Script for the vhb lecture

Programming in C++
Part 1

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

6 The concept of classes in C++

In chapter 6 you will learn how to implement the concept of classes, described in chapter 5, in C++.

6.1 What is a class?

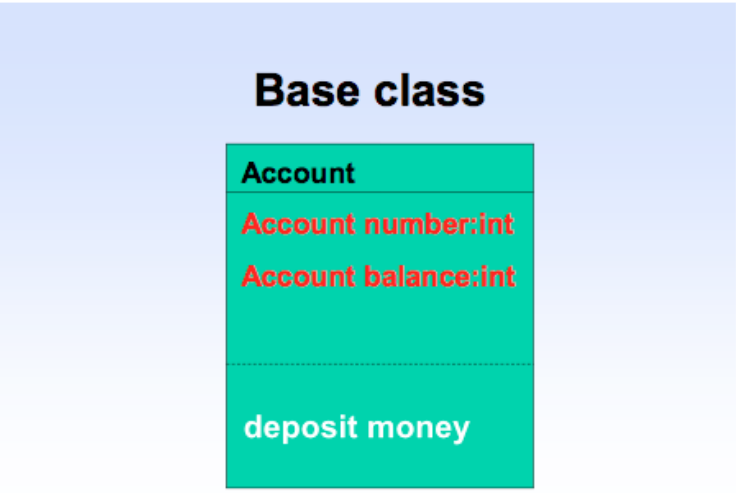
A *class* is a group of data elements and functions that work together to perform a specific programming task. You can also say that the class *encapsulates* the task.

Classes have the following characteristics:
- Data members
- Member functions
- Data encapsulation for access restrictions on the data members
- Constructors
- Destructors

A class is like an abstract blueprint for a concrete object (= instance).
Each instance of a class is an independent object. Each instance has its own data elements, and the objects are independent of each other. They are all of the same type, but in the memory they form separate instances.

Let´s take another look at the base class "Account", which we discussed as an example in the last chapter.

The **name** of the class is "Account".
Two **data members** of the class were specified as "AccountNumber" and "AccountBalance", both of the *integer* data type.
A **member function** "deposit money" was specified.



6.2 Data members of a class in C++

6.2.1 Declaration, definition und access

A class must first be **declared**. After the declaration, it is considered as a data type.

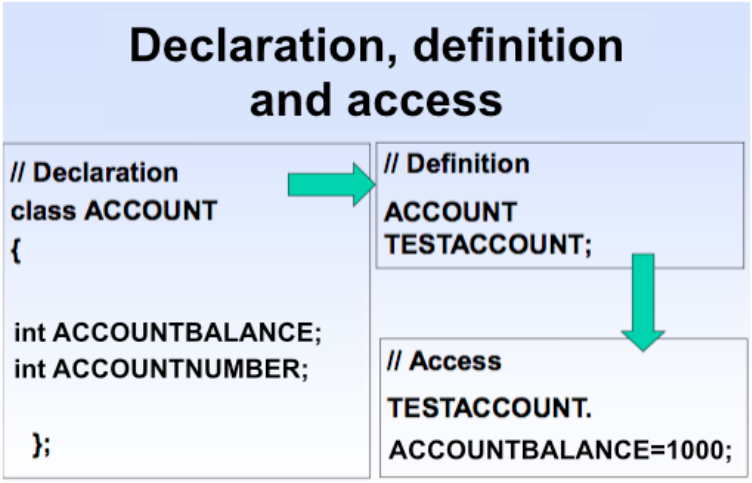
```
class ACCOUNT           // Declaration of the ACCOUNT class
{
public:
    int ACCOUNTBALANCE;
};
```

We can then use it like any other data type and build variables (instances) in the main program.

```
ACCOUNT TESTACCOUNT;            // Creating an instance of the ACCOUNT class
```

You access an instance by using its name together with the name of the element, separated by the **member access operator** (for accessing the elements of an object).

```
TESTACCOUNT.ACCOUNTBALANCE = 1000; // Sets the ACCOUNTBALANCE of TESTACCOUNT to 100
```



Usually you can find the declaration of a class in a header file. In straightforward cases, both the declaration and the definition of the class can be placed in the same source code file.

Normally, you create a source code file whose name largely corresponds to the name of the class and extends with .CPP. Since Windows 95 and Windows NT support long filenames, you can name your file in the same way as your class, if you like. The header file of the class is best named after the source code file, except that the extension .H is used.

Example:

```
class ACCOUNT                    // Declaration in the header file: Save header file as ACCOUNT.
{
  int ACCOUNTNUMBER;
  int ACCOUNTBALANCE;
.......
};
```

```
#include “ACCOUNT.H“      // Including the self-defined header file
.....
                          // Save as ACCOUNT.CPP
```

Can you define multiple accounts?

Yes, that's another strength of object orientation.

If you also need a CURRENTACCOUNT, a SAVINGSACCOUNT and a CASHACCOUNT within your program in addition to the TESTACCOUNT, you only need to define them.
In the following example, 3 instances of class “ACCOUNT” are created:

Example:

```
ACCOUNT CURRENTACCOUNT;
ACCOUNT SAVINGSACCOUNT;
ACCOUNT CASHACCOUNT;
```

These three accounts are defined on the basis of the declaration of the ACCOUNT class and are then available for processing.

6.2.2 The limited access principle in C++

A fundamental paradigm of object orientation is the **limited access principle**.

The decisive point is that the values of the data members of an object can only be changed using defined member functions. The data member values are therefore encapsulated from outside the interface.

Let's take a look at the implementation of the limited access principle in the C++ programming language.

I would like to return to our last example.

The ACCOUNTNUMBER and ACCOUNTBALANCE data members were specified in the declaration of the ACCOUNT class.

6.2.2.1 private und public

These two data members were declared as "private" (*access specifier*) in the default without any further information.

This enables us to achieve the optimum of data encapsulation.

You can no longer access the data members and data member values of this class "from the outside" of the class. Private data members of a class are only accessible from members of the same class.

But be careful!

```
class ACCOUNT                                  // Definition of the ACCOUNT class
{
int ACCOUNTNUMBER;
};

int main()
{
ACCOUNT CurrentAccount;                        // Instantiated current account

CurrentAccount. ACCOUNTNUMBER =555666;         // Compiler error!!!
}
```

The direct access to the data member values of a class, as specified in the *above* example, leads to an error message such as "ACCOUNT::ACCOUNTNUMBER is not accessible" when compiling!

How can the data of a class object, such as the ACCOUNTBALANCE, be accessed?

There are two options.

On the one hand, we could use member functions that can change the data.
We'll get to know member functions a little later.

On the other hand, we can loosen the data encapsulation by declaring the corresponding data members using the keyword "**public**".

Let's take another look at our example.

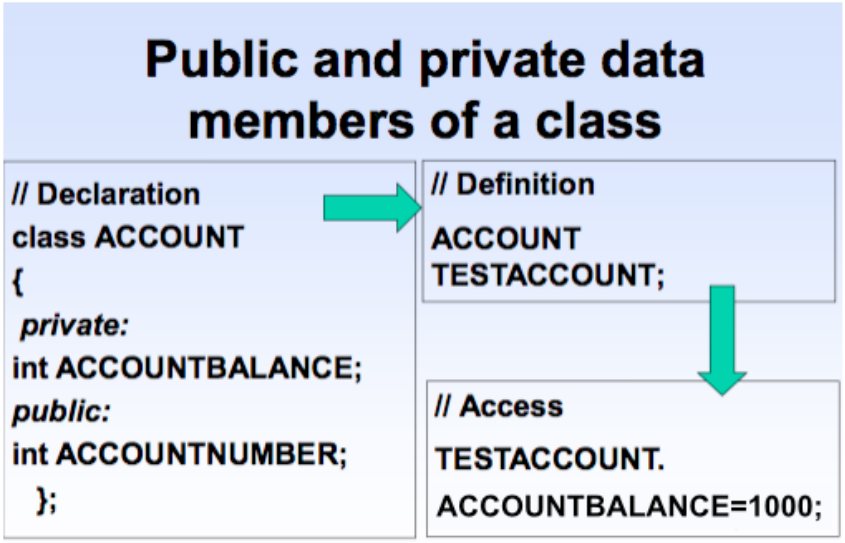
-
```
class ACCOUNT                                  // Definition of the ACCOUNT class
{
public: int ACCOUNTNUMBER;
};

int main()
{
ACCOUNT CurrentAccount;                        // Instantiated current account

CurrentAccount. ACCOUNTNUMBER =555666;         // Access the instance
}
```

This way the compiler will no longer generate an error message.

You should, however, specify data members as "**public**" very carefully, otherwise the limited access principle will be undermined. The default setting for the declaration of class data members should therefore be "**private**".



The access specifiers control how a class can be used. As a C++ developer you are both, the creator and the user of the class. In a team, however, you will also make your class available to the other programmers. To understand the meaning of access specifiers, you must first learn how to use classes. Each class contains a public section (*public*) that can be accessed from the outside of the class. The private section (*private*) is only meant for the internal use within the class. A well-defined class should not take things that the user does not need to know into the public sphere.

Data abstraction means that the user only knows as much about a class as is necessary for its "operation". For example, you get into your car and turn the ignition key to start the engine. Would you like to know all the details of what's going on in your car? Of course not!

You just want to know how to handle your car. In this analogy, the steering wheel, the pedals, the speedometer and so on form the interface between the car and the driver. The driver knows when and how to act in order to get from location A to location B. Conversely, the engine, the chassis and the electrics of the car are covered and hidden so that you hardly have anything to do with them. These are details you do not need to know anything about, so they are hidden from you - in our terminology, they would be called “private”.

Imagine what it would be like if you had to keep an eye on everything the car does: Does the carburettor get enough petrol? Is the differential sufficiently lubricated? Does the alternator supply the right voltage level? Do the inlet valves open smoothly? Who needs this?
In the same way, a class keeps its internal work private so that the user does not have to worry about what happens “under the hood”. The internal information is private, whereas the user interface is public.

6.2.2.2 protected

The access specifier which is called *protected* is a bit more difficult to explain. Like private members of a class, the user cannot alter protected class members. However, derived classes can access them. The access specifier called *protected* will be discussed in more detail in chapter 9 in the subject of "inheritance".

Let's go back to our car example. Suppose you wanted to extend your car to turn it into an impressive limousine. To do this, you would need to understand the structure of the car. You should at least know how to modify the drive shaft and the frame of the vehicle. You would have to get your hands dirty, and as a designer you would get in touch with those parts of the vehicle that you would not have been interested in beforehand (the protected parts). The actual function of the engine, however, is still hidden (private) because you do not need to know how the engine works to extend the car frame. It is similar with the "public" parts of the car, which you usually do not change, but to which you can add some new "public" elements, for example, the control elements for an intercom system.

However, we have now somewhat deviated from the topic and have gained some insight into the problem area of inheritance. A topic that I will discuss in more detail later on.

In C++, an access specifier is one of the following three keywords: *public*, *private* and *protected*.

- *public*: Data members of the class are public, i.e. they can also be accessed from outside the class.
- *private*: Private data members of a class are only accessible by other members of the same class.
- *protected*: Derived classes (see **Chapter 9**) get access to the data members of the class.

In the declaration, you determine whether and how the access to a class element is restricted. The class itself is declared with the keyword “*class”*. A class declaration is similar to the declaration of a structure with additional access specifiers.

6.3 Member functions of a class in C++

Let's now turn to the **member functions** of a class in C++.

Member functions are, as we discussed in detail in the last chapter, functions that belong to a particular class.

Member functions are able to access **private members** of a class.

Member functions are therefore used to ensure that the limited access principle is observed.

Let's take another look at the example of a **base class**.

In addition to the name and data members of the class, the member functions of the class also get specified.
In our above example, only the member function “deposit money” was specified.

The validity range of member functions is the class; outside the class they do not exist. They can only be called within the class or by an instance of the class. You can use the member functions to access all data members of the class, regardless of their respective access restrictions. Member functions can also be declared as public, protected and private.

However, think carefully about which access restrictions you define for the member functions. The public member functions form the interface between class and user. The member functions enable the user to access and work with a class.

Let’s assume you have a class that plays and records sound files. The public member functions of the class should include functions such as:
- Open(), Play(), Record(), Safe(), FastForward()

```
#include <iostream>
#include <cstdlib>
using namespace std;

class Sound
{
   private:
          int timer;
          void setTimer(int t) { timer = t; };

   public:
          void Open() { setTimer (0); };
          void Play() { };
          void Record() { };
          void Safe() { };
          void FastForward() { };
};

int main()
{
          Sound k;
          k.Open(); // possible
          // k.setTimer (0); // not possible

          system("pause");
          return 0;
}
```

The private member functions are used for the internal work of the class. They are not intended to be used by the user; because after all, they are private to hide them from the outside world.

An example:

When you create class instances, you often have to do extensive work involving a large number of source codes (for example, initializing variables and reserving memory space). In order to keep this member function clear, you could place these statements in an init() function, which then can be called. The user should never directly use this function. It would probably be devastating if a user did this at the wrong time. Such a function would therefore be declared as private to ensure the integrity of both the class and the user.

Protected member functions cannot be called by the user either. However, derived classes can access them.

6.3.1 Declaration of member functions

In C++, a member function must first be **declared** and **defined** before it can be used.

First, let's have a look at the declaration of a method.

In addition to the data members, also the member functions are declared for a class

Like the class data members, also member functions can be declared as **public, private** or **protected**.

Example:

```
int READ_ACCOUNTBALANCE();
```

Declaration of a member function

```
class ACCOUNT
{ private: // Declaration of a data member
      int ACCOUNTBALANCE;
  public: // Declaration of a public member
          // function
      int READ_ACCOUNTBALANCE(void);
};
```

6.3.2 Definition of member functions

The next step is to define a declared member function.
The definition determines how the member function works in detail.
In our example, we have defined a member function that only performs one reading operation.

As with the definition of a function, the member function is defined in the function header and function body.

Member functions can be defined inside or outside the class.

6.3.2.1 Definition within the class

```
class ACCOUNT
{
        private:
                int ACCOUNTBALANCE;     // Declaration of a data member

        public:         // Definition of the member function READ_ACCOUNTBALANCE
                int READ_ACCOUNTBALANCE ()
                {
                        return(ACCOUNTBALANCE);
                }
};
```

6.3.2.2 Definition outside the class

For more extensive member functions, the definition within the class becomes too confusing and should therefore take place outside the class declaration.
Please note that in this case the **class operator "::"** must be used.
It is also called the **scope resolution operator** and indicates for which class the specific member functions are defined.

Example:

member function header: `int ACCOUNT:: READ_ACCOUNTBALANCE ()`
member function body: `{ return (ACCOUNTBALANCE); }`

Definition of a member function

```
int ACCOUNT::READ_ACCOUNTBALANCE(void)
{
// The member function READ_ACCOUNTBALANCE
// returns the value of the data member
// ACCOUNTBALANCE of the class ACCOUNT
return(ACCOUNTBALANCE);
}
```

6.3.2.3 friend

In C++, it is possible to make *private* or *protected* data members or member functions accessible to other classes by declaring them as "friends" (*friend*).

Example:

```
class TESTCLASS
{
friend class FRIENDCLASS; // friend-declaration

private:        // Declaration of data members and member functions of class TESTCLASS
…
};
```

This allows all objects of class “FRIENDCLASS” to access the private members of class “TESTCLASS”.

Note: “Friends" of a class have the same access rights as the members of the class itself.

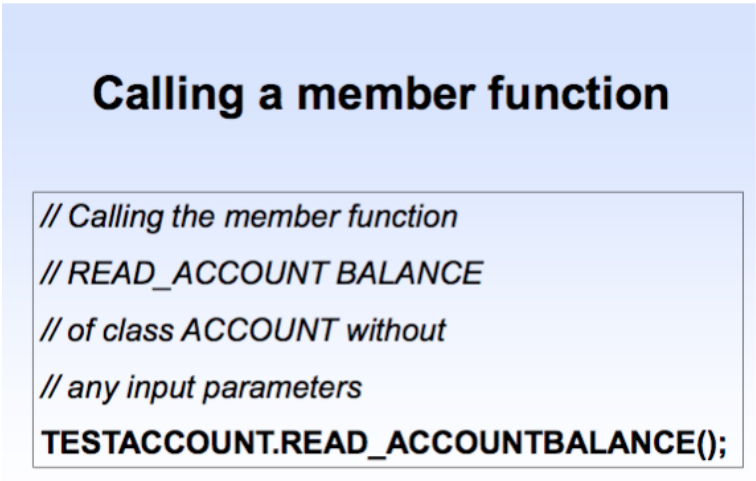
6.3.3 Calling a member function

You can call a class member function by using the following information:

- **the name of the object**
- **the member access operator (for accessing elements of an object)**
- **the names of the member functions and**
- **the parameters of the member functions**

Beispiel:

TESTACCOUNT.READ_ACCOUNTBALANCE();



7 Example application: ACCOUNT MANAGEMENT

7.1 Requirements

In an example of ACCOUNT MANAGEMENT we will now demonstrate the implementation of the concepts of object orientation.
What should our ACCOUNT MANAGEMENT be able to do?
Our program is quite simple.
We want to be able to CHANGE ACCOUNT BALANCE and READ ACCOUNT BALANCE.
This should of course be possible several times.

SCREEN OUTPUT

```
ACCOUNT MANAGEMENT
1 = CHANGE ACCOUNT BALANCE
2 = READ ACCOUNT BALANCE
0 = EXIT PROGRAM

PLEASE SELECT:
```

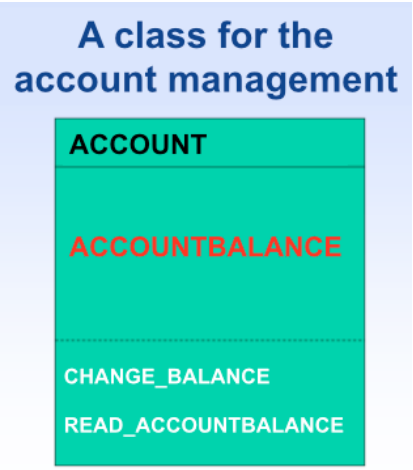
7.2 Analysis

First of all, we have to think about what
- classes
- data members and
- member functions

we need in order to solve our problem.

An initial analysis will show that we have to create an ACCOUNT class with the data member
- ACCOUNTBALANCE

and the member functions
- CHANGE_BALANCE and
- READ_ACCOUNTBALANCE



7.3 Declaration of a class

The ACCOUNT class is declared as follows:

```
class ACCOUNT
{
   private:
          int ACCOUNTBALANCE;

   public:
          bool CHANGE_BALANCE (int);
          int READ_ACCOUNTBALANCE();
};
```

In the class, the two member functions are only declared. It is now necessary to specify exactly what they will be used for outside the class (definition). It would also be possible to declare and define member functions within the class at the same time.
The CHANGE_BALANCE member function as an input parameter takes a value that is to be entered from the keyboard. The *if statement* then checks whether the value is plausible (between 0 and 100000). If the input is correct, the value of the ACCOUNTBALANCE data member is set to the input value and the member function returns 1 (true) as return value.
Otherwise, there is no change in value and 0 (false) is returned.

```
bool ACCOUNT::CHANGE_BALANCE(int INPUT)    // Definition of the member function "CHANGE_BALANCE"
{
          if((INPUT>=0)&&(INPUT<=100000))  // The entered value must be between 0 and 100000
          {
            ACCOUNTBALANCE=INPUT;          // ACCOUNTBALANCE is changed
            return(1);                     // Change is made
          }
          return(0);                       // Change failed (e.g. when entering a negative value)


int ACCOUNT::READ_ACCOUNTBALANCE()         // Definition of member function "READ_ACCOUNTBALANCE"
          { return(ACCOUNTBALANCE); }
```

7.4 Main program

First, an instance of the ACCOUNT class is created in the main program and the CHANGE_BALANCE member function is initialized to 0.

```
int main()                                 // main program
{
   ACCOUNT TESTACCOUNT;                    // Definition of TESTACCOUNT
```

Then the CHANGE_BALANCE member function is initialized to 0. The member function has a variable as an input parameter. The variable must also be initialized to 0.

```
   int     BALANCE=0;                          // Initialization of a variable

  TESTACCOUNT.CHANCE_BALANCE(BALANCE);    // Initialization of a the member function CHANCE_BALANCE to 0
```

The user menu is displayed in a do-while loop. An ALTERNATIVE auxiliary variable is needed, which is first initialized to 0.
The menu is displayed as long as the user enters 1 or 2. If the user enters 0, the loop and the program terminates.

```
   int AUXILIARY=0;                                    // Initialization of a variable

   cout << "A C C O U N T M A N A G E M E N T" << endl;  //  User menu
        do                                             //  ! do-while loop !
        {
        cout << "1 = CHANGE ACCOUNT BALANCE\n";        // User makes a selection
        cout << "2 = READ ACCOUNT BALANCE\n";
        cout << "0 = Exit program" << endl << endl;
        cout << "Please select:";
        cin >> AUXILIARY;                              // Keyboard input

                                                       // Insert switch statement at this position.

        } while(AUXILIARY);                            // As long as AUXILIARY is required!
  system("pause");
}                                                      // End of the program!
```

According to the user input, the CHANGE_BALANCE or READ_ACCOUNTBALANCE member function is called within a switch statement.

```
        switch(AUXILIARY)                      // Following function
        {
        case 1:                                // Entering a new VALUE
                cout << "New BALANCE:";
                cin >> BALANCE;                                  // Value is read from the keyboard
                if(TESTACCOUNT.CHANGE_BALANCE(BALANCE))  // Calling function and query: 1 (true) or 0 (false)
                        cout << "BALANCE has been changed." << endl << endl;
                else
                        cout << " BALANCE has not been changed." << endl << endl;
                break;

        case 2:                                // Query of the current BALANCE
                cout << "Actual BALANCE:";
                cout << TESTACCOUNT. READ_ACCOUNTBALANCE () << endl << endl;
                break;
        }
```

7.5 Complete program

The complete program now looks like this:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class ACCOUNT                                   // Definition of the class "ACCOUNT"
{
   private:
          int ACCOUNTBALANCE;

   public:
          bool CHANGE_BALANCE (int);
          int READ_ACCOUNTBALANCE();
};

bool ACCOUNT::CHANGE_BALANCE(int INPUT)     // Definition of the member function "CHANGE_BALANCE"
{
          if((INPUT>=0)&&(INPUT<=100000))          // The entered value must be between 0 and 100000
          {
            ACCOUNTBALANCE=INPUT;                  // ACCOUNTBALANCE is changed
            return(1);                             // Change is made
          }
          return(0);                               // Change failed (e.g. when entering a negative value)

int ACCOUNT::READ_ACCOUNTBALANCE()      // Definition of member function "READ_ACCOUNTBALANCE"
          { return(ACCOUNTBALANCE); }

int main()                                      // main program
{
   ACCOUNT TESTACCOUNT;                         // Definition of TESTACCOUNT
   int AUXILIARY =0, BALANCE =0;                // Initialization of variables

   TESTACCOUNT.CHANCE_BALANCE(BALANCE); // Initialization of a the member
                                                  // function CHANCE_BALANCE to 0

    cout << "A C C O U N T M A N A G E M E N T" << endl;    //  User menu
          do                                               //  ! do-while loop !
          {
          cout << "1 = CHANGE ACCOUNT BALANCE\n";      // User makes a selection
          cout << "2 = READ ACCOUNT BALANCE\n";
          cout << "0 = Exit program" << endl << endl;
          cout << "Please select:";
          cin >> AUXILIARY;                                // Keyboard input

          switch(AUXILIARY)                                // Following functions
          {
          case 1:                                          // Entering a new VALUE
          cout << "New BALANCE:";
          cin >> BALANCE;                                  // Value is read from the keyboard
          if(TESTACCOUNT.CHANGE_BALANCE(BALANCE))  // Calling function and query: 1 (true) or 0 (false)
                    cout << "BALANCE has been changed." << endl << endl;
          else
                    cout << " BALANCE has not been changed." << endl << endl;
          break;

          case 2:                                          // Query of the current BALANCE
          cout << "Actual BALANCE:";
          cout << TESTACCOUNT. READ_ACCOUNTBALANCE () << endl << endl;
          break;
          }

          } while(AUXILIARY);                              // As long as AUXILIARY is required!
  system("pause");
}                                                          // End of the program!
```