**Inheritance**

So far, we have only dealt with freestanding classes, let us now put three classes

together in a *class hierarchy*. We start with the baseclass Person, which has a

constructor that initializes the field m\_stName, and a method Print that writes the

name. Print is marked as virtual; this means that dynamic binding will come into

effect when we, in the main function, define a pointer to an object of this class. We

do not have to mark the methods as virtual in the subclasses, it is sufficient to do so

in the baseclass. The constructors of a class cannot be virtual. However, every other

member, including the destructor, can be virtual and I advise you to always mark

them as virtual in your baseclass. Non-virtual methods are not a part of the objectoriented

model and were added to C++ for performance reasons only.

//Person.h

class Person

{

public:

Person(string stName);

virtual void Print() const;

private:

string m\_stName;

};

//Person.cpp

#include <iostream>

#include <string>

using namespace std;

#include "Person.h"

Person::Person(string stName)

:m\_stName(stName)

{

// Empty.

}

void Person::Print() const

{

cout << "Name: " << m\_stName << endl;

}

Student and Employee are subclasses to Person, which means that they inherit all

public members of Person. Generally, they also inherit all protected members, even

though we do not have any of those in the baseclass in this case. They define their

own versions of Print with the same parameter list (in this case no parameters at

all). It is called *overriding*. This is not to be confused with *overloading*, which was

refers to freestanding functions or methods of the same class. For overriding to

come into effect, the methods must have the same name and parameters list and

the method of the baseclass must be virtual. As constructors cannot be virtual, they

cannot be overridden. Two overridden methods cannot have different return types.

//Student.h

class Student : public Person

{

public:

Student(string stName, string stUniversity);

void Print() const;

private:

string m\_stUniversity;

};

//Student.cpp

#include <iostream>

#include <string>

using namespace std;

#include "Person.h"

#include "Student.h"

Student::Student(string stName, string stUniversity)

:Person(stName),

m\_stUniversity(stUniversity)

{

// Empty.

}

void Student::Print() const

{

Person::Print();

cout << "University: " << m\_stUniversity << endl;

}

//Employee.h

class Employee : public Person

{

public:

Employee(string stName, string stEmployer);

void Print() const;

private:

string m\_stEmployer;

};

//Employee.cpp

#include <iostream>

#include <string>

using namespace std;

#include "Person.h"

#include "Employee.h"

Employee::Employee(string stName, string stEmployer)

:Person(stName),

m\_stEmployer(stEmployer)

{

// Empty.

}

void Employee::Print() const

{

Person::Print();

cout << "Company: " << m\_stEmployer << endl;

}

**Dynamic Binding**

In the example above, it really is no problem to create static objects of the classes.

When we call Print on each object, the corresponding version of Print will be

called. It becomes a bit more complicated when we introduce a pointer to a Person

object and let it point at an object of one of the subclasses. As Print in Person is

virtual, dynamic-binding comes into force. This means that the version of Print in

the object the pointer actually points at during the execution will be called. Had it not

been virtual, Print in Person would always have been called. To access a member

of an object given a pointer to the object, we could use the dot notation together

with the dereferring operator. However, the situation is so common that an arrow

notation equivalent to those operations has been introduced. The following two lines

are by definition interchangeable:

pPerson->Print();

(\*pPerson).Print();

//Main.cpp

#include <iostream>

using namespace std;

#include "Person.h"

#include "Student.h"

#include "Employee.h"

void main()

{

Person person("John Smith");

person.Print();

cout << endl;

Student student("Mark Jones", "MIT");

student.Print();

cout << endl;

Employee employee("Adam Brown", "Microsoft");

employee.Print();

cout << endl;

Person\* pPerson = &person;

pPerson->Print(); // Calls Print in Person.

cout << endl;

pPerson = &student;

pPerson->Print(); // Calls Print in Student.

cout << endl;

pPerson = &employee;

pPerson->Print(); // Calls Print in Employee.

}

Had we omitted the word virtual in the class Person above, we would not have

dynamic-binding, but rather static-binding. In that case, Print in Person would

always be called. As mentioned above, static-binding is present for performance

reasons only and I suggest that you always mark every method of the baseclass in

the class hierarchy as virtual.

Pure virtual methods

Let us take the next logical step and continue with abstract baseclasses and pure

virtual methods. An abstract baseclass cannot be instantiated into an object, but

can be used as a baseclass in a class hierarchy. In the example above, we became

acquainted with virtual methods. In this section, we look into pure virtual methods.

A pure virtual method does not have a definition, just a prototype. A class becomes

abstract if it has at least one pure virtual method, which implies that a class cannot

be abstract without a pure virtual method. A subclass to an abstract class can choose

between defining all pure virtual methods of all its baseclasses, or become abstract

itself. In this manner, it is guaranteed that a concrete (not abstract) subclass always

has definitions of all its methods.

The next example is a slightly different version of the hierarchy of the previous

section. This time, Person is an abstract baseclass because it has the pure virtual

method Print. Its prototype is virtual and succeeded with = 0.

The field m\_stName is now protected, which means that it is accessible by

methods in subclasses, but not by methods of other classes or by freestanding

functions. Another difference in these classes is the use of constant references in the

constructor. Instead of sending the object itself as an actual parameter, which might

be time and memory consuming, we can send a reference to the object. To make sure

that the fields of the object are not changed by the method, we mark the reference

as constant. Compare the constructor of

Person in this case with the previous case.

The change does not really affect the program, it is just a way to make the program

execute faster and use less memory.

//Person.h

class Person

{

public:

Person(const string& stName);

virtual void Print() const = 0;

protected:

string m\_stName;

};

//Person.cpp

#include <string>

using namespace std;

#include "Person.h"

Person::Person(const string& stName)

:m\_stName(stName)

{

// Empty.

}

As Person is an abstract class, Student must define Print in order not to become

abstract itself.

//Student.h

class Student : public Person

{

public:

Student(const string& stName, const string& stUniversity);

void Print() const;

private:

string m\_stUniversity;

};

In the Student class of the previous example, Print called Print in Person. This

time, Person does not have a definition of Print, so there is no method to call.

Instead, we access the Person field m\_stName directly; this is allowed because in this

version it is protected in Person.

**//Student.cpp**

#include <string>

#include <iostream>

using namespace std;

#include "Person.h"

#include "Student.h"

Student::Student(const string& stName, const string& stUniversity)

:Person(stName),

m\_stUniversity(stUniversity)

{

// Empty.

}

void Student::Print() const

{

cout << "Name: " << m\_stName << endl;

cout << "University: " << m\_stUniversity << endl;

}

Employee works in the same way as Student.

**//Employee.h**

class Employee : public Person

{

public:

Employee(const string& stName, const string& stEmployer);

void Print() const;

private:

string m\_stEmployer;

};

**Employee.cpp**

#include <string>

#include <iostream>

using namespace std;

#include "Person.h"

#include "Employee.h"

Employee::Employee(const string& stName,

const string& stEmployer)

:Person(stName),

m\_stEmployer(stEmployer)

{

// Empty.

}

void Employee::Print() const

{

cout << "Name: " << m\_stName << endl;

cout << "Company: " << m\_stEmployer << endl;

}

In the main function, we cannot create an object of the Person class as it is an abstract

class. Neither can we let the pointer pPerson point at such an object. However,

we can let it point at an object of the class Student or Employee. The condition for

Student and Employee being concrete classes was that they defined every pure

virtual method, so we can be sure that there always exists a definition of Print

to call.

//**Main.cpp**

#include <string>

#include <iostream>

using namespace std;

#include "Person.h"

#include "Student.h"

#include "Employee.h"

void main()

{

// Does not work as Person is an abstract class.

// Person person("John Smith");

// person.Print();

// cout << endl;

Student student("Mark Jones", "Berkeley");

student.Print();student.Print();

cout << endl;

Employee employee("Adam Brown", "Microsoft");

employee.Print();

cout << endl;

// In this version, there is no object person to point at.

// Person\* pPerson = &person;

// pPerson->Print();

// cout << endl;

Person\* pPerson = &student;

pPerson->Print(); // Calls Print in Student.

cout << endl;

pPerson = &employee;

pPerson->Print();// Calls Print in Employee.

}

**Arrays of Objects**

An array of objects is not really so much different from an array of values. However,

one issue to consider is that there is no way to call the constructor of each object

individually. Therefore, the class must have a default constructor or no constructor

at all. Remember that if a class has one or more constructors, one of them must be

called every time an object of the class is created.

// The default constructor is called for each car object.

Car carArray[3];

carArray[2].IncreaseSpeed(100);

// The default constructor is called for each car object.

Car \*pDynamicArray = new Car[5];

pDynamicArray[4].IncreaseSpeed(100);

delete [] pDynamicArray;

Just as for values, we can also initialize an object array with a list. In that case,

we can call constructors other than the default constructor. Note that when we

introduce a new object in an array initialization list and call the default constructor,

we have to add parentheses unlike when creating freestanding objects by calling the

default constructor.

Car carArray[] = {Car(), Car(100, 90)};

carArray[0].TurnLeft(90);

**Streams and File Processing**

We can open, write to, read from, and close files with the help of *streams*. Streams

are predefined classes. ifstream is used to read from files, and ofstream is used to

write to files. They are subclasses of istream and ostream in the operator overload

section above. The program below reads a series of integers from the text file

input.txt

and writes their squares to the file output.txt. The stream operator returns

false when there are no more values to be read from the file. Note that we do not

have to close the file at the end of the program, the destructor will take care of that.

//TextStream.cpp

#include <iostream>

#include <fstream>

using namespace std;

void main(void)

{

ifstream inFile("Input.txt", ios::in);

ofstream outFile("Output.txt", ios::out);

int iValue;

while (inFile >> iValue)

{

outFile << (iValue \* iValue) << endl;

}

}

The text files are written in plain text and can be viewed by the editor.

Input.txt

1

2

3

4

5

Output.txt

1

4

9

16

25

We can also read and write binary data with the stream classes. The program below

writes the numbers 1 to 10 to the file Numbers.bin and then reads the same series of

values from the file. The methods write and read take the address of the value to be

read or written and the size of the value in bytes. They return the number of bytes

actually read or written. When reading, we can check whether we have reached the

end of the file by counting the number of read bytes; if it is zero, we have reached the

end. Even though we do not have to close the file, it is appropriate to do so when the

file has been written so that the values are safely saved before we open the same file

for reading.

//BinaryStreams.cpp

#include <iostream>

#include <fstream>

using namespace std;

void main(void)

{

ofstream outFile("Numbers.bin", ios::out);

for (int iIndex = 1; iIndex <= 10; ++iIndex)

{

outFile.write((char\*) &iIndex, sizeof iIndex);

}

outFile.close();

ifstream inFile("Numbers.bin", ios::in);

int iValue;

while (inFile.read((char\*) &iValue, sizeof iValue) != 0)

{

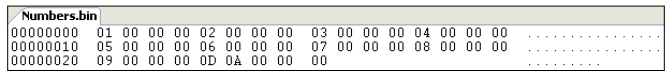
cout << iValue << endl;

}

}

The values are stored in compressed form in the binary file Numbers.bin, which is

why they are not readable in the editor. Here is a screen dump of the file:



Even though these file processing techniques are of use in many situations, we will

not use them in the applications of this book. Instead, we will use the technique of

*Serialization*, described in Chapter 3.

**Summary**

\*The object-oriented model rests on the three cornerstones *inheritance*,

*encapsulation*, and *dynamic binding* as well as the five relations *instance*,

*inheritance*, *aggregation*, *connection*, and *call*.

\*An object can be created as an *instance* of a class. A class consists of two

types of *members*: *methods* (member functions) and *fields* (member variables).

A member can be *private*, *protected*, or *public*. The methods of a class can be

divided into constructors, inspectors, modifications, and one destructor.

\*A class can *inherit* one or more, other baseclasses with its members. A

method of the baseclass can be *virtual*, resulting in *dynamic binding*.

\*An array can hold a sequence of objects. The classes of those objects have to

have a default constructor or no constructor at all in order for the objects to

be thoroughly initialized.

\*With the help of pointers and classes, we can create a linked list, which is a

very useful structure. With its help, we can construct a stack.

\*We can overload the usual operators so they take objects as operands.

However, we cannot affect the number of operands, nor the precedence or

associativity of the operators.

\*We can use the *this* pointer to access our own object and we can define

functions as *friends* to a class.

\*Exception handling is an elegant error handling method. When an error

occurs, we throw an exception. The point in the exception may or may not be

handled in another part of the code. In either case, we do not have to worry

about that when the error occurs.

\*We can define template classes, which are instantiated with suitable types

when we instantiate objects. We can also define template functions that take

parameters of different types.

\*We can organize our classes, freestanding functions, and global variables

into namespace.

\*We can read from and write to text and binary files with the predefined

clases ifstream and ofstream.

REFERENCIAS

**Björnander, S. (2008).** Microsoft Visual C++ Windows Applications by Example (1/a edición). Packt Publishing. www.packtpub.com