

A Complete Guide to Programming in C++

Ulla Kirch-Prinz
Peter Prinz

JONES AND BARTLETT PUBLISHERS

A Complete Guide to Programming in C++

Ulla Kirch-Prinz

Peter Prinz



JONES AND BARTLETT PUBLISHERS

Sudbury, Massachusetts

BOSTON TORONTO LONDON SINGAPORE

World Headquarters

Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers
Canada
2406 Nikanna Road
Mississauga, ON L5C 2W6
CANADA

Jones and Bartlett Publishers
International
Barb House, Barb Mews
London W6 7PA
UK

Copyright © 2002 by Jones and Bartlett Publishers, Inc.

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or any information storage or retrieval system, without written permission from the copyright owner.

Cover Image: Stones on shore-line and yellow leaf, Bjorkliden, Sweden, by Peter Lilja

Library of Congress Cataloging-in-Publication Data

Prinz, Peter.

[C++ Lernen und professionell anwenden. English]

A complete guide to programming in C++ / Peter Prinz, Ulla Kirch-Prinz; translated by Ian Travis.

p. cm.

ISBN: 0-7637-1817-3

1. C++ (Computer program language) I. Kirch-Prinz, Ulla. II. Title.

QA76.73.C153 P73713 2001

005.13'3—dc21

2001029617

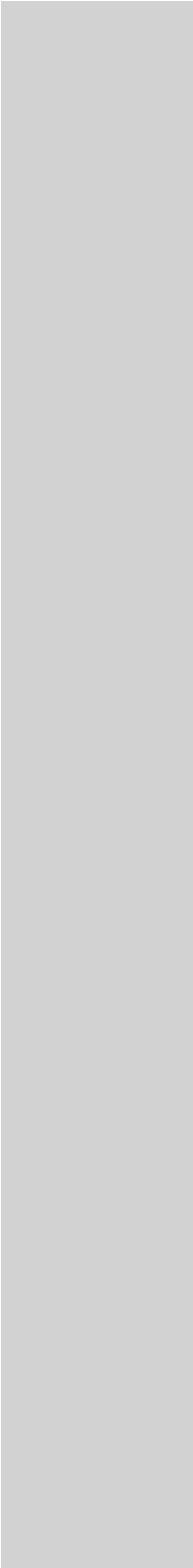
2090

Chief Executive Officer: Clayton Jones
Chief Operating Officer: Don W. Jones, Jr.
V.P., Managing Editor: Judith H. Hauck
V.P., Design and Production: Anne Spencer
V.P., Manufacturing and Inventory Control: Therese Bräuer
Editor-in-Chief: Michael Stranz
Development and Product Manager: Amy Rose
Marketing Manager: Nathan Schultz
Production Assistant: Tara McCormick
Cover Design: Night & Day Design
Composition: Northeast Compositors
Text Design: Mary McKeon
Printing and Binding: Courier Westford
Cover printing: John Pow Company, Inc.

This book was typeset in QuarkXpress 4.11 on a Macintosh G4. The font families used were Goudy, Gill Sans, Courier, Rubino Serif, and Seven Sans. The first printing was printed on 50 lb. Finch Opaque.

Printed in the United States of America

05 04 03 02 01 10 9 8 7 6 5 4 3 2 1



chapter

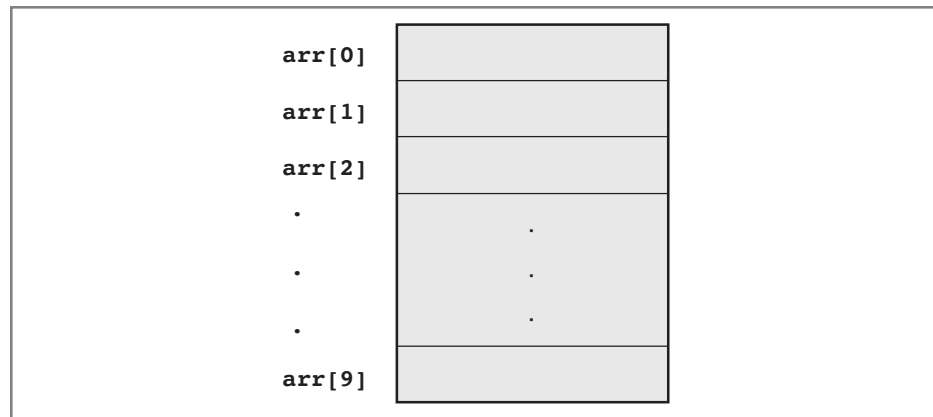
16

Arrays

This chapter describes how to define and use arrays, illustrating one-dimensional and multidimensional arrays, C strings and class arrays.

■ DEFINING ARRAYS

The array `arr` in memory



Sample program

```
// array.cpp
// To input numbers into an array and output after.
// -----
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    const int MAXCNT = 10;           // Constant
    float arr[MAXCNT], x;           // Array, temp. variable
    int i, cnt;                     // Index, quantity

    cout << "Enter up to 10 numbers \n"
          << "(Quit with a letter):" << endl;
    for( i = 0; i < MAXCNT && cin >> x; ++i)
        arr[i] = x;
    cnt = i;
    cout << "The given numbers:\n" << endl;
    for( i = 0; i < cnt; ++i)
        cout << setw(10) << arr[i];
    cout << endl;
    return 0;
}
```

An *array* contains multiple objects of *identical types* stored sequentially in memory. The individual objects in an array, referred to as *array elements*, can be addressed using a number, the so-called *index* or *subscript*. An array is also referred to as a *vector*.

□ Defining Arrays

An array must be defined just like any other object. The definition includes the *array name* and the *type* and *number of array elements*.

Syntax: `type name[count];` `// Array name`

In the above syntax description, `count` is an integral constant or integral expression containing only constants.

Example: `float arr[10];` `// Array arr`

This statement defines the array `arr` with 10 elements of `float` type. The object `arr` itself is of a *derived type*, an “array of `float` elements” or “`float` array.”

An array always occupies a *contiguous* memory space. In the case of the array `arr`, this space is `10*sizeof(float) = 40` bytes.

□ Index for Array Elements

The subscript operator `[]` is used to access individual array elements. In C++ an *index* always begins at zero. The elements belonging to the array `arr` are thus

`arr[0], arr[1], arr[2], ..., arr[9]`

The index of the last array element is thus 1 lower than the number of array elements. Any `int` expression can be used as an index. The subscript operator `[]` has high precedence, just like the class member operators `.` and `->`.

No error message is issued if the index exceeds the valid index range. As a programmer, you need to be particularly careful to avoid this error! However, you can define a class to perform range checking for indices.

You can create an array from any type with the exception of some special types, such as `void` and certain classes. Class arrays are discussed later.

Example: `short number[20];` `// short array`
 `for( int i=0; i < 20; i++ )`
 `number[i] = (short)(i*10);`

This example defines an array called `number` with 20 `short` elements and assigns the values 0, 10, 20, ..., 190 to the elements.

■ INITIALIZING ARRAYS

Sample program

```

// fibo.cpp
// The program computes the first 20 Fibonacci
// numbers and the corresponding Fibonacci quotients.
// -----
#include <iostream>
#include <iomanip>
#include <cmath>           // Prototype of sqrt()
#include <string>
using namespace std;

#define COUNT 20

long fib[COUNT + 1] = { 0, 1 };

string header =
" Index   Fibonacci number Fibonacci quotient  Deviation"
"\n                                     of limit "
"\n-----";

int main()
{
    int i;
    double q, lim;

    for( i=1; i < COUNT; ++i )        // Computing the
        fib[i+1] = fib[i] + fib[i-1]; // Fibonacci numbers

    lim = ( 1.0 + sqrt(5.0)) / 2.0;    // Limit

    // Title and the first two Fibonacci numbers:
    cout << header << endl;
    cout << setw(5) << 0 << setw(15) << fib[0] << endl;
    cout << setw(5) << 1 << setw(15) << fib[1] << endl;
    // Rest of the table:
    for( i=2; i <= COUNT; i++ )
    {
        // Quotient:
        q = (double)fib[i] / (double)fib[i-1];
        cout << setw(5) << i << setw(15) << fib[i]
            << setw(20) << fixed << setprecision(10) << q
            << setw(20) << scientific << setprecision(3)
            << lim - q << endl;
    }
    return 0;
}

```

□ Initialization List

Arrays can be initialized when you define them. A *list* containing the values for the individual array elements is used to initialize the array:

Example: `int num[3] = { 30, 50, 80 };`

A value of 30 is assigned to `num[0]`, 50 to `num[1]`, and 80 to `num[2]`. If you initialize an array when you define it, you do not need to state its length.

Example: `int num[] = { 30, 50, 80 };`

In this case, the length of the array is equal to the number of initial values. If the array length is explicitly stated in the definition and is larger than the number of initial values, any remaining array elements are set to zero. If, in contrast, the number of initial values exceeds the array length, the surplus values are ignored.

Locally defined arrays are created on the stack at program runtime. You should therefore be aware of the following issues when defining arrays:

- Arrays that occupy a large amount of memory (e.g., more than one kbyte) should be defined as *global* or *static*.
- Unless they are initialized, the elements of a local array will not necessarily have a definite value. Values are normally assigned by means of a loop.

You cannot assign a vector to another vector. However, you can overload the assignment operator within a class designed to represent arrays. This topic will be discussed in depth later.

□ The Sample Program Opposite

The example on the opposite page contains the first twenty Fibonacci numbers and their quotients. Fibonacci numbers are useful for representing natural growth. In computer science, Fibonacci numbers are used for things like memory management and hashing. Their definition is as follows:

- the first Fibonacci number is 0, the second is 1
- each subsequent Fibonacci number is the sum of its two immediate predecessors.

This results in the following sequence: 0, 1, 1, 2, 3, 5, 8, 13, ...

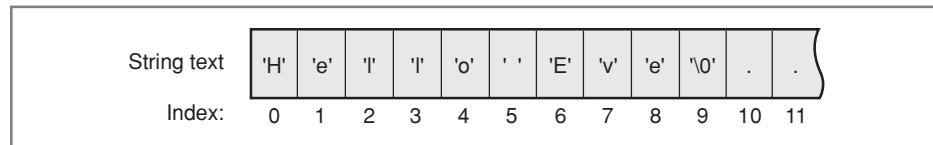
The quotient of a Fibonacci number and its predecessor is referred to as a Fibonacci quotient. The sequence of Fibonacci quotients, $1/1$, $2/1$, $3/2$, ..., converges towards the threshold value $(1 + \sqrt{5})/2$.

■ C STRINGS

□ Initializing

```
char text[40] = "Hello Eve";
```

String text in memory:



NOTE

The array `text` has length of 40, whereas the string "Hello Eve" only occupies the first 9 bytes.

Sample program

```
// C-string.cpp : Using C strings.
// -----
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;
char header[] = "\n    *** C Strings ***\n\n";
int main()
{
    char hello[30] = "Hello ", name[20], message[80];

    cout << header << "Your first name: ";
    cin >> setw(20) >> name;    // Enter a word.
    strcat( hello, name);      // Append the name.
    cout << hello << endl;
    cin.sync();                // No previous input.
    cout << "\nWhat is the message for today?"
         << endl;
    cin.getline( message, 80); // Enter a line with a
                               // max of 79 characters.
    if( strlen( message) > 0)  // If string length is
    {                          // longer than 0.
        for( int i=0; message[i] != '\0'; ++i)
            cout << message[i] << ' ';    // Output with
        cout << endl;                    // white spaces.
    }
    return 0;
}
```

□ char Arrays

Arrays whose elements are of `char` type are often used as data communication buffers.

Example: `char buffer[10*512]; // 5 Kbyte buffer`

However, their most common use is for string storage. One way of representing a string is to store the string and the terminating null character `'\0'` in a `char` array. When you define an array, you can use a string constant to initialize the array.

Example: `char name[] = "Hugo";`

This definition is equivalent to

```
char name[] = { 'H', 'u', 'g', 'o', '\0' };
```

As you can see, the string `name` occupies five bytes, including an additional byte for the null character. If you need to allocate more memory, you can state the size of the array explicitly as shown opposite.

In the C language, strings are usually represented as `char` vectors with a terminating null character. In C++, strings of this type are referred to as C strings to distinguish them from objects of the `string` class.

□ C Strings and the `string` Class

C strings are simple `char` arrays, which means that the functionality of the `string` class is not available for them. Thus, for example, assignments and comparisons are not defined.

Example: `char str1[20], str2[20] = "A string";`
`str1 = str2; // Error!`
`strcpy( str1, str2); // ok!`

The standard functions of the C language, such as `strlen()`, `strcpy()`, `strcmp()`, and others, are available for C strings. These global functions all begin with the `str` prefix.

As the program on the opposite page shows, I/O streams are overloaded for `char` arrays, too. Input and output are as easily achieved as with `string` class objects. However, the program must make sure not to overrun the end of the `char` array when reading data into the array. You can use the `width()` method or the `setw()` manipulator for this purpose.

Example: `cin >> setw(20) >> name; // 19 characters`

C strings are preferable to the `string` class if only a few operations are needed and you want to avoid unnecessary overheads.

■ CLASS ARRAYS

Sample program

```

// AccountTab.cpp
// An array containing objects of class Account.
// -----

#include "account.h"      // Definition of class Account
#include <iostream>
using namespace std;

Account giro("Lucky, Peter", 1234567, -1200.99 );
Account accountTab[] =
{
    Account("Tang, Sarah", 123000, 2500.0),
    Account("Smith, John", 543001),
    Account(),              // Default constructor
    "Li, Zhang",           // Account("Li, Zhang"),
    giro                   // Account(giro)
};

int cnt = sizeof(accountTab) / sizeof(Account);

int main()
{
    // To set some values:
    accountTab[1].setState( 10000.00);
                                // Assignment ok:
    accountTab[2] = Account("Pit, Dave", 727003, 200.00);

    cout << "The accounts in the table:" << endl;
    for( int i = 0; i < cnt; ++i)
    {
        accountTab[i].display();
        if( i % 3 == 2)
        {
            cout << "Press return to go on!\n";
            cin.get();
        }
    }
    cout << endl;

    return 0;
}

```

□ Declaring Class Arrays

Array elements can also be objects of a class type. The array is known as a *class array* in this case. When you declare an array of this type, you only need to state the type of the array elements.

Example: `Result temperatureTab[24];`

This statement defines the class array `temperatureTab` that stores 24 objects of type `Result`. This class was introduced at the beginning of the last chapter.

As the statement does not initialize the array explicitly, the default constructor is automatically called for each array element.



NOTE

Class arrays can only be defined without explicit initialization if a default constructor exists for the class.

Thus, the previous example is only valid for the first version of the `Result` class as this class contains a default constructor.

□ Explicit Initialization

A class array is initialized as usual by an *initialization list*. The list contains a constructor call for each array element.

Example:

```
Result temperatureTab[24] =
{
    Result( -2.5, 0,30,30),
    Result( 3.5),           // At present time
    4.5,                   // Just so
    Result( temp1),         // Copy constructor
    temp2                   // Just so
};
```

The first five array elements are initialized by the constructor calls implicitly contained in these statements. Instead of using a constructor with one argument, you can simply supply the argument. The default constructor is then called for the remaining elements.

If the size of an array is not stated explicitly, the number of values in the initialization list defines the size of the array.

The public interface of the objects in the array is available for use as usual.

Example: `temperatureTab[2].setTime( 2,30,21);`

No additional parentheses are needed in this statement since the subscript operator `[]` and the class member operator `.` are read from left to right, although they have the same precedence.

■ MULTIDIMENSIONAL ARRAYS

Sample program

```
// multidim.cpp
// Demonstrates multidimensional arrays.
// -----
#include <iostream>
#include <iomanip>
using namespace std;

char representative[2][20] = {"Armstrong, Wendy",
                             "Beauty, Eve"};

    // Each representative has five different
    // articles available, having sold the following:
int articleCount[2][5] = { { 20, 51, 30, 17, 44},
                          {150, 120, 90, 110, 88}
                          };

int main()
{
    for( int i=0; i < 2; i++ )
    {
        cout << "\nRepresentative: " << representative[i];
        cout << "\nNumber of items sold: ";

        for( int j = 0; j < 5; j++ )
            cout << setw(6) << articleCount[i][j];
        cout << endl;
    }
    return 0;
}
```

Screen output:

```
Representative:  Armstrong, Wendy
Items sold:      20    51    30    17    44

Representative:  Beauty, Eve
Items sold:      150   120    90   110    88
```

□ Defining Multidimensional Arrays

In C++ you can define multidimensional arrays with any number of dimensions. The ANSI standard stipulates a minimum of 256 dimensions but the total number of dimensions is in fact limited by the amount of memory available.

The most common multidimensional array type is the two-dimensional array, the so-called *matrix*.

Example: `float number[3][10]; // 3 x 10 matrix`

This defines a matrix called `number` that contains 3 *rows* and 10 *columns*. Each of the 30 (3×10) elements is a `float` type. The assignment

Example: `number[0][9] = 7.2; // Row 0, column 9`

stores the value 7.2 in the last element of the first row.

□ Arrays as Array Elements

C++ does not need any special syntax to define multidimensional arrays. On the contrary, an n -dimensional array is no different than an array with only one dimension whose elements are $(n-1)$ -dimensional arrays.

The array `number` thus contains the following three elements:

`number[0]    number[1]    number[2] .`

Each of these elements is a `float` array with a size of 10, which in turn forms the rows of the two-dimensional array, `number`.

This means that the same rules apply to multidimensional arrays as to one-dimensional arrays. The initialization list of a two-dimensional array thus contains the values of the array elements, that is, the one-dimensional rows.

Examples: `int arr[2][3] = { {5, 0, 0}, {7, 0, 0} };
int arr[][3] = { {5}, {7} };`

These two definitions are equivalent. When you initialize an array, you can only omit the size of the first dimension. It is necessary to define any other dimensions since they define the size of array elements.

□ The Example on the Opposite Page

The program opposite defines the two-dimensional arrays `representative` and `articleCount`, which have two rows each. The `representative[i]` rows are `char` arrays used for storing the names of the representatives. You can also use a one-dimensional `string` array.

Example: `string representative[2] = {"La..", "Fo.."};`

■ MEMBER ARRAYS

Class TelList

```

// telList.h
// Class TelList to represent a list
// containing names and telephone numbers.
// -----
#ifndef _TelList_
#define _TelList_

#include <string>
using namespace std;
#define PSEUDO -1          // Pseudo position
#define MAX 100           // Maximal number of elements

// Type of a list element:
struct Element { string name, telNr; };

class TelList
{
private:
    Element v[MAX];        // The array and the current
    int count;             // number of elements

public:
    TelList(){ count = 0;}
    int getCount() const { return count; }
    Element *retrieve( int i )
    {
        return (i >= 0 && i < count)? &v[i] : NULL;
    }
    bool append( const Element& el )
    {
        return append( el.name, el.telNr);
    }
    bool append( const string& name,
                  const string& telNr);
    bool erase( const string& name);
    int  search( const string& name);
    void print();
    int  print( const string& name);
    int  getNewEntries();
};

#endif // _TelList_

```

□ Encapsulating Arrays

A programmer often needs to handle objects of the same type, such as company employees, bank accounts, or the articles in stock. A class designed to perform this task can use an array for ease of data management. An array allows you to access individual objects directly and perform searches.

A class that encapsulates an array will provide methods for simple array operations, such as inserting and deleting objects. When you design a class of this type, one aim will be to perform automatic range checking. This helps avoid overrunning the end of an array when performing read or write operations. The resulting class will contain a comfortable and safe interface for object data management.

□ The Class `TelList`

The class `TelList` on the opposite page is designed to manage a simple telephone list.

Each entry in the list contains a dataset containing a name and a phone number. The `Element` type, which comprises two strings, was defined for this purpose. The array `v` can store up to `MAX` entries of the `Element` type. The data member `count` records the number of elements currently stored in the array. When a phone list is created, this number will initially be 0. When an element is inserted or deleted, the number is modified correspondingly.

The `TelList` class uses a single default constructor that sets the counter, `count`, to zero. It is not necessary to provide an initial value for the `MAX` elements in the array `v` since the default constructor of the `string` class is executed for all strings.

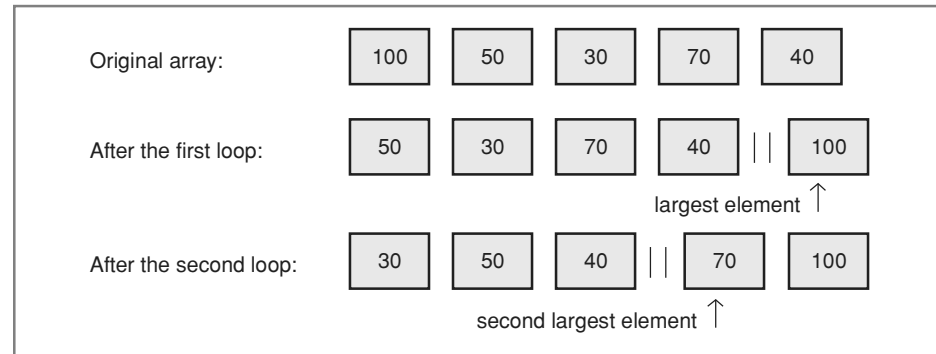
The tasks performed by the other methods are easily deduced from their names. The `retrieve()` method returns to a given index a pointer to the corresponding element. Using a pointer makes it possible to return a `NULL` pointer if the index is invalid.

The `append()` methods add a new entry to the list. The data passed to a method is copied to the next free array element and the counter is incremented. If there is no space available, the name field is empty, or the name is already in use, nothing happens. In this case, the method returns `false` instead of `true`.

The exercises for this chapter contain further details on these methods. You can implement the methods for the `TelList` yourself and go on to test them.

■ EXERCISES

Example of a bubble sort algorithm



Sieve of Eratosthenes

For this task you can define an array of boolean values in which each element is initially `true`. To eliminate a number n you simply set the n^{th} element in the array to `false`.

Result:

Index	0	1	2	3	4	5	6	7	8	9	...
Array	false	false	true	true	false	true	false	true	false	false	...

Screen shot of exercise 4

```

      * * B R E A K * * * *
--- Press interrupt key to terminate (^C) ---

```

✓ NOTE

The output of a scrolling string has to be performed at the same cursor position. The screen control characters make it possible to locate the cursor, and that independent of the current compiler (see appendix).

Exercise 1

Write a C++ program that reads a maximum of 100 integers from the keyboard, stores them in a `long` array, sorts the integers in ascending order, and displays sorted output. Input can be terminated by any invalid input, such as a letter.

**NOTE**

Use the *bubble sort algorithm* to sort the array. This algorithm repeatedly accesses the array, comparing neighboring array elements and swapping them if needed. The sorting algorithm terminates when there are no more elements that need to be swapped. You use a flag to indicate that no elements have been swapped.

Exercise 2

Chapter 14 introduced the sample class `DayTime` and the `isLess()` method. Define and initialize an array with four `DayTime` class objects.

Then write a `main` function that first uses the `print()` method to display the four elements. Finally, find the largest and smallest elements and output them on screen.

Exercise 3

Write a program that outputs all prime numbers less than 1000. The program should also count the number of prime numbers less than 1000. An integer ≥ 2 is a prime number if it is not divisible by any number except 1 and itself. Use the *Sieve of Eratosthenes*:

To find primary numbers simply eliminate multiples of any primary numbers you have already found, i.e.:

first eliminate any multiples of 2 (4, 6, 8, ...),
 then eliminate any multiples of 3 (6, 9, 12, ...),
 then eliminate any multiples of 5 (10, 15, 20, ..) // 4 has already been eliminated and so on.

Exercise 4

Write a C++ program to create the screen output shown opposite. The following banner

```
* * * B R E A K * * *
```

is to be displayed in the center of the window and scrolled left. You can scroll the banner by beginning string output with the first character, then the second, and so on. Handle the string like a loop where the first letter follows the last letter and output continues until the starting position is reached.

You can use a wait loop to modify the speed of the banner after each string is output.

Exercise 5**Methods to be implemented for the TelList class**

```
bool append( const string& name,
             const string& telNr);

bool erase( const string& name);

int  search( const string& name);

void print();

int  print( const string& name);

int  getNewEntries();
```

Menu of the application program

```
***** Telephone List *****

D = Display all entries
F = Find a telephone number
A = Append an entry
E = Erase an entry
Q = Quit the program

Your choice:
```

Exercise 5

The sample class `TelList` was introduced in this chapter; however, some methods still need to be implemented and tested.

- Implement the `TelList` class methods shown opposite.

The name is used as an unambiguous key. This means the `append()` method can only be used to append an entry provided the name is neither blank nor already in use.

The method `erase()` deletes an array element. The position of the element to be deleted is first located using the `search()` method. If the element does not exist, `erase()` returns a value of `false`. In any other case, the last element in the array is used to overwrite the element that is to be deleted and the counter `count` is decremented.

The `search()` method finds the position in the array that contains the search name. If the search operation is unsuccessful, the value `PSEUDO` is returned.

The `print` method without parameters outputs all available entries. You can pass the first letter or letters of a name to the second method to output any entries beginning with these letters. Use the method `compare()` from the `string` class to help you with this task.

Example: `str1.compare( 0, 5, str2) == 0`

This expression is true if the five characters subsequent to position 0 in the strings `str1` and `str2` are identical.

The `getNewEntries()` method is used to read new phone list entries from the keyboard. Each new entry is appended using the `append()` method. Reading should be terminated if the user types an empty string. The method returns the number of new entries.

Write an application program that creates a phone list of type `TelList` and displays the menu shown on the opposite page.

- The menu must be placed in a function of your own that can return the command input. The menu must be called in the main loop of the program. Depending on the command input, one of the methods defined in the class `TelList` should be called. If the menu item “Erase” or “Search” is chosen, you must also read a name or the first letters of a name from the keyboard.



NOTE

The phone list will not be stored permanently in a file. This is just one of the enhancements (another would be variable length) that will be added at a later stage.

■ SOLUTIONS

Exercise I

```
// -----
// bubble.cpp
// Inputs integers into an array,
// sorts in ascending order, and outputs them.
// -----
#include <iostream>
#include <iomanip>
using namespace std;

#define MAX 100 // Maximum number
long number[MAX];

int main()
{
    int i, cnt; // Index, quantity

    cout << "\nS o r t i n g   I n t e g e r s \n"
          << endl;
    // To input the integers:
    cout << "Enter up to 100 integers \n"
          << "(Quit with any letter):" << endl;
    for( i = 0; i < MAX && cin >> number[i]; ++i)
        ;
    cnt = i;
    // To sort the numbers:
    bool sorted = false; // Not yet sorted.
    long help; // Swap.
    int end = cnt; // End of a loop.

    while( !sorted) // As long as not
    { // yet sorted.
        sorted = true;
        --end;
        for( i = 0; i < end; ++i) // Compares
        { // adjacent integers.
            if( number[i] > number[i+1])
            {
                sorted = false; // Not yet sorted.
                help = number[i]; // Swap.
                number[i] = number[i+1];
                number[i+1] = help;
            }
        }
    }
}
```

```

// Outputs the numbers
cout << "The sorted numbers:\n" << endl;

for( i = 0; i < cnt; ++i)
    cout << setw(10) << number[i];
cout << endl;

return 0;
}

```

Exercise 2

```

// -----
// DayTime.h
// The class DayTime represents the time in hours,
// minutes and seconds.
// -----
#ifndef _DAYTIME_
#define _DAYTIME_
#include <iostream>
#include <iomanip>
using namespace std;

class DayTime
{
private:
    short hour, minute, second;
    bool overflow;
public:
    DayTime( int h = 0, int m = 0, int s = 0)
    {
        overflow = false;
        if( !setTime( h, m, s))          // this->setTime(...)
            hour = minute = second = 0;
    }
    bool setTime(int hour, int minute, int second = 0)
    {
        if(    hour    >= 0    &&    hour < 24
            && minute >= 0    &&    minute < 60
            && second >= 0    &&    second < 60 )
        {
            this->hour    = (short)hour;
            this->minute  = (short)minute;
            this->second  = (short)second;
            return true;
        }
        else
            return false;
    }
}

```

```

int getHour()    const { return hour;    }
int getMinute() const { return minute; };
int getSecond() const { return second; };

int asSeconds() const      // Daytime in seconds
{
    return (60*60*hour + 60*minute + second);
}

bool isLess( DayTime t) const // Compares
                                // *this and t.
{
    return asSeconds() < t.asSeconds();
} // this->sSeconds() < t.asSeconds();

void print() const
{
    cout << setfill('0')
          << setw(2) << hour    << ':'
          << setw(2) << minute  << ':'
          << setw(2) << second  << " Uhr" << endl;
    cout << setfill(' ');
}

void swap( DayTime& t)          // Just one parameter!
{                               // Swaps *this and t:
    DayTime temp(t);  t = *this; *this = temp;
}

};

#endif // _DAYTIME_

// -----
// TimeTab.cpp
// An array containing objects of class DayTime.
// -----
#include "DayTime.h" // Definition of class DayTime
#include <iostream>
using namespace std;
char header[] =
"\n\n    *** Table with Daytimes ***\n\n";

int main()
{
    DayTime timeTab[4] =
        { 18, DayTime(10,25), DayTime(14,55,30) };
    int i;
    timeTab[3].setTime( 8,40,50); // Last element.
    cout << header << endl;

```

```
// Output:
for( i = 0; i < 4; ++i)
{
    timeTab[i].print();
    cout << endl;
}

// To compute shortest and longest time:
int i_min = 0, i_max = 0;           // Indices for shortest
                                   // and longest elements.
for( i = 1; i < 4; ++i)
{
    if( timeTab[i].isLess( timeTab[i_min]) )
        i_min = i;

    if( timeTab[i_max].isLess( timeTab[i]) )
        i_max = i;
}

cout << "\nShortest time: ";  timeTab[i_min].print();

cout << "\nLongest time: ";  timeTab[i_max].print();

return 0;
}
```

Exercise 3

```
// -----
// sieve.cpp
// Identifies prime numbers using the Sieve of
// Eratosthenes.
// -----
#include <iostream>
#include <iomanip>
using namespace std;
#define LIMIT 1000           // Upper limit
bool flags[LIMIT] = { false, false}; // Array with flags

int main()
{
    register int i, j;           // Indices
    for( i = 2; i < LIMIT; ++i)
        flags[i] = true;        // Sets flags to true
    // Sieving:
    for( i = 2; i < LIMIT/2; ++i)
    {
        if( flags[i])           // Is i a prime number?
        {                       // Yes -> Delete multiples.
            for( j = i+i; j < LIMIT; j += i)
                flags[j] = false;
        }
    }
    // To count:
    int count = 0;               // Counter
    for( i = 2; i < LIMIT; ++i)
        if(flags[i])             // If i is a prime number
            ++count;             // -> count
    // Output:
    cout << "There are"<< count <<" prime numbers less than"
         << LIMIT << endl;
    cout << "\nTo output prime numbers? (y/n) ";
    char reply; cin.get(reply);
    if( reply == 'y' || reply == 'Y')
    { for( i = 2; i < LIMIT; ++i)
        if(flags[i])             // If i is a prime number
        {                       // -> to output it.
            cout.width(8); cout << i;
        }
    }
    cout << endl;    return 0;
}
```

Exercise 4

```
// -----
// scroll.cpp
// Scrolling a message.
// -----
#include <iostream>
#include <iomanip>
using namespace std;
#define DELAY 1000000L // Output delay

inline void cls() // Clear screen
{
    cout << "\033[2J\n";
}

inline void locate(int z, int s) // Put cursor in row z
{                               // and column s
    cout << "\033[" << z << ';' << s << 'H';
}

char msg[] = "* * * B R E A K * * * ";

int main()
{
    int i, start = 0, len = strlen(msg);

    cls(); locate(24, 20); // Row 24, column 20
    cout << "--- Press interrupt key to terminate (^C) ---";

    while( true )
    {
        locate( 12, 25); // Row 12, column 25
        i = start; // Output from index start
        do
        {
            cout << msg[i++];
            i = i % len; // if( i == len) i = 0;
        }
        while( i != start);
        cout << endl; // Outputs buffer to screen
        // Wait in short
        for( int count = 0; count < DELAY; ++count)
            ;
        ++start; // For next output
        start %= len; // start = start % len;
    }
    cls();
    return 0;
}
```

Exercise 5

```
// -----
// telList.h
// The class TelList representing a list
// with names and telephone numbers.
// -----
//
// As before in this chapter.

// -----
// telList.cpp
// Implements the methods of class TelList.
// -----
#include "telList.h"      // Definition of class TelList
#include <iostream>
#include <iomanip>
using namespace std;

bool TelList::append( const string& name,
                      const string& telNr)
{
    if( count < MAX           // Space available,
        && name.length() > 1   // 2 characters at least
        && search(name) == PSEUDO) // not yet existing
    {
        v[count].name = name;
        v[count].telNr = telNr;
        ++count;
        return true;
    }
    return false;
}

bool TelList::erase( const string& key )
{
    int i = search(key);
    if( i != PSEUDO )
    {
        // Copies the last
        v[i] = v[count-1]; --count; // element to position i
        return true;
    }
    return false;
}
```

```

int TelList::search(const string& key )
{
    for( int i = 0; i < count; i++ )           // Searching.
        if( v[i].name == key )
            return i;                           // Found

    return PSEUDO;                             // Not found
}
// Functions to support the output:
inline void tabHeader()                       // Title of the table
{
    cout << "\n  Name                               Telephone #\n"
         << "-----"
         << endl;
}
inline void printline( const Element& el)
{
    cout << left << setw(30) << el.name.c_str()
         << left << setw(20) << el.telNr.c_str()
         << endl;
}

void TelList::print()                       // Outputs all entries
{
    if( count == 0 )
        cout << "\nThe telephone list  is empty!" << endl;
    else
    {
        tabHeader();
        for( int i = 0; i < count; ++i)
            printline( v[i]);
    }
}

int TelList::print( const string& name) const // Entries
{                                           // beginning with name.
    int matches = 0, len = name.length();

    for( int i = 0; i < count; ++i)
    {
        if( v[i].name.compare(0, len, name) == 0 )
        {
            if( matches == 0) tabHeader(); // Title before
                                           // first output.
            ++matches;
            printline( v[i]);
        }
    }
    if( matches == 0 )
        cout << "No corresponding entry found!" << endl;
    return matches;
}

```

```

int TelList::getNewEntries()           // Input new entries
{
    int inputCount = 0;
    cout << "\nEnter new names and telephone numbers:"
         "\n(Terminate by empty input) "
         << endl;
    Element el;
    while( true)
    {
        cout << "\nNew last name, first name: ";
        cin.sync(); getline( cin, el.name);
        if( el.name.empty())
            break;
        cout << "\nTelephone number: ";
        cin.sync(); getline( cin, el.telNr);
        if( !append( el))
        {
            cout << "Name has not been found!" << endl;
            if( count == MAX)
            {
                cout << "The Table is full!" << endl;
                break;
            }
            if( search( el.name) != PSEUDO)
                cout << "Name already exists!" << endl;
        }
        else
        {
            ++inputCount;
            cout << "A new element has been inserted!"
                 << endl;
        }
    }
    return inputCount;
}

// -----
// telList_t.cpp
// Manages a telephone list.
// -----
#include "telList.h"           // Definition of class TelList
#include <iostream>
#include <string>
#include <cctype>
using namespace std;
inline void cls()
{ cout << "\033[2J\n"; // Output only new-lines, if ANSI
}                          // control characters are not available.

```

```

inline void go_on()
{
    cout << "\n\nGo on with return! ";
    cin.sync(); cin.clear();           // No previous input
    while( cin.get() != '\n')
        ;
}

int menu();                          // Reads a command

char header[] =
"\n\n          ***** Telephone List *****\n\n";

TelList myFriends;                   // A telephone list

int main()
{
    int action = 0;                   // Command
    string name;                      // Reads a name

    myFriends.append("Lucky, Peter", "0203-1234567");

    while( action != 'B')
    {
        action = menu();
        cls();
        cout << header << endl;

        switch( action)
        {
            case 'D':                 // Show all
                myFriends.print();
                go_on();
                break;

            case 'F':                 // Search
                cout <<
                "\n--- To search for a phone number ---\n"
                "\nEnter the beginning of a name: ";
                getline( cin, name);
                if( !name.empty())
                {
                    myFriends.print( name);
                    go_on();
                }
                break;

            case 'A':                 // Insert
                myFriends.getNewEntries();
                break;
        }
    }
}

```

```

        case 'E':                                // Delete
            cout <<
                "\n--- To delete a telephone entry. ---\n "
                "\nEnter the complete name: ";
            getline( cin, name);
            if( !name.empty())
            {
                if( !myFriends.erase( name))
                    cout << name << " not found!"
                        << endl;
                else
                    cout << "Entry for " << name
                        << " deleted!" << endl;
                go_on();
            }
            break;

        case 'T':  cls();                        // To terminate
                    break;
    }
} // End of while

return 0;
}

int menu()
{
    static char menuStr[] =
        "\n\n          D = Display all entries"
        "\n\n          F = Find a telephone number"
        "\n\n          A = Append a new entry "
        "\n\n          E = Erase an entry "
        "\n\n          Q = Quit the program"
        "\n\n Your choice: ";

    cls();
    cout << header << menuStr;

    char choice;
    cin.sync(); cin.clear();                    // No previous input
    if( !cin.get(choice))
        choice = 'B';
    else
        choice = toupper(choice);

    cin.sync();                                // Clear input buffer
    return choice;
}

```

Arrays and Pointers

This chapter describes the relationship between pointers and arrays. This includes:

- pointer arithmetic
- pointer version of functions
- pointers as return values and read-only pointers
- pointer arrays

Operations that use C strings illustrate how to use pointers for efficient programming. String access via the command line of an application program is used to illustrate pointer arrays.

■ ARRAYS AND POINTERS (I)

Sample program

```

// textPtr.cpp
// Using arrays of char and pointers to char
// -----
#include <iostream>
using namespace std;

int main()
{
    cout << "Demonstrating arrays of char "
          << "and pointers to char.\n"
          << endl;

    char text[] = "Good morning!",
          name[] = "Bill!";
    char *cPtr = "Hello ";           // Let cPtr point
                                     // to "Hello ".

    cout << cPtr << name << '\n'
          << text << endl;

    cout << "The text \"" << text
          << "\" starts at address " << (void*)text
          << endl;

    cout << text + 6           // What happens now?
          << endl;

    cPtr = name;               // Let cPtr point to name, i.e. *cPtr
                              // is equivalent to name[0]
    cout << "This is the " << *cPtr << " of " << cPtr
          << endl;
    *cPtr = 'k';
    cout << "Bill can not " << cPtr << "!\n" << endl;
    return 0;
}

```

Sample output:

```

Demonstrating arrays of char and pointers to char.
Hello Bill!
Good morning!
The text "Good morning!" starts at address 00451E40
morning!
This is the B of Bill!
Bill can not kill!

```

□ Name and Address of an Array

In C++ the name of an array is also the starting address for that array. To be more precise, an array name is a pointer to the first array element.

Example: `char town[] = "Beijing";`

In this case, `town` is a `char` pointer to `town[0]`, that is, a pointer to the memory address that stores the 'B' character. Expressions `town` and `&town[0]` are thus equivalent.

Example: `cout << town; // or: cout << &town[0];`

A pointer to the first character of the string `town` is passed. The characters forming the string are read and displayed from this point onward until the terminating null character, `'\0'`, is reached.

□ Pointer Variables and Arrays

An array name is not a pointer variable but a constant that cannot be modified. However, you can assign this constant to a pointer variable.

Example: `char *cPtr;
cPtr = town; // or: cPtr = &town[0];
cout << cPtr; // To output "Beijing"`

Now `cPtr` points to the array element `town[0]` just like `town`. But, in contrast to `town`, `cPtr` is a variable that can be moved.

Example: `cPtr = "Hello!";`

After this statement, `cPtr` points to the 'H' character. String constants such as `"Hello!"` are also `char` arrays and thus represent the address of the first array element.

□ Typeless Pointers

If you need to display the address rather than the string, you should pass a `void*` type pointer rather than a `char` pointer.

Example: `cout << (void *)town;`

This casts the `char` pointer to a `void *` type pointer and passes it as an argument to the `<<` operator, which in turn outputs the address in hexadecimal format. The `<<` operator belongs to the `ostream` class and is overloaded for `void *` types for this purpose.

A `void *` pointer represents a memory address without establishing a certain type. `void *` pointers are also referred to as *typeless pointers* for this reason. When you use a typeless pointer for memory access, you must therefore name the type being accessed explicitly by means of type casting.

■ ARRAYS AND POINTERS (2)

Sample program

```
// arrPtr.cpp
// Outputs addresses and values of array elements.
// -----

#include <iostream>
using namespace std;

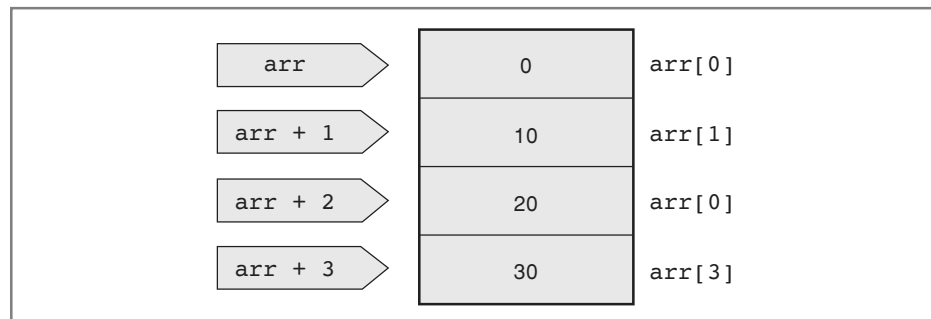
int arr[4] = { 0, 10, 20, 30 };

int main()
{
    cout << "\nAddress and value of array elements:\n"
          << endl;

    for( int i = 0; i < 4; i++ )
        cout << "Address: " << (void*)(arr+i)    // &arr[i]
              << "    Value: " << *(arr+i)        // arr[i]
              << endl;

    return 0;
}
```

Interrelation between pointers and array elements



□ Addressing Array Elements

Access to individual array elements in C++ is very closely related to pointer arithmetic. Now let's look at an `int` array to illustrate this point.

Example: `int arr[4] = { 0, 10, 20, 30 };`

As you already know, the name of the array `arr` is an `int` pointer to `arr[0]`.

Now it is possible to add or subtract pointers and integral values. The size of the object referenced by the pointer is automatically taken into consideration.

Since `arr` is an `int` pointer to `arr[0]`, `arr+1` points to the next array element `arr[1]`, i.e., to an address that is `sizeof(int)` bytes higher in memory. The memory space between the two entries will be two or four bytes, depending on the size of the type `int`. Thus the following applies to any given number, `i`:

`arr + i` points to the array element `arr[i]` ,
`*(arr + i)` is the array element `arr[i]` ,

This technique can also be used to address memory spaces outside of the array. Thus, `arr - 1` addresses the word that precedes `arr[0]`. But generally this does not make much sense, since you have no means of knowing what is stored at this memory address.

□ Addressing with Pointer Variables

Array elements can also be addressed using pointer variables.

Example: `int *ptr = arr; // ptr points to arr[0]`

In this case, both `ptr` and `arr` are pointers to the array element `arr[0]`. Thus, `ptr + 1`, `ptr + 2`, . . . point to the array elements `arr[1]`, `arr[2]`, . . .

For any given integer, `i`, the following expressions are thus equivalent:

`&arr[i]` `arr + i` `ptr + i`

The following thus represent equivalent values:

`arr[i]` `*(arr + i)` `*(ptr + i)` `ptr[i]`

At first it might seem surprising that you can use the array notation `ptr[i]` for pointers. The compiler translates `arr[i]` to `*(arr + i)`—in other words: “Start at address `arr`, move `i` objects up, and access the object!” This also applies to `ptr[i]`.

■ POINTER ARITHMETIC

Examples for arithmetic with pointers

```
float v[6] = { 0.0, 0.1, 0.2, 0.3, 0.4, 0.5 },
          *pv, x;

pv = v + 4;      // Let pv point to v[4].
*pv = 1.4;       // Assign 1.4 to v[4].
pv -= 2;         // Reset pv to v[2].
++pv;           // Let pv point to v[3].

x = *pv++;       // Assign v[3] to x and
                // increment pv.
x += *pv--;      // Increment x by v[4] and let
                // pv point to v[3] again.
--pv;           // Reset pv to v[2].
```

To step through an array of classes

```
// Searches for a given account number in a table of
// accounts and outputs the found account.
// -----
#include "account.h"      // Definition of class Account.

Account accountTab[100]; // Table containing accounts.

int main()
{
    int cnt;                // Actual number of accounts.
    Account *aPtr;          // Pointer to Account-objects.
    // To input data into accountTab and actualize cnt.
    // To search for the account number 1234567:
    bool found = false;
    for( aPtr = accountTab; aPtr < accountTab+cnt; ++aPtr)
        if( aPtr->getNr() == 1234567 )
        {
            found = true;
            break;
        }
    if( found)               // Found?
        aPtr->display();     // Yes -> display.
    // To continue
}
```

In C++ you can perform arithmetic operations and comparisons with pointers, provided they make sense. This primarily means that the pointer must always point to the elements of an array. The following examples show some of your options with pointer arithmetic:

Example:

```
float v[6], *pv = v; // pv points to v[0]
int i = 3;
```

□ Moving a Pointer in an Array

As you already know, the addition `pv + i` results in a pointer to the array element `v[i]`. You can use a statement such as `pv = pv + i;` to store the pointer in the variable `pv`. This moves the pointer `pv` `i` objects, that is, `pv` now points to `v[i]`.

You can also use the operators `++`, `--`, and `+=` or `-=` with pointer variables. Some examples are shown opposite. Please note that the indirection operator, `*`, and the operators `++` and `--` have the same precedence. Operators and operands thus are grouped from right to left:

Example: `*pv++` is equivalent to `*(pv++)`

The `++` operator increments the pointer and not the variable referenced by the pointer. Operations of this type are not possible using the pointer `v` since `v` is a *constant*.

□ Subtracting Pointers

An addition performed with two pointers does not return anything useful and is therefore invalid. However, it does make sense to perform a *subtraction* with two pointers, resulting in an `int` value that represents the number of array elements between the pointers. You can use this technique to compute the index of an array element referenced by a pointer. To do so, you simply subtract the starting address of the array. For example, if `pv` points to the array element `v[3]`, you can use the following statement

Example:

```
int index = pv - v;
```

to assign a value of 3 to the variable `index`.

□ Comparing Pointers

Finally, *comparisons* can be performed with two pointers of the same type.

Example:

```
for( pv = v + 5; pv >= v; --pv)
    cout << setw(10) << *pv;
```

This loop outputs the numbers contained in `v` in reverse order. In the example on the opposite page, the pointer `aPtr` walks through the first `cnt` elements of the array `accountTab`, as long as `aPtr < accountTab + cnt`.

■ ARRAYS AS ARGUMENTS

Sample program

```

// reverse.cpp
// Defines and calls the function reverse().
// reverse() copies a C string into another C string
// and reverses the order of characters.
// -----

#include <iostream>
using namespace std;
#include <string.h>          // Header-File for Cstrings,
                           // here for strlen().
void reverse( char str[], char umstr[]); // Prototype

int main()                  // Read a word and
{                            // output in reversed order.
    const int CNT = 81;
    char word[CNT], revword[CNT];

    cout << "Enter a word: ";
    cin.width(CNT);          // maximal CNT-1 characters
    cin >> word;

    reverse( word, revword); // Call

    cout << "\nThe \"reversed\" word:  " << revword
         << endl ;

    return 0;
}

void reverse( char s1[], char s2[]) // Copies the
{                                  // reversed C string s1 to s2
    int j = 0;

    for( int i = strlen(s1)-1; i >= 0; i--, j++)
        s2[j] = s1[i];

    s2[j] = '\0';              // Terminating character
}

```

Sample output:

```

Enter a word: REGAL
The "reversed" word:  LAGER

```

If an array name is passed as an argument when calling a function, the function actually receives the address of the first array element. The called function can then perform read or write operations for any element in the array.

□ Declaring Parameters

If the argument is an array, there are two equivalent methods of declaring parameters. This point is illustrated by the example using `strlen()` to return the length of a C string. For example, calling `strlen("REGAL")` returns a value of 5.

1. You can declare the parameter as an array.

Example:

```
int strlen( char str[])    // Compute length of
{   int i;                // str without '\0'.
    for( i = 0; str[i] != '\0'; ++i)
        ;
    return (i);
}
```

2. You can declare the parameter as a pointer.

Example:

```
int strlen( char *str)
{   /* as above */ }
```

In both cases the parameter `str` is a pointer that stores the starting address of the array. Array notation is preferable if you intend to use an index to access the elements of an array. Calling `strlen("REGAL")`; leads to the following situation:

'R'	'E'	'G'	'A'	'L'	'\0'
<code>str[0]</code>	<code>str[1]</code>	<code>str[2]</code>	<code>str[3]</code>	<code>str[4]</code>	<code>str[5]</code>

As you can see, the length of a C string is equal to the index of the element containing the terminating null character.

The function `reverse()` on the opposite page copies the characters of a C string to a second char array in reverse order, first copying the last character in `s1`, that is, the character with the index `strlen(s1) - 1`, to `s2[0]`, then the second to last character `s2[1]`, and so on.

□ Array Length

A function to which an array is passed initially knows only the starting address of the array but not its length. In the case of C strings, the length is derived implicitly from the position of the terminating null character. In most other cases the length must be supplied explicitly.

Example:

```
void sort( Account aTab[], int len )
{   /* To sort array aTab of length len */ }
```

■ POINTER VERSIONS OF FUNCTIONS

□ Function `strcpy`

The standard function `strcpy()` copies C strings.

Example: `char dest[30], source[] = "A string";
strcpy(dest, source);`

Here the string `source` is copied to `dest` “from left to right” just like an assignment.

The following function `strcpy()` is somewhat simpler than the standard function since it has no return value.

Index Version of `strcpy()`

```
void strcpy( char s1[], char s2[])    // Copies s2 to s1
{
    int i;                           // Index
    for( i = 0; s2[i] != '\0'; ++i)  // Copy.
        s1[i] = s2[i];
    s1[i] = '\0';                     // Append terminating
                                     // character.
}
```

Pointer version 1 of `strcpy()`

```
void strcpy( char *s1, char *s2)    // Copies s2 to s1
{
    for( ; *s2 != '\0'; ++s1, ++s2) // Copy
        *s1 = *s2;
    *s1 = '\0';                     // Append terminating
                                     // character.
}
```

Pointer version 2 of `strcpy()`

```
void strcpy( char *s1, char *s2)    // Copy s2 to s1.
{
    while( (*s1++ = *s2++) != '\0' ) // Copy and append
        ;                             // terminating
                                     // character.
}
```

□ Using Pointers Instead of Indices

As we have already seen, a parameter for an array argument is always a pointer to the first array element. When declaring parameters for a given type *T*:

T name[] is always equivalent to *T* *name.

So far, in previous sample functions, the pointer has been used like a fixed base address for the array, with an index being used to access the individual array elements. However, it is possible to use pointers instead of indices.

Example: A new version of the standard function `strlen()`:

```
int strlen( char *str)      // Computes length
{
    char* p = str;
    for( p = str; *p != '\0'; ++p) // Search
        ;                       // for \0
    return (p - str);
}
```

In this case, the difference between two pointers results in the string length.

□ The Sample Functions Opposite

The first version of the function `strcpy()` “string copy” opposite uses an index, whereas the second does not. Both versions produce the same results: the string *s2* is copied to *s1*. When you call the function, you must ensure that the `char` array referenced by *s1* is large enough.

As the parameters *s1* and *s2* are pointer variables, they can be shifted. The second “pointer version” of `strcpy()`, which is also shown opposite, uses this feature, although the function interface remains unchanged.

Generally, pointer versions are preferable to index versions as they are quicker. In an expression such as *s1*[*i*] the values of the variables *s1* and *i* are read and added to compute the address of the current object, whereas *s1* in the pointer version already contains the required address.

□ Multidimensional Arrays as Parameters

In a parameter declaration for *multidimensional* arrays, you need to state every dimension with the exception of the first. Thus, a parameter declaration for a two-dimensional array will always contain the number of columns.

Example: `long func( int num[][10] );` // ok.
`long func( int *num[10] );` // also ok.

■ READ-ONLY POINTERS

Sample program

```

// accountFct.cpp
// Defines and calls a function, which outputs
// a list of overdrawn accounts.
// -----
#include "account.h"      // Definition of class Account.

Account accountTab[] =    // Table with Account-objects.
{ Account("Twain, Mark", 1234567, -3434.30),
  Account("Crusoe, Robinson", 200000, 0.00),
  Account("Temple, Shirley", 543001, +777.70),
  Account("Valentin, Carl", 543002, -1111.10),
};
int cnt = sizeof(accountTab) / sizeof(Account);
// Prototype:
int displayOverdraw( const Account *aTab, int cnt,
                    double limit);

int main()
{
    double limit = 0.0;
    cout << "Output the overdrawn accounts!\n"
          << "These are the accounts, which fell below \n"
          << "the limit, ex. -1000.00.\n" << endl;
    cout << "What is the limit? ";
    cin >> limit;

    cout << "Listing the overdrawn accounts:\n" << endl;
    if( displayOverdraw( accountTab, cnt, limit) == 0)
        cout << "\nNo account found!"
              << endl;
    return 0;
}

int displayOverdraw( const Account *aTab, int cnt,
                    double limit)
{
    int count = 0;
    const Account* aPtr;
    for( aPtr = aTab; aPtr < aTab + cnt; ++aPtr)
        if( aPtr->getState() < limit ) // Below the limit?
        {
            aPtr->display();           // Yes -> display.
            ++count;
        }
    return count;
}

```

□ Pointers to `const` Objects

You can use a normal pointer for both read and write access to an object. However, just like the definition used for a reference, you can also define a *read-only pointer*, that is, a pointer that can be used for read operations only. In fact, a read-only pointer is obligatory if you need to point to a constant object.

□ Declaration

You use the keyword `const` to define a read-only pointer.

Example: `const int a = 5, b = 10, *p = &a;`

This statement defines the constants `a` and `b`, and a pointer `p` to a constant object of type `int`. The referenced object `*p` can be read but not modified.

Example: `cout << *p; // To read is ok.`
`*p = 1; // Error!`

The pointer itself is not a constant, so it can be modified:

Example: `p = &b; // ok!`

The referenced object also does not need to be a constant. In other words, a read-only pointer can also point to a non-constant object.

Example: `Account depo("Twain, Mark", 1234, 4321.90);`
`const Account* ptr = &depo; // ok!`
`ptr->display(); // ok!`
`ptr->setState( 7777.70); // Error!`

But `ptr` can only be used for read access to the non-constant object `depo`.

□ Read-Only Pointers as Parameters

Read-only pointers are most commonly found in parameter lists. This guarantees that arguments cannot be modified.

Example: `int strlen( const char *s);`

In this example, the parameter `s` is a read-only pointer. This allows you to pass constant C strings to the standard function `strlen()`. You cannot remove the “write protection” by assigning the read-only pointer `s` to a normal pointer.

Example: `char *temp = s; // Error!`

You need to declare a read-only pointer if a constant object may be passed as an argument.

■ RETURNING POINTERS

Sample program

```
// search1.cpp
// A filter to output all lines containing a given
// pattern. The function strstr() is called.
// Call:      search1 [ < text.dat ]
// -----
#include <iostream>
using namespace std;
#define MAXL 200           // Maximum length of line
namespace MyScope
{
    // Self-defined version of function strstr():
    char *strstr( const char *str, const char *patt);
}
char line[500],           // For a line of text.
patt[] = "is";           // The search pattern.
int main()
{
    int lineNr = 0; // As long as a line is left over:
    while( cin.getline( line, MAXL))
    {
        ++lineNr;
        if( MyScope::strstr( line, patt) != NULL)
        {
            // If the pattern is found:
            cout.width(3);
            cout << lineNr << ": "    // Output the line
                << line << endl;    // number and the line
        }
    }
    return 0;
}
```

```
// strstr.cpp
// A self-defined version of the function strstr()
// -----
#include <string.h>        // For strlen() and strncmp()
namespace MyScope
{
    char *strstr( const char *s1, const char *s2)
    {
        // To search for the string s2 within s1.
        int len = strlen( s2);
        for( ; *s1 != '\0'; ++s1)
            if( strncmp( s1, s2, len) == 0) // s2 found?
                return (char *)s1;        // Yes -> return pointer
                                           // to this position, or
        return NULL;                      // else the NULL pointer.
    }
}
```

A function can return a pointer to an object. This makes sense for a function that searches for a particular object, for example. Such a function will return either a pointer to the required object or a NULL pointer if the object cannot be found.

The standard C library functions often use pointers as return values. For example, the functions `strcpy()`, `strcat()`, and `strstr()` each return a pointer to the first character in a C string.

□ The Functions `strcpy()` and `strcat()`

In contrast to the example on the page entitled “Pointer versions of functions,” the standard function `strcpy()` has a return value. The function returns its first argument, that is, a pointer to the target string and leads to the following:

Prototype: `char* strcpy( char* s1, const char* s2 );`

The second parameter is a read-only pointer, since the source string is read-only.

The standard function `strcat()` concatenates two C strings, adding the C string passed as the second argument to the first argument. When you call this function, make sure that the char array for the first string is large enough to store both strings. The return value is the first argument. The following example shows one possible implementation.

Example:

```
char *strcat( char *s1, const char *s2 )
{
    char *p = s1 + strlen(s1); // End of s1
    strcpy(p, s2);
    return s1;
}
```

□ Notes on the Sample Program

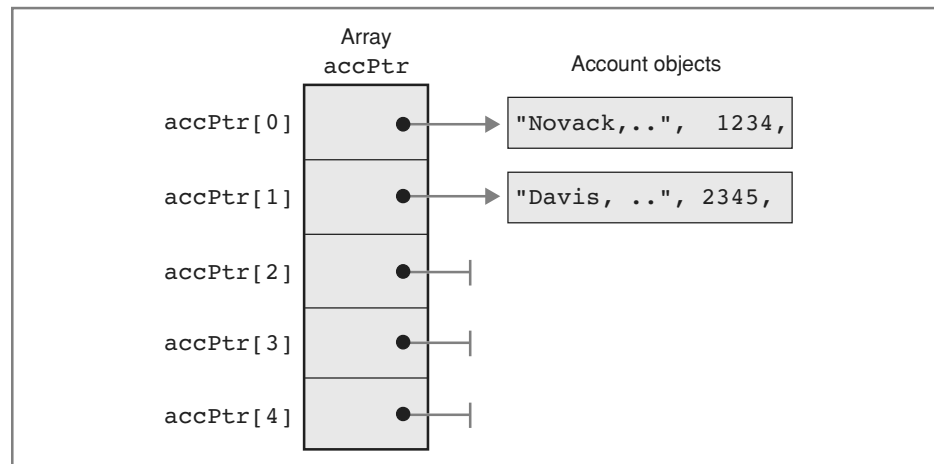
The program on the opposite page shows a self-defined version of the standard function `strstr()`. This version was placed in the `MyScope` namespace to distinguish it from the standard function.

The function `strstr()` searches for a given character sequence within a string. The standard function `strncmp()` is used to compare two strings. This function returns zero if the first *n* characters are identical.

The program uses the `strstr()` function to display all the lines in the text containing the letters “is” with line numbers. The exercises for this chapter contain a program called `search.cpp` where you can supply a search pattern.

■ ARRAYS OF POINTERS

Pointers in the array `accPtr`



Sample function with pointers to `char`

```
// The function displayError() outputs an error message
// to a corresponding error number.
// -----
#include <iostream>
using namespace std;
void displayError ( int errorNr)
{
    static char* errorMsg[] = {
        "Invalid error number",
        "Error 1: Too much data ",
        "Error 2: Not enough memory ",
        "Error 3: No data available " };
    if( errorNr < 1 || errorNr > 3)
        errorNr = 0;
    cerr << errorMsg[errorNr] << endl;
}
```



NOTE

A string literal, such as "Error..." is a `char` pointer to the first character in the string. Thus, such a pointer can be used to initialize another `char` pointer.

Due to its `static` declaration, the array is generated only once and remains valid until the program ends.

Pointers offer various possibilities for simple and efficient handling of large amounts of data. For example, when you are sorting objects it makes sense to define pointers to those objects and simply place the pointers in order, instead of rearranging the actual order of the objects in memory.

□ Defining Arrays of Pointers

Whenever you need a large number of pointers, you can define an array whose elements are pointers. An array of this type is referred to as a *pointer array*.

Example: `Account* accPtr[5];`

The array `accPtr` contains five `Account` pointers `accPtr[0]`, `accPtr[1]`, ... , `accPtr[4]`. The individual pointers in the array can now be assigned object addresses. Any pointers not currently in use should have the value `NULL`.

Example:

```
Account save("Novack, Kim", 1111, 9999.90);
Account depo("Davis, Sammy", 2222, 1000.);
accPtr[0] = &save;
accPtr[1] = &depo;
for( int i=2; i<5; ++i) accPtr[i] = NULL;
```

□ Initialization

As usual, an initialization list is used to initialize the array. In the case of a pointer array, the list contains either valid addresses or the value `NULL`.

Example: `Account* accPtr[5] = { &depo, &save, NULL};`

The value `NULL` is automatically assigned to any objects for which the list does not contain a value. This produces the same result as in the previous example.

□ Usage

The individual objects addressed by the pointers in an array do not need to occupy a contiguous memory space. Normally these objects will be created and possibly destroyed dynamically at runtime (this will be discussed in detail in a later chapter). This allows for extremely flexible object handling. The order is defined only by the pointers.

Example:

```
for( int i=0; i<5; ++i)
    if( accPtr[i] != NULL)
        accPtr[i]->display(); // To output
```

The function `displayError()` opposite displays the error message for a corresponding error number, using an array of `char` pointers to the error messages.

■ COMMAND LINE ARGUMENTS

Sample program

```
// hello.cpp
// Demonstrates the command line arguments.
// Call:  hello name1 name2
// -----
#include <iostream>
using namespace std;

int main( int argc, char *argv[])
{
    if( argc != 3 )
    {
        cerr << "Use: hello name1 name2" << endl;
        return 1;
    }
    cout << "Hello " << argv[1] << '!' << endl;
    cout << "Best wishes \n"
         << "\tyours " << argv[2] << endl;
    return 0;
}
```

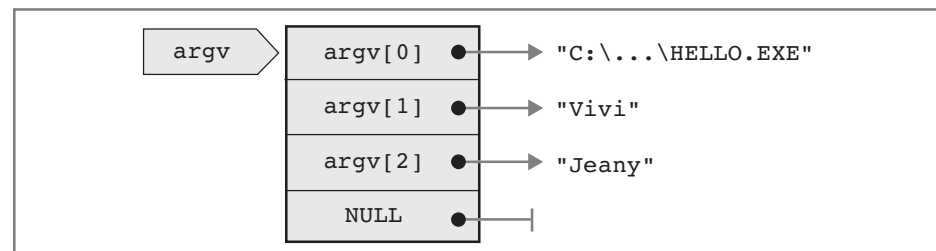
Example of calling the program:

hello Jeany Vivi

Screen output

```
Hello Jeany!
Best wishes
    Yours Vivi
```

Array `argv` in memory



□ Arguments for a Program

When you launch a program, you can use the command line to supply additional character sequences other than the program name. These *command line arguments* are typically used to govern how a program is executed or to supply the data a program will work with.

Example: `copy file1 file2`

In this case, the program `copy` is launched with the arguments `file1` and `file2`. The individual arguments are separated by spaces. Characters used for redirecting input and output (`>` or `<`) and a following word are evaluated by the operating system and not passed to the program. If an argument contains space or redirection characters, you must place it in double quotes.

□ Parameters of the Function `main()`

So far we have only used the function `main()` without parameters. However, if you intend to process command line arguments, you must define parameters for `main()`.

Example:

```
int main(int argc, char * argv[] )
{   . . . // Function block }
```

`argc` contains the number of arguments passed via the command line. The program name is one of these, so `argc` will have a value of at least 1.

The parameter `argv` is an array of char pointers:

<code>argv[0]</code>	points to the program name (and path)
<code>argv[1]</code>	points to the first real argument, that is, the word after the program name
<code>argv[2]</code>	points to the second argument
.....	
<code>argv[argc-1]</code>	points to the last argument
<code>argv[argc]</code>	is the NULL pointer

The parameters are traditionally named `argc` and `argv` although any other name could be used.

Various operating systems, for example WINDOWS 98/00/NT and UNIX, allow you to declare a third parameter for `main()`. This parameter is an array with pointers to environment strings. The exercises for this chapter contain a program that displays the program environment.

■ EXERCISES

For exercise 3**Index version of the standard function `strcmp()`**

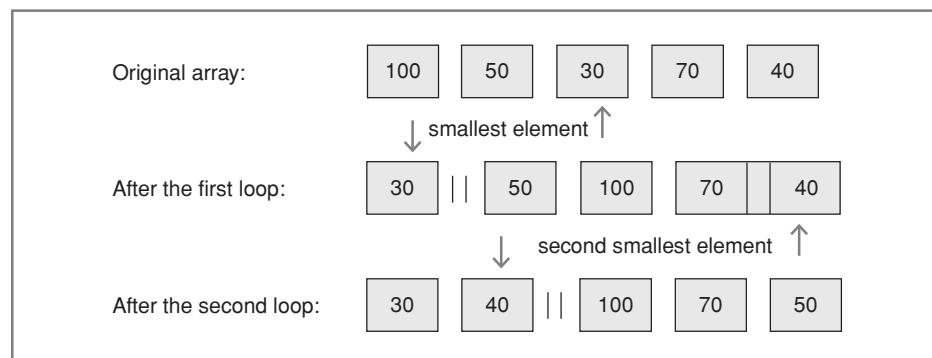
```
// strcmp() compares two C strings lexicographically.
// Return value:      < 0, if str1 < str2
//                  = 0, if str1 == str2
//                  > 0, if str1 > str2 .
// -----

int strcmp( const char str1[], const char str2[])
{
    int i;
    for( i=0; str1[i] == str2[i] && str1[i] != '\0'; ++i)
        ;
    return (str1[i] - str2[i]);
}
```

Notes on exercise 4**The selection sort algorithm****Method**

First find the smallest element in the array and exchange it with the first element.

This procedure is repeated while $i > 0$ for the remainder of an array containing array elements with an initial index of i .

Example

Exercise 1

Given an array `v` with the following definition:

```
int v[] = { 10, 20, 30, 40 }, i, *pv;
```

What screen output is caused by the following statements?

- a.

```
for( pv = v; pv <= v + 3; pv++ )
    cout << "      *pv = " << *pv;
```
- b.

```
for( pv = v, i = 1; i <= 3; i++ )
    cout << "    pv[i] = " << pv[i];
```
- c.

```
for( pv = v, i = 0; pv+i <= &v[3]; pv++, i++)
    cout << " *(pv + i) = " << *(pv + i);
```
- d.

```
for( pv = v + 3; pv >= v; --pv )
    cout << "      v[" << (pv - v) << "] = "
        << v[pv - v];
```

Exercise 2

Write a program that uses the `cin` method `get()` to read a line character by character and stores it in a `char` array. The line is then output in reverse order. Use a pointer, not an index, to address the array elements.

Exercise 3

The standard function `strcmp()` performs a lexicographical comparison of two C strings. The opposite page contains an index version of `strcmp()`. The return value is the difference between two character codes.

Write a pointer version of the function `strcmp()`. Call this function `str_cmp()` to distinguish it from the standard function.

To test the function, use a loop to read two lines of text and output the results of the comparison. The loop should terminate when both strings are empty.

Exercise 4

Define and test the function `selectionSort()` that sorts an array of `int` values in ascending order. The principle of the selection sort algorithm is shown opposite.

Arguments: An `int` array and its length

Return values: None

Develop both an index version and a pointer version. Test the functions with random numbers between -10000 and +10000.

Notes on exercise 5**Sample environment strings for DOS/Windows**

```
. . .  
COMSPEC=C:\COMMAND.COM  
PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\DOS;D:\TOOLS;  
PROMPT=$p$g  
TEMP=C:\TEMP  
. . .
```

Frequency table for exercise 7

Age \ Blood-pressure					
	<120	120–129	130–139	140–149	>= 160
20–29	25	34	26	12	8
30–39	19	27	24	11	4
40–49	6	15	35	36	18

Exercise 5

- a. Write a program that outputs its own name and all command line arguments, each in a separate line.
- b. Now extend the program to output its own environment. The environment is a memory area containing strings in the format

NAME=String

A third parameter for the function `main()` allows access to the environment. This parameter is an array of pointers just like `argv`. The array elements are `char` pointers to the environment strings, the last element being a `NULL` pointer.

Exercise 6

A sample filter program called `search1`, which outputs lines and the relevant line numbers for lines containing the search pattern "ei", was introduced in this chapter.

Modify the program to produce a useful tool called `search`, to which you can pass any search pattern via the command line. The program should issue an error message and terminate if the command line does not contain a search string. Use the standard function `strstr()`.

Sample call:

```
search Shanghai < news.txt
```

Exercise 7

The following frequency was observed during an examination of the relationship between age and blood pressure for 300 males.

Write a function that calculates the sums of the rows and columns in an `int` matrix with three rows and five columns. Store the sums of the rows and columns separately in a one-dimensional row or column array.

Arguments: The matrix, the row array, and the column array.

Return value: The sum of all the matrix elements.

To test the function, output the matrix, as shown in the graphic opposite along with the computed sums in your `main` function.

■ SOLUTIONS

Exercise 1

Screen Output:

```

a.      *pv = 10      *pv = 20      *pv = 30      *pv = 40
b.      pv[i] = 20    pv[i] = 30    pv[i] = 40
c.      *(pv+i) = 10  *(pv+i) = 30
d.      v[3] = 40     v[2] = 30     v[1] = 20     v[0] = 10

```

Exercise 2

```

// -----
// reverse.cpp
// Exercise on pointer arithmetic:
// Reads a line and outputs the line in reverse order.
// -----
#include <iostream>
using namespace std;

#define MAXLEN 80

int main()
{
    char line[MAXLEN], *p;

    cout << "Enter a line of text: " << endl;

    // Input a line:
    for( p = line;
        p < line+MAXLEN && cin.get(*p) && *p != '\n';
        ++p )
        ;

    // Output the line in reverse order:
    while( --p >= line)
        cout << *p;

    cout << endl;

    return 0;
}

```

Exercise 3

```
// -----
// str_cmp.cpp
// Define and test the pointer version str_cmp()
// of the standard function strcmp().
// -----
#include <iostream>
using namespace std;
#define MAXLEN 100          // Maximum length of C strings
// Prototype:
int str_cmp( const char* str1, const char* str2);
int main()                  // Test str_cmp()
{
    char text1[MAXLEN], text2 [MAXLEN];
    cout << "Testing the function str_cmp()" << endl;
    while( true)
    {
        cout << "Enter two lines of text!\n"
                "End with two empty lines.\n" << endl;
        cout << "1. line: ";
        cin.sync(); cin.clear(); cin.get(text1,MAXLEN);
        cout << "2. line: ";
        cin.sync(); cin.clear(); cin.get(text2,MAXLEN);
        if( text1[0] == '\0' && text2[0] == '\0')
            break;                // Both lines empty.
        int cmp = str_cmp( text1, text2);
        if( cmp < 0)
            cout << "The 1st string is smaller!\n";
        else if( cmp == 0)
            cout << "Both strings are equal!\n";
        else
            cout << "The 1st string is greater!\n";
        cout << endl;
    }
    return 0;
}
// -----
// Function str_cmp()
// Pointer version of the standard function strcmp().
// -----
int str_cmp( const char* str1, const char* str2)
{
    for( ; *str1 == *str2 && *str1 != '\0'; ++str1, ++str2)
        ;
    return (*str1 - *str2);
}
```

Exercise 4

```
// -----  
// selSort.cpp  
// Implement the selection sort algorithm  
// for int-arrays.  
// -----  
  
#include <iostream>  
#include <iomanip>  
#include <cstdlib>           // For srand(), rand()  
#include <ctime>             // For time()  
using namespace std;  
  
// Prototype:  
void selectionSort( int arr[], int len);  
  
const int len = 200;  
int intArr[len];           // int-array  
  
int main()  
{  
    cout << "\n      *** Selection Sort Algorithm ***\n"  
        << endl;  
  
    // To initialize an int-array with random numbers:  
    srand( (unsigned int)time(NULL)); // Initialize the  
                                       // random number generator.  
    for( int n=0; n < len; ++n)  
        intArr[n] = (rand() % 20000)-10000;  
  
    // To sort the numbers  
    selectionSort( intArr, len);  
  
    // To output the numbers  
    cout << "The sorted numbers:" << endl;  
  
    for( int i = 0; i < len; ++i)  
        cout << setw(8) << intArr[i];  
    cout << endl;  
  
    return 0;  
}  
  
inline void swap( int& a, int& b)  
{  
    int temp = a;  a = b;  b = temp;  
}
```

```

// Index version:
/*
void selectionSort( int arr[], int len)
{
    register int j, mini;          // Indices

    for( int i = 0; i < len-1; ++i)
    {
        mini = i;                  // Search for minimum
        for( j = i+1; j < len; ++j) // starting with index i.
            if( arr[mini] > arr[j])
                mini = j;

        swap( arr[i], arr[mini]);  // Swap.
    }
}
*/

// Pointer version:
void selectionSort( int *arr, int len)
{
    register int *p, *minp;        // Pointer to array elements,
    int *last = arr + len-1;      // pointer to the last element

    for( ; arr < last; ++arr)
    {
        minp = arr;                // Search for minimum
        for( p = arr+1; p <= last; ++p) // starting with arr
            if( *minp > *p)
                minp = p;

        swap( *arr, *minp);        // Swap.
    }
}

```

Exercise 5

```
// -----
// args.cpp
// The program outputs the program name including the path,
// command line arguments and the environment.
// -----

#include <iostream>
using namespace std;

int main( int argc, char *argv[], char *env[])
{
    cout << "Program: " << argv[0] << endl;

    cout << "\nCommand line arguments:" << endl;

    int i;
    for( i = 1; i < argc; ++i)           // Arguments
        cout << argv[i] << endl;

    cout << "Type <Return> to go on";
    cin.get();

    cout << "\nEnvironment strings:" << endl;

    for( i = 0; env[i] != NULL; ++i)     // Environment
        cout << env[i] << endl;

    return 0;
}
```

Exercise 6

```
// -----
// search.cpp
// A filter that outputs all lines containing a certain
// pattern. The standard function strstr() is called.
//
// Call:      search pattern [ < text.dat ]
//
// If no file name is passed the input is read from the
// keyboard. In this case end input with <Ctrl> + <Z>.
// -----

#include <iostream>
#include <cstring>           // Standard functions for C strings
using namespace std;
#define MAXL 200           // Maximum length of line
char line[500];           // For a line of text.
```

```

int main( int argc, char *argv[])
{
    if( argc != 2)
    {
        cerr << "Call:  search  pattern [ < text.dat ]"
              << endl;
        return 1;
    }

    int lineNr = 0;
                                // As long as a line exists:
    while( cin.getline( line, MAXL))
    {
        ++lineNr;
        if( strstr( line, argv[1]) != NULL)
        {
                                // If the pattern was found:
            cout.width(3);
            cout << lineNr << ": "           // Output the line
                << line << endl;           // number and the line
        }
    }
    return 0;
}

```

Exercise 7

```

// -----
// matrix.cpp
// To compute the sums of rows and columns in a matrix.
// -----

#include <iostream>
#include <iomanip>
using namespace std;

// Define and initiate a two-dimensional array:

int matrix[3][5] = { { 25, 34, 26, 12,  8 },
                    { 19, 27, 24, 11,  4 },
                    {  6, 15, 35, 36, 18 } };

int rowsum[3];           // For the sums of the rows
int colsum[5];           // For the sums of the columns

// Prototype of function matrixsum():
int matrixsum( int arr2D[][5], int vlen,
               int rsum[], int csum[]);

```

```

int main()
{
    cout << "Testing the function matrixsum().\n"
          << endl;

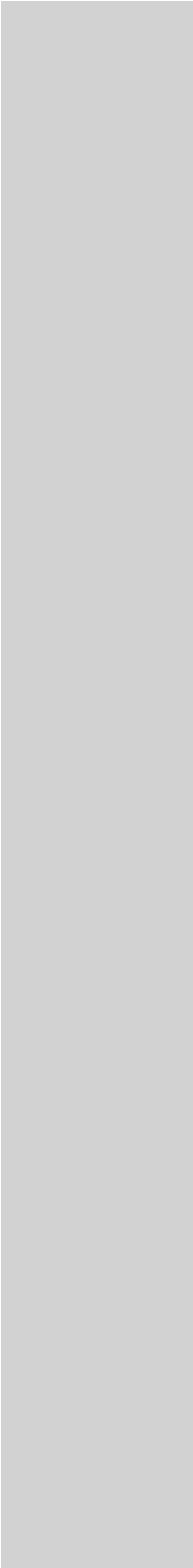
    // Compute sums:
    int totalsum =
        matrixsum( matrix, 3, rowsum, colsum);

    // Output matrix and sums:
    cout << "The matrix with the sums "
          << "of rows and columns:\n"
          << endl;

    int i,j;
    for( i = 0 ; i < 3 ; ++i)      // Output rows of the
    {                               // matrix with row sums.
        for( j = 0 ; j < 5 ; ++j)
            cout << setw(8) << matrix[i][j];
        cout << " | " << setw(8) << rowsum[i] << endl;
    }
    cout << " -----"
          << endl;
    for( j = 0 ; j < 5 ; ++j )
        cout << setw(8) << colsum[j];
    cout << " | " << setw(8) << totalsum << endl;
    return 0;
}

// -----
int matrixsum( int v[][5], int len,
               int rsum[], int csum[])
{
    int ro, co;                               // Row and column index
    for( ro = 0 ; ro < len ; ++ro)           // To compute row sums
    {
        rsum[ro] = 0;
        for( co = 0 ; co < 5 ; ++co)
            rsum[ro] += v[ro][co];
    }
    for(co = 0 ; co < 5 ; ++co)               // Compute column sums
    {
        csum[co] = 0;
        for( ro = 0 ; ro < len ; ++ro)
            csum[co] += v[ro][co];
    }
    return (rsum[0] + rsum[1] + rsum[2]); // Total sum =
                                          // sum of row sums.
}

```



chapter

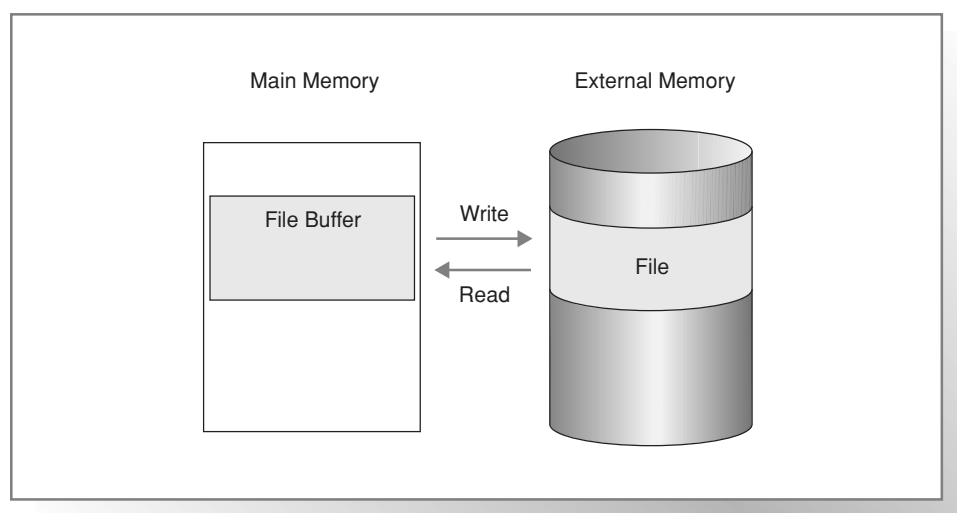
18

Fundamentals of File Input and Output

This chapter describes sequential file access using file streams. File streams provide simple and portable file handling techniques.

■ FILES

File operations



When a program is terminated, the program data stored in main memory is lost. To store data permanently, you need to write that data to a file on an external storage medium.

□ File Operations

Single characters or character strings can be written to text files just like they can be output on screen. However, it is common practice to store records in files. A record contains data that forms a logical unit, such as the human resource information for a person. A *write operation* stores a record in a file, that is, the existing record in the file is updated or a new record is added. When you *read* a record, this record is taken from the file and copied to the data structure of a program.

Objects can be put into permanent storage using similar techniques. However, this normally involves more than just storing an object's data. You also need to ensure that the object can be correctly reconstructed when it is read, and this in turn involves storing type information and references to other objects.

External mass storage media, such as hard disks, are normally block-oriented—that is, data is transferred in blocks whose size is a multiple of 512 bytes. Efficient and easy file management thus implies putting the data you need to store into temporary storage in main memory, in a so-called *file buffer*.

□ File Positions

From the viewpoint of a C++ program, a file is simply a long byte array. The structure of the file, using records for example, is entirely the programmer's responsibility, allowing for a maximum degree of flexibility.

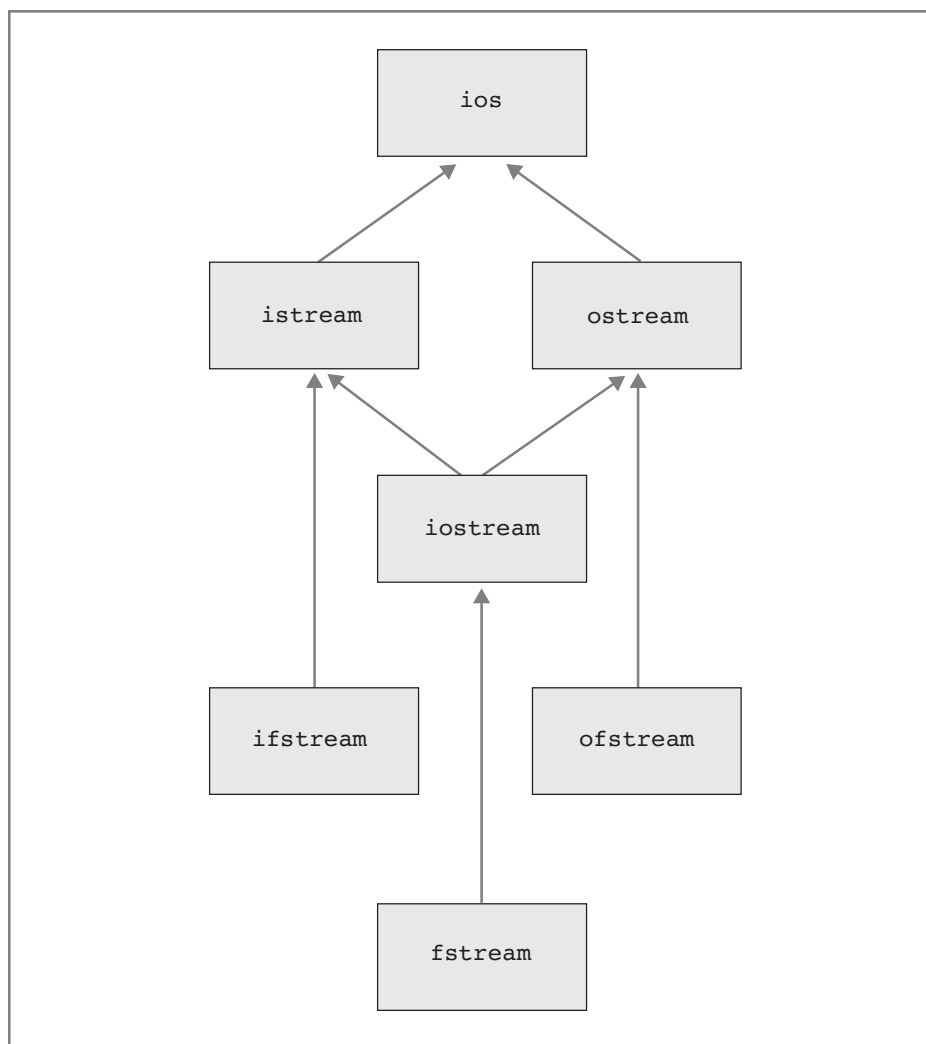
Every character in a file occupies a byte position. The first byte occupies position 0, the second byte position 1, and so on. The *current file position* is the position of the byte that will be read or written next. Each byte that is transferred automatically increases the current file position by 1.

In the case of *sequential access*, the data is read or written byte by byte in a fixed order. The first read operation starts at the beginning of the file. If you need access to some piece of information in a file, you must read the file content from start to finish. Write operations can create a new file, overwrite an existing file, or append new data to an existing file.

Easy access to given data in a file implies being able to set the current file position as required. This technique is known as *random file access* and will be discussed in one of the following chapters.

■ FILE STREAMS

Stream classes for file access



C++ provides various standard classes for file management. These so-called *file stream classes* allow for easy file handling. As a programmer you will not need to concern yourself with file buffer management or system specifics.

Since the file stream classes have been standardized, you can use them to develop portable C++ programs. One program can thus process files on a Windows NT or UNIX platform. You simply need to recompile the program for each platform you use.

□ The File Stream Classes in the `iostream` Library

The class hierarchy on the opposite page shows that the file stream classes contain the stream classes, with which you are already familiar, as *base classes*:

- the `ifstream` class derives from the `istream` class and allows file reading
- the `ofstream` class derives from the `ostream` stream class and supports writing to files
- the `fstream` class derives from the `iostream` stream class. As you would expect, it supports both read and write operations for files.

The file stream classes are declared in the `fstream` header file. An object that belongs to a file stream class is known as a *file stream*.

□ Functionality

The file stream classes inherit the functionality of their base classes. Thus, the methods, operators, and manipulators you have already used for `cin` and `cout` are also available here. Thus every file stream has:

- methods for non-formatted writing and reading of single characters and/or data blocks
- the operators `<<` or `>>` for formatted reading and writing from or to files
- methods and manipulators for formatting character sequences
- methods for state queries.

File handling methods, particularly methods for opening and closing files, round off the package.

■ CREATING FILE STREAMS

Sample program

```

// showfile.cpp
// Reads a text file and outputs it in pages,
// i.e. 20 lines per page.
// Call: showfile filename
// -----
#include <iostream>
#include <fstream>
using namespace std;

int main( int argc, char *argv[])
{
    if( argc != 2 )                // File declared?
    {
        cerr << "Use: showfile filename" << endl;
        return 1;
    }
    ifstream file( argv[1]);       // Create a file stream
                                   // and open for reading.
    if( !file )                   // Get status.
    {
        cerr << "An error occurred when opening the file "
              << argv[1] << endl;
        return 2;
    }

    char line[80];
    int cnt = 0;
    while( file.getline( line, 80)) // Copy the file
    {                                // to standard
        cout << line << endl;        // output.
        if( ++cnt == 20)
        {
            cnt = 0;
            cout << "\n\t ---- <return> to continue ---- "
                  << endl;
            cin.sync(); cin.get();
        }
    }
    if( !file.eof() )              // End-of-file occurred?
    {
        cerr << "Error reading the file "
              << argv[1] << endl;
        return 3;
    }
    return 0;
}

```

□ Opening a File

You need to open a file before you can manipulate it. To do so, you can

- state the *file name*, which can also contain a path
- define a so-called *file access mode*.

If the path is not explicitly stated, the file must be in the current directory. The file access mode specifically defines whether read and/or write access to the file is permitted.

Any files still open when a program terminates are automatically closed.

□ File Stream Definition

You can open a file when you create a file stream—you simply state the file name to do so. In this case default values are used for the file access mode.

Example: `ifstream myfile("test.file");`

The file name `test.file` is passed to the constructor of the `ifstream` class, which opens the file for reading. Since the path was not stated, the file must be in the current directory. When a file is opened, the current file position is the beginning of the file.

If you create a file stream for write-only access, the file you state need not exist. In this case a new file is created.

Example: `ofstream yourfile("new.file");`

This statement creates a new file called `new.file` and opens the file for writing. But be careful! If the file already exists, it will be truncated to a length of zero bytes, or in other words deleted.

You can create a file stream which does not reference a specific file and use the `open()` method to open a file later.

Example: `ofstream yourfile;
yourfile.open("new.file");`

This example has the same effect as the previous example. More specifically, `open()` uses the same default values for file access when opening a file as the default constructor for the class.

It rarely makes sense to use fixed file names. In the case of the sample program on the opposite page, you state the file name in the command line when you launch the program. If no file name is supplied, the program issues an error message and terminates. Using interactive user input is another possible way to define a file name.

■ OPEN MODES

Flags for the open mode of a file

Flag	Effects
<code>ios::in</code>	Opens an existing file for input.
<code>ios::out</code>	Opens a file for output. This flag implies <code>ios::trunc</code> if it is not combined with one of the flags <code>ios::in</code> or <code>ios::app</code> or <code>ios::ate</code> .
<code>ios::app</code>	Opens a file for output at the end-of-file.
<code>ios::trunc</code>	An existing file is truncated to zero length.
<code>ios::ate</code>	Open and seek to end immediately after opening. Without this flag, the starting position after opening is always at the beginning of the file.
<code>ios::binary</code>	Perform input and output in binary mode.

✓ NOTE

1. These flags are defined in the baseclass `ios`, which is common to all stream classes, and are of the `ios::openmode` type.
2. By default a file is opened as a *text file* in so-called text mode. When you read from or write to a text file, control characters to indicate newlines or the end-of-file are interpreted separately and adapted to the current platform (so-called “cooked mode”). When a file is opened in binary mode, the file contents are left unchanged (the so called “raw mode”).

Default settings when opening a file

The constructor and the method `open()` of all stream classes use the following default values:

Class	Flags
<code>ifstream</code>	<code>ios::in</code>
<code>ofstream</code>	<code>ios::out</code> <code>ios::trunc</code>
<code>fstream</code>	<code>ios::in</code> <code>ios::out</code>

To open a file in any but the default mode, you must supply both the file name and the open mode. This is necessary, for example, to open an existing file for write access without deleting the file.

□ Open Mode Flags

In addition to the file name, you can pass a second argument for the open mode to the constructors and the `open()` method. The open mode is determined by using flags. A *flag* represents a single bit in a computer word. If the flag is raised, the bit in question will contain the value 1, with 0 representing all other cases.

You can use the bit operator, `|`, to combine various flags. Either the flag `ios::in` or `ios::out` must be stated in all cases. If the flag `ios::in` is raised, the file must already exist. If the flag `ios::in` is not used, the file is created, if it does not already exist.

Example: `fstream addresses("Address.file", ios::out | ios::app);`

This opens a file for writing at end-of-file. The file is created, if it does not already exist. The file will automatically grow after every write operation.

You can use the default mode for the `fstream` class, that is, `ios::in | ios::out`, to open an existing file for reading and writing. This so-called update mode is used for updating the information in a file and is often seen in conjunction with random file access.

□ Error Handling

Errors can occur when opening a file. A user may not have the required access privileges, or the file you want to read may not exist. The state flag `failbit` of the `ios` base class is raised in this case. The flag can either be queried directly using the `fail()` method, or indirectly by querying the status of a file stream in an `if` condition.

Example: `if( !myfile) // or: if( myfile.fail())`

The `fail` bit is also set if a read or write error occurs. If a read operation fails, the end of the current file may have been reached. To distinguish this normal behavior from a read error, you can use the `eof()` method (`eof` = end-of-file) to query the `eof` bit:

Example: `if( myfile.eof()) // At end-of-file?`

The `eof` bit is set if you try to carry on reading at the end of a file. The sample program on the previous page illustrates the potential issues.

■ CLOSING FILES

Sample program

```
// fcopy1.cpp : Copies files.
// Call: fcopy1 source [ destination ]
// -----
#include <iostream>
#include <fstream>
using namespace std;
inline void openererror( const char *file)
{
    cerr << "Error on opening the file " << file << endl;
    exit(1); // Ends program closing
} // all opened files.
void copy( istream& is, ostream& os); // Prototype

int main(int argc, char *argv[])
{
    if( argc < 2 || argc > 3)
    { cerr << "Call: fcopy1 source [ destination ]"
      << endl;
      return 1; // or: exit(1);
    }
    ifstream infile(argv[1]); // Open 1st file
    if( !infile.is_open())
        openererror( argv[1]);

    if( argc == 2) // Just one sourcefile.
        copy( infile, cout);
    else // Source and destination
    {
        ofstream outfile(argv[2]); // Open 2nd file
        if( !outfile.is_open() )
            openererror( argv[2]);
        copy( infile, outfile);
        outfile.close(); // Unnecessary.
    }
    infile.close(); // Unnecessary.
    return 0;
}

void copy( istream& is, ostream& os) // Copy it to os.
{
    char c;
    while( is.get(c) )
        os.put(c); // or: os << c ;
}
```

□ Motivation

After you have completed file manipulation, the file should always be closed for the following reasons:

- data may be lost, if for some reason the program is not terminated correctly
- there is a limit to the number of files that a program can open simultaneously.

A program that terminates correctly will automatically close any open files before exiting. A file stream destructor will also close a file referenced by a stream. However, if the file is no longer in use before this point, you should close the file explicitly.

□ Methods `close()` and `is_open()`

Each of the file stream classes contains a definition of a `void` type method called `close()`, which is used to close the file belonging to the stream.

Example: `myfile.close();`

However, the file stream continues to exist. It is therefore possible to use the stream to open and manipulate another file.

If you are not sure whether a file stream is currently accessing a file, you can always perform a test using the `is_open()` method. In the case of the `myfile` file stream, the test is as follows:

Example:

```
if( myfile.is_open() )
{ /* . . . */ }           // File is open
```

□ The `exit()` Function

Open files are also closed when you call the global function `exit()`. The actual reason for using this function is to terminate a program in an orderly manner and return an error code to the calling process.

Prototype: `void exit( int status );`

The calling process, to which the `status` error code is passed for evaluation, will often be the command interpreter—a Unix shell, for example. Successful program execution normally produces the error code 0. The statement `return n;` is thus equivalent to the statement `exit(n);` when used in the `main()` function.

The program on the opposite page copies a file stated in the command line. If the user forgets to state a second (target) file, the source file is copied to standard output. In this case, the source file will need to be a text file.

■ READING AND WRITING BLOCKS

Sample program

```

// Pizza_W.cpp
// Demonstrating output of records block by block.
// -----
#include <iostream>
#include <fstream>
using namespace std;

char header[] =
"    * * *   P I Z Z A   P R O N T O   * * *\n\n";
// Record structure:
struct Pizza { char name[32]; float price; };
const int MAXCNT = 10;
Pizza pizzaMenu[MAXCNT] =
{
    { "Pepperoni", 9.90F },    { "White Pizza", 15.90F },
    { "Ham Pizza", 12.50F },  { "Calzone", 14.90F } };
int cnt = 4;
char pizzaFile[256] = "pizza.file";

int main()                                // To write records.
{
    cout << header << endl;

    // To write data into the file:
    int exitCode = 0;
    ofstream outFile( pizzaFile, ios::out|ios::binary );
    if( !outFile)
    {
        cerr << "Error opening the file!" << endl;
        exitCode = 1;
    }
    else
    {
        for( int i = 0; i < cnt; ++i)
            if( !outFile.write( (char*)&pizzaMenu[i],
                                sizeof(Pizza)) )
            {
                cerr << "Error writing!" << endl;
                exitCode = 2;
            }
    }
    if( exitCode == 0)
        cout << "\nData has been added to file "
              << pizzaFile << "\n" << endl;
    return exitCode;
}

```

The file stream classes can use all the `public` operations defined in their base classes. This means you can write formatted or unformatted data to a file or read that data from the file block by block or character by character.

□ Formatted and Unformatted Input and Output

The previous sample programs illustrated how to use the methods `get()`, `getline()`, and `put()` to read or write data from or to text files. Formatted input and output of numerical values, for example, requires the `>>` and `<<` operators and appropriate manipulators or formatting methods.

Example:

```
double price = 12.34;
ofstream textFile("Test.txt");
textFile << "Price: " << price << "Dollar" << endl;
```

The file `Test.txt` will contain a line of text, such as `"Price ... "` that exactly matches the screen output.

Converting binary data to legible text is not practicable if you are dealing with large amounts of data. It makes sense to write the data for a series of measurements to a binary file in the order in which they occur in the program. To do so, you simply open the file in binary mode and write the data to the file, or read it from the file, block by block.

□ Transferring Data Blocks

The `ostream` method `write()` transfers given number of bytes from main memory to a file.

Prototype: `ostream& write( const char *buf, int n);`

Since `write()` returns a reference to the stream, you can check to ensure that the write operation was successful.

Example:

```
if( ! fileStream.write("An example ", 2) )
    cerr << "Error in writing!" << endl;
```

A warning is issued if an error occurs while writing the characters `"An"`. You can use the `read()` method to read data blocks from the file. The method transfers a data block from a file to a program buffer.

Prototype: `istream& read( char *buf, int n);`

The methods `read()` and `write()` are often used for files with fixed length records. The block that needs to be transferred can contain one or more records. The buffer in main memory is either a structure variable or an array with elements belonging to the structure type. You need to cast the address of this memory area to `(char *)` as shown in the example opposite.

■ OBJECT PERSISTENCE

Class Account

```
// Class Account with methods read() and write()
// -----
class Account
{
    private:
        string name;           // Account holder
        unsigned long nr;      // Account number
        double balance;        // Balance of account
    public:
        . . . // Constructors, destructor,
              // access methods, ...
        ostream& Account::write(ostream& os) const;
        istream& Account::read(istream& is)
};
```

Implementing methods read() and write()

```
// write() outputs an account into the given stream os.
// Returns: The given stream.
ostream& Account::write(ostream& os) const
{
    os << name << '\0';           // To write a string
    os.write((char*)&nr, sizeof(nr) );
    os.write((char*)&balance, sizeof(balance) );
    return os;
}

// read() is the opposite function of write().
// read() inputs an account from the stream is
// and writes it into the members of the current object

istream& Account::read(istream& is)
{
    getline( is, name, '\0'); // Read a string
    is.read( (char*)&nr, sizeof(nr) );
    is.read( (char*)&balance, sizeof(balance));
    return is;
}
```

□ A Scenario

Imagine you want to save the data for various account types, including current and savings accounts to a file. Since the objects you need to save belong to different types, you must save both the data members and the type of object. This is the only way to ensure that an object will be correctly reconstructed when read.

The methods in the class you are going to define should throw exceptions if errors occur. The exception type thrown by a method is stated in the exception specification.

□ The `AccFile` Class

The `AccFile` class, which is used for random access to a file with account data, is defined opposited. The data members are an `fstream` type file stream and a string used for storing the file name.

The constructor saves the file name and opens a given file for reading and appending at end-of-file. If the file cannot be opened, the constructor throws an `OpenError` class exception.

The `append()` method writes an account passed to it as an argument at end-of-file and returns the position where the account was written into the file.

In order to get the current type of the argument, the virtual method `getTypeid()` is called. Depending on this type the `append()` method will write the appropriate type field to the file and then call the virtual method `write()`, allowing the current object to write itself to the file. If a write error occurs, the method will throw a `WriteError` type exception.

The `retrieve()` method first reads the type identifier and then determines the type of object it needs to allocate memory for. The data from the file is then transferred to the dynamically allocated object by a call to the virtual method `read()`. Here too, an exception will be thrown if a stream access error occurs.

■ APPLICATION: INDEX FILES

The insert() method of class IndexFile

```

// index.cpp: (continued)
// Implements the methods of class IndexFile.
// -----
// . . .
void IndexFile::insert(long k, long n)
                        throw(ReadError, WriteError)
{
    IndexEntry entry;
    int size = entry.recordSize();    // Length of an
                                    // index entry.

    index.clear();
    index.seekg(0, ios::end);
    long nr = index.tellg();          // Get file length
                                    // 0, if file is empty.
    if(!index) throw ReadError(name);

    nr -= size;                       // Last entry.
    bool found = false;
    while( nr >= 0 && !found )        // Search for position
    {                                // to insert
        if( !entry.read_at(index, nr))
            throw ReadError(name);

        if( k < entry.getKey() )      // To shift.
        {
            entry.write_at(index, nr + size);
            nr -= size;
        }
        else
        {
            found = true;
        }
    }

    entry.setKey(k); entry.setPos(n); // insert
    entry.write_at(index, nr + size);

    if(!index)
        throw WriteError(name);
}

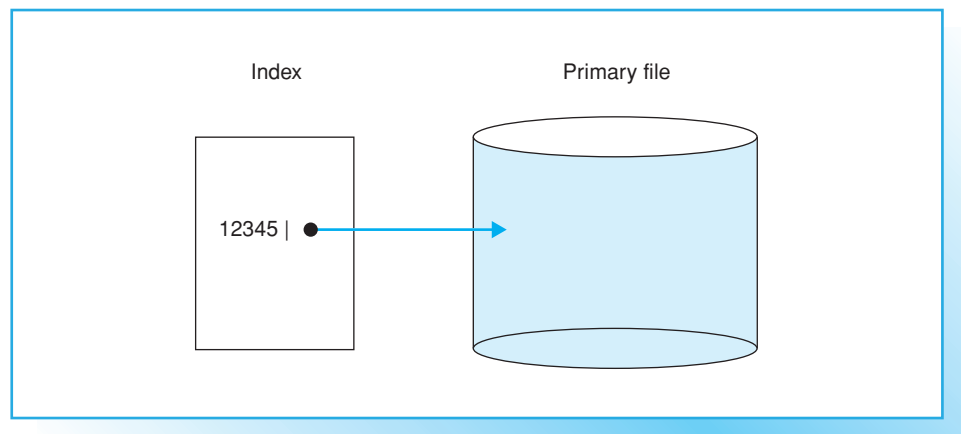
```

It makes sense to organize data sequentially in files if you need to walk through all the records regularly. This is particularly true for files used to store salary data or phone bills.

However, most applications need to provide quick access to specific data. For example, a user would definitely prefer to be able to locate an account quickly by reference to an account number, rather than searching through a file from top to bottom. Index files can mean a real performance boost in cases like this.

□ Index Files

An *index file* comprises a so-called *primary file* containing the live data, and an index. The *index* consists of pairs of keys and record positions for the primary file. A key stored in the index will identify a record in the primary file. This situation can be more easily explained by the following graphic:



The index is sorted by reference to the keys for speed of access, allowing you to perform a binary search to locate the position of a record.

□ Inserting into the Index

We can use the `IndexFile` class definition to represent an index. The `insert()` method, which correctly inserts a new record into the sorted index, is defined opposite.

The read pointer is first set to end-of-file for insertions. If the current position is 0, that is, the file is empty, the entry is inserted at offset 0. In all other cases, any entries whose keys are greater than the new key are shifted down to make room for the new entry.

■ IMPLEMENTING AN INDEX FILE

Representing the index file

```
// index.h:   Defines the class IndexFile
// -----
class IndexFileSystem : public AccFile, public IndexFile
{
    private:
        string name;
    public:
        IndexFileSystem(const string s)
            : AccFile(s + ".prim"), IndexFile(s + ".ind")
            { name = s; }

        void    insert    ( Account& acc );
        Account* retrieve( long key );
};
```

Methods insert() and retrieve() of class IndexFileSystem

```
// index.cpp:   Implementing the methods.
// -----
void IndexFileSystem::insert(Account& acc)
{
    // No multiple entries:
    if(search(acc.getNr()) == -1)
    {
        long pos = append(acc); // Insert in primary file
        if(pos != -1)           // Insert in
            IndexFile::insert(acc.getNr(), pos); // index file.
    }
}

Account* IndexFileSystem::retrieve(long key )
{
    long pos = search(key); // Get the record address:
    if( pos == -1 )         // Account number found?
        return NULL;
    else
    {
        IndexEntry entry; // Read the index entry:
        IndexFile::retrieve(entry, pos);
        // Read from primary file:
        return( AccFile::retrieve( entry.getPos() ) );
    }
}
```

□ Index File for Account Management

Since an index file consists of a primary file and an index, it makes sense to derive the class used to represent an index file from the classes of the primary file and the index file. Let's now look at a sample index file, used for managing bank accounts.

The `IndexFileSystem` class, which is derived from the two previously defined classes `AccFile` and `IndexFile`, is defined on the opposite page. The only data member is a string for the file name. The constructor expects a file name as an argument and composes names for the primary file and the index by adding a suitable suffix. Base initializers are then used to open the corresponding files.

It is not necessary to define a destructor, since files are automatically closed when the base class destructors are called.

□ Inserting and Retrieving Records

The `insert()` method was defined to insert new records. It first calls the `search()` method to check whether the account number already exists in the index. If not, a new record is appended to the end of the primary file using the `append()` method. Then the key and the address of the record are inserted into the index.

The `IndexFileSystem` class also contains the `retrieve()` method, which is used to retrieve records from the primary file. The key, `key`, which is passed to the method, is used by the `search()` method to look up the address of the required record in the index. Then the record is retrieved from the primary file by the `AccFile` class `retrieve()` method.

Only the `retrieve()` methods for the `IndexFile` and `AccFile` classes and the `search()` method, which performs a binary search in the index, are needed to complete the index file implementation. It's your job to implement these three methods as your next exercise!

Using a sorted file to implement an index has the disadvantage that records need to be shifted to make room to insert new records. As shifting is time-consuming, an index is normally represented by a tree, which needs less reorganization.

■ EXERCISES

Exercise I**Class IndexFile**

```

class IndexFile
{
    private:
        fstream index;
        string name;           // Filename of the index

    public:
        IndexFile(const string s) throw(OpenError);
        ~IndexFile() { index.close(); }
        void insert( long key, long pos)
                                throw(ReadError, WriteError);
        long search( long key) throw(ReadError);
        void retrieve(IndexEntry& entry, long pos )
                                throw( ReadError);
};

```

Class AccFile

```

enum TypeId { ACCOUNT, DEPOSIT, SAVINGS };
class AccFile
{
    private:
        fstream f;
        string name;           // Filename of primary file
    public:
        AccFile(const string s) throw(OpenError);
        ~AccFile() { f.close(); }
        long append( Account& acc) throw(WriteError);
        Account* retrieve( long pos ) throw(ReadError);
};

```

Exercise I

Complete and test the implementation of the `IndexFileSystem` class. The methods should throw exceptions of an appropriate `FileError` type if an error occurs.

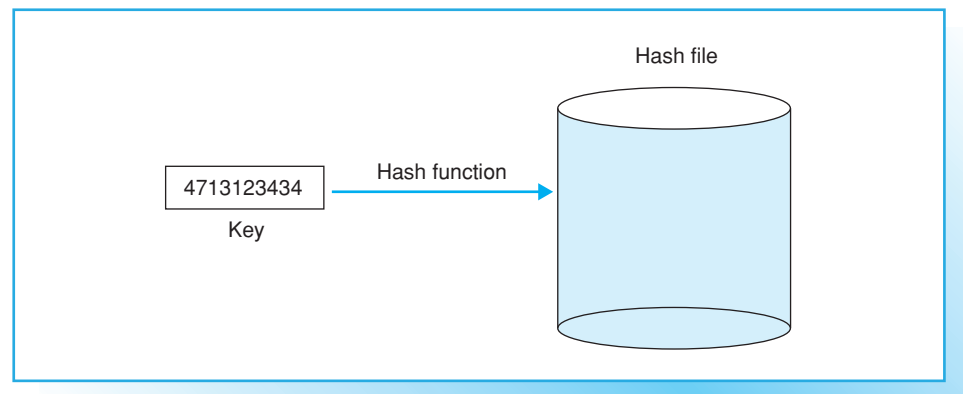
- a. Complete the constructor of the `IndexFile` class in order to throw an exception of the type `OpenError` if the file can not be opened.
- b. Write the `retrieve()` method for the `IndexFile` class. The method retrieves a record at a given position in the index.
- c. Define the `search()` method, which looks up an index entry for an account number passed to it as an argument. Base the method on the binary search algorithm.

Return value: The position of the record found, or -1 if the account number is not found in the index.

- d. Then define the `retrieve()` method for the `AccFile` class, which first evaluates the `type` field at a given position in the account file, then dynamically allocates memory for an object of the appropriate type, and finally reads the data for an account from the file.
- e. Write a `main()` function that uses a `try` block to create an `IndexFileSystem` type object and to insert several accounts of various types into the index file. The subsequent user dialog reads a key and displays the corresponding record on screen. Write an exception handler to handle the various errors that could occur. The name of the file and the cause of the error must be output in any case of error.

Hash Files

Hash Files profit from random file access in localizing file records directly. Each file record must have the same length and it must be identified by a unique key, the so-called *hash key*.



The idea behind hashing is to provide a function, the *hash function*, which is applied to the hash key of a record and yields the address of the file record. If the file records are numerated and if the hash key equals the record number, a simple hash function can be the identical function. However, hash keys, such as account or insurance numbers, consist of a fixed number of digits that do not start with 0.

The following example shows a frequently used hash function

Example: $\text{Hash}(\text{key}) = \text{key} \% b;$

This hash function transforms the hash value *key* into a record number between 0 and $b-1$. The number 0 is the first record number and $b-1$ is the last record number in the *address space* of the hash file. For a sufficiently large prime number b , the function $\text{Hash}()$ yields a good distribution of file records in the address space.

However, the hash function maps the hash key values key , $\text{key} + b$, $\text{key} + 2*b$, etc. to the same record number. In this case *collisions* may occur, for example, when a new record being inserted hashes to an address that already contains a different record.

To solve this conflict, the new record must be inserted at some other position. The process of finding another position at which to insert the new record is called *collision resolution*.

Linear solution is a common collision resolution technique: Starting at the occupied position, subsequent positions are checked sequentially until an available position is found. The new file record is then inserted at this position.

Exercise 2

A hash file is required for speed of access to customer data. The concept of hash files is explained on the opposite page. To keep things simple, each record in the hash file contains only a customer id and a name. The customer id is the key used by the hash function opposite to compute the address of the record. Use linear solution as collision resolution technique.

Note: Linear solution provides for adequate access times if the address space is sufficiently large and not too full. It is also important to distribute the record numbers yielded by the hash function evenly throughout the available address space. The hash function opposite will guarantee a good distribution if b is a sufficiently large prime number.

- Develop the `HashEntry` class used to represent customer data. You need to store the customer id as an `unsigned long` value and the name of the customer as a `char` array with a length of 30. Supply default values for the constructor declaration and additionally declare the `read_at()` and `write_at()` methods that read customer information at a given position in a stream or write information at that position. Both methods expect the position and the stream as arguments.
- Define the `HashFile` class to represent a hash file. The private members of the class comprise an `fstream` type file stream, a string used to store the file name, an `int` variable used to store the number b , and the hash function shown opposite as a method. The public members comprise a constructor that expects a file name and a number b as arguments. It opens the corresponding file for read and write access. The destructor closes the file.

Additionally declare the methods `insert()` and `retrieve()` to insert or retrieve single records. Both methods use a call to the hash function to compute the appropriate record number in the hash file. If a collision occurs, the methods perform a sequential search to locate the next free slot in the address space (mod of address space size), or the desired customer data.

- Test the `HashFile` by writing a `main` function that creates hash file with a small address space (e.g. $b = 7$). Add various customer information records to the hash file and then retrieve this data. Deliberately provoke collisions using the customer ids 5, 12, and 19, for example.

■ SOLUTIONS

Exercise I

```
// -----
// exceptio.h : Error classes for file processing
// -----

// Unchanged (cf. earlier in this chapter).

// -----
// account.h :
// Defines the classes
//      Account, DepAcc, and SavAcc
// with virtual read and write methods as well as
// the class AccFile to represent account files.
// -----
#ifndef _ACCOUNT_H
#define _ACCOUNT_H

#include <fstream>
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

#include "exceptio.h"

enum TypeId { ACCOUNT, DEP_ACC, SAV_ACC };

class Account
{
private:                                // Data elements:
    string name;
    unsigned long nr;
    double balance;

public:                                // Constructor:
    Account( const string c_name = "X",
              unsigned long c_nr  = 1111111L,
              double c_balance    = 0.0)
        : name(c_name), nr(c_nr), balance(c_balance)
    { }

    virtual ~Account() {}                // Virtual destructor
}
```

```

        // Access methods here:
        long getNr() const { return nr; }
        void setNr(unsigned long n){ nr = n; }
        // . . .

        // The other methods:
        virtual TypeId getTypeId() const { return ACCOUNT; }

        virtual ostream& write(ostream& fs) const;
        virtual istream& read(istream& fs);

        virtual void display() const
        {
            cout << fixed << setprecision(2)
                 << "-----\n"
                 << "Account holder:      " << name    << endl
                 << "Account number:      " << nr      << endl
                 << "Balance of account:  " << balance << endl
                 << "-----\n"
                 << endl;
        }
};

class DepAcc : public Account
{
private:
    double limit;           // Overdrawn limit
    double interest;        // Interest rate

public:
    DepAcc( const string s = "X",
            unsigned long n = 1111111L,
            double bal = 0.0,
            double li  = 0.0,
            double ir  = 0.0)
        : Account(s, n, bal), limit(li), interest(ir)
    { }

    // Access methods:
    // . . .

    // The other methods are implicit virtual:
    TypeId getTypeId() const { return DEP_ACC; }

    ostream& write(ostream& fs) const;
    istream& read(istream& fs);

```

```
void display() const
{
    Account::display();
    cout << "Overdrawn limit:      " << limit << endl
         << "Competitive interest: " << interest
         << "\n-----\n"
         << endl;
}

};

class SavAcc: public Account
{
private:
    double interest;           // Compound interest

public:
    // Methods as in class DepAcc.

};

// -----
// The definition of class AccFile

class AccFile
{
private:
    fstream f;
    string name;               // Filename

public:
    AccFile(const string& s) throw(OpenError);
    ~AccFile(){ f.close(); }

    long append( Account& acc)   throw(WriteError);
    Account* retrieve( long pos) throw(ReadError);

    void display() throw( ReadError);
};

#endif
```

```

// -----
// account.cpp
// Implement methods of the classes
// Account, DepAcc, SavAcc, and AccFile.
// -----
#include "account.h"
#include <typeinfo>

ostream& Account::write(ostream& os) const
{
    os << name << '\0';
    os.write((char*)&nr, sizeof(nr) );
    os.write((char*)&balance, sizeof(balance) );
    return os;
}

istream& Account::read(istream& is)
{
    getline( is, name, '\0');
    is.read((char*)&nr, sizeof(nr) );
    is.read((char*) &balance, sizeof(balance));
    return is;
}

ostream& DepAcc::write(ostream& os) const
{
    if(!Account::write(os))
        return os;
    os.write((char*)&limit, sizeof(limit) );
    os.write((char*)&interest, sizeof(interest) );
    return os;
}

istream& DepAcc::read(istream& is)
{
    if(!Account::read(is))
        return is;
    is.read((char*)&limit, sizeof(limit) );
    is.read((char*)&interest, sizeof(interest));
    return is;
}

// ostream& SavAcc::write(ostream& os) const
// istream& SavAcc::read(istream& is)
// as in class DepAcc.

```

```

// ---- Methods of class AccFile ----
AccFile::AccFile(const string& s) throw( OpenError)
{
    ios::openmode mode = ios::in | ios::out | ios::app
                        | ios::binary;
    f.open( s.c_str(), mode);
    if(!f)
        throw OpenError(s);
    else
        name = s;
}

void AccFile::display() throw(ReadError)
{
    Account acc, *pAcc = NULL;
    DepAcc depAcc;
    SavAcc savAcc;
    TypeId id;

    if( !f.seekg(0L))
        throw ReadError(name);

    cout << "\nThe account file: " << endl;

    while( f.read((char*)&id, sizeof(TypeId)) )
    {
        switch(id)
        {
            case ACCOUNT:  pAcc = &acc;
                           break;
            case DEP_ACC:  pAcc = &depAcc;
                           break;
            case SAV_ACC:  pAcc = &savAcc;
                           break;

            default: cerr << "Invalid flag in account file"
                        << endl;
                     exit(1);
        }

        if(!pAcc->read(f))
            break;

        pAcc->display();
        cin.get();           // Go on with return
    }
}

```

```

        if( !f.eof())
            throw ReadError(name);
        f.clear();
    }

long AccFile::append( Account& acc) throw( WriteError)
{
    f.seekp(0L, ios::end);    // Seek to end,
    long pos = f.tellp();    // save the position.

    if( !f )
        throw WriteError(name);

    TypeId id = acc.getTypeId();
    f.write( (char*)&id, sizeof(id));    // Write the TypeId

    if(!f)
        throw WriteError(name);
    else
        acc.write(f);            // Add an object to the file.

    if(!f)
        throw WriteError(name);
    else
        return pos;
}

Account* AccFile::retrieve( long pos) throw(ReadError)
{
    f.clear();
    f.seekg(pos);                // Set the get pointer

    if( !f )
        throw ReadError(name);

    TypeId id;
    f.read( (char*)&id, sizeof(id) );    // Get TypeId

    if(!f)
        throw ReadError(name);

    Account* buf;
    switch( id )
    {
        case ACCOUNT:    buf = new Account;
                        break;
        case SAV_ACC:    buf = new SavAcc;
                        break;
    }
}

```

```

        case DEP_ACC:  buf = new DepAcc;
                      break;
    }

    if( !(buf->read(f)) )           // Get data
        throw ReadError(name);

    return buf;
}

// -----
// index.h: Contains definitions of classes
//          IndexEntry representing an index entry,
//          Index      representing the index and
//          IndexFile  representing an index file.
// -----
#ifndef _INDEX_H
#define _INDEX_H

#include <fstream>
#include <iostream>
#include <string>
#include "account.h"
using namespace std;

class IndexEntry
{
private:
    long key;           // Key
    long recPos;        // Offset

public:
    IndexEntry(long k=0L, long n=0L){ key=k; recPos=n; }

    void setKey(long k) { key = k; }
    long getKey() const { return key; }
    void setPos(long p) { recPos = p; }
    long getPos() const { return recPos; }

    int recordSize() const
    { return sizeof(key) + sizeof(recPos); }

    fstream& write( fstream& ind) const;
    fstream& read(  fstream& ind);

    fstream& write_at(fstream& ind, long pos) const;
    fstream& read_at( fstream& ind, long pos);

```

```

        void display() const
        {   cout << "Account Nr: " << key
            << "   Position: " << recPos << endl;
        }
    };

class IndexFile
{
private:
    fstream index;
    string name;           // Filename of index

public:
    IndexFile( const string& s) throw (OpenError);
    ~IndexFile() { index.close(); }

    void insert( long key, long pos)
        throw(ReadError, WriteError);
    long search( long key) throw(ReadError);
    void retrieve(IndexEntry& entry, long pos )
        throw(ReadError);
    void display() throw(ReadError);
};

class IndexFileSystem : public AccFile, public IndexFile
{
private:
    string name;           // Filename without suffix
public:
    IndexFileSystem(const string& s)
        : AccFile(s + ".prim"), IndexFile(s + ".ind")
        { name = s; }

    bool    insert( Account& acc);
    Account* retrieve( long key);
};

#endif

```

```

// -----
// index.cpp : Methods of the classes
//             IndexEntry, Index, and IndexFile
// -----
#include "index.h"

fstream& IndexEntry::write_at(fstream& ind, long pos) const
{
    ind.seekp(pos);
    ind.write((char*)&key, sizeof(key) );
    ind.write((char*)&recPos, sizeof(recPos) );
    return ind;
}

fstream& IndexEntry::read_at(fstream& ind, long pos)
{
    ind.seekg(pos);
    ind.read((char*)&key, sizeof(key) );
    ind.read((char*)&recPos, sizeof(recPos));
    return ind;
}

fstream& IndexEntry::write(fstream& ind) const
{
    ind.write((char*)&key, sizeof(key) );
    ind.write((char*)&recPos, sizeof(recPos) );
    return ind;
}

fstream& IndexEntry::read(fstream& ind)
{
    ind.read((char*)&key, sizeof(key) );
    ind.read((char*)&recPos, sizeof(recPos));
    return ind;
}

// -----
// Methods of class IndexFile
IndexFile::IndexFile(const string& file) throw (OpenError)
{
    ios::openmode mode = ios::in | ios::out | ios::binary;
                        // Open file if it already exists:
    index.open( file.c_str(), mode);
    if(!index)          // If the file doesn't exist
    {
        index.clear();
        mode |= ios::trunc;
        index.open( file.c_str(), mode);
        if(!index)
            throw OpenError(name);
    }
    name = file;
}

```

```

void IndexFile::display() throw(ReadError)
{
    IndexEntry entry;

    index.seekg(0L);
    if(!index)
        throw ReadError("IndexFile: Setting the get pointer");

    cout << endl << "The Index: " << endl;
    while( true)
    {
        if( !entry.read(index))
            break;
        entry.display();
    }
    if( !index.eof())
        throw ReadError(name);
    index.clear();
}

long IndexFile::search(long k) throw(ReadError)
{
    IndexEntry entry;
    long key;
    long mid, begin = 0, end;    // Number of file records.
    int size = entry.recordSize(); // Length of an index
                                   // entry.

    index.clear();
    index.seekg(0L, ios::end);
    end = index.tellg() / size;

    if(!index)
        throw ReadError(name);

    if( end == 0)
        return -1;

    end -= 1;                // Position of the last entry

    while( begin < end )
    {
        mid = (begin + end + 1)/2 ;

        entry.read_at(index, mid*size);
        if(!index)
            throw ReadError(name);

        key = entry.getKey();
        if( k < key)
            end = mid - 1;
        else
            begin = mid;
    }
}

```

```

        entry.read_at(index, begin * size);
        if(!index)
            throw ReadError(name);

        if( k == entry.getKey() )           // Key found?
            return begin * size;
        else return -1;
    }

void IndexFile::insert(long k, long n)
    throw(ReadError, WriteError)
{
    IndexEntry entry;
    int size = entry.recordSize(); // Length of an index
                                   // entry.

    index.clear();
    index.seekg(0, ios::end);
    long nr = index.tellg();        // Get file length
                                   // 0, if file is empty.

    if(!index) throw ReadError(name);

    nr -= size;                     // Last entry.
    bool found = false;
    while( nr >= 0 && !found )       // Search position
    {                                // to insert
        if(!entry.read_at(index, nr))
            throw ReadError(name);

        if( k < entry.getKey() )     // To shift.
        {
            entry.write_at(index, nr + size);
            nr -= size;
        }
        else
        {
            found = true;
        }
    }

    entry.setKey(k); entry.setPos(n); // Insert
    entry.write_at(index, nr + size);

    if(!index)
        throw WriteError(name);
}

```

```

void IndexFile::retrieve( IndexEntry& entry, long pos)
    throw(ReadError)
{
    index.clear();
    if(!entry.read_at(index, pos))
        throw ReadError(name);
}

// -----
// Implementing the methods of class IndexFileSystem.

bool IndexFileSystem::insert( Account& acc)
    throw(ReadError, WriteError)
{
    if(search(acc.getNr()) == -1) // No multiple entries.
    {
        long pos = append(acc); // Add to primary file.
        IndexFile::insert(acc.getNr(), pos); // Add to Index
        return true;
    }
    else
        return false;
}

Account* IndexFileSystem::retrieve(long key )
{
    // Get the record address from the index:
    long pos = search(key); // Byte offset of
                           // index entry.

    if( pos == -1 ) // Account number doesn't exist.
        return NULL;
    else // Account number does exist:
    {
        IndexEntry entry; // To read the index eintry
        IndexFile::retrieve( entry, pos);
        // Get from primary file:
        return( AccFile::retrieve( entry.getPos() ));
    }
}

// -----
// index_t.cpp : Testing the index file
// -----
#include <iostream>
#include <string>
using namespace std;
#include "index.h"
#include "account.h"

```

```

int main()
{
    try
    {
        IndexFileSystem database("AccountTest");

        Account acc1( "Vivi", 490UL, 12340.57);
        database.insert( acc1 );

        SavAcc acc2( "Ulla", 590UL, 4321.19, 2.5);
        database.insert( acc2 );

        DepAcc acc3( "Jeany", 390UL, 12340.20, 10000.0, 12.9);
        database.insert( acc3 );

        database.IndexFile::display();
        cin.get();

        database.AccFile::display();

        unsigned long key;
        cout << "Key? "; cin >> key;
        if(database.search(key) != -1)
            cout << "Key " << key << " found" << endl;
        else
            cout << "Key " << key << " not found" << endl;

        Account* pAcc = database.retrieve(key);
        if( pAcc != NULL )
        {
            pAcc->display();
            delete pAcc;
            pAcc = NULL;
        }
        else cout << "Retrieving failed" << endl;
    }
    catch(OpenError& err)
    {
        cerr << "Error on opening the file:" << err.getName()
            << endl;
        exit(1);
    }
    catch(WriteError& err)
    {
        cerr << "Error on writing into the file: "
            << err.getName() << endl;
        exit(1);
    }
}

```