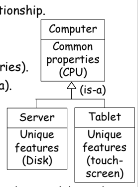
INHERITANCE

- 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 properties in common (e.g., they all have CPUs and memories).
  - They also have some abilities in common (e.g., running programs and storing data).
- We can say "A server **is a** computer" and "A tablet is **a kind of** computer".
- In addition to the common properties, each can also have its own unique features.
  For example, a server has a magnetic disk and can process big data, a tablet has a touchscreen, a smartphone can make phone calls, etc.

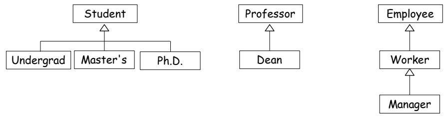


Generalization – Specialization:

- With the help of inheritance, we can create more specialized types (classes) of general types (classes).
- Specialized types may have more features (properties) than general types.
  - For example, the computer is a general type; all computers contain a CPU and memory.
  - A server and a tablet are special types of computers. A server can run programs like all other computers. In addition, it can process big data.
  - In programming, classes that represent specialized types may have more members (data and methods) than classes that represent general types.

Examples:

- Undergraduate students, master's students, and Ph.D. students are all students.
  - They have **common attributes**, e.g, ID and **abilities** (behavior, responsibility), e.g., printing the ID.
  - However, the procedure to calculate the GPA can be different for undergraduate and master's students.
  - Ph.D students must take a qualification exam; this requirement does not exist for other student types.
- Professor ← Dean: A dean is a professor; a dean can teach and conduct research like a regular professor. In addition, the dean manages the faculty.
- 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 specialized.



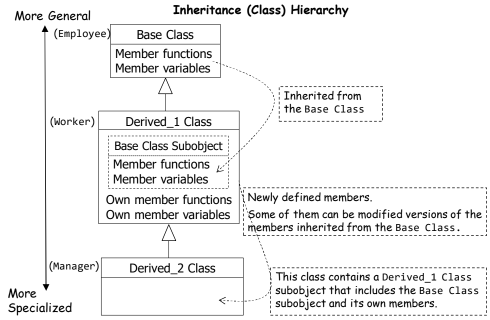
Inheritance in Programming - Modification During Specialization:

- In OOP, **inheritance** enables the **modification** and **extension** of a class without changing its code.
  - When we create "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 (general) class, called the ***base (parent) class***, is not modified.
- However, the new (specialized) class, called the ***derived (child) class***, can
  - **use** the features of the base class,
  - **add** new features (attributes and methods), and
  - **modify** some features of the base class.

Example:

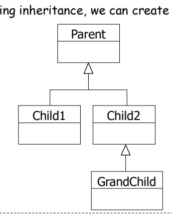
- A manager is a worker. The Worker is the general type, and the Manager is the special type of Worker.
- Managers have all of the properties of the Workers and some additional features.
- Workers have a procedure to calculate their salaries.
- Managers also have a procedure for salary calculation, but it may differ from the Workers' procedure.
- The Manager type should modify the procedure derived from the general Worker type.

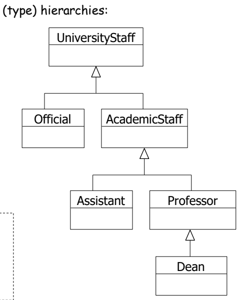
Inheritance (Class) Hierarchy



Inheritance (Class) Hierarchy (cont'd)

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



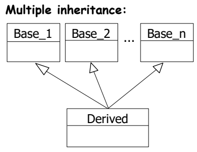


Terminology:

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

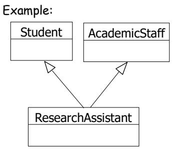
Inheritance (Class) Hierarchy (cont'd)

Multiple inheritance:



- C++ supports multiple inheritance.
- Some programming languages, such as Java and C#, use a different mechanism (interfaces) instead of multiple inheritance.
- Multiple inheritance can cause a variety of ambiguity problems and increase the complexity of the code (Each base class might contain data members/member functions with the same name).
- It must be used judiciously and only when necessary (slide 7.57).

Example:



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

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

- In inheritance, the objects of the derived class contain a subobject of the base class; however, this is not a composition (not a has-a relationship).
- Remember, **composition** in OOP models the real-world situation in which objects are composed (or part) of other objects.
  - For example, a 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 of employee.
  - Every kind of employee has certain 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:

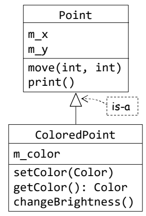
- We have already developed a `Point` class.
- Now, we need points with colors and related functions.
- This is a specialized version of the `Point` class we already defined.
- We do not need to define a new `ColoredPoint` class from scratch.
- We can **reuse** the existing `Point` class and derive the 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
};
```

General (common) features

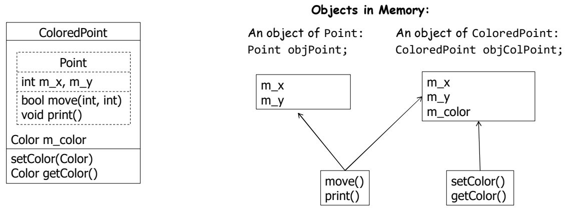
+ Inherited (added)

```
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` (cont'd)

Objects in Memory:



Example: `ColoredPoint` is a `Point` (cont'd)

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

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

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

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)` to a double literal.

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

The derived class, `ColoredPoint`, inherits this function, so its objects can use it.

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

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

Example e07_1b.cpp

Access Control in Inheritance

Remember: The **private** access specifier states 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::writeX(int in_x) { m_x = in_x; }    // Error! m_x is private in Point
```

- The derived class can 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::writeX(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 class creator sets the rules, and the client programmer must follow them.
- 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 3.10).

Access Control (cont'd)

Protected Members:

- When we 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 `private`.
- Protected members cannot be accessed outside the class, except by functions specified as `friend` functions.
- 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 (cont'd):

Example:

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

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

// Member function of the Derived Class, ColoredPoint
// ColoredPoint accesses 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
}
```

Example e07_2.cpp

Protected vs. Private Members

Remember the **information (data) hiding** principle (see slide 3.10).

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

Some potential problems protected members can cause:

- 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 `protected` members:

- 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 (cont'd)

- 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 become useless.
- Methods of the derived class `ColoredPoint` can modify the coordinates of a point object directly and move it beyond the allowed limits. The rules set by the class creator become useless.

```
// ColoredPoint accesses 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
}

int main(){
  ColeredPoint colored_point{};
  colored_point.setAll(-100, -500, Color::Red);     // The point moves beyond the limits
```

Example e07_3.cpp

Base Class Access Specification

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

Example:

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

Base class specifier is `public`

`public`, `protected`, or `private`

- 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 also `public` in the derived class, so the objects of the derived class can access them.
  In the example `e07_1a.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.
- The access status of `private` members cannot be changed; they always remain `private`.

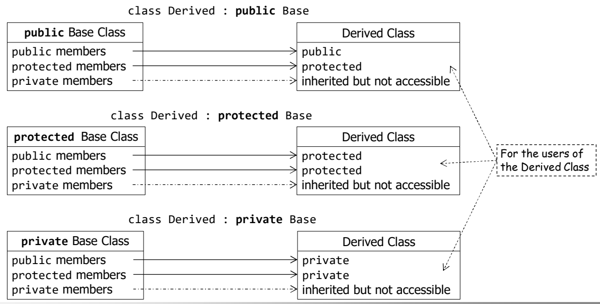
Protected inheritance (*protected derivation*):

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

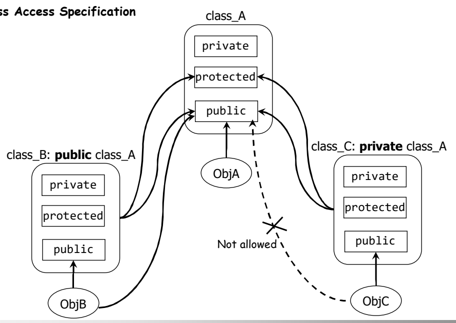
Private inheritance (*private derivation*):

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

Base Class Access Specification (cont'd)



Base Class Access Specification (cont'd)



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 col_point;       // A derived object
col_point.move(10, 20);       // Error! move is private
col_point.print();            // OK. Print is public again
```


Example: Private inheritance

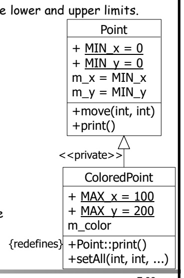
Problem:

- 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 `Point` class already checks the lower limits. We only need to add upper limits and implement mechanisms to check them.
- 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 check 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()`, which checks the upper limits.
- The `Point` class checks the lower limits, while the `ColoredPoint` checks the upper ones.
- 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, 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;
};
// The derived class checks the upper limit values
void ColoredPoint::setAll(int in_x, int in_y, Color in_color){
   if (in_x <= MAX_x) setX(in_x);            // setters of Point check 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.
- For each class, the responsibilities are clearly defined (***separation of concerns***).

Example: The `ColoredPoint` class has lower and upper limits (cont'd)

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

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

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

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

Example e07_4a.cpp

Redefining Access Specifications (cont'd):

- 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;          // Nonconstant methods are made private
  using Point::setX;
  using Point::setY;
    :
};

int main(){
ColoredPoint colored_point{ Color::Green };     // A green point
colored_point.setX(200);                        // Error! setX function in ColoredPoint is private
colored_point.move(200,200);                    // Error! move in ColoredPoint is private
colored_point.Point::move(200, 200);            // OK! Using the base name explicitly
```

Under `public` inheritance, the `move` in `Point` is still `public`. You can redefine it (7.28).

Example e07_4b.cpp

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 functions) may not be suitable for the derived class. These members should be redefined in the derived class.

Example:

- The `Point` class has a `setAll` and `print` function that sets and prints the properties of the `Point` objects.
- However, these functions are not appropriate for the class `ColoredPoint` because colored (specialized) points have more properties (e.g., color) to be set and printed. The algorithms can also be different.
- So, the `setAll` and `print` functions must be **redefined** in the `ColoredPoint` class.

```
class Point {
public:
  bool setAll(int, int);        // sets the coordinates
  void print() const;           // prints coordinates to the screen
    :                           // Other methods, e.g, reset()
};

class ColoredPoint : public Point {
public:
  bool setAll(int, int, Color);  // redefines the setAll function to set the color
  void print() const;            // redefines the print function, the signature is the same
    :                            // The method reset() is not redefined
};
```

The signatures of the original and redefined methods can be the same or different.

Example (cont'd): Redefining the methods of the `Point` class

- The `bool setAll(int, int, Color)` function of the `ColoredPoint` class hides the `setAll(int, int)` 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 `setAll` and two `print` functions.
- The base class members with the same name can be accessed using the scope resolution operator (`::`).

```
// ColoredPoint redefines the setAll function. This function sets the color as well
bool ColoredPoint::setAll(int new_x, int new_y, Color in_color){
   if (Point::setAll(new_x, new_y)) {          // calls setAll inherited from Point
              m_color = in_color;              // sets the color
              return true;                     // new values are accepted
    }
    return false;                               // new values are not accepted
}
// ColoredPoint redefines the print function. 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
}
```


Example (cont'd): Redefining the methods of the `Point` class

- The users (objects) of the `ColoredPoint` class normally employ the redefined methods.
- If the base class is public, `ColoredPoint` objects can also access the methods of `Point` using the scope resolution operator (`::`) when needed.

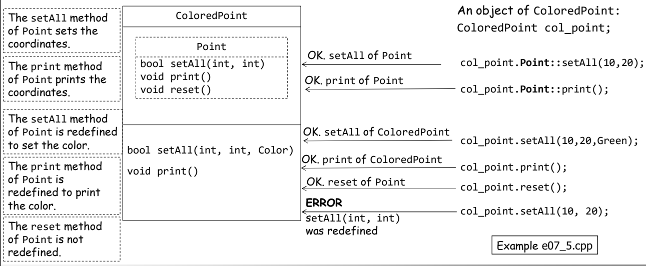
```
int main()
{
  ColoredPoint col_point{ Color::Green };   // A green point
  col_point.print();                        // print function of ColoredPoint
  col_point.Point::print();                 // print function inherited from Point
  col_point.setAll(10, 20, Color::Blue);    // setAll function of ColoredPoint
  col_point.setAll(10, 20);                 // ERROR! setAll of Point was redefined
  col_point.Point::setAll(10, 20);          // OK! setAll of Point, if public inheritance
    :
}
```

If the base class access specifier is `public`

The `ColoredPoint` class contains this method but it has been redefined. This is not function overloading.

Example (cont'd): Redefining the methods of `Point` in `ColoredPoint`

In this example, `Point` is the public base of `ColoredPoint`.



Example e07_5.cpp

Data members of the base class can also be redefined:

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 <u>two</u> methods: `void method()` and `void method(int)`.
- It has <u>four</u> data members: `int m_data1`, `string m_data1`, `int m_data2` inherited from `Base`, and `int m_data2`.

Example (cont'd): Name Hiding

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

OK, if `method()` of `Base` is public or protected.

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. If method() of Base is public
}
```

If the method in the `Base` is `public`, the objects can still access the redefined method using the name `Base`.

Since the `Derived` class redefines (hides) the `method()` of the `Base`, its objects cannot access the method of the `Base` directly (implicitly).

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_point` of the `ColoredPoint` class can also access the `print()` function of the `Point` class.

  ```
  col_point.Point::print();        // calls the redefined method of the Base
  ```
- However, this is inappropriate when the members of the base are not suitable 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 `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 creator 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 (cont'd):

Redefining the `move` function of the `Point` class under `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 creator of `ColoredPoint` 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
   :
};
```

Example e07_6.cpp

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


Function Overloading and Name Hiding in C++:

Function Overloading:

- Remember: Overloading occurs when two or more methods of the <u>same class</u> or multiple nonmember functions in the same namespace have the <u>same name</u> but <u>different parameters</u> (Slide 2.38).
- Since the <u>overloaded functions have different signatures</u>, the compiler treats them as distinct functions, so there is no uncertainty when we call them.
- Summary:
  - Overloading applies to methods of the <u>same class</u> or nonmember <u>functions in the same namespace</u> that have the same name.
  - Functions have <u>the same names</u> but <u>different input parameters</u>.

Name Hiding:

- Name hiding occurs when a derived class redefines the methods of the base class.
- The methods may have <u>the same or different parameters</u>, but they will have different bodies.
- Summary:
  - Name hiding happens only with <u>inheritance</u>.
  - Functions have <u>the same names.</u> The <u>parameters</u> can be <u>the same or different</u>.

Example e07_7.cpp

Overriding:

- Function **overriding** in inheritance facilitates **dynamic polymorphism**, which we will discuss in Chapter 8.

Constructors and Destructors in Inheritance

Default Constructor:

- If the `Base` class contains a default constructor, the `Derived` class 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}
{ }
```

The default constructor of `Point` (the base class) is called implicitly.
If `Point` does not contain a default constructor, this code will not compile.

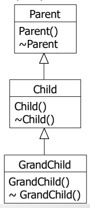
The order of object construction in inheritance:

- Since a derived class's object contains a base class's object inside it, the base object must be constructed before the rest of the object.
- Firstly, the subobject inherited from the `Base` is constructed (`Base` class constructor runs).
- Then, the remaining part of the `Derived` object is initialized (`Derived` class constructor runs).
- If a base class is derived from another class, this applies throughout the entire hierarchy.

Destructors in inheritance:

- 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 the reverse order of construction, i.e., the derived object is destroyed before the subobject inherited from the `Base`.

Example:




```
int main()
{
  GrandChild grandchild_object;
  std::print("..Program terminates..");
  return 0;
}
```

The Output:

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

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 member initializer 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{
 public:
   Point(int, int);   // Constructor to initialize x and y coordinates
     :
```

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

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

- 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 `Point`. **Error!**
There is no default constructor in `Point`.

Example e07_9a.cpp

Constructors with parameters (cont'd):

- If the `Base` class contains multiple constructors, the creator of the `Derived` class can call one of them in the member initializer list of its constructors.
- Unlike the default constructor, constructors with parameters are not invoked automatically.
- The creator 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{
 public:
   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 their member initializer lists.
  - `ColoredPoint::ColoredPoint(int in_x, int in_y) : Point{in_x, in_y} { }`
  - `ColoredPoint::ColoredPoint(Color in_color): Point{1}, m_color{in_color} { }`
  - `ColoredPoint::ColoredPoint(){ }`

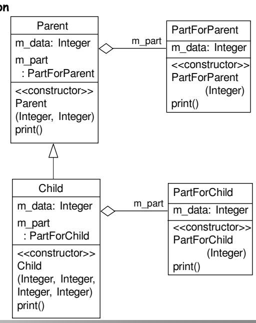
Example e07_9b.cpp

Constructors and destructors in inheritance with composition

- In OOD, "is-a" and "has-a" relationships can coexist.
- Remember: In composition, the objects' lifecycles are tied (6.14).

Example:

- 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 inheritance with composition (cont'd)

Default Constructors:

Example 1:

In this example, all classes have default constructors.

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

Example e07_10a.cpp

```
// *** The Derived Class
class Child : public Parent {
public:
  Child() {std::println("Child constructor"); }
  ~Child() { std::println("Child destructor"); }
private:
  PartForChild m_part;  // Part of the Child
  int m_data{4};
};

// -------- Main Program -------------
int main() {
  Child child_object{};  // An object of the Child
   ...
}
```

The order of construction:

```
PartForParent
Parent
PartForChild
Child
```

The order of destruction:

```
Child
PartForChild
Parent
PartForParent
```

Constructors and destructors in inheritance with composition (cont'd)

Constructors with parameters:

Example 2:

In this example, all classes only have constructors that accept parameters.

- Constructors of owners must initialize their parts (see 6.17).
- Constructors of child classes must initialize their parents.

Initialize the part

Initialize the data member

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


Example 2 (cont'd):

The order in the member initializer list is not important.
The Parent subobject is always 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 }
   {}
   ...
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();
  :
```


Example 2 (cont'd):

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

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

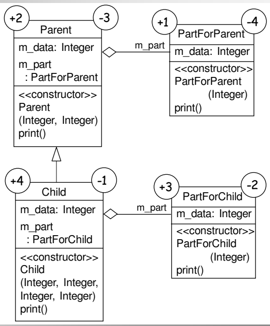
The order of construction:

PartForParent
Parent
PartForChild
Child

The order of destruction:

Child
PartForChild
Parent
PartForParent

Example e07_10b.cpp



Constructors and destructors in inheritance with composition (cont'd)

Dynamic Member objects (Pointers as members)

Remember: Instead of automatic objects, data members of an owner class can 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 initializer list of the constructor, not in the body.

Example 3a: Pointers as members. Dynamic objects are created in the member initializer 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 (cont'd): Pointers as members. Dynamic objects are created in the member initializer 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 member
{};
// Destructor of the Child
Child::~Child() {
   delete m_part;                 // Delete the part object
};
```

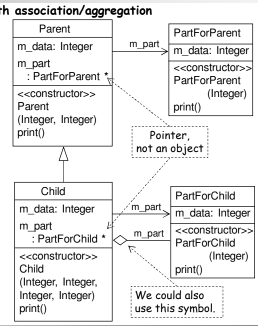
Example e07_10c.cpp

Constructors and destructors in inheritance with association/aggregation

Remember: In association and aggregation, the objects' lifecycles are NOT tied (6.9).

The whole (i.e., the owner) can exist without the part and vice versa.

- In the design on the right, the Parent and Child contain **pointers** to their parts, rather than objects of those members.
- Now, the programmer can determine the order of creation between the owner and the parts.
- The owner object can be created before the part, if necessary.
- The data members of the owner can be used to initialize the parts.
- The order between Parent and Child cannot be changed. The Child's constructor must always initialize the subobject inherited from the Parent first.



Dynamic Member objects (Pointers as members)

Changing the order of construction between the whole and the parts

If the owner class has pointers to parts,

- The programmer can decide when the parts will be created and destroyed.
- The dynamic objects can be created in the constructor's body instead of in the member initializer 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 parts.

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.

Example e07_10d.cpp

```
// Constructor of the Parent
Parent::Parent(int in_data1): m_data{ in_data1 }    // Only the data member is initialized
{                                                    // The body of the constructor
  m_part = new PartForParent{ m_data };              // Part object is created
}        // 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 creator 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 `Point`

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


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

- 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 Point
   :
};
int main()
{
 ColoredPoint colored_point1{ 10, 20 };    //Inherited constructor of Point
 ColoredPoint colored_point2{ 30 };        //Inherited constructor of Point
```

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

- In addition the `ColoredPoint` class can also have its own constructors:
  `ColoredPoint (int, int, Color);`

Example e07_11.cpp

The Copy constructor in inheritance

The default copy constructor:

- Remember: If the class creator does not write a copy constructor, the compiler supplies one by default.
- The default copy constructor will simply copy member-by-member the contents of the original into the new object.
- The default copy constructor will also copy the subobject inherited from the base class.

Example:

- What happens if we do not supply a copy constructor for our `Point` and `ColoredPoint` classes?
- Can the statement below compile and run correctly?
  `ColoredPoint colored_point2{ colored_point1 };`
  Check the example. Example e07_12a.cpp
- This program runs correctly because the compiler supplies default copy constructors for both classes.
- The default copy constructor of `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 in inheritance (cont'd)

The programmer-defined copy constructor in the derived class:

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

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

Only the data member is initialized.
It is not specified which constructor of `Point` to call.

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

Example e07_12b.cpp

- When we run this program, we see that the object `colored_point2` **is not an exact copy** of `colored_point1` (coordinates are different).
- The programmer-written `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 `Point` automatically.

The programmer-defined copy constructor in the derived class (cont'd):

- To fix the problem in the program `e07_12b.cpp`, we must call the `Point` copy constructor in the member initializer 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 `Point` is called explicitly.

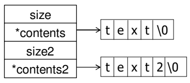
Example e07_12c.cpp

- Note: The `Point` copy constructor is called using the object of `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 no 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.

The Copy constructor and the assignment operator under inheritance (cont'd)

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 `String` class we have 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 String
     :
```

Example e07_13.cpp

Inheriting from the library

We can also derive new classes from classes in a library, just like we did from programmer-written classes.

Example: A colored string

- Assume that according to requirements, we need strings with colors.
- We can derive a class `ColoredString` from class **`std::string`**.

  `class ColoredString : public std::string {...}`
- This new class will inherit all members (constructors, operators, getters, setters, etc.) of `std::string`. So, we **reuse** `std::string`.
- Remember that we can add new members and redefine inherited members.
- We can use objects of `ColoredString` as we use standard `std::string` objects.

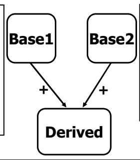
```
int main() {
 ColoredString firstString{ "First String", Color::Blue }; // Constructor
 ColoredString secondString{ firstString };                // Copy constructor
 secondString += thirdString;                              // += operator of std::string
 secondString.insert(12, "-");           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

Example e07_14.cpp

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

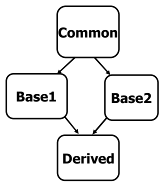
```
class Derived : public Base1 , public Base2{
public:
  Derived();
  ~Derived();
  void f1();
  void f2(int, char);
  void f5();
};
```

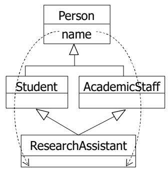
```
int main() {
  Derived d;
  d.f1();              // Derived::f1
  //d.f2(1);           // Error!
  d.Base2::f2(1);      // Base2::f2
  //d.f3();            // Error! Ambiguous
  d.f4();              // Base1::f4
```

Example: e07_15.cpp

Repeated base classes in multiple inheritance

(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) (cont'd)

- Suppose there is a data item, `common_data`, in `Common`. `Base1` and `Base2` are derived from `Common`.

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

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

The objects of `Derived`, which is derived from both `Base1` and `Base2`, 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
  }
};
```

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 the diamond problem (repeated base classes) 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
   {   };
```

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 outside 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

In public inheritance:

- If a class Derived has a public 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.

- Conversion in the opposite direction (from a pointer to Base to a pointer to Derived) must be explicit.

  A point is not always a colored point.

  Example:

```
class Base {...};
class Derived : public Base {...};
int main() {
   Derived derived_obj;
   Base *base_ptr = &derived_obj;                     // OK! implicit conversion
   Derived *derived_ptr = base_ptr;                   // ERROR! The base is not derived
   derived_ptr = static_cast<Derived *>(base_ptr);    // 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 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.

  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 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 example e07_17.cpp on the next slide.

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

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

Example: e07_17.cpp

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, **a reference to Base can also point to objects of Derived**.
  - If a function expects a reference to Base as a parameter, we can call this function by sending a reference to the Derived object as an argument.

    Remember, in slide 7.54, we call the copy constructor of Point by sending an object of ColoredPoint 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 no type mismatch because ColoredPoint is a Point.

References to objects and inheritance (cont'd)

Example:

Remember: In example e06_5.cpp, there is a class GraphicTools that contains tools that can operate on Point objects.

For example, the method distanceFromOrigin of GraphicTools calculates the distance of a Point object from the origin (0,0).

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

Since a colored point is a point, we can also use this method of GraphicTools 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
   :
```

Example: e07_18.cpp

Pointers to objects in private inheritance

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

The creator of the derived class does not permit users of that class to use the inherited members because 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 exhibit the same behaviors as their base objects.

Pointers to objects under `private` inheritance (cont'd)

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 creator.

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.

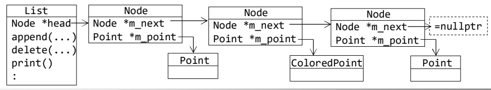
Example: A linked list that contains `Point` and `ColoredPoint` objects.

- 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 place `Point` and its child objects (e.g., `ColoredPoint`) into a list, without modifying their code, we will define another type of class called `Node`.

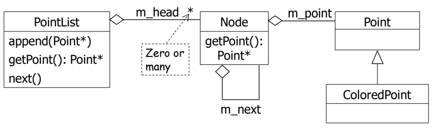
  A `Node` object will have two members:

  `m_point`: A pointer to the `Point` type (the element in the list). It can also point to child objects.

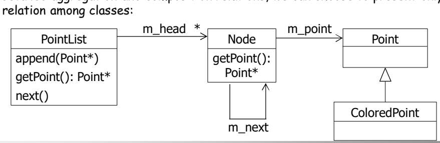
  `m_next`: A pointer to the next node in the list.



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 choose to present only the general association relation among classes:



Example: A linked list that contains `Point` and `ColoredPoint` objects (cont'd)

```
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 do not 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 (cont'd)

Example: e07_19.zip

```
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!

  delete ptrPoint1;
  delete ptrColPoint1;
  :
```

`setColor` is not a member of `Point`.
You cannot access members of `Derived` through a pointer to `Base`.

In Chapter 8, we will extend this program by adding **`virtual` (polymorphic)** methods.

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.
- Inheritance enables us to use polymorphism, which we will cover in Chapter 8.