8 Inheritance and Polymorphism

With the knowledge of the previous lectures, we can finally look into working with inheritance and polymorphism in C++. You can find more details on this topic in (Prata 2012, Chapter 13). For a comparison with Java, see (Weiss 2004, Chapter 6). More details on specific aspects can be found in Scott Meyers's book (Meyers 2005, Chapter 6).

We will use (mostly) C++ terminology for inheritance: a ***derived class*** inherits from a ***base class***. The following table shows alternative terminology used in other languages and in general OOP discourse: 260

C++	Alternatives	
Base class	Parent	Super class
Derived class	Child	Subclass

Inheritance may be `public` (the most common case) or `private`, depending on the keyword used before the base class in the derived class declaration: 261

```
class Base { .... };
class D1 : public Base { .... };     // public inheritance
class D2 : private Base { .... };    // private inheritance
```

8.1 What is inheritance?

Perhaps the most confusing aspect of the word "inheritance" is that in C++, it is used with several quite different meanings:

1. Inheritance as an "is-a" relationship. is-a

   Inheritance as an "is-a" relationship models the human urge to form hierarchies of classification ("taxonomies"). Each of the following sets contains its successors: living things, animals, mammals, dogs, terriers. Read backwards, these sets form an is-a hierarchy: a terrier ***is a*** dog, a dog ***is a*** mammal, a mammal ***is an*** animal, etc. taxonomy

   The important feature of a taxonomy is that a smaller set ***inherits*** all of the behaviour of the larger sets that contain it. A dog has ***all*** of the properties of a mammal – otherwise it wouldn't be a mammal. (We discuss apparent exceptions to this rule below.)

   There are two kinds of is-a relationship, depending on what gets inherited. A base class declares various functions, and the derived class(es) may inherit:

a) the declarations of the function only, or

b) the declarations and definitions of the functions.

interface inheritance and implementation inheritance

We refer to the first case as ***interface inheritance*** and the second case as ***implementation inheritance***.[31]

2. Inheritance as "implemented using" (a "has-a" relation). For example, we could implement a `Stack` by inheriting a `vector`, giving a "stack implemented using vector" class. The important point here is that `vector` and `Stack` provide different sets of functions, but the `Stack` functions are easily implemented using the `vector` functions. However, users of `Stack` must not be allowed to use the `vector` functions because that would violate the integrity of the stack.

The implementation mechanisms for these kinds of inheritance in C++ are, roughly:

1. `public` inheritance implements "is-a"
   a) abstract base class with pure `virtual` functions (see Section 8.2.1 on page 154)
   b) base class with `virtual` declarations and default definitions
2. `private` inheritance implements "using" (or "has-a")

abstract base class

The distinction between 1(a) and 1(b) can become blurred. A base class with one or more pure virtual functions is called an ***abstract base class*** (or "ABC") and defines, in effect, an interface: this is 1(a). A base class with implemented virtual functions is a complete, working class that can be specialized by derived classes that inherit some or most of its implementation and modify the rest: this is 1(b). In between, there are base classes with a mixture of pure and impure virtual functions, defining a sort of partially-implemented interface.

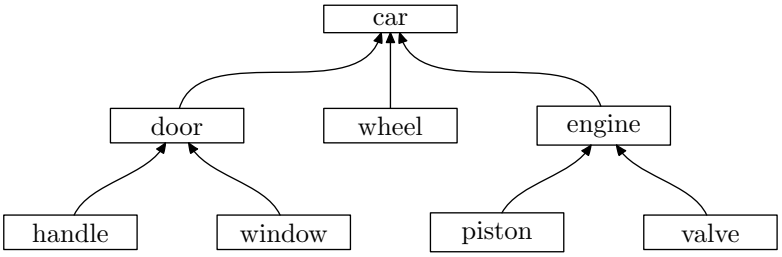

Figure 76: Not an "is-a" inheritance hierarchy

is-a vs. has-a

262

It is very important not to confuse "is-a" with "has-a". In some (older) books, you may see things like Figure 76 described as "inheritance hierarchies". But this is a "has-a" hierarchy: a car has an engine, an engine has pistons, etc. It is absurd to say "a valve is an engine" or "a handle is a door". Hierarchies of this kind are represented by ***layering*** (also called ***composition***), in which an instance of class `Car` contains instances of classes `Door`, `Wheel`, and `Engine`, etc. This confusion stems from the fact that C++ can support both "has-a" hierarchies (with the `private`

[31] These correspond roughly to `implements` and `extends` in Java. Note, however, that the Java keyword `implements` corresponds to ***interface inheritance*** in C++ and the Java keyword `extends` corresponds to ***implementation inheritance*** in C++.

inheritance) and "is-a" hierarchies (through `public` inheritance). However, in modern object-oriented programming, inheritance generally refers to "is-a" hierarchies (also called ***taxonomies***) only; and in this course, we focus almost exclusively on this type of inheritance (we'll briefly discuss `private` inheritance in Section 8.5.1 on page 174).

Figure 77 shows a popular example used to demonstrate the "difficulties" of inheritance. There are various solutions for this "problem". One solution is to redefine `Penguin::fly`:

263 264

```
class Penguin : public Bird
{
    void fly() { throw PENGUINS_CANNOT_FLY_EXCEPTION; }
};
```

This is not a good solution, because it violates the assumption in base class `Bird` that "birds fly". Figure 78 on the following page shows a better solution. It is obtained by realizing that the class hierarchy that we have defined is incomplete: there are birds that fly and birds that don't fly. The revised class hierarchy easily accommodates kiwis, ostriches, turkeys, and cassowaries.

265 266

8.1.1 Constructors and destructors

Constructors and destructors work in a special and well-defined way with derived classes.

- When a constructor of a derived class is called, the constructor of the base class is invoked first, then the constructor of the derived class.
- When the destructor of a derived class is called, the destructor for the derived class is invoked first, then the destructor for the base class.

```
class Bird
{
public:
    virtual void fly();
    ....
};

class Penguin : public Bird
{
    ....
};

int main()
{
    Bird *pb = new Penguin;
    pb->fly();                // oops - penguins can't fly!
}
```

Figure 77: The problem of the flightless penguin

```
class Bird
{
public:
    // no definition of fly()
    ....
};

class FlyingBird : public Bird
{
    virtual void fly();
};

class NonFlyingBird : public Bird
{
public:
    // no definition of fly()
    ....
};

class Penguin : public NonFlyingBird
{
    ....
};
```

Figure 78: Recognizing that some birds don't fly

267 This behaviour is illustrated by the code in Figure 79 on the next page. Executing this program
269 gives the output:

```
Construct Parent
Construct Child
Construct GrandChild
Destroy GrandChild
Destroy Child
Destroy Parent
```

8.1.2 Slicing

A derived class inherits data members from its base class and may define its own data members. Consequently, a derived class instance `d` is as large or larger than an instance `b` of the corresponding
270 base class. There are several consequences: see Figure 80 on page 154.

- The assignment `b = d` is allowed. Since the data in `d` does not fit into `b`, it is not copied: we say that `d` is ***sliced***. Slicing occurs not only during assignment but also when a derived object is passed or returned by value and the destination is a base class object.
- The assignment `d = b` is not allowed because it would leave the derived data in `d` undefined.

```
class Parent
{
public:
    Parent() { cout << "Construct Parent" << endl; }
    ~Parent() { cout << "Destroy Parent" << endl; }
};

class Child : public Parent
{
public:
    Child() { cout << "Construct Child" << endl; }
    ~Child() { cout << "Destroy Child" << endl; }
};

class GrandChild : public Child
{
public:
    GrandChild() { cout << "Construct GrandChild" << endl; }
    ~GrandChild() { cout << "Destroy GrandChild" << endl; }
};


int main()
{
    GrandChild g;
}
```

Figure 79: Constructors and destructors in inheritance hierarchies

- It is usually a mistake to use the base class of a hierarchy as a template argument. Consider: 271

  ```
  vector<Base> bases;
  Derived der( .... );
  bases.push_back(der);
  ```

  Since the argument of `push_back` is passed by value, `der` is sliced; data in `Derived` but not in `Base` is lost. To avoid this problem, use pointers, as in `vector<Base*>`.

Slicing does not occur when we address the object through references or pointers. Suppose that `pb` is a pointer to a base class instance and `pd` is a pointer to a derived class instance. Then:

- The assignment `pb2 = pd` is allowed. After the assignment, `pb2` will point to the ***complete object***, containing base and derived data, but only base class functions and data will be accessible, because of the type of `pb2`. Assignments of this kind are called ***upcasts*** because they go "up" the class hierarchy, from derived class to base class. upcasts
- The assignment `pd = pb1` is not allowed. Allowing it would give the program apparent access to functions and data of the derived class, but those fields do not exist. Assignments of this kind are called ***downcasts***, because they go "down" the class hierarchy. downcasts

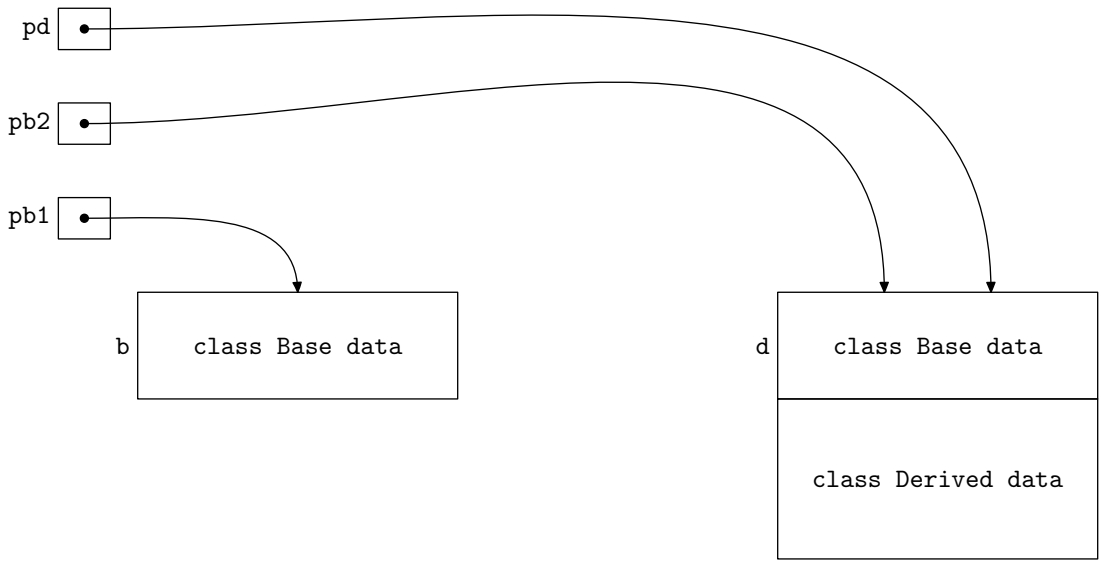

Figure 80: Base and derived objects

- References work in the same way as pointers: whenever we have the address of an object (that is, a reference or a pointer to it), slicing does not occur and dynamic binding works.

For now, the thing to remember is: ***upcast good, downcast bad***. We will discuss these casts in more detail later.

8.2 Designing a base class

Designing a class is difficult because you have to think about all the ways in which the class might be used. Designing a base class is harder still, because you have to think about the people who want to inherit from your class as well as the people who want to use it directly (if it is possible to do so). The differences between a simple class and a potential base class are:

- `virtual` functions
- `protected` attributes

polymorphism

Virtual functions open up the possibility of redefinition in derived classes, leading to polymorphic behaviour. Protected attributes can be accessed by derived classes but not by outsiders. These two features combine to make inheritance useful and manageable.

8.2.1 `virtual` functions

A ***virtual function*** is a function that is defined in a base class and can be redefined in a derived class.[32] Virtual functions are qualified by the keyword `virtual`.

[32] In Java, ***all*** functions are virtual in the C++ sense. Java programmers must remember to write `virtual` when writing C++ programs!

Inheritance polymorphism, which means simply class-dependent behaviour, requires a particular set of circumstances. All of the following conditions must hold for polymorphism to take place: 272

- There must be a base class `B` containing a `virtual` function `f`.
- There must be a class `D`, derived `public` from `B`, that redefines `f`.
- There must be an instance, `d`, of `D`.
- There must be a reference, `&rd`, or a pointer, `*pd`, to the object `d`. These might be defined as:

```
B & rd = d;
B * pd = &d;
```

- The function `f` must be invoked using either a reference or a pointer, as in these statements:

```
rd.f();
pd->f();
```

  Each of these statements will call the redefined function `D::f()`, even though `rd` (`pd`) is a reference (pointer) to the base type. This is ***inheritance polymorphism*** or ***dynamic binding*** (i.e., binding a function name to a function at run-time – as opposed to ***static binding***, where the function to be called is determined at compile-time).

dynamic binding vs. static binding

A virtual function may be ***pure virtual***, meaning that it has no implementation. The body of a pure virtual function is written using the special notation `=0`. A class that has one or more pure virtual functions cannot be instantiated. It may have constructors, but these constructors can be called only by derived classes that provide definitions for the virtual functions.

pure virtual =0

There are essentially three different kinds of member functions in a base class: normal functions, virtual functions, and pure virtual functions. Base class `Animal` has one example of each kind. 273

```
class Animal
{
public:
    string getName() { .... }
    virtual void move() { /* how to walk */ };
    virtual void communicate() = 0; // no implementation
    ....
};
```

From this declaration, we can infer the following intentions of the designer of the `Animal` class hierarchy. Assume that `pa` is a pointer declared as `Animal*` but actually pointing to an instance of a class derived from `Animal`.

- `getName` is a non-virtual function with an implementation. The idea in this example is that the mechanism for accessing the name of an animal is the same for all derived classes. A non-virtual function should not be redefined in a derived class for reasons explained below (Section 8.2.2 on page 157).

> ***Use a non-virtual function for behaviour that is common to all classes in the hierarchy and should not be overridden in derived classes.***

- Function `move` is virtual and has an implementation. In this example, `Animal::move` defines walking, which is a way of moving shared by many animals, and is a kind of ***default behaviour***. A derived class has the option of either inheriting this method of walking or of redefining `move` in a special way for some other kind of movement. Calling `pa->move()` invokes movement that depends on the particular kind of animal.

> ***Use a virtual function to define default behaviour that may be overridden in derived classes.***

- Function `communicate` is a pure virtual function. In this example, the use of a pure virtual function suggests that there is no default behaviour for communication. It corresponds to the fact that animals communicate in many different and unrelated ways (using sound, gesture, smell, touch, etc.). Since class `Animal` has a pure virtual function, it cannot be instantiated: there are no objects of type `Animal`. This corresponds to the real world, where if you see an "animal", it is always a particular kind of animal: aardvark, beaver, cat, dog, elephant, giraffe, hedgehog, etc. Any class for which instances are needed must provide an implementation of `communicate`.

> ***Use a pure virtual function to require behaviour for which there is no reasonable default.***

There is one function that should ***always*** be virtual in a base class: it is the destructor. If a
274 class is intended as a base, declare:

```
class Base
{
public:
    virtual ~Base() { .... }
    ....
};
```

even if the body of the destructor is empty.

275 To see the importance of this, consider the code in Figure 81 on the facing page. Running this
276 program produces the following output:

```
Deleting Base
```

We have constructed an instance of `Derived`, containing data members `x` and `y`, but the run-time system has deleted an instance of `Base`, with data member `x` only. The memory occupied by `y` will almost certainly ***not*** be deallocated – a memory leak!

277 If we change the declaration of class `Base` to

```
class Base
{
public:
    virtual ~Base() { cout << "Deleting Base" << endl; }
    ....
```

```
class Base
{
public:
    ~Base() { cout << "Deleting Base" << endl; }
private:
    int x;
};

class Derived : public Base
{
public:
    ~Derived() { cout << "Deleting Derived" << endl; }
private:
    int y;
};

int main()
{
    Base * pb = new Derived;
    delete pb;
}
```

Figure 81: The danger of a non-virtual destructor

and run the program again, it displays

```
Deleting Derived
Deleting Base
```

By making `Base::~Base` virtual, we have ensured that `delete pb` calls `Derived::~Derived`. This destructor first executes its own code, then invokes the destructor for the base class, and finally deallocates the memory for all of the variables of the derived object.

A base class must have a virtual destructor.

8.2.2 Don't redefine non-virtual functions

C++ allows you to redefine non-virtual functions, as in this example: 278

```
class Animal
{
public:
    Animal() : name("Freddie") {}
    string getName() { return name; }
private:
```

```
    string name;
};

class Bison : public Animal
{
public:
    string getName() { return "Grrrrr"; }
};
```

Although it is allowed, redefining non-virtual functions is not a good idea, for several reasons
279 (Meyers 2005, Item 36). Suppose that we define some variables:

```
Bison b;
Animal *pa = &b;
Bison *pb = &b;
```

These definitions give us two pointers, `pa` and `pb`, pointing to the same object, `b`. It is therefore something of a surprise when we execute

```
cout << pa->getName() << endl;
cout << pb->getName() << endl;
```

and obtain

```
Freddie
Grrrrr
```

What has happened here is that, since `Animal::getName` is not `virtual`, the compiler uses the type of the pointer to choose the function. Thus `pa` causes `Animal::getName` to be called and `pb` causes `Bison::getName` to be called.

If `Animal::getName` was a `virtual` function, then the function called would depend on the type of the object, not the pointer. This is more natural behaviour: we expect a bison to answer `"Grrrrr"` whether we refer to it as an animal or as a bison (at least in this example).

More formally, overriding a non-virtual function is a ***violation of the "is-a" relationship***. The purpose of a non-virtual function in a base class is to define behaviour that is invariant over the whole class hierarchy. In the example, to be an `Animal` (that is, to satisfy the predicate "is-a `Animal`"), you must respond to `getName` by returning your name. If you don't do that, you are not an `Animal` and do not belong in the `Animal` hierarchy. There are two possibilities:

- It is essential that bisons have a way of saying `"Grrrrr"`. In this case, the function that does this should not be called `getName`.
- It is correct modelling to say that bisons reply `"Grrrrr"` when asked their name. In this case, `Animal::getName` should be a virtual function.

Never redefine non-virtual functions in derived classes.

8.2.3 Virtual methods in C++11: `override` and `final`

280

In C++11, you can use the new `override` keyword to make it explicit that a function is intended to override an *existing* virtual base class version: override

```
void foo() override;
```

If the base class does not have a `virtual` function `foo()`, the compiler will issue an error. This protects you from a number of mistakes, including accidentally hiding a function from a base class (see 'Dominance' in Section 8.3.3).

> ***In C++11, always add `override` when you are overriding a `virtual` function.***

Similar to Java, C++11 now also has a `final` specifier, which indicates that a function cannot be further overridden in derived classes: final

```
void foo() final;
```

Fun fact: `override` and `final` are not ordinary keywords: you can still use variables or functions in your code named `override` or `final` (although that is emphatically not recommended for new code). This works because it they are designed as so-called *contextual keywords*, which helps to preserve compatibility with existing code.

C++11 has a number of additional new features related to inheritance that we do not cover in this introduction; see (Prata 2012, Chapter 18) for an overview.

8.2.4 `protected` attributes

Attributes in the `public` part of a class declaration are accessible to anyone who owns an instance of the object. Attributes in the `private` part of the declaration are accessible only to members and friends of the object.

A class declaration may contain a third part, introduced by the keyword `protected`, for attributes that are accessible to members and friends of classes derived from the class. protected

According to (Stroustrup 1994, page 301), the keyword `protected` was introduced into C++ at the request of Mark Linton, who was writing ***InterViews***, an extensible ***X/Window***[33] toolkit. Five years later, Linton banned the use of `protected` data members in InterViews because they had become "a source of bugs: 'novice users poking where they shouldn't have in ways they ought to have known better than'". Stroustrup goes on to say (Stroustrup 1994, page 302):

[33] See http://en.wikipedia.org/wiki/X_Window_System

In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly. In fact, one of my concerns about `protected` is exactly that it makes it too easy to use a common base the way one might sloppily have used global data.

Fortunately, you don't have to use protected data in C++; `private` is the default in classes and is usually the better choice. Note that none of these objections are significant for protected member ***functions***. I still consider `protected` a fine way of specifying operations for use in derived classes.

> ***Prefer*** `private` ***to*** `protected` ***for data members of base classes.***

> ***Use*** `protected` ***functions to access*** `private` ***base class data.***

8.2.5 Example: A base class for bankers

As an example of base class design, we will take class `Account` from Section 6 (see page 101) and redesign it as a base class. Class `Account` is quite appropriate as a base class, because banks provide many kinds of account, and the various kinds are often represented as a hierarchy of classes in banking software. A number of design decisions are mentioned or implied in the following notes; few of them are cast in stone, and most might be made differently in specific
281 circumstances.

282 Figure 82 on the next page shows the new version of class `Account`. The type of `Account::name` has been changed from `char*` to `std::string`, to avoid the memory management problems that we encountered in Section 6 on page 99. Some of the functions, and the class `AccountException`, are not changed, and we do not repeat their descriptions here. The redesign takes into account the following considerations:

- Constructors cannot be virtual, so we leave them unchanged.
- The destructor must be virtual, as explained in Section 8.2.1 on page 154.
- The assignment, `operator=`, cannot be virtual, so we leave it unchanged.
- We provide some accessors (`getName`, `getID`, and `getBalance`) that were not present in the original class but are likely to be useful. They are `public` and non-virtual, since they access private data members and there should never be a need to redefine them. As explained above, these functions express ***invariant properties*** of the account class hierarchy: the meaning of `getBalance` is independent of the type of account we are dealing
283 with. Figure 83 on page 162 shows their definitions.
- The definitions of `deposit`, `withdraw`, and `transfer` are not changed. However, it is quite possible that a derived class might need to modify their behaviour; consequently, we declare them to be `virtual`.

```
class Account
{
public:
    Account();
    Account(std::string name, long id, long balance = 0);
    Account(const Account & other);
    virtual ~Account();
    Account & operator=(const Account & other);
    const std::string getName() const;
    long getID() const;
    long getBalance() const;
    virtual void deposit(long amount);
    virtual void withdraw(long amount);
    virtual void transfer(Account & other, long amount);
    friend std::ostream & operator<<(std::ostream & os,
                                     const Account & acc);

protected:
    void setID(long newID);
    void setBalance(long newBalance);

private:
    std::string name;
    long id;
    long balance;
};

bool operator==(const Account & left, const Account & right);
bool operator!=(const Account & left, const Account & right);
```

Figure 82: Class `Account` as a base class

- There are some functions that should be available to derived classes but not to everybody. These functions are declared in the `protected` section of the class declaration. This group consists of the new functions `setID` and `setBalance`, which have the obvious definitions shown in Figure 84 on the next page. 284
- The comparison operators compare accounts by comparing their addresses. Consequently, they do not have to be defined as `friend`s, and their declarations can be moved outside the class declaration.

8.3 Designing a derived class

Designing a derived class is usually easier than designing a base class because there are fewer options. The base class designer has decided what gets inherited and what can be redefined; the derived class designer has to decide:

```
const string Account::getName() const
{
    return name;
}

long Account::getID() const
{
    return id;
}

long Account::getBalance() const
{
    return balance;
}
```

Figure 83: Accessors for class `Account`

```
void Account::setID(long newID)
{
    id = newID;
}

void Account::setBalance(long newBalance)
{
    balance = newBalance;
}
```

Figure 84: Mutators for class `Account`

- which functions to inherit without change
- which functions to redefine
- what new data to introduce
- what functions to introduce to operate on the new data

If there are no new data or functions, there is probably no need for a derived class at all. In a banking application, for example, it would be pointless to introduce new classes for different rates of interest: the interest rate is just a data member of an appropriate base class.

functions that cannot be inherited

Some functions ***cannot be*** inherited and must be defined if the derived class needs them. The functions that are not inherited are:

- constructors
- the assignment operator (`operator=`)
- friends

Derived class constructors can, and usually should, call the base class constructors explicitly, as in this example.[34] Note how the notation for initialization in the constructor of class `Derived` provides a natural way of calling the class `Base` constructor: Figure 85. 285

The ***Liskov Substitution Principle*** provides a helpful guideline for designing derived classes:

> ***Subclasses must be usable through the base class interface without the need for the user to know the difference.***

"Liskov" is Barbara Liskov of MIT, who has made many contributions to object-oriented programming. An application of the principle for class `Account` would be as follows: if we have a pointer `p` to an instance of class `Account` or any of its derived classes, we should be able to call `p->withdraw(500)` without knowing or caring the actual class of the object `*p`.

8.3.1 Example: A derived class for bankers

In this section, we develop a class `SavingsAccount` derived from the base class `Account`.[35] The important features of a savings account are:

- A savings account ***is an*** account. `SavingsAccount` provides all of the functionality of `Account`.
- `SavingsAccount` has a default constructor and a full constructor that allows all data for the account to be initialized.
- `SavingsAccount` provides a copy constructor and an assignment operator.

```
class Base
{
public:
    Base(int n) { .... }
};

class Derived : public Base
{
public:
    Derived(int k, char c) : Base(k), c(c) { .... }
private:
    char c;
};
```

Figure 85: The constructor of a derived class should call the constructor of its base class.

[34] If you do not call a base class constructor explicitly, the compiler will implicitly call the default constructor, as seen in Figure 79 on page 153. Note that this will fail if your class does not have a default constructor.

[35] The savings account is designed to demonstrate features of object-oriented programming. Any resemblance to actual banking practice is entirely coincidental.

- A savings account collects interest. `SavingsAccount` has a data member for the interest rate and a member function for computing interest and adding the interest to the balance.
- Interest is paid only if the balance exceeds a specified minimum. `SavingsAccount` updates the minimum balance when money is withdrawn from the account.
- The insertion `operator<<` shows the interest rate and minimum balance for the `SavingsAccount` in addition to the information shown for an `Account`.

286

Figure 86 shows the class declaration for `SavingsAccount`. The class declaration contains exactly those functions that we wish to define or redefine. Functions that are inherited from the base class are not mentioned in the declaration of the derived class.

The data members of `SavingsAccount` have the following roles:

- `rate` is the interest rate for the account.
- `minimumBalance` is the minimum balance that the client must maintain over each period in order to earn interest.
- `lowestInPeriod` is the actual minimum balance during the period.

Since constructors are ***not*** inherited, we must define constructors for `SavingsAccount`. These constructors can, and should, invoke the base class constructors when necessary.

287

- The default constructor in Figure 87 on the next page(a) uses the default constructor of class `Account` and sets the new data members to zero.[36]

```
class SavingsAccount : public Account
{
public:
    SavingsAccount();
    SavingsAccount(std::string name, long id, long balance = 0,
        double rate = 0, long minimumBalance = 0);
    SavingsAccount(const SavingsAccount & other);
    SavingsAccount & operator=(const SavingsAccount & other);
    void withdraw(long amount);
    void addInterest();
    friend std::ostream & operator<<(std::ostream & os,
        const SavingsAccount & sacc);
private:
    double rate;
    long minimumBalance;
    long lowestInPeriod;
};
```

Figure 86: Class `SavingsAccount` derived from `Account`

[36] The use of `Account` in the constructor, and `Account::` in other functions, is similar to the use of `super` in Java. Java can use a single keyword because the parent of a class is unique; C++ allows multiple inheritance and must therefore indicate the base class name explicitly.

As we have seen, it is a good idea to provide a default constructor for any class. It is an even better idea for a class that might be used as a base class because, if the base class does not have a default constructor, then none of the derived classes can have a default constructor.

- The normal constructor in Figure 87(b) allows the caller to specify all of the fields of an account with default values for the `balance`, `rate`, and `minimumBalance`. Like the default constructor, it calls the base class constructor and initializes the new data members separately. 288
- The copy constructor in Figure 87(c) must call the copy constructor for the base class and then initialize the new data members explicitly. Calling the copy constructor for `Account` with the `SavingsAccount other` is an example of upcasting. 289

The assignment operator is implemented by using `*this` (implicitly) to call the assignment operator of the base class, and then assigning the new data members explicitly: see Figure 88 on the following page. 290

We must update the value of `lowestInPeriod` whenever the balance might get smaller. This can happen only when money is withdrawn from the account. Figure 89 on the next page shows the new withdrawal function. We override the definition of `Account::withdraw` with a function that updates `lowestInPeriod`. 291

The new function `addInterest`, which we assume is called periodically, checks that the client has the required minimum balance and, if so, adds interest. Since `balance` is private, `addInterest`

```
SavingsAccount::SavingsAccount()
    : Account(), rate(0), minimumBalance(0), lowestInPeriod(0)
{ }
```

(a) Default Constructor

```
SavingsAccount::SavingsAccount(string name, long id,
      long balance, double rate, long minimumBalance)
    : Account(name, id, balance), rate(rate),
      minimumBalance(minimumBalance), lowestInPeriod(balance)
{ }
```

(b) Normal Constructor

```
SavingsAccount::SavingsAccount(const SavingsAccount & other)
:   Account(other),
    rate(other.rate),
    minimumBalance(other.minimumBalance),
    lowestInPeriod(other.lowestInPeriod)
{ }
```

(c) Copy Constructor

Figure 87: Constructors for class `SavingsAccount`

```
SavingsAccount & SavingsAccount::operator=
                  (const SavingsAccount & other)
{
    Account::operator=(other);
    rate = other.rate;
    minimumBalance = other.minimumBalance;
    lowestInPeriod = other.lowestInPeriod;
    return *this;
}
```

Figure 88: Assignment for class `SavingsAccount`

```
void SavingsAccount::withdraw(long amount)
{
    Account::withdraw(amount);
    if (lowestInPeriod > getBalance())
        lowestInPeriod = getBalance();
}
```

Figure 89: Withdrawing from a savings account

must call the protected member function `Account::setBalance` to update the balance. This
292 function also reinitializes `lowestInPeriod`: see Figure 90.

The insertion operator calls `Account(sacc)` to upcast the `SavingsAccount` to an `Account` before inserting it into the stream. It then inserts the new data members into the stream: see
293 Figure 91.

If `sa1` and `sa2` are savings account objects, the expressions `sa1 == sa2` and `sa1 != sa2` compile and evaluate correctly. This is ***not*** because `operator==` and `operator!=` are inherited from
294 `Account` but because the compiler can convert `sa1` and `sa2` to `Account`.

Figure 92 on the next page shows a short test program for accounts and savings accounts. When
295 it is run, this program displays:

```
void SavingsAccount::addInterest()
{
    if (lowestInPeriod >= minimumBalance)
    {
        long interest = static_cast<long>(rate * getBalance());
        setBalance(getBalance() + interest);
    }
    lowestInPeriod = getBalance();
}
```

Figure 90: Adding interest

```
std::ostream & operator<<(std::ostream & os,
                          const SavingsAccount & sacc)
{
    return os <<
        Account(sacc) <<
        fixed << setprecision(6) << setw(10) << sacc.rate <<
        setw(6) << sacc.minimumBalance;
}
```

Figure 91: Inserting a savings account into a stream

```
Anne Bailey           1234567    1000
Anne Bailey           1234567    1000
Before interest: Charles Daumier    7654321   1500  0.010000  1000
After  interest: Charles Daumier    7654321   1515  0.010000  1000
Before interest: Charles Daumier    7654321    515  0.010000  1000
After  interest: Charles Daumier    7654321    515  0.010000  1000
```

The first two lines show that an account can be constructed and copied. The next two lines show that Charles earns $15 interest because his balance is greater than $1,000. After transferring $1,000 to Anne, his balance drops to $515 and does not earn interest, as shown in the last two lines.

Note that the transfer is performed using a reference to an `Account`, not a `SavingsAccount`. (This is realistic, because the object doing the transferring should not have to know the kind of accounts involved in the transfer.) Since `transfer` is not redefined in `SavingsAccount`, the

```
Account anne("Anne Bailey", 1234567, 1000);
cout << anne << endl;

Account & ra = anne;
cout << ra << endl;

SavingsAccount chas("Charles Daumier",
                    7654321, 1500, 0.01, 1000);
cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;

Account *pa = & chas;
pa->transfer(anne, 1000);

cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;
```

Figure 92: Testing the banker's hierarchy

function called is `Account::transfer`. When `transfer` called `withdraw`, it uses `this`, which is a pointer to a `SavingsAccount` object. Consequently, `SavingsAccount::withdraw` gets called, and updates the lowest balance.

8.3.2 Additional base class functions

After completing the base class `Account` and the derived class `SavingsAccount`, we notice that there is information that might be helpful to users: does a particular account pay interest? It is
296 easy to add a function to `SavingsAccount`:

```
bool paysInterest() { return true; }
```

This is of little use, however, to the owner of a pointer to an `Account` that may or may not be a `SavingsAccount`. If `paysInterest` is to be useful, it must be declared `virtual` in the base class, `Account`:

```
virtual bool paysInterest() { return false; }
```

Thus, by default, an account class does not pay interest. Any class corresponding to an account that does pay interest must redefine `paysInterest` to return `true`.

This seems fairly innocuous: after all, the question 'does it pay interest?' can be asked of any account, and therefore the function `paysInterest` should be defined in the base class. The problem – trivial in this example – is that the concept of 'paying interest' is not otherwise mentioned in class `Account`.

In a hierarchy with many classes, there will be many functions of this kind. The base class will become cluttered with accessors that provide specialized information that is only relevant to particular derived classes. This can become a maintenance problem because, each time one of these functions is added, the base class is changed, requiring recompilation of the entire hierarchy. Unfortunately, there does not seem to be a clean solution to this problem.

8.3.3 Dominance

297 There is a somewhat surprising aspect of overloading a virtual function: A function with the same name as a function in a base class hides it (instead of overriding it), even when they have different signatures. This is called ***dominance***: a derived class dominates all aspects of its base class, including functions.

As a consequence, care must be taken when trying to overload functions from a base class: this is only possible by ***forwarding*** all calls to the corresponding version of the function in the base class. E.g., for the example in Figure 93, the derived class must define a function with the same signature as the base class and forward each function call:

```
void foo(double d) { Base::foo(d); };
```

shadowing To ensure that you are not accidentally hiding (or *shadowing*) a function from a base class that you intended to override, always use the `override` keyword in new code (C++11 or newer, see Section 8.2.3).

```
class Base
{
public:
   virtual void foo(double);
};

class Derived : public Base
{
public:
   virtual void foo(string);
};

Base *pb = new Derived;
pb->foo("bar");            // calls Derived::foo
pb->foo(3.14);             // error!
pb->Base::foo(3.14);       // ok, calls Base::foo()
```

Figure 93: Dominance

8.4 Conversion and Casting

Casting a value means converting its type. Casting is needed when the type of a value must actually be changed or when the compiler is confused about the type and has to be told what to do.

8.4.1 Implicit Conversion

There are many situations in which the compiler will perform a conversion implicitly. An assignment statement in which the left and right sides have different types may cause an implicit conversion: a "small" value (e.g., `char`) may be implicitly converted to a "large" value (e.g., `int`), or an `int` may be implicitly converted to `double`.

Similar implicit conversions occur when a value of one type is passed as an argument to a function expecting another type.

8.4.2 C-style Casting

The following definition causes a warning message ("possible loss of data"): 298

```
int k = 3.5;
```

If this is really what you want to do, you can eliminate the error message with a C-style cast:

```
int k = (int)3.5;
```

function-style cast

C++ provides an alternative, equivalent form that resembles constructor syntax. It is called a *function-style cast*:

```
int k = int(3.5);
```

8.4.3 Modern C++ Casting

C++ also provides a deliberately ugly syntax for casting. There are four kinds of cast. The first three have the same effect as the old-style casts above; `dynamic_cast` is a new feature that cannot be simulated with the old syntax.

`static_cast`: Use a `static_cast` when the compiler can check that the conversion is feasible and can generate any required code at compile-time.

299 For example,

```
int i = 5;
int j = 7;
double q1 = i/j;
```

will set `q1` to zero, which may not be the intended result. If you want to obtain 0.714286, use a `static_cast`:

```
double q2 = static_cast<double>(i) / j;
```

For reasons we have discussed previously, the compiler allows conversion from a derived class `D` to a base class `B` (although information may be lost by slicing) but does not allow conversion
300 from a base class `B` to a derived class `D`:

```
b = static_cast<B>(d);   // OK
d = static_cast<D>(b);   // compile-time error
```

In other words, casting does not allow you to do anything that you could not do with a simple assignment:

```
b = d;   // OK
d = b;   // compile-time error
```

301 With pointers or references, conversions in both directions are allowed:

```
B *pb = static_cast<B*>(&d);   // OK - upcast
D *pd = static_cast<D*>(&b);   // OK - downcast
```

However, downcasting (converting from base class to derived class) is not recommended: use `dynamic_cast` instead.

dynamic_cast: A dynamic_cast converts only between reference or pointer types. Its main use is for safe downcasting. The run-time system checks that the value to be converted is of the expected type. The target type must be polymorphic.

Suppose that you have a pointer pb of type Base*. You believe that the object pointed to is an instance of a derived class Derived. Then you could use the following code: 302

```
Derived *pd = dynamic_cast<Derived*>(pb);
```

If your belief is correct (*pb is a Derived instance), then pd will be set pointing to it. If you are wrong, pd will be a null pointer.

The target type for a dynamic cast must be polymorphic. For this reason, dynamic casts may not compile even though the corresponding assignment does compile:

```
Base *pb = pd;                          // OK
Base *pb = dynamic_cast<Base*>(pd);  // compile-time error:
                                        //   Base* is not polymorphic
```

References can be cast dynamically:

```
B bb = dynamic_cast<B&>(d);
```

However, if this cast fails (because d is not an instance of a class derived from Base), the run-time system throws the exception std::bad_cast.

const_cast: Use const_cast when you have a const value in a context that requires a non-const value.

you can cast away const!

The program in Figure 94 on the following page does not compile, because the parameter m of function g cannot be converted from const int to int. One way to get around this problem is to change the call to g, in function f, to 303 304

```
g( const_cast<int&>(n));
```

With this change, the program compiles, and running it gives the following output, showing that const_cast really does "cast away const".

```
g: 6
g: 7
f: 7
```

Next, consider the following test of the functions f and g: 305

```
void g(int & m)
{
   cout << "g: " << m << endl;
   ++m;
   cout << "g: " << m << endl;
}

void f(const int & n)
{
   g(n);
   cout << "f: " << n << endl;
}

int main()
{
   f(6);
}
```

Figure 94: A problem with `const`

```
int main()
{
   int k1 = 6;
   f(k1);
   cout << "main 1: " << k1 << endl << endl;

   const int k2 = 6;
   f(k2);
   cout << "main 2: " << k2 << endl;
   return 0;
}
```

This test gives the following output; note the difference between "`main 1`" and "`main 2`":

```
g: 6
g: 7
f: 7
main 1: 7

g: 6
g: 7
f: 7
main 2: 6
```

306 You could avoid the `const_cast` by coding `f` like this:

```
void f(const int & n)
```

```
{
   int m = n;
   g(m);
   cout << "f: " << n << endl;
}
```

The output of this program is:

```
g: 6
g: 7
f: 6
main 1: 6

g: 6
g: 7
f: 6
main 1: 6
```

These examples show the dangers of casting away `const`-ness. In a large application, the programmer writing `main` (or any client code) might only be able to see the declaration

```
void f(const int & n);
```

and would infer that the argument passed to `f` cannot change. But, if `f` uses `const_cast`, this assumption is invalid (we will later see a more sensible example for using `const_cast`).

`reinterpret_cast`: On very rare occasions, you may have to change the interpretation of a bit-string. The following function is intended to create a hash code from an address. It converts the address to an `unsigned long` and shifts the result right to obtain the high order bits. 307

```
int hash(void *p)
{
   return reinterpret_cast<unsigned long>(p) >> 12;
}
```

However, `reinterpret_cast` cannot be used for arbitrary type conversions. This attempt to see how a `double` value is represented by printing it in hexadecimal does not compile. The compiler explains that there is a better way to perform the indicated conversion:

```
cout << hex << reinterpret_cast<unsigned long>(3.14159) << endl;
```

```
casting.cpp(60) : error C2440: 'reinterpret_cast' :
          cannot convert from 'double' to 'unsigned long'
Conversion is a valid standard conversion, which can be performed
implicitly or by use of static_cast, C-style cast or function-style cast
```

8.5 Advanced inheritance topics

We covered the concepts that are most important for modern C++ programming: (single) *is-a* inheritance and templates. However, C++ has a number of additional concepts, like multiple inheritance, which are significantly more complex. We only mention some aspects here; should you come across these in a project, it is time to pick up one of the advanced C++ reference books.

8.5.1 Private Inheritance

Inheritance using the `private` keyword (or no keyword at all, as it is the default behaviour)
308 represents a "using" or "is-implemented-by" relationship between classes.

Most importantly, there is ***no*** polymorphism in `private` inheritance. The compiler will never convert an object of a derived class to a base class. Everything inherited from a base class becomes `private` in the derived class.

```
class Image {
public:
    virtual void draw() const;
    ...
};

class Timer {
public:
    explicit Timer(int tickFrequency);
    virtual void onTick() const;
    ...
}

class Image : private Timer {
public:
    virtual void draw() const;
private:
    virtual void onTick() const;
    ...
};

Image * pi = new Image;
pi->draw();                 // ok
pi->onTick()                // error!

Timer pt = new Image;     // error!
```

Figure 95: `private` inheritance

So, when should you use `private` inheritance? As the "is-implemented-by" relation indicates, it is purely an implementation technique: a class is implemented using an already implemented base class. This is typically not decided during the object-oriented design phase, but rather during implementation.

An example is shown in Figure 95. Suppose you are developing an `Image` manipulation class and want to add some profiling functionality to better determine the performance of your class. You rummage in your toolchest and come up with a `Timer` class that does exactly what you need: at a pre-defined interval, it will call the `onTick()` function. To make it work, you need to overload this function in your `Image` class. But this is not a *is-a* relationship: an `Image` is certainly not a `Timer`, and it should never be possible to use an `Image` object in place of a `Timer` object in a program. With `private` inheritance, you can inherit the `onTick` function in your `Image` class and add whatever profiling code you need. A similar example would be the implementation of a `stack` data structure using an already existing `vector` class: implementing the `stack` will be trivial by using the existing `vector` as a base, but users of `stack` should never be able to access elements of a `stack` directly (using `[]`). With `private` inheritance, the `stack` can be implemented using the `vector` class, without its public interface being exposed to the `stack` users.

309 310

Together with `private` inheritance, you might also encounter the `using` declaration for class members, which can be used to "unhide" the public and protected members of the base class to the derived class' users.

`using`

Note that a similar effect to `private` inheritance can be achieved in most cases with the *Composite* design pattern, which is easier to understand and manage than private inheritance.

For more details on private inheritance, see (Prata 2012, Chapter 14).

Protected Inheritance. Finally, there exists a variant of `private` inheritance: ***protected inheritance***. With the keyword `protected`, derived classes gain `protected` access to members of the base class in an otherwise private inheritance:

```
class Derived :  protected Base
```

As with `private` inheritance, polymorphism does ***not*** work with `protected` inheritance.

Templates and Inheritance. Template classes can also make use of inheritance. We will see an example for this later; but we do not cover the design of templates with inheritance in detail in this course.

8.5.2 Multiple Inheritance

C++ allows multiple inheritance, that is, inheriting from more than one base class. Unlike Java, where only multiple *interface* inheritance is possible, C++ allows multiple inheritance of the *implementation*:

```
class Derived :  public Base1, public Base2
```

The consequences are rather complex and we will not discuss them here: If you ever need to develop (or modify) C++ code containing multiple implementation inheritance, you should consult an advanced reference book.

311 An issue that becomes important with multiple implementation inheritance is the ambiguity
312 introduced when the same function or member name is inherited from different base classes. In this case, it becomes necessary to explicitly indicate which base class version should be used.

Together with multiple inheritance, you will probably encounter ***virtual inheritance***, specified with the `virtual` keyword. This is important in multiple (implementation) inheritance, indicating that a member is *shareable* across a number of derived classes:

```
  class Derived :  virtual public Base
or class Derived :  public virtual Base
```

The rules for initializing virtual base classes are rather complicated and not as intuitive as for non-virtual bases, and we will not discuss them in this introductory course.

References

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.