent
PartForParent
```


Constructors and destructors in the inheritance with composition (contd)

Constructors with parameters:

Example 2:

In this example, all classes have constructors with parameters.

- Constructors of owners must initialize their parts.
- Constructors of child classes must initialize their parents.

Initialize the part

Initialize the data member

```
// *** Base Class
class Parent {
public:
  Parent(int in_data1, int in_data2) : m_part{in_data1}, m_data{in_data2}
     {}        // The body of the constructor
  ...
private:
  PartForParent m_part;      // Parent contains (has) a part (composition)
  int m_data{};              // data of Parent
};
```

Example 2 (contd):

The order in the list is not important.
Always the `Parent` subobject is initialized first.
Then the part is initialized.

```
// *** The Derived Class
class Child : public Parent {
public:
  Child(int in_data1, int in_data2, int in_data3, int in_data4)
    : Parent{ in_data1, in_data2 }, m_part{ in_data3 }, m_data{ in_data4 }
   {}          // The body of the constructor
   ...
private:
  PartForChild m_part;         // Child contains (has) a part (composition)
  int m_data{};                // data of Child
};


int main() {
  Child child_object{ 1, 2, 3, 4 };   // An object of the Child
  child_object.print();
  :
```

See Example e07_10b.cpp

Example 2 (contd):

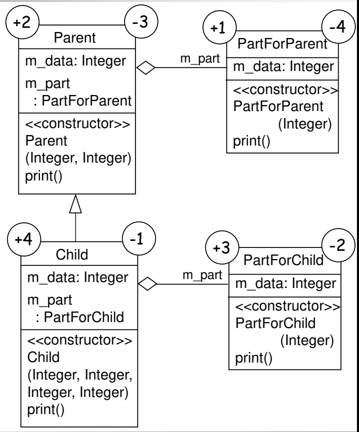
The construction and destruction order is the same as in Example 1 with default constructors.

The order of construction:

```
PartForParent
Parent
PartForChild
Child
```

The order of destruction:

```
Child
PartForChild
Parent
PartForParent
```



Constructors and destructors in the inheritance with composition (contd)
Dynamic Member objects (Pointers as members)

Remember: Instead of automatic objects, data members of an owner class may also be pointers to parts.

- If the relationship is composition, the whole must create and initialize part objects in the constructor.
- To preserve the order of creation (first parts, then the whole), objects must be created in the member initialization list of the constructor, not in the body.

Example 3a: Pointers as members. Dynamic objects are created in the member initialization list

A dynamic part object is created

```
class Parent {              // *** Base Class
public:
  Parent(int in_data1, int in_data2)
             : m_part{ new PartForParent {in_data1} },
               m_data{in_data2}
     {}                            // The body of the constructor
  ~Parent (){delete m_part;}    // Destructor is required to release memory
private:
  PartForParent * m_part;       // Parent contains a pointer to the part
```


Example 3a (contd): Pointers as members. Dynamic objects are created in the member initialization list

```
 // *** Derived Class
 class Child : public Parent {
 public:
    Child(int, int, int, int);  // Constructor of the Child
    ~Child() ;                  // Destructor of the Child
  ...
 private:
    PartForChild *m_part;      // Child contains a pointer to the part
  ...
// Constructor of the Child
Child::Child(int in_data1, int in_data2, int in_data3, int in_data4)
     : Parent{ in_data1, in_data2 },    // Intialize the Parent subobject
       m_part{ new PartForChild {in_data3} },  // Create the part object
       m_data{ in_data4 }                      // Initialize data memeber
{};
// Destructor of the Child
Child::~Child() {
  delete m_part;             // Delete the part object
};
```

See Example e07_10c.cpp

Dynamic Member objects (Pointers as members)
Changing the order of construction

If the owner class has pointers to parts,

- The programmer can decide when the parts are to be created and destroyed.
- The dynamic objects can be created in the body of the constructor instead of in the member initialization list.

  In this case, the owner will be created first, then the parts.
- Data members of the owner can be used to initialize the parts because the owner is created before its part.

Example 3b:

- Pointers as members. Dynamic objects are created in the body of the constructor. The owner is created before the part.
- Data members of the owners are used to initialize the parts.

See Example e07_10d.cpp

```
// Constructor of the Parent
Parent::Parent(int in_data1)
       :m_data{ in_data1 }          // The data member is initialized
{                                   // The body of the constructor
  m_part = new PartForParent{ m_data };  // m_data is a data of the owner
}         // The part object is created and initialized using the data member
```


Inheriting constructors

- Constructors must do different things in the base and derived classes.

  The base class constructor must create the base class data, and the derived class constructor must create the derived class data.
- Because the derived class and base class constructors create different data, normally, one constructor cannot be used in place of another.
- Base class constructors are inherited in a derived class as regular member functions but not as the constructors of the derived class.
- However, the author of the derived class can decide to use the base class's constructor as the derived class's constructor.
- To inherit the base class constructor, we should put a `using` declaration in the derived class.

Example: The `ColoredPoint` inherits constructors of the `Point`

```
class ColoredPoint : public Point {
public:
  using Point::Point; // Inherits all constructors of the Point
    :
};
```


Example: The `ColoredPoint` inherits constructors of the `Point`

We assume that the `Point` class has two constructors.

```
class Point {
public:
  Point(int, int);   // Constructor with two integers to initialize x and y
  Point(int);        // Initializes x and y to the same value, e.g., (10,10)
:
}

class ColoredPoint : public Point {
public:
  using Point::Point;  // Inherits all constructors of the Point
   :
};
int main()
{
 ColoredPoint colored_point1{ 10, 20 };//Inherited constructor of the Point
 ColoredPoint colored_point2{ 30 };    //Inherited constructor of the Point
```

Without the `using` declaration, these definitions will not compile.

The `ColoredPoint` class can also have its own constructors:

```
  ColredPoint (int, int, Color);
```

See Example e07_11.cpp

The Copy constructor under inheritance

The default copy constructor:

- Remember: If the class author does not write a copy constructor, the compiler supplies one by default.
- The default copy constructor will simply copy the contents of the original into the new object as a member-by-member copy.
- In most cases, this copy is sufficient.

Example:

What happens if we do not supply a copy constructor for our `Point` and `ColoredPoint` classes?

See Example e07_12a.cpp

This program runs correctly because the compiler supplies copy constructors for both classes.

The default copy constructor of the `ColoredPoint` calls the default copy constructor of the `Point` class, and all members are copied from the original object into the new object.

The Copy constructor under inheritance (contd)

The programmer-defined copy constructor in the derived class:

Although not necessary in our example, the programmer can write copy constructor for the `ColoredPoint`.

```
ColoredPoint::ColoredPoint(const ColoredPoint& in_col_point)
                              : m_color{ in_col_point.m_color }
{}
```

It is not specified which constructor of the `Point` to call.

See Example e07_12b.cpp

```
int main() {
 ColoredPoint colored_point1{ 10, 20, Color::Blue}; // Constructor
 ColoredPoint colored_point2{colored_point1};       // Copy constructor
```

- When we run this program, we see that the object `colored_point2` **is not the exact copy** of `colored_point1` (coordinates are different).
- The `ColoredPoint` copy constructor does not call the `Point` copy constructor automatically if we do not tell it to do so.
- The compiler knows it has to create a `Point` subobject but does not know which constructor to use.
- If we do not specify a constructor, the compiler will call the default constructor of the `Point` automatically.

The programmer-defined copy constructor in the derived class (contd):

To fix the problem in the program `e07_12b.cpp`, we must call the `Point` copy constructor in the initialization list of the `ColoredPoint` copy constructor.

```
ColoredPoint::ColoredPoint(const ColoredPoint& in_col_point)
                : Point{in_col_point}, m_color{in_col_point.m_color}
{}
```

The copy constructor of the `Point`

The `Point` copy constructor is called with the object of the `ColoredPoint` (`in_col_point`) as an argument.

However, the input parameter of the `Point` copy constructor is a reference to `Point` objects, i.e., `Point(const Point &);`

There is not a type mismatch, thanks to the is-a relationship.

Remember, `ColoredPoint` is a `Point`.

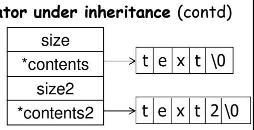
Therefore, `ColoredPoint` objects can be sent as arguments to the functions that expect `Point` objects as parameters.

We will discuss this topic in detail later.

See Example e07_12c.cpp

The Copy constructor and the assignment operator under inheritance (contd)

Example: Double String



- Assume that according to new requirements, we need a string type with two contents'.
- We can derive the new class `DoubleString` from the existing class `String` that we already developed.
- Since the base and derived classes both contain pointers, we must supply copy constructors and copy assignment operators for these classes.
- The `DoubleString` copy constructor must call the `String` copy constructor.

```
DoubleString::DoubleString(const DoubleString& in_object)
                                         : String{ in_object }
```

The `DoubleString` assignment operator function must call the `String` assignment operator.

```
const DoubleString& DoubleString::operator=(const DoubleString& in_object)
{
  if (this != &in_object) {          // checking for self-assignment
    String::operator=(in_object);    // call the operator of the String
     :
```

See Example e07_13.cpp

Inheriting from the library

Just like from programmer-written classes, we can also derive new classes from the classes in a library.

Example: A colored string

- Assume that according to requirements, we need strings with a color.
- We can derive a class `ColoredString` from the class **`std::string`**.
- This new class will inherit all members (constructors, operators, getters, setters, etc.) of the `std::string`. So, we **reuse** the `std::string`.
- As you know, we can add new members and redefine inherited members.

```
class ColoredString : public std::string {...}
```

We can use objects of `ColoredString` like standard `std::string` objects.

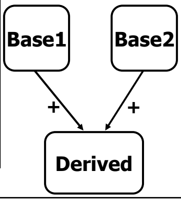
See Example e07_14.cpp

```
int main() {
 ColoredString firstString{ "First String", Color::Blue }; // Constructor
 ColoredString secondString{ firstString };              // Copy constructor
 secondString += thirdString;                          // += operator of std::strig
 secondString.insert(12, "-");                       // Insert "-" to the position 12
 ColoredString fourthString;                               // Default constructor
 fourthString = secondString;                             // Assignment operators
```

Multiple Inheritance

Multiple inheritance occurs when a class inherits from two or more base classes.

```
class Base1{
public:
   Base1();
   ~Base1();
   void f1();
   void f2();
   void f3();
   void f4();
};
```




```
class Base2{
public:
   Base2();
   ~Base2();
   void f1();
   void f2(int);
   void f3(int);
};
```

Remember:

- The derived class includes all members of both base classes.

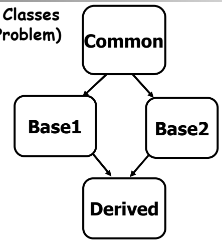
  For example, the class `Derived` contains three `f1` and two `f3` functions.
- In inheritance, functions are not overloaded. They are **redefined** or **overridden**.

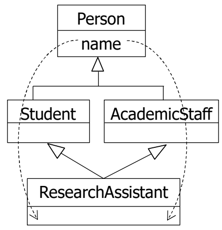
```
class Derived : public Base1 , public Base2{
public:
  Derived();
  ~Derived();
  void f1();
  void f2(int, char);
  void f5();
};
```

See Example: e07_15.cpp

```
int main() {
Derived d;
d.f1();           // Derived::f1
//d.f2(1);        // Error!
d.Base2::f2(1);   // Base2::f2
//d.f3();         // Error! Ambiguous
d.f4();           // Base1::f4
```


Repeated Base Classes (The Diamond Problem)





`Base1` and `Base2` inherit from `Common`, and `Derived` inherits from `Base1` and `Base2`.

Recall that each object created through inheritance contains a base class subobject.

A `Base1` object and a `Base2` object will contain subobjects of `Common`, and a `Derived` object will contain subobjects of Base1 and Base2, so a `Derived` object will also contain two `Common` subobjects, one inherited via `Base1` and one inherited via `Base2`.

This is a strange situation. There are two subobjects when there should be only one.

Repeated Base Classes (The Diamond Problem) (contd)

Suppose there is a data item in `Common`:

```
class Common
{
 protected:
   int common_data;
 };
```

```
class Base1 : public Common
    {   };
class Base2 : public Common
    {   };
```

The derived objects will contain two `common_data`.

```
class Derived : public Base1, public Base2  {
public:
  void setCommonData(int in) {
     common_data = in;           // ERROR! Ambiguous
     Base1::common_data = in;    // OK but confusing
     Base2::common_data = in;    // OK but confusing
  }
};
```

See Example: e07_16a.cpp

The compiler will complain that the reference to `common_data` is ambiguous.
It does not know which version of `common_data` to access: the one in the Common subobject in the Base1 subobject or the Common subobject in the Base2 subobject.

Virtual Base Classes

You can fix this using a new keyword, **`virtual`**, when deriving `Base1` and `Base2` from `Common` :

```
class Common
   {   };
class Base1 : virtual public Common
   {   };
class Base2 : virtual public Common
   {   };
class Derived : public Base1, public Base2
   {   };
```

See Example: e07_16b.cpp

The `virtual` keyword tells the compiler to inherit only one subobject from a class into subsequent derived classes.

That fixes the ambiguity problem, but other more complicated issues may arise that are out of the scope of this course.

In general, you should avoid multiple inheritance, although if you have considerable experience in C++, you might find reasons to use it in some situations.

Pointers to objects and inheritance

Under `public` inheritance:

- If a class `Derived` has a `public` base class `Base`, then the address of a `Derived` object can be assigned to a pointer to `Base` without explicit type conversion.

  In other words, a pointer to `Base` can store the address of an object of `Derived`.

  A pointer to `Base` can also point to objects of `Derived`.

  For example, a pointer to `Point` can point to objects of `Point` and also to objects of `ColoredPoint`.

  A colored point **is a** point.
- The opposite conversion must be explicit for a pointer to `Base` to a pointer to `Derived`.

  A point is not always a colored point.

```
class Base {...};
class Derived : public Base {...};
int main() {
  Derived d_obj;
  Base *bp = &d_obj;                    // implicit conversion
  Derived *dp = bp;                     // ERROR! Base is not Derived
  dp = static_cast<Derived *>(bp);      // explicit conversion
```


Accessing members of the Derived class via a pointer to the Base class:

- When a **pointer to the Base** class points to objects of the Derived class, only the members inherited from the Base can be accessed via this pointer.

  In other words, members just defined in the Derived class cannot be accessed via a pointer to the Base class.

  For example, a pointer to Point objects can store the address of an object of the ColoredPoint type.

  Using a pointer to the Point class, it is only possible to access the "point" properties of a colored point, i.e., only the members that the ColoredPoint inherits from the Point class.
- Using a **pointer to the Derived** type (e.g., `ColoredPoint`), it is possible to access, as expected, all (`public`) members of the `ColoredPoint` (both inherited from the `Point` and defined in the `ColoredPoint`).

  See the example `e07_17.cpp` in the next slide.

We will investigate some additional issues about pointers under inheritance (such as accessing overridden functions) in Chapter 8 (Polymorphism).

Example: Pointers to `Point` and `ColoredPoint` classes

```
class Point {                       // The Point Class (Base Class)
public:
   bool move(int, int);             // Points behavior
      :
};
class ColoredPoint : public Point { // Derived Class, public inheritance
public:
   void setColor(Color)             // ColoredPoints behavior
      :
};
```

See Example: e07_17.cpp

```
int main(){
  ColoredPoint  objColoredPoint{ 10, 20, Color::Blue };
  Point* ptrPoint = &objColoredPoint;      // Point* ptr ← &ColoredPoint
  ptrPoint->move(30, 40);                  // OK. Moving is Points behavior
  ptrPoint->setColor(Color::Green);        // ERROR! Setting the color is not
                                           // Points behavior
  ColoredPoint* ptrColoredPoint = &objColoredPoint; // ColoredPoint* ptr
  ptrColoredPoint->move(100, 200);             // OK. ColoredPoint is a Point
  ptrColoredPoint->setColor(Color::Green);   // OK. ColoredPoints behavior
```


References to objects and inheritance

- Remember, like pointers, references can also point to objects.
  We pass objects to functions as arguments, usually using their references for two reasons:
  a. To avoid copying large-sized objects, e.g., `void function(const ClassName&);`
  b. To modify original objects in the function, e.g., `void function(ClassName&);`
- If a class `Derived` has a `public` base class `Base`, **a reference to `Base` can also point to objects of `Derived`.**

  If a function gets a reference to `Base` as a parameter, we can call this function, sending a reference to the `Derived` object as an argument.

  Remember, on slide 7.52, we call the copy constructor of the `Point` by sending the object of the `ColoredPoint` (`in_col_point`) as an argument.

  However, the input parameter of the `Point` copy constructor is a reference to `Point` objects, i.e., `Point(const Point &);`

References to objects and inheritance (contd)

Example:

Remember the example `e06_5.cpp`. We have a class called `GraphicTools` that contains tools that can operate on `Point` objects.

For example, the method `distanceFromZero` of the `GraphicTools` calculates the distance of a `Point` object from zero (0,0).

```
double GraphicTools::distanceFromZero(const Point&) const;
```

Since a colored point is a point, we can use this method of the `GraphicTools` also for the `ColoredPoint` objects without modifying it.

Since the method's parameter in `GraphicTools` is a reference to `Point` objects, we can call the same method without any modification by passing references to `ColoredPoint` objects as arguments.

```
int main() {
  GraphicTools gTool;                              // A GraphicTools object
  Point point1{ 10, 20 };                          // A Point object
  distance = gTool.distanceFromZero(point1);       // ref. to Point object

  ColoredPoint col_point1{ 30, 40, Color::Blue };// A ColoredPoint object
  distance = gTool.distanceFromZero(col_point1);  // ref. to ColoredPoint
    :
```

See Example: e07_18.cpp

Pointers to objects under `private` inheritance

Remember, if the base class is `private`, derived objects cannot access public members inherited from the base (see slide 7.20).

It is because the author of the derived class does not permit users of the derived class to use these inherited members since they are not suitable for the derived class.

Therefore, if the class `Base` is a **private** base of `Derived`, the implicit conversion of a `Derived*` to `Base*` will not be done.

In this case, a pointer to the `Base` type cannot point to `Derived` objects.

If the base class is private, derived objects may not show the same behaviors as their base objects.

Pointers to objects under `private` inheritance (contd)

Example:

```
class Base {
public:
  void methodBase();
};
class Derived : private Base {   // Private inheritance
};
int main(){
  Derived dObj;          // A Derived object
  dObj.methodBase();     // ERROR! methodBase is a private member of Derived
  Base* bPtr = &dObj;    // ERROR! private base
  Base* bPtr = reinterpret_cast<Base*>(&dObj); // OK. explicit conversion
                                               // AVOID!
  bPtr->methodBase();                          // OK but AVOID!
```

Accessing members of the private base after an explicit conversion is possible but not preferable.

By doing so, we break the rules set by the `Derived` class author.

As a result, the program may behave unexpectedly.

A heterogeneous linked list of objects

Since a pointer to `Base` can also point to `Derived` objects, we can create **heterogeneous** linked lists comprising both `Base` and `Derived` objects.

Example: A linked list that contains `Point` and `ColoredPoint` objects.

A `Point` object has no built-in pointer to link it with another `Point` object.

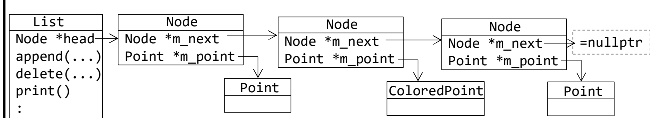
Changing the definition of the `Point` class and adding a pointer to the next object violates the "separation of concerns" principle because linking is not a task (responsibility) of a point.

To put `Point` and its child objects (e.g., colored points) into a list, we will define another type of class called `Node`.

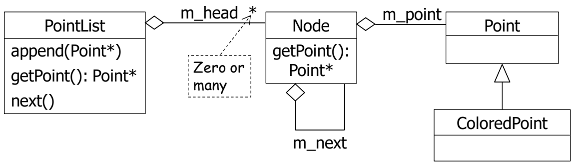
A `Node` object will have two members:

`m_point`: A pointer to the `Point` type (the element in the list).

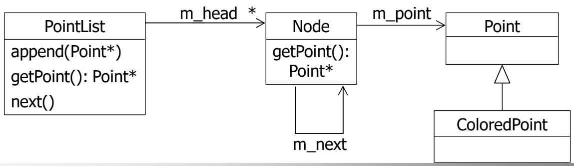
`m_next`: A pointer to the next node in the list.



The UML class diagram of the design of the list for point and colored point objects:



Instead of detailed aggregation and composition relations, we can present only the general association relation among classes:



Example: A linked list that contains `Point` and `ColoredPoint` objects (contd)

```
class Node{
public:
  Node(Point *);
  Point* getPoint() const { return m_point; }
  Node* getNext() const { return m_next; }
   :
private:
  Point* m_point{};  // The pointer to the element of the list
  Node* m_next{};    // Pointer to the next node
};
class PointList{
public:
  :
  void append(Point *);          // Add a point to the end of the list
  Point* getPoint() const;        // Return the current Point
  void next();              // Move the current pointer to the next node
private:
  Node* m_head{};       // The pointer to the first node in the list
  Node* m_current{};    // The pointer to the current node in the list
};
```

You don't need to create your own classes for linked lists. `std::list` is already defined in the standard library. We provide this example for educational purposes.

Example: A linked list that contains `Point` and `ColoredPoint` objects (contd)

```
int main() {
  PointList listObj;                                   // Empty list
  ColoredPoint  col_point1{ 10, 20, Color::Blue };   // ColoredPoint type
  listObj.append(&col_point1);         // Append a colored point to the list

  Point *ptrPoint1 = new Point {30, 40};       // Dynamic Point object
  listObj.append(ptrPoint1);                   // Append a point to the list

  ColoredPoint  *ptrColPoint1 = new ColoredPoint{ 50, 60, Color::Red };
  listObj.append(ptrColPoint1);        // Append a colored point to the list

  Point* local_ptrPoint;                     // A local pointer to Point objects
  local_ptrPoint = listObj.getPoint(); //Get the (pointer to) first element
  std::print("X = {}", local_ptrPoint->getX() );
  std::println(", Y = {}", local_ptrPoint->getY() );

  local_ptrPoint->setX(0);                 // OK. setX is a member of Point
  local_ptrPoint->setColor(Color::Red);    // Error! not a member of Point

  delete ptrPoint1;
  delete ptrColPoint1;
  :
```

See Example: e07_19.zip

Conclusion about Inheritance:

- We use inheritance to represent the "**is-a**" ("**kind-of**") relationship between objects.
- We can create special types from general types.
- We can **reuse** the base class without changing its code.
- We can add new members, redefine existing members, and redefine access specifications of the base class without modifying its code.
- It enables us to use polymorphism, which we will cover in Chapter 8.