Simple Inheritance

Overview

- Colon Syntax
- Inheritance and IS A
- Derived Object Creation
- Initializing Base Object
- Visibility of Base Members
- Overriding Members

Colon Syntax (1/2)

- Normal initialisation of data members
  - Data member is created
  - Data member is assigned

```
Point::Point()
{
  m_x = 0.0;
  m_y = 0.0;
}
```

3

Colon Syntax

Normally you initialise all the data members of a class in its constructor. When an object of that specific class is created the constructor is called and initialises all the data members for that object.

Inside the body of the constructor the system already created the members in memory and initialised them with a value unknown to us. Therefore we initialise the values with a value we want them to be.

The creation steps normally for members are:

- Data members are created and initialised by the system.
- We initialise them with our values.

Colon Syntax (2/2)

- Syntax

```
ClassName::ClassName(): Member(constructor arguments)
{
}
```

- With colon syntax
  - Data member is assigned at creation
  - Save extra assignment 节省
  - Only for constructors

```
Point::Point(): m_x(0.0), m_y(0.0) {}

Circle::Circle(): m_centre(0.0, 0.0), m_radius(10) {}
```

18

COLON SYNTAX

The colon syntax enables us to give a value the system uses to initialise the data members upon creation. Using this colon syntax the data members are initialised upon creation.

When data members are of a user-defined type we can specify which constructor the system needs to call to create and initialise that specific member. The object otherwise would be created (constructor call) and initialised (a call to another member function) two member function calls which can be substituted for one call using he colon syntax.

Const/Reference Data Members

- Members need to be initialised before object is created
- Not possible to use normal assignment in constructor code
- Colon syntax must be used for
  - Const data members
  - Reference data members

5

CONST DATA MEMBERS

There are some situations where the colon syntax is mandatory. The use of const data members and reference members is one of them. These members need to be assigned upon creation, the only way to accomplish this is by using the colon syntax.

Example Const/ Reference Data Members

```
class Simple
{
private:
  const int m_val;
  const Other& m_smp;

public:
  Simple(int newVal, const Other& o); // Constructors
};


Simple::Simple(int newVal, const Other& o): m_val(newVal), m_smp(o)
{
}
```

6

CONST AND REFERENCE DATA MEMBERS

Reference data members seem to be quite logical. When you have two objects that need each other that need some kind of reference. For example an employee object and a company object. They often need to reference one another. In C we would use a pointer to reference two variables. Some C++ programmers prefer to use the reference operator. We advise however to not use this one inside a class. When you use a reference inside a class it needs to be initialised using a constructor. The constructor will need some object reference.

```
class Company
{
};

class Employee
{private:
       Company& m_Comp;
public:
       Employee(Company& comp);
};

Employee::Employee(Company& comp) : m_Comp(comp)
{
}
```

If we call this constructor we need to supply an object to be referenced by the data member.

```
void main()
{
      Company comp;
      Employee emp(comp)
}
```

The above example will correctly run because our company object comp lives longer than the employee object.

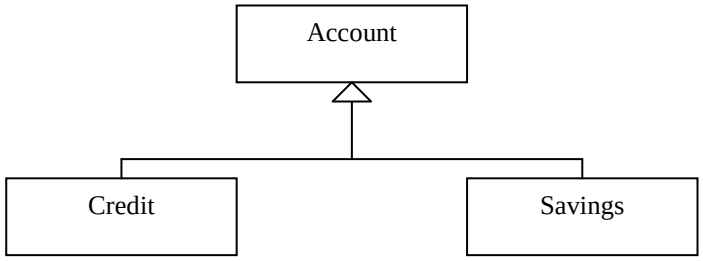
```
void main()
{
      Employee emp(Company())
}
```

The above example will compile as well. When we run the program there is however big change that it will crash. The object we supply to store a reference to is only temporarily. It is an anonymous object. We therefore suggest using pointers for these types of references because pointers already alert the programmer of a life-expectancy problem.

INHERITANCE EXAMPLE

ACCOUNTING

In the next two modules we will show how inheritance is implemented in C++. These two modules will use an inheritance scheme of different types of accounts. There are two types of accounts: a credit account with a credit limit and a savings account with a savings percentage.
The UML object diagram would look as follows:



The specific attributes of the classes are:

Account

balance	: Current balance of account
account_nr	: Account number

Credit

credit_limit	: The credit limit of the credit account

Savings

interest	: Interest rate for savings account

Aggregation: a relationship where the child (embedded object) can exist independently of the parent (outer object). Example: Class (parent) and Student (child). Delete the Class and the Students still exist.
Composition: a relationship where the child (embedded object) cannot exist independent of the parent (object). Example: House (parent) and Room (child). Rooms don't exist separate to a House.

Inheritance and IS A

- Class is a kind of ...
- New class as specialisation of given class
- New class is called derived class
- Given class called base class
- Derived class ADDS functionality

7

INHERITANCE

We now come to the important topic of inheritance and how it is implemented in C++. Having constructed a number of classes we may at some stage wish to adapt these to work in other, possibly more specialised situations. The approach adopted in C++ (as in other object oriented languages) is to create a so-called derived class of the original class and add the new functionality or modify existing functionality to suit our needs. The original class is not modified in any way and in fact its functionality, which is not changed, also remains valid for the new derived class.
A given class may have one or more classes from which it is derived. In the former case we speak of single inheritance and in the latter case we speak of multiple inheritance.

Specialisation Scenarios

- Derived class can
  - Add extra state
  - Add member functions
  - Override member functions of base class

8

SPECIALISATION; CREATING NEW DERIVED CLASSES

Having constructed a given class we can create new ones by the inheritance mechanism. You should be careful however, not to use inheritance when aggregation is more suitable. For example, a Circle is often created as a derived class of Point, which is incorrect. The correct approach is to say that a Circle consists of a Point and a radius. ISA — wrong! HASA — bingo!

In the following sheets we will use an inheritance scenario of a class Credit which is derived of a class Account.

For the moment we can state the following facts concerning the relationship between the base class Account and its derived class Credit:

- Class Credit inherits all the member data and member functions of Account with the exception of constructors, destructors and the assignment operator.
- Class Credit can add completely new member data and member functions to its declaration.
- Class Credit can 'undo' or 'overwrite' a member function of Account; that is, Credit can create a member function with the same name as in class Account.
- Class Credit has at least as much state as class Account (i.e. member data); it never contains less. Thus, state cannot be 'undone' in going from a base class to a derived class.

Inheritance Syntax

```
class Credit: public Account
{
};
```

- Credit is publicly derived from Account
- Credit inherits functionality and structure of Account

9

PUBLIC INHERITANCE

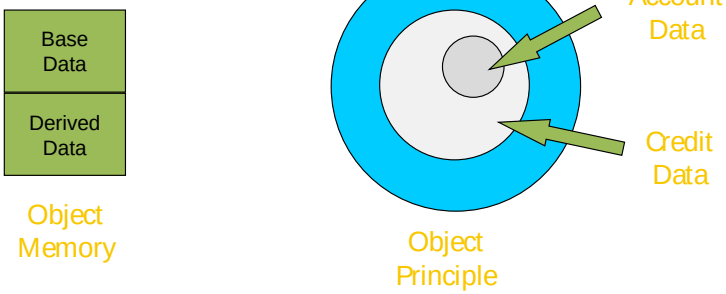
The example shown on the sheet is called public inheritance. The derived class Credit is publicly derived of Account. The Credit class inherits all the behaviour of the Account class.

PUBLIC INHERITANCE AND PRIVATE BASE MEMBERS

When deriving a class from a base class the private part of the base class is inherited but cannot be accessed by clients (including the derived class) in general. When we have a derived object we speak of a derived part and a base part. The private members of the base class can only be accessed by the base part of the object.

Data Members and Inheritance

- Derived object has base data in it but it is not accessible by derived members



10

DATA MEMBERS AND INHERITANCE

When deriving a class from a base class you inherit all the members of that base class. Inheriting members does not necessarily mean that you can access those members in the derived class. The data members of the base class (private part) cannot be used in member functions of the derived class although they are part of that derived class.

Base Class

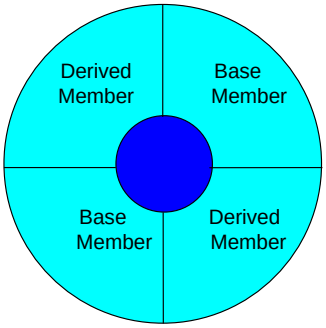
```
class Account
{
private:
       double balance;     // Balance of account
       long account_nr;    // Account number

public:
       long accountnr();   // Account number
       ...
};
```

Derived Class

```
class Credit: public Account
{
private:
       int credit_limit;   // Credit limit of credit account

public:
       void display();     // Display chracteristics
       ...
};

void Credit::display()
{
       cout << "Account number : ";
       // cout << account_nr << endl; // NOT POSSIBLE private of base class
       cout << accountnr() << endl;
}
```

Member Functions and Inheritance

- Member functions of base become part of public part of derived
- Derived object guarantees base functionality



11

MEMBER FUNCTIONS AND INHERITANCE

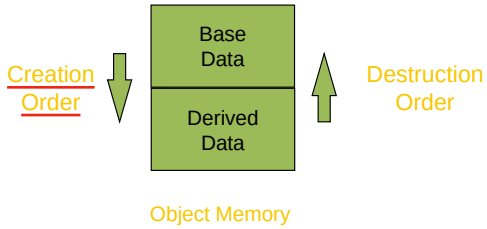
The public members of the base class become part of the public interface of the derived class. The derived class inherits the base functionality.

Members in Derived

```
class Account
{
private:
public:
       long accountnr();   // Account number
       ...
};

class Credit : public Account
{
private:
public:
       void display();     // Display characteristics
       ...
};

void main()
{
       Account acnt;
       Credit crt;

       acnt.accountnr();   // Account::accountnr()
       crt.accountnr();    // Account::accountnr()
       crt.display();      // Credit::display()
}
```

Inheritance and Object Creation

- When data of derived object is created also base data is created



12

Objects And Inheritance

When we create an object the object's constructor is called. If the object is a derived object it will consist of a base part and a derived part. Both these parts have data members which need to be initialised. The constructors take care of the initialisation. The base constructor initialises the base part of the object and the derived constructor the derived part.

Creation Order

If the derived object is created the data members of the base part will be created first. The base class constructor will initialise those members. After the base data members are created and initialised the data members of the derived object are created after which the object exists.

Destruction Order

At object destruction the destructor of the object to be destroyed is called. If the object is a derived object its destructor gets called first after which the destructor for the base part is called.

Using Base Class Constructors

- Specify which base constructor to call in source file

```
Credit::Credit(): Account()
{
    The constructor of derived class
}
```

- Pass values to base constructors

```
Credit::Credit(long accountNumber): Account(accountNumber)
{
}
```

- Only for constructors

13

USING BASE CLASS CONSTRUCTORS

As mentioned when the derived object gets created the base part of that object is created first. For the initialisation of that base part the base constructor is called. In the constructors of the derived class you can specify which constructor to call for the base part.

In the source file we have to place a colon(:) after the function prototype followed by the specification of the base class constructor to call.

```
Credit::Credit() : Account()
{
}
```

Colon Syntax

When we do not specify which constructor to call, the default constructor of the base class is called. It is therefore better to specify which constructor needs to be called.

PASSING VALUES TO BASE CONSTRUCTORS

It is possible to pass values to a constructor of the base class.

```
Credit::Credit() : Account(12345l)
{
}

Credit::Credit(long accnr) : Account(accnr)
{
}
```

Passing Values to Base Class

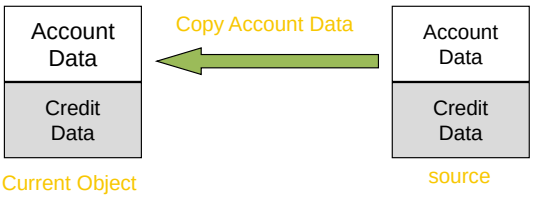
Not For Destructors

There is only one destructor in the base class so we do not have to specify which base class destructor to call.

Copy Constructor of Derived Class

- Divided in two steps
- Let base class constructor copy base part of object

```
Credit::Credit(const Credit& source): Account(source)
{
  …
}
```



14

Copy Constructors And Inheritance

The copy constructor also plays an important role when using inheritance. The copy constructor functionality of the derived class can be split into two parts.

Step 1 Calling The Base Class Copy Constructor

When creating constructors for the derived class we have to specify which constructor needs to be called from the base class.
The idea of the copy constructor is to copy the contents of the argument object into the current object. The argument object is a derived object consisting of a derived part and a base part. Both these parts need to be copied in the current object. The base part is not accessible to the derived constructors. Therefore we need to call a constructor of the base class that copies the base data members in the current object.

```
Credit::Credit(const Credit& crt) : Account(crt)
{
}
```

Call Base Copy Constructor

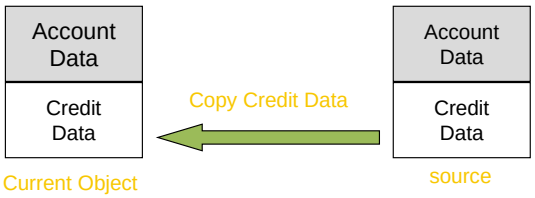
The argument for this copy constructor is the object to be copied. In this case we give the copy constructor of Account the Credit object that is the argument of the current copy constructor. This is possible although the copy constructor of the base class needs a base object; the derived object behaves (IS A) like a base object.

If we do not specify the copy constructor as the constructor of the base class that needs to be called the default constructor is called. This will result in an incorrect copy; the base part will not be copied.

Copy Constructor of Derived Class

- Copy own data members

```
Credit::Credit(const Credit& source): Account(source)
{
  m_credit=source.m_credit;
}
```



15

STEP 2 COPYING DERIVED DATA

When the base class constructor copied the base data we need to copy the data members of the derived part as well. After copying these derived data members we have created a complete copy of the argument object.

Overriding Functions

- Declare same function as in base class (same signature)
- If no overridden function then inherited from base class

16

OVERRIDING FUNCTIONS

If we declare a member function in the derived class that has the same signature as a member function of the base class it is called overriding. An overridden function is called for derived objects.

Overriding

```
class Account
{
private:
        ...
public:
        void display();      // Display characteristics
        ...
};

class Credit: public Account
{
private:
public:
        void display();      // Display characteristics
        ...
};

void main()
{
        Account acnt;
        Credit crt;

        acnt.display();      // Account::display()
        crt.display();       // Credit::display()
}
```

This is useful when you want to define new behaviour for certain function of the base class.

CALLING OVERRIDDEN FUNCTIONS OF THE BASE CLASS

It is still possible to call the overridden function by specifying the class where the function is situated.

Calling Base Function

```
void main()
{
       Credit crt;

       crt.display();              // Credit::display()
       crt.Account::display();     // Account::display()
}
```

Calling Base Class Functions in Derived Member Functions

- If you want to use the base class behaviour use the scope resolution operator

```
void Credit::Display()
{
  Account::Display();
  cout << "Credit limit : " << m_credit;
}
```

17

CALLING BASE CLASS FUNCTIONS IN DERIVED MEMBER FUNCTIONS

Sometimes you want to use the behaviour of a base class member function plus something extra. The copy constructor is a good example. The copy constructor called the base class copy constructor using the colon syntax after which the derived members were copied.

When you want to use behaviour of the base class other than its constructors you have to call that function in the member function of the derived class.
The example on the sheet calls the display() function of the base class (Account).

What is NOT Inherited

- Constructors are not inherited

```
Credit crt(3456); // Constructor with long not inherited from Account
```

- Destructor is not inherited
- "Partial" inheritance of operators
- Friends are not inherited

18

INHERITANCE OF MEMBER DATA AND MEMBER FUNCTIONS

When creating new derived classes from existing classes you should inherit that functionality which is common to both the base and derived classes. All member data is inherited in going from the base class to the derived class.
Member functions and operators are inherited with the exception of the following:

- Constructors.
- Destructors.
- Partial operators.

Destructors are also a special case and are discussed in the module on polymorphism.

INHERITANCE AND CONSTRUCTORS

Constructors are not inherited from base to derived classes. This is a deliberate design policy.

No inherited Constructor

```
class Account
{
private:
       ...
public:
       Account();          // Default constructor
       Account(long accnr);// Constructor with long
       ...
};

class Credit : public Account
{
public:
       Credit();           // Default constructor
       ...
};

void main()
{
       Account acnt;        // Default constructor Account::Account()
       Account acnt2(4567l);// Constructor Account::Account(long)

       Credit crt;               // Default constructor Credit::Credit()
       // Credit crt2(12345l);   // Account::Account(long) not inherited
       ...
}
```

It is not possible to create a Credit object by using the constructor with a long from the base class. If the base class has a constructor with a certain argument the derived class needs to define the same functionality if it is required that the same kind of constructor should be available.

If we want to be able to create a Credit object with a long we have to create a Credit constructor with a long argument that will call the base class version via the colon syntax.

Partial Operator Inheritance

- Operator that is inherited only has behaviour for base object

```
Credit crt1, crt2;
if (crt1 == crt2) { … }  // Account::operator == ()
```

- Can give unwanted results
- Have to override overloaded operators in derived class if needed

Indeed, for any class, if we don't create an assignment operator in the derived class, the system will automatically create one for us.

19

PARTIAL COMPARISON

If we do not create the comparison operator in the derived class we inherit it from the base class. This means that we inherit the comparison behaviour of the base class. The comparison operator of the base class compares the elements of the base class and not the derived members.
Suppose we did not specify the comparison operator in the class Credit.

Incorrect Comparison

```
class Account
{
public:
        ...
        bool operator == (const Account& acnt);
};

class Credit : public Account
{
public:
        // No comparison operator
};

void main()
{
        Credit crt1, crt2;

        if (crt1 == crt2) { ... }  // Account::operator==() is called
}
```

The comparison operator of the Account class compares the data members of the base part with the crt object. It will not compare the members of the derived class. This is why it is called partial inheritance.

The rule is to call the operator of the base class in the derived class

Correct Comparison

```
bool Credit::operator==(const Credit& crt)
{
       bool result=Account::operator==(crt);

       result &= credit_limit == crt.credit_limit;
    and operator ,  且运算
       return result;
}
```

Access Specifier Protected

- Protected members
  - Can be accessed in derived classes
  - Cannot be accessed by outside world

20

A NEW LEVEL OF ACCESS CONTROL; PROTECTED AREAS

We have discussed the topics of private and public members. Only member functions and friends of the class in which they are declared can use private members. Public members can be used b any code.
C++ has a third level of access control (by using the protected specifier).
Protected members can be used only by the following:

- Member functions of the class.
- Member functions of friends of the class.
- Member functions of derived classes of the class.
- Friends of the derived class.

Private:
1. member function of the class
2. friend function or member function of friends of the class

Protected and Private Inheritance

- Public members of Account protected in Credit

```
class Credit: protected Account {};
```

- Public members of Account private in Credit

```
class Credit: private Account {};
```

- Private inheritance almost the same as aggregation

21

PROTECTED AND PRIVATE INHERITANCE

When you derive a class from another class you have to specify how this derivation is done. Normally you use the public access specifier. This way the public member functions of the base class will be made public in the derived class. Using inheritance this way is straightforward and 'good' inheritance. The users of the derived class can use the functions of the base class. It is however possible to use protected or private inheritance, private being the default if the specifier is left out. Using this technique you make the public part of the base class private or protected in the derived class. The derived class is derived of the base class but its functions are not accessible to the users of the derived class. This sounds much like aggregation and should be avoided in your code. Whenever you want to use this technique try using aggregation or composition instead.

Behaviour of Derived Objects

- Derived objects can behave as base objects
  - Derived object guarantees to have base functionality
- Where a base object reference is expected derived object can be passed (substitutability principle)
- Can assign derived object to base object not vice versa
  - Base object does not guarantee derived functionality

22

BEHAVIOUR OF DERIVED OBJECTS

Because a derived class is a base class you substitute a base class reference with a derived reference. So all functions that have an argument of base reference can be called using a derived object. This is not true if you have used protected or private inheritance, one more reason to avoid it.