When you create a derived class (a class that inherits from another class), C++ requires a specific syntax structure.  
The order matters because the compiler must first know:

1. It is a class
2. What the new class name is
3. Which base class(es) it inherits from
4. How it inherits from them (access level)

**Required Elements in Order**

Here’s the pattern:

class DerivedClassName : accessSpecifier BaseClassName {

// class body (data members, member functions, etc.)

};

Let’s explain each part:

| Element | Purpose |
| --- | --- |
| class keyword | Tells the compiler you are declaring a class. |
| DerivedClassName | The name of the new class you are creating. |
| : (colon) | Indicates that what follows is the base class list (the classes you inherit from). |
| accessSpecifier | Sets how base class members are inherited (public, protected, or private). |
| BaseClassName | The class you are inheriting from (must be declared earlier). |
| { ... } (braces) | The body of the derived class: new variables and functions you want to add or override. |
| ; (semicolon) | Ends the class definition. Mandatory. |

**Example**

Base class:

class Person {

public:

void showName();

};

Derived class:

class Employee : public Person {

int empId;

public:

void showDetails();

};

Explanation:

* class → starts the class definition
* Employee → new class name
* : public Person → inherits from Person using public inheritance
* { ... } → contains new members
* ; → ends the class definition

**What the Access Specifier Does**

| Access in Base Class | Public Inheritance | Protected Inheritance | Private Inheritance |
| --- | --- | --- | --- |
| public members | remain public | become protected | become private |
| protected members | remain protected | remain protected | become private |
| private members | not inherited | not inherited | not inherited |

This matters because it controls how outside code can access the base class members through the derived class.

**Key Points**

* You can list multiple base classes separated by commas, each with its own access specifier.
* Always put the semicolon ; at the very end of the class definition, or you get a compile error.

When you create a derived class (child class) from a base class (parent class), most of the base class’s members are inherited — but not everything is automatically inherited.

And also, private members of the base are never directly accessible from the derived class.

**Items That Are Not Inherited**

Even though the derived class conceptually "has" these from the base, the compiler does not copy them into the derived class. These include:

| Item | Why it’s not inherited |
| --- | --- |
| Constructors | Each class is responsible for building its own objects. The derived class must call the base’s constructor explicitly in its initializer list, not inherit it. |
| Destructors | Each class must clean up its own members. The base destructor runs automatically after the derived destructor finishes, but it isn’t inherited. |
| Friend functions | A friend is not a class member, just a non-member function with special access to private/protected parts. Friend status is not passed to derived classes. |
| Overloaded new operator | If the base class overloads operator new, it applies only when you allocate base objects. Derived class objects must define their own if they want custom allocation. |
| Overloaded assignment operator (operator=) | Each class must implement its own correct copy behavior (deep copy/shallow copy), because member composition often differs between base and derived. |

So: these things exist in the base, but are not automatically copied down to the derived class definition.

**Access to Private Members**

Even if something is inherited (like ordinary public or protected data members and methods), the private members of the base are never directly accessible from the derived class. The derived class must use base class public/protected member functions to interact with those values.

**What You Must Do Instead**

If your derived class needs any of these items (constructors, destructor, friend, etc.), you must explicitly define them again in your derived class, for example:

class Base {

public:

Base(int x) { /\* ... \*/ }

~Base() { /\* ... \*/ }

};

class Derived : public Base {

public:

// Must explicitly define its own constructor

Derived(int x, int y) : Base(x) {

// derived part initialization

}

// Its own destructor (optional)

~Derived() { /\* ... \*/ }

};

* Notice how the Derived constructor calls the Base(x) constructor in its initializer list.
* This is the only way to "use" base constructors — they aren’t inherited automatically.

**Summary**

Not inherited → must explicitly write them in derived class if needed

* Constructors
* Destructors
* Friend functions
* Overloaded operator new
* Overloaded operator=

Private base members → not accessible directly from derived

* Access only via base’s public/protected methods.