A Complete Guide to Programming in C++

Ulla Kirch-Prinz

Peter Prinz

JONES AND BARTLETT PUBLISHERS
Sudbury, Massachusetts
BOSTON TORONTO LONDON SINGAPORE

preface

This book was written for readers interested in learning the C++ programming language from scratch, and for both novice and advanced C++ programmers wishing to enhance their knowledge of C++. It was our goal from the beginning to design this text with the capabilities of serving dual markets, as a textbook for students and as a holistic reference manual for professionals.

The C++ language definition is based on the **A**merican **N**ational **S**tandards **I**nstitute *ANSI Standard* X3J16. This standard also complies with ISO norm 14882, which was ratified by the **I**nternational **S**tandardization **O**rganization in 1998. The C++ programming language is thus platform-independent in the main with a majority of C++ compilers providing ANSI support. New elements of the C++ language, such as exception handling and templates, are supported by most of the major compilers. Visit the Jones and Bartlett web site at www.jbpub.com for a listing of compilers available for this text.

The ***chapters*** in this book are organized to guide the reader from elementary language concepts to professional software development, with in-depth coverage of all the C++ language elements en route. The order in which these elements are discussed reflects our goal of helping the reader to create useful programs at every step of the way.

discussed. Students learn that templates allow the construction of functions and classes based on types that have not yet been stated. Thus, templates are a powerful tool for automating program code generation.

Chapter 33 explains standard class templates used to represent containers for more efficient management of object collections. These include sequences, such as lists and double ended queues; container adapters, such as stacks, queues, and priority queues; associative containers, such as sets and maps; and bitsets. In addition to discussing how to manage containers, the chapter also looks at sample applications, such as bitmaps for raster images, and routing techniques.

Additional Features

Chapter Goals A concise chapter introduction, which contains a description of the chapter's contents, is presented at the beginning of each chapter. These summaries also provide students with an idea of the key points to look for throughout the chapter.

Chapter Exercises Each chapter contains exercises, including programming problems, designed to test students' knowledge and understanding of the main ideas. The exercises also provide reinforcement for key chapter concepts. Solutions are included to allow students to check their work immediately and correct any possible mistakes.

Case Studies Every chapter contains a number of case studies that were designed to introduce the reader to a wide range of application scenarios.

Notes This feature provides students with helpful tips and information useful to learning C++. Important concepts and rules are highlighted for additional emphasis and easy access.

Hints These are informative suggestions for easier programming. Also included are common mistakes and how to avoid making them.

Acknowledgements

Our thanks go out to everyone who helped produce this book, particularly to

Ian Travis, for his valuable contributions to the development of this book.
Alexa Doehring, who reviewed all samples and program listings, and gave many valuable hints from the American perspective.
Michael Stranz and **Amy Rose** at Jones and Bartlett Publishers, who managed the publishing agreement and the production process so smoothly.
Our children, **Vivi** and **Jeany**, who left us in peace long enough to get things finished!
And now all that remains is to wish you, Dear Reader, **lots of fun with C++**!

Ulla Kirch-Prinz
Peter Prinz

■ A BEGINNER'S C++ PROGRAM

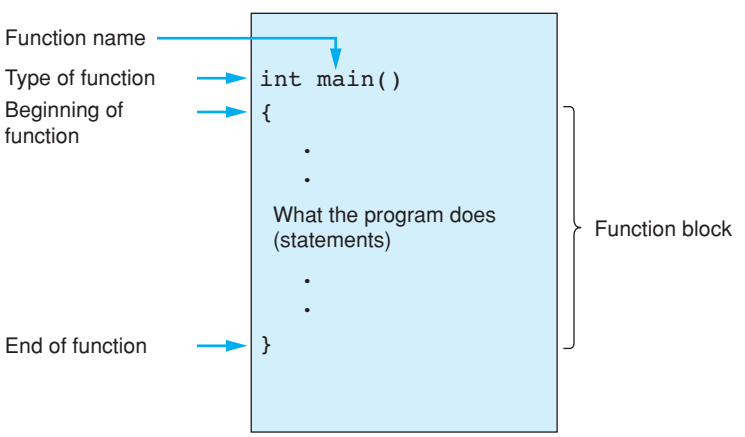
Sample program

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

int main()
{
   cout <<  "Enjoy yourself with C++!"  << endl;
   return 0;
}
```


Screen output

```
Enjoy yourself with C++!
```

Structure of function main()



A C++ program is made up of objects with their accompanying *member functions* and *global functions*, which do not belong to any single particular class. Each function fulfills its own particular task and can also call other functions. You can create functions yourself or use ready-made functions from the standard library. You will always need to write the global function `main()` yourself since it has a special role to play; in fact it is the main program.

The short programming example on the opposite page demonstrates two of the most important elements of a C++ program. The program contains only the function `main()` and displays a message.

The first line begins with the number symbol, `#`, which indicates that the line is intended for the *preprocessor*. The preprocessor is just one step in the first translation phase and no object code is created at this time. You can type

```
#include <filename>
```

to have the preprocessor copy the quoted file to this position in the source code. This allows the program access to all the information contained in the header file. The header file `iostream` comprises conventions for input and output streams. The word *stream* indicates that the information involved will be treated as a flow of data.

Predefined names in C++ are to be found in the `std` (standard) namespace. The `using` directive allows direct access to the names of the `std` namespace.

Program execution begins with the first instruction in function `main()`, and this is why each C++ program must have a main function. The structure of the function is shown on the opposite page. Apart from the fact that the name cannot be changed, this function's structure is not different from that of any other C++ function.

In our example the function `main()` contains two *statements*. The first statement

```
cout << "Enjoy yourself with C++!" << endl;
```

outputs the text string `Enjoy yourself with C++!` on the screen. The name `cout` (console output) designates an object responsible for output.

The two less-than symbols, `<<`, indicate that characters are being "pushed" to the output stream. Finally `endl` (end of line) causes a line feed. The statement

```
return 0;
```

terminates the function `main()` and also the program, returning a value of `0` as an exit code to the calling program. It is standard practice to use the exit code `0` to indicate that a program has terminated correctly.

Note that statements are followed by a semicolon. By the way, the shortest statement comprises only a semicolon and does nothing.

A program can use several data to solve a given problem, for example, characters, integers, or floating-point numbers. Since a computer uses different methods for processing and saving data, the data *type* must be known. The type defines

1. the internal representation of the data, and
2. the amount of memory to allocate.

A number such as `-1000` can be stored in either 2 or 4 bytes. When accessing the part of memory in which the number is stored, it is important to read the correct number of bytes. Moreover, the memory content, that is the bit sequence being read, must be interpreted correctly as a signed integer.

The C++ compiler recognizes the *fundamental types*, also referred to as *built-in types*, shown on the opposite page, on which all other types (vectors, pointers, classes, ...) are based.

☐ The Type `bool`

The result of a comparison or a logical association using AND or OR is a *boolean* value, which can be true or false. C++ uses the `bool` type to represent boolean values. An expression of the type `bool` can either be `true` or `false`, where the internal value for `true` will be represented as the numerical value 1 and `false` by a zero.

☐ The `char` and `wchar_t` Types

These types are used for saving character codes. A *character code* is an integer associated with each character. The letter `A` is represented by code 65, for example. The *character set* defines which code represents a certain character. When displaying characters on screen, the applicable character codes are transmitted and the "receiver," that is the screen, is responsible for correctly interpreting the codes.

The C++ language does not stipulate any particular characters set, although in general a character set that contains the *ASCII code* (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) is used. This 7-bit code contains definitions for 32 control characters (codes 0 – 31) and 96 printable characters (codes 32 – 127).

The `char` (character) type is used to store character codes in one byte (8 bits). This amount of storage is sufficient for extended character sets, for example, the ANSI character set that contains the ASCII codes and additional characters such as German umlauts.

The `wchar_t` (wide character type) type comprises at least 2 bytes (16 bits) and is thus capable of storing modern Unicode characters. *Unicode* is a 16-bit code also used in Windows NT and containing codes for approximately 35,000 characters in 24 languages.

■ FUNDAMENTAL TYPES (CONTINUED)

Integral types

Type	Size	Range of Values (decimal)
`char`	1 byte	−128 to +127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	−128 to +127
`int`	2 byte resp.	−32768 to +32767 resp.
	4 byte	−2147483648 to +2147483647
`unsigned int`	2 byte resp.	0 to 65535 resp.
	4 byte	0 to 4294967295
`short`	2 byte	−32768 to +32767
`unsigned short`	2 byte	0 to 65535
`long`	4 byte	−2147483648 to +2147483647
`unsigned long`	4 byte	0 to 4294967295

Sample program

```
#include <iostream>
#include <climits>       // Definition of INT_MIN, ...
using namespace std;

int main()
{
  cout << "Range of types int and unsigned int"
       << endl << endl;
  cout << "Type              Minimum            Maximum"
       << endl
       << "-----------------------------------------------"
       << endl;

  cout << "int             " <<  INT_MIN << "         "
                             <<  INT_MAX << endl;

  cout << "unsigned int    " <<  "           0         "
                             << UINT_MAX << endl;
  return 0;
}
```

chapter 3

Using Functions and Classes

This chapter describes how to

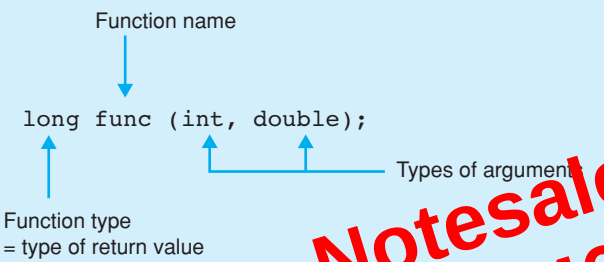
- declare and call standard functions and
- use standard classes.

This includes using standard header files. In addition, we will be working with string variables, i.e. objects belonging to the standard class `string` for the first time.

Functions and classes that you define on your own will not be introduced until later in the book.

■ DECLARING FUNCTIONS

Example of a function prototype



The prototype above yields the following information to the compiler:

- `func` is the function name
- the function is called with two arguments: the first argument is of type `int`, the second of type `double`
- the return value of the function is of type `long`.

Mathematical standard functions

```
double sin (double);           // Sine
double cos (double);           // Cosine
double tan (double);           // Tangent
double atan (double);          // Arc tangent
double cosh (double);          // Hyperbolic Cosine
double sqrt (double);          // Square Root
double pow (double, double);   // Power
double exp (double);           // Exponential Function
double log (double);           // Natural Logarithm
double log10 (double);         // Base-ten Logarithm
```

☐ Declarations

Each name (*identifier*) occurring in a program must be known to the compiler or it will cause an error message. That means any names apart from keywords must be *declared*, i.e. introduced to the compiler, before they are used.

Each time a variable or a function is defined it is also declared. But conversely, not every declaration needs to be a definition. If you need to use a function that has already been introduced in a library, you must declare the function but you do not need to redefine it.

☐ Declaring Functions

A function has a name and a type, much like a variable. The function's type is defined by its *return value*, that is, the value the function passes back to the program. In addition, the type of arguments required by a function is important. When a function is declared, the compiler must therefore be provided with information on

- the name and type of the function and
- the type of each argument.

This is also referred to as the function *prototype*.

Examples:
```
int toupper(int);

double pow(double, double);
```

This informs the compiler that the function `toupper()` is of type `int`, i.e. its return value is of type `int`, and it expects an argument of type `int`. The second function `pow()` is of type `double` and two arguments of type `double` must be passed to the function when it is called. The types of the arguments may be followed by names, however, the names are viewed as a comment only.

Examples:
```
int toupper(int c);

double pow(double base, double exponent);
```

From the compiler's point of view, these prototypes are equivalent to the prototypes in the previous example. Both junctions are standard junctions.

Standard function prototypes do not need to be declared, nor should they be, as they have already been declared in standard header files. If the header file is included in the program's source code by means of the `#include` directive, the function can be used immediately.

Example:
```
#include <cmath>
```

Following this directive, the mathematical standard functions, such as `sin()`, `cos()`, and `pow()`, are available. Additional details on header files can be found later in this chapter.

Exercise 1

Create a program to calculate the square roots of the numbers

4 12.25 0.0121

and output them as shown opposite. Then read a number from the keyboard and output the square root of this number.

To calculate the square root, use the function `sqrt()`, which is defined by the following prototype in the `math.h` (or `cmath`) header file:

```
double sqrt( double x);
```

The return value of the `sqrt()` function is the square root of `x`.

Exercise 2

The program on the opposite page contains several errors! Correct the errors and ensure that the program can be executed.

Exercise 3

Create a C++ program that defines a string containing the following character sequence:

```
I have learned something new again!
```

and displays the length of the string on screen.

Read two lines of text from the keyboard. Concatenate the strings using `" * "` to separate the two parts of the string. Output the new string on screen.

■ FORMATTED OUTPUT OF INTEGERS

Manipulators formatting integers

Manipulator	Effects
`oct`	Octal base
`hex`	Hexadecimal base
`dec`	Decimal base (by default)
`showpos`	Generates a + sign in non-negative numeric output.
`noshowpos`	Generates non-negative numeric output without a + sign (by default).
`uppercase`	Generates capital letters in hexadecimal output.
`nouppercase`	Generates lowercase letters in hexadecimal output (by default).

Sample program

```
// Reads integral decimal values and
// generates octal, decimal, and hexadecimal output.

#include <iostream>      // Declarations of cin, cout and
using namespace std;     // manipulators oct, hex, ...

int main()
{
   int number;
   cout << "Please enter an integer: ";
   cin >> number;
   cout << uppercase                 // for hex-digits
        << " octal  \t decimal  \t hexadecimal\n "
        << oct << number << "        \t "
        << dec << number << "        \t "
        << hex << number << endl;
   return 0;
}
```

■ OUTPUT IN FIELDS

Element functions for output in fields

Method	Effects
`int width() const;`	Returns the minimum field width used
`int width(int n);`	Sets the minimum field width to `n`
`int fill() const;`	Returns the fill character used
`int fill(int ch);`	Sets the fill character to `ch`

Manipulators for output in fields

Manipulator	Effects
`setw(int n)`	Sets the minimum field width to `n`
`setfill(int ch)`	Sets the fill character to `ch`
`left`	Left-aligns output in fields
`right`	Right-aligns output in fields
`internal`	Left-aligns output of the sign and right-aligns output of the numeric value

NOTE

The manipulators `setw()` and `setfill()` are declared in the header file `iomanip`.

Examples

```
#include <iostream>          // Obligatory
#include <iomanip>           // declarations
using namespace std;

1st Example: cout << '|' << setw(6) << 'X' << '|';
Output:      |     X|          // Field width 6
2nd Example: cout << fixed << setprecision(2)
                  << setw(10) << 123.4 << endl
                  << "1234567890" << endl;
Output:         123.40          // Field width 10
             1234567890
```

■ FORMATTED INPUT

Sample program

```
// Inputs an article label and a price

#include <iostream>     // Declarations of cin, cout,...
#include <iomanip>      // Manipulator setw()
#include <string>
using namespace std;

int main()
{
   string label;
   double price;

   cout << "\nPlease enter an article label:";

   // Input the label (15 characters maximum):
   cin >> setw(16);        // Or:  cin.width(16);
   cin >> label;

   cin.sync();    // Clears the buffer and resets
   cin.clear();   // any error flags that may be set

   cout << "\nEnter the price of the article: ";
   cin >> price;           // Input the price

   // Controlling output:
   cout << fixed << setprecision(2)
        << "\nArticle:"
        << "\n  Label:  " << label
        << "\n  Price:  " << price << endl;

   // ... The program to be continued

   return 0;
}
```


NOTE

The input buffer is cleared and error flags are reset by calling the `sync()` and `clear()` methods. This ensures that the program will wait for new input for the price, even if more than 15 characters have been entered for the label.

The >> operator, which belongs to the `istream` class, takes the current number base and field width flags into account when reading input:

- the number base specifies whether an integer will be read as a decimal, octal, or hexadecimal
- the field width specifies the maximum number of characters to be read for a string.

When reading from standard input, `cin` is buffered by lines. Keyboard input is thus not read until confirmed by pressing the <Return> key. This allows the user to press the backspace key and correct any input errors, provided the return key has not been pressed. Input is displayed on screen by default.

Input Fields

The >> operator will normally read the next input field, convert the input by reference to the type of the supplied variable, and write the result to the variable. Any white space characters (such as blanks, tabs, and new lines) are ignored by default.

Example:
```
char ch;
cin >> ch;          // Enter a character
```

When the following keys are pressed

```
<return> <tab> <blank> <X> <return>
```

the character `'X'` is stored in the variable `ch`.

An input field is terminated by the first white space character or by the first character that cannot be processed.

Example:
```
int i;
cin >> i;
```

Typing `123FF<Return>` stores the decimal value `123` in the variable `i`. However, the characters that follow, `FF` and the newline character, remain in the input buffer and will be read first during the next read operation.

When reading strings, only one word is read since the first white space character will begin a new input field.

Example:
```
string city;
cin >> city;      // To read just one word!
```

If `Lao Kai` is input, only `Lao` will be written to the `city` string. The number of characters to be read can also be limited by specifying the field width. For a given field width of `n`, a maximum of `n-1` characters will be read, as one byte is required for the null character. Any initial white space will be ignored. The program on the opposite page illustrates this point and also shows how to clear the input buffer.

☐ Inputting Integers

You can use the `hex`, `oct`, and `dec` manipulators to stipulate that any character sequence input is to processed as a hexadecimal, octal, or decimal number.

Example:
```
int n;
cin >> oct >> n;
```

An input value of `10` will be interpreted as an octal, which corresponds to a decimal value of `8`.

Example:
```
cin >> hex >> n;
```

Here, any input will be interpreted as a hexadecimal, enabling input such as `f0a` or `-F7`.

☐ Inputting Floating-Point Numbers

The `>>` operator interprets any input as a decimal floating-point number if the variable is a floating-point type, `float`, `double`, or `long double`. The floating-point number can be entered in fixed point or exponential notation.

Example:
```
double x;
cin >> x;
```

The character input is converted to a `double` value in this case. Input, such as `123`, `-22.0`, or `3e10` is valid.

☐ Input Errors

But what happens if the input does not match the type of variable defined?

Example:
```
int i, j;     cin >> i >> j;
```

Given input of `1A5` the digit `1` will be stored in the variable `i`. The next input field begins with `A`. But since a decimal input type is required, the input sequence will not be processed beyond the letter `A`. If, as in our example, no type conversion is performed, the variable is not written to and an internal error flag is raised.

It normally makes more sense to read numerical values individually, and clear the input buffer and any error flags that may have been set after each entry.

Chapter 6, "Control Flow," and Chapter 28, "Exception Handling," show how a program can react to input errors.

exercises

■ EXERCISES

Screen output for exercise 3

```
Article Number    Number of Pieces   Price per piece
  .......           ......       ...... Dollar
```

Program listing for exercise 5

```
// A program with resistant mistakes

#include <iostream>
using namespace std;

int main()
{
   char ch;
   string word;

   cin >> "Let's not press the return key: " >> ch;

   cout << "Enter a word containing
            three characters at most: ";

   cin  >> setprecision(3) >> word;

   cout >> "Your input: " >> ch >> endl;

   return 0;
}
```

Exercise 1

What output is generated by the program on the page entitled "*Formatted output of floating-point numbers*" in this chapter?

Exercise 2

Formulate statements to perform the following:

a. Left-justify the number 0.123456 in an output field with a width of 15.
b. Output the number 23.987 as a fixed point number rounded to two decimal places, right-justifying the output in a field with a width of 12.
c. Output the number –123.456 as an exponential and with four decimal spaces. How useful is a field width of 10?

Exercise 3

Write a C++ program that reads an article number, a quantity, and a unit price from the keyboard and outputs the data on screen as displayed on the opposite page.

Exercise 4

Write a C++ program that reads any given character code (a positive integer) from the keyboard and displays the corresponding character and the character code as a decimal, an octal, and a hexadecimal on screen.

TIP

The variable type defines whether a character or a number is to be read or output.

Why do you think the character P is output when the number 336 is entered?

Exercise 5

Correct the mistakes in the program on the opposite page.

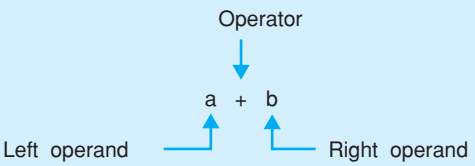
```
   cout << "Number of pieces:      ";
   cin  >> count;

   cout << "Price per piece:     ";
   cin  >> price;

                                                  // Output:
   cout <<
   "\n\tArticle Number    Quantity    Price per piece ";

   cout << "\n\t"
        << setw(8)  << number
        << setw(16) << count
        << fixed    << setprecision(2)
        << setw(16) << price << " Dollar" << endl;

   return 0;
}
```

Exercise 4

```
#include <iostream>
#include <iomanip>          // Manipulator setw()
using namespace std;

int main()
{
   unsigned char c = 0;
   unsigned int  code = 0;

   cout << "\nPlease enter a decimal character code: ";
   cin  >> code;

   c = code;                            // Save for output

   cout << "\nThe corresponding character: " << c << endl;

   code = c;            // Character code. Is only
                        // necessary, if input is > 255.
   cout << "\nCharacter codes"
        << "\n  decimal:     " << setw(3) << dec << code
        << "\n  octal:       " << setw(3) << oct << code
        << "\n  hexadecimal: " << setw(3) << hex << code
        << endl;

   return 0;
}
```

■ BINARY ARITHMETIC OPERATORS

Binary operator and operands



The binary arithmetic operators

Operator	Significance
+	Addition
	Subraction
	Multiplication
/	Division
%	Remainder

Sample program

```
#include <iostream>
using namespace std;
int main()
{
   double x, y;
   cout << "\nEnter two floating-point values: ";
   cin >> x >> y;
   cout << "The average of the two numbers is: "
        << (x + y)/2.0 << endl;
   return 0;
}
```

Sample output for the program

```
Enter two floating-point values:  4.75   12.3456
The average of the two numbers is: 8.5478
```

■ UNARY ARITHMETIC OPERATORS

The unary arithmetic operators

Operator	Significance
+ -	Unary sign operators
++	Increment operator
--	Decrement operator

Precedence of arithmetic operators

Precedence	Operator	Grouping
High	++ -- (postfix)	left to right
	++ -- (prefix) + - (sign)	right to left
	* / %	left to right
Low	+ (addition) - (subtraction)	left to right

Effects of prefix and postfix notation

```
#include <iostream>
using namespace std;
int main()
{
   int i(2), j(8);

   cout << i++ << endl;        // Output:  2
   cout << i   << endl;        // Output:  3
   cout << j-- << endl;        // Output:  8
   cout << --j << endl;        // Output:  6

   return 0;
}
```

☐ Initializing and Reinitializing

A typical loop uses a *counter* that is initialized, tested by the controlling expression and reinitialized at the end of the loop.

Example:

```
int count = 1;              // Initialization
while( count <= 10)         // Controlling
{                           // expression
  cout << count
       << ". loop" << endl;
  ++count;                  // Reinitialization
}
```

In the case of a `for` statement the elements that control the loop can be found in the loop header. The above example can also be expressed as a `for` loop:

Example:

```
int count;
for( count = 1; count <= 10; ++count)
  cout << count
       << ". loop" << endl;
```


Any expression can be used to initialize and reinitialize the loop. Thus, a `for` loop has the following form:

Syntax:

```
for( expression1; expression2; expression3 )
   statement
```

`expression1` is executed first and only once to initialize the loop. `expression2` is the controlling expression, which is always evaluated prior to executing the loop body:

- if `expression2` is `false`, the loop is terminated
- if `expression2` is `true`, the loop body is executed. Subsequently, the loop is reinitialized by executing `expression3` and `expression2` is re-tested.

You can also define the loop counter in `expression1`. Doing so means that the counter can be used within the loop, but not after leaving the loop.

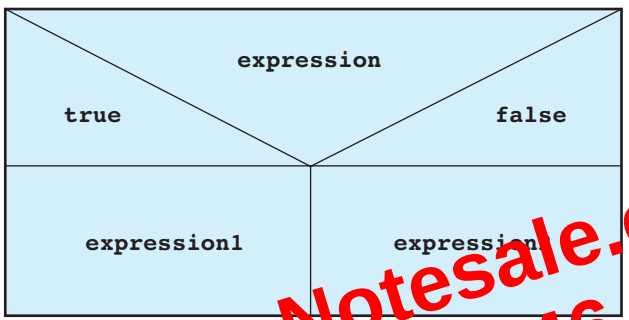
Example:

```
for( int i = 0; i < 10; cout << i++ )
   ;
```

As this example illustrates, the loop body can be an empty statement. This is always the case if the loop header contains all necessary statements. However, to improve readability, even the empty statement should occupy a line of its own.

■ CONDITIONAL EXPRESSIONS

Structogram for a conditional expression



Sample program

```
// greater.cpp
#include <iostream>
using namespace std;

int main()
{
   float x, y;

   cout << "Type two different numbers:\n";
   if( !(cin >> x && cin >> y) ) // If the input was
   {                             // invalid.
      cout << "\nInvalid input!" << endl;
   }
   else
   {
      cout << "\nThe greater value is: "
           << (x > y ? x : y)  << endl;
   }

   return 0;
}
```

Sample output for this program

```
Type two different numbers:
173.2
216.7
The greater value is: 216.7
```

☐ Conditional Operator

The conditional operator `?:` is used to form an expression that produces either of two values, depending on the value of some condition. Because the value produced by such an expression depends on the value of a condition, it is called *conditional expression*.

In contrast to the `if-else` statement the selection mechanism is based on expressions: one of two possible expressions is selected. Thus, a conditional expression is often a concise alternative to an `if-else` statement.

Syntax: `expression ? expression1 : expression2`

`expression` is evaluated first. If the result is `true`, `expression1` is evaluated; if not `expression2` is executed. The value of the conditional expression is therefore either the value of `expression1` or `expression2`.

Example: `z = (a >= 0) ? a : -a;`

This statement assigns the absolute value of `a` to the variable `z`. If `a` has a positive value of `12`, the number `12` is assigned to `z`. But if `a` has a negative value, for example `-8`, the number `8` is assigned to `z`.

Since this sample program stores the value of the conditional expression in the variable `z`, the statement is equivalent to

```
if( a > 0 )
   z = a;
else
   z = -a;
```


☐ Precedence

The conditional operator is the only C++ operator with three operands. Its precedence is higher than that of the comma and assignment operators but lower than all other operators. In other words, you could omit the brackets in the first example.

You can use the result of a conditional evaluation without assigning it, as the sample program on the opposite page shows. In this example, `x` is printed on screen if `x` is greater than `y`, and `y` is printed otherwise.

However, you should assign the result of complex expressions to a variable explicitly to improve the readability of your program.

☐ break

The `break` statement exits from a `switch` or loop immediately. You can use the `break` keyword to jump to the first statement that follows the `switch` or loop.

The program on the opposite page, which outputs a group of 20 ASCII characters and their corresponding codes, uses the `break` keyword in two places. The first `break` exits from an infinite `while(true) { ... }` loop when a maximum value of 256 has been reached. But the user can also opt to continue or terminate the program. The second `break` statement is used to terminate the `while` loop and hence the program.

☐ continue

The `continue` statement can be used in loops and has the opposite effect to `break`, that is, the next loop is begun immediately. In the case of a `while` or `do-while` loop the program jumps to the test expression, whereas a `for` loop is reinitialized.

Example:

```
for( int i = 0; i < 100; i++ )
{
   . . .  // Process all integers.
   if( i % 2 == 1)
      continue;
   . . .                  // Process even
                          // numbers only.
}
```


☐ goto and Labels

C++ also offers a `goto` statement and labels. This allows you to jump to any given point marked by a label within a function. For example, you can exit from a deeply embedded loop construction immediately.

Example:

```
for( . . . )
   for( . . . )
     if (error) goto errorcheck;
 . . .
errorcheck: . . .        // Error handling
```

A *label* is a name followed by a colon. Labels can precede any statement.

Any program can do without `goto` statements. If you need to use a `goto` statement, do so to exit from a code block, but avoid entering code blocks by this method.

chapter 7

Symbolic Constants and Macros

This chapter introduces you to the definition of symbolic constants and macros illustrating their significance and use. In addition, standard macros for character handling are introduced.

exercises

■ EXERCISES

Hints for Exercise 2

You can use the function `kbhit()` to test whether the user has pressed a key. If so, the function `getch()` can be used to read the character. This avoids interrupting the program when reading from the keyboard.

These functions have not been standardized by ANSI but are available on almost every system. Both functions use operating system routines and are declared in the header file `conio.h`.

The function `kbhit()`

Prototype: `int kbhit();`
Returns: `0`, if no key was pressed, otherwise `!= 0`.

When a key has been pressed, the corresponding character can be read by `getch()`.

The function `getch()`

Prototype: `int getch();`
Returns: The character code. There is no special return value on reaching end-of-file or if an error occurs.

In contrast to `cin.get()`, `getch()` does not use an input buffer when reading characters, that is, when a character is entered, it is passed directly to the program and not printed on screen. Additionally, control characters, such as return (= 13), Ctrl+Z (= 26), and Esc (= 27), are passed to the program "as is."

Example:

```
int c;
if( kbhit() != 0)   // Key was pressed?
{
   c = getch();     // Yes -> Get character
   if( c == 27 )    // character == Esc?
   // . . .
}
```

NOTE

When a function key, such as F1, F2, ..., Ins, Del, etc. was pressed, the function `getch()` initially returns `0`. A second call yields the key number.

Exercise 1

Please write

a. the macro `ABS`, which returns the absolute value of a number,

b. the macro `MAX`, which determines the greater of two numbers.

In both cases use the conditional operator `?:`.

Add these macros and other macros from this chapter to the header file `myMacros.h` and then test the macros.

If your system supports screen control macros, also add some screen control macros to the header. For example, you could write a macro named `COLOR(f,b)` to define the foreground and background colors for the following output.

Exercise 2

Modify the program `ball1.cpp` to

a. display a white ball on a blue background,

b. terminate the program when the Esc key is pressed,

c. increase the speed of the ball with the + key and decrease the speed with the - key.

You will need the functions `kbhit()` and `getch()` (shown opposite) to solve parts b and c of this problem.

Exercise 3

Write a filter program to display the text contained in any given file. The program should filter any control characters out of the input with the exception of the characters `\n` (end-of-line) and `\t` (tabulator), which are to be treated as normal characters for the purpose of this exercise. Control characters are defined by codes `0` to `31`.

A sequence of control characters is to be represented by a single space character.

A single character, that is, a character appearing between two control characters, is not to be output!

NOTE

Since the program must not immediately output a single character following a control character, you will need to store the predecessor of this character. You may want to use two counters to count the number of characters and control characters in the current string.

```
#define  ESC   27          // ESC terminates the program
unsigned long delay = 5000000;         // Delay for output

int main()
{
   int x = 2, y = 2, dx = 1, speed = 0;
   bool end = false;
   string floor(80, '-'),
          header   = "****  BOUNCING BALL ****",
          commands = "[Esc] = Terminate      "
                     "[+] = Speed up      [-] = Slow down";

   COLOR(WHITE,BLUE);    CLS;
   LOCATE(1,25);  cout << header;
   LOCATE(24,1);  cout << floor;
   LOCATE(25,10); cout << commands;

   while( !end )                  // As long as the flag is not set
   {
     LOCATE(y,x);  cout << 'o';          // Show the ball
     for( long wait = 0; wait < delay; ++wait)
       ;
     if(x == 1 || x == 79) dx = -dx;  // Bounce off a wall?
     if( y == 23 )                          // On the floor?
     {
       speed = - speed;
       if( speed == 0 ) speed = -7;       // Kick
     }
     speed += 1;                            // Speed up = 1

     LOCATE(y,x); cout <<  ' ';             // Clear screen
     y += speed;  x += dx;                  // New position

     if( kbhit() != 0 )                     // Key pressed?
     {
        switch(getch())                     // Yes
        {
          case '+':  delay -= delay/5;    // Speed up
                     break;
          case '-':  delay += delay/5;    // Slow down
                     break;
          case ESC:  end = true;          // Terminate
        }
     }
   }
   NORMAL;  CLS;
   return 0;
}
```

chapter 8

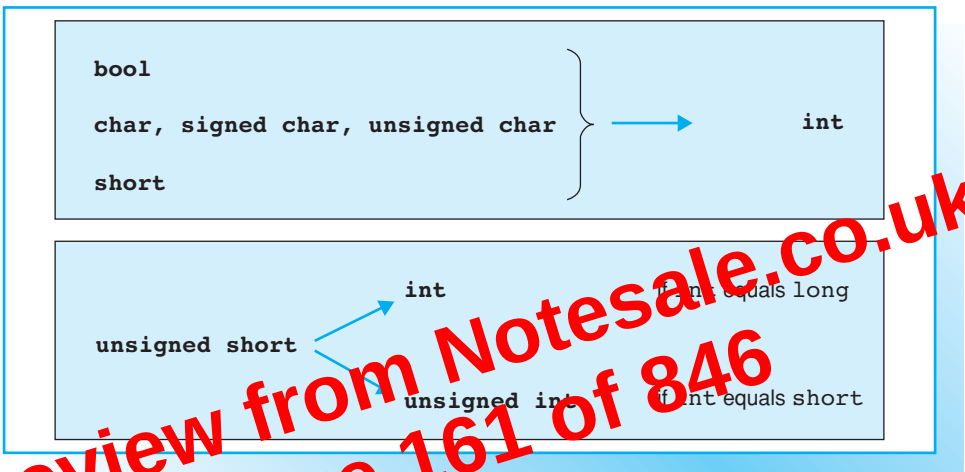
Converting Arithmetic Types

This chapter introduces implicit type conversions, which are performed in C++ whenever different arithmetic types occur in expressions.

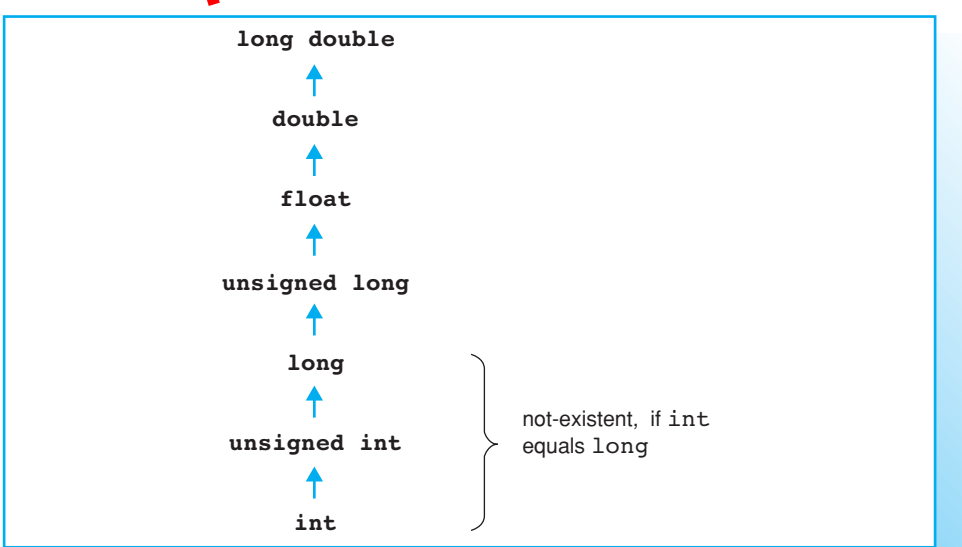
Additionally, an operator for explicit type conversion is introduced.

■ IMPLICIT TYPE CONVERSIONS

Integer promotions



Type hierarchy



Example

```
short size(512);  double res, x = 1.5;
res = size / 10 * x;     // short -> int -> double
      int
```

Exercise 1

A function has the following prototype

```
void func( unsigned int n);
```

What happens when the function is called with `-1` as an argument?

Exercise 2

How often is the following loop executed?

```
unsigned int limit = 1000;
for (int i = -1; i < limit; i++)
//  . . .
```

Exercise 3

What is output when the program opposite is executed?

Exercise 4

Write a C++ program to output the sine curve on screen as in the graphic shown on the opposite page.

NOTE

1. Plot one point of the curve in columns 10, 10+1, ..., 10+64 respectively. This leads to a step value of 2*PI/64 for x.
2. Use the following extended ASCII code characters to draw the axes:

Character	Decimal	Octal
–	196	304
+	197	305
▲	16	020
▶	30	036

Example: `cout << '\020';  // up arrowhead`

Searching

You can search strings to find the first or last instance of a substring. If the string contains the required substring, the position of the substring found by the search is returned. If not, a pseudo-position `npos`, or `-1`, is returned. Since the `npos` constant is defined in the `string` class, you can reference it as `string::npos`.

The `find()` method returns the position at which a substring was first found in the string. The method requires the substring to be located as an argument.

Example:
```
string youth("Bill is so young, so young");
int first = youth.find("young");
```

The variable `first` has a value of `11` in this example.

You can use the "right find" method `rfind()` to locate the last occurrence of a substring in a string. This initializes the variable `last` with a value of `21` in our example.

Example:
```
int last = youth.rfind("young");
```


Replacing

When replacing in strings, a string overwrites a substring. The string lengths need not be identical.

You can use the `replace()` method to perform this operation. The first two arguments supply the starting position and the length of the substring to be replaced. The third argument contains the replacement string.

Example:
```
string s1("There they go again!"),
       s2("Bob and Bill");
int pos = s1.find("they");       // pos == 6
if( pos != string::npos )
   s1.replace(pos, 2, s2);
```

This example uses the string `s2` to replace 4 characters, `"they"`, starting at position 6 in `s1`. After this operation `s1` contains the string `"There Bob and Bill go again!"`.

If you only need to insert part of a string, you can use the fourth argument to define the starting position and the fifth to define the length of the substring.

Example:
```
string s1("Here comes Mike!"),
                 s2("my love?");
          s1.replace(11, 4, s2, 0, 7);
```

The string `s1` is changed to `"Here comes my love!"`.

chapter 10

Functions

This chapter describes how to write functions of your own. Besides the basic rules, the following topics are discussed:

- passing arguments
- definition of `inline` functions
- overloading functions and default arguments
- the principle of recursion.

The program opposite shows how the function `area()` is defined and called. As previously mentioned, you must declare a function before calling it. The *prototype* provides the compiler with all the information it needs to perform the following actions when a function is called:

- check the number and type of the arguments
- correctly process the return value of the function.

A function declaration can be omitted only if the function is defined within the same source file immediately before it is called. Even though simple examples often define and call a function within a single source file, this tends to be an exception. Normally the compiler will not see a function definition as it is stored in a different source file.

When a function is called, an argument of the same type as the parameter must be passed to the function for each parameter. The arguments can be any kind of expressions, as the example opposite with the argument `1+2` shows. The value of the expression is always copied to the corresponding parameter.

Return Statement

When the program flow reaches a *return statement* or the end of a function code block, it branches back to the function that called it. If the function is any type other than `void`, the `return` statement will also cause the function to return a value to the function that called it.

Syntax: `return [expression]`

If `expression` is supplied, the value of the expression will be the return value. If the type of this value does not correspond to the function type, the function type is converted, where possible. However, functions should always be written with the `return` value matching the function type.

The function `area()` makes use of the fact that the `return` statement can contain any expression. The `return` expression is normally placed in parentheses if it contains operators.

If the expression in the `return` statement, or the `return` statement itself, is missing, the return value of the function is undefined and the function type must be `void`. Functions of the `void` type, such as the standard function `srand()`, will perform an action but not return any value.

Exercise 2

```
// ------------------------------------------------------
// max.cpp
// Defines and calls the overloaded functions Max().
// ------------------------------------------------------

// As long as just one function Max() is defined, it can
// be called with any arguments that can be converted to
// double, i.e. with values of type char, int or long.
// After overloading no clear conversion will be possible.

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

inline double Max(double x, double y)
{
   return (x < y ? y : x);
}

inline char Max(char x, char y)
{
   return (x < y ? y : x);
}

string header(
         "To use the overloaded function Max().\n"),
       line(50,'-');

int main()     // Several different calls to function Max()
{
   double x1 = 0.0, x2 = 0.0;

   line += '\n';
   cout << line << header << line << endl;

   cout << "Enter two floating-point numbers:"
        << endl;
   if( cin >> x1  &&  cin >> x2)
   {
       cout << "The greater number is " << Max(x1,x2)
            << endl;
   }
   else
       cout << "Invalid input!" << endl;

   cin.sync(); cin.clear();   // Invalid input
                              // was entered.
```

```
    cout << line
         << "And once more with characters!"
         << endl;

    cout << "Enter two characters:"
         << endl;

    char c1, c2;
    if( cin >> c1  &&  cin >> c2)
    {
        cout << "The greater character is " << Max(c1,c2)
             << endl;
    }
    else
        cout << "Invalid input!" << endl;

    cout << "Testing with int arguments." << endl;
    int  a = 30, b = 50;
    cout << Max(a,b) << endl;       // Error! Which
                                    // function Max()?
    return 0;
}
```

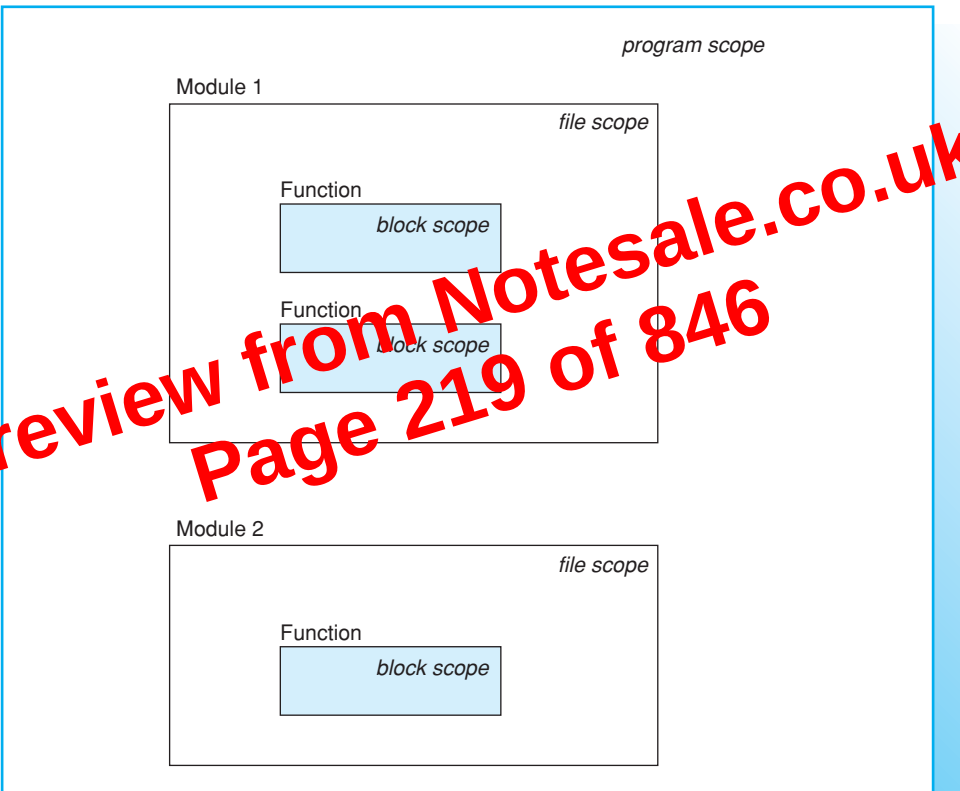

Exercise 3

```
// ------------------------------------------------------
// factorial.cpp
// Computes the factorial of an integer iteratively,
// i.e. using a loop, and recursively.
// ------------------------------------------------------
#include <iostream>
#include <iomanip>
using namespace std;

#define N_MAX   20

long double fact1(unsigned int n);  // Iterative solution
long double fact2(unsigned int n);  // Recursive solution

int main()
{
   unsigned int n;

  // Outputs floating-point values without
  // decimal places:
   cout << fixed << setprecision(0);
```

■ STORAGE CLASSES OF OBJECTS

☐ Availability of Objects

C++ program



☐ Storage Class Specifiers

The storage class of an object is determined by

- the position of its declaration in the source file
- the storage class specifier, which can be supplied optionally.

The following storage class specifiers can be used

```
extern   static   auto   register
```

Exercise 1

In general, you should use different names for different objects. However, if you define a name for an object within a code block and the name is also valid for another object, you will reference only the new object within the code block. The new declaration hides any object using the same name outside of the block. When you leave the code block, the original object once more becomes visible.

The program on the opposite page uses identical variable names in different blocks. What does the program output on screen?

Exercise 2

You are developing a large-scale program and intend to use two commercial libraries, `tool1` and `tool2`. The names of types, functions, macros, and so on are declared in the header files `tool1.h` and `tool2.h` for users of these libraries.

Unfortunately, the libraries use the same global names in part. In order to use both libraries, you will need to define namespaces.

Write the following program to simulate this situation:

- Define an inline function called `calculate()` that returns the sum of two numbers for the header file `tool1.h`. The function interface is as follows:

```
double calculate(double num1, double num2);
```

- Define an inline function called `calculate()` that returns the product of two numbers for a second header file `tool2.h`. This function has the same interface as the function in `tool1.h`.
- Then write a source file containing a `main` function that calls both functions with test values and outputs the results.

  To resolve potential naming conflicts, define the namespaces `TOOL1` and `TOOL2` that include the relevant header files.

☐ Passing by Reference

A *pass by reference* can be programmed using references or pointers as function parameters. It is syntactically simpler to use references, although not always permissible.

A parameter of a reference type is an alias for an argument. When a function is called, a reference parameter is initialized with the object supplied as an argument. The function can thus directly manipulate the argument passed to it.

Example: `void test( int& a) { ++a; }`

Based on this definition, the statement

```
test( var);      // For an int variable var
```

increments the variable `var`. Within the function, any access to the reference `a` automatically accesses the supplied variable, `var`.

If an object is passed as an argument when passing by reference, the object is not copied. Instead, the address of the object is passed to the function internally, allowing the function to access the object with which it was called.

☐ Comparison to Passing by Value

In contrast to a normal *pass by value* an expression, such as `a+b`, cannot be used as an argument. The argument must have an address in memory and be of the correct type.

Using references as parameters offers the following *benefits*:

- arguments are not copied. In contrast to passing by value, the run time of a program should improve, especially if the arguments occupy large amounts of memory
- a function can use the reference parameter to return *multiple* values to the calling function. Passing by value allows only one result as a return value, unless you resort to using global variables.

If you need to read arguments, but not copy them, you can define a *read-only reference* as a parameter.

Example: `void display( const string& str);`

The function `display()` contains a string as an argument. However, it does not generate a new string to which the argument string is copied. Instead, `str` is simply a reference to the argument. The caller can rest assured that the argument is not modified within the function, as `str` is declared as a `const`.

Every C++ expression belongs to a certain type and also has a value, if the type is not `void`. Reference types are also valid for expressions.

☐ The Stream Class Shift Operators

The `<<` and `>>` operators used for stream input and output are examples of expressions that return a reference to an object.

Example:
```
cout << " Good morning "
```

This expression is not a `void` type but a reference to the object `cout`, that is, it represents the object `cout`. This allows you to repeatedly use the `<<` on the expression:

```
cout << "Good morning" << '!'
```

The expression is then equivalent to

```
(cout << " Good morning ") << '!'
```

Expressions using the `<<` operator are composed from left to right, as you can see from the table of precedence contained in the appendix.

Similarly, the expression `cin >> variable` represents the stream `cin`. This allows repeated use of the `>>` operator.

Example:
```
int a;  double x;
cin >> a >> x;        // (cin >> a) >> x;
```

☐ Other Reference Type Operators

Other commonly used reference type operators include the simple assignment operator `=` and compound assignments, such as `+=` and `*=`. These operators return a reference to the operand on the left. In an expression such as

```
a = b  or  a += b
```

`a` must therefore be an object. In turn, the expression itself represents the object `a`. This also applies when the operators refer to objects belonging to class types. However, the class definition stipulates the available operators. For example, the assignment operators `=` and `+=` are available in the standard class `string`.

Example:
```
string name("Jonny ");
name += "Depp";           //Reference to name
```

Since an expression of this type represents an object, the expression can be passed as an argument to a function that is called by reference. This point is illustrated by the example on the opposite page.

Efficient program logic often requires access to the memory addresses used by a program's data, rather than manipulation of the data itself. Linked lists or trees whose elements are generated dynamically at runtime are typical examples.

☐ Pointers

A *pointer* is an expression that represents both the *address* and *type* of another object. Using the address operator, `&`, for a given object creates a pointer to that object. Given that `var` is an `int` variable,

Example:
```
&var          // Address of the object var
```

is the address of the `int` object in memory and thus a pointer to `var`. A pointer points to a memory address and simultaneously indicates by its type how the memory address can be read or written to. Thus, depending on the type, we refer to *pointers to char*, *pointers to int*, and so on, or use an abbreviation, such as *char pointer*, *int pointer*, and so on.

☐ Pointer Variables

An expression such as `&var` is a constant pointer; however, C++ allows you to define *pointer variables*, that is, variables that can store the address of another object.

Example:
```
int *ptr;             // or:  int* ptr;
```

This statement defines the variable `ptr`, which is an `int*` type (in other words, *a pointer to int*). `ptr` can thus store the address of an `int` variable. In a declaration, the star character `*` always means "pointer to."

Pointer types are derived types. The general form is `T*`, where `T` can be any given type. In the above example `T` is an `int` type.

Objects of the same base type `T` can be declared together.

Example:
```
int a, *p, &r = a;  // Definition of a, p, r
```

After declaring a pointer variable, you must point the pointer at a memory address. The program on the opposite page does this using the statement

```
ptr = &var;.
```

☐ References and Pointers

References are similar to pointers: both refer to an object in memory. However, a pointer is not merely an alias but an individual object that has an identity separate from the object it references. A pointer has its own memory address and can be manipulated by pointing it at a new memory address and thus referencing a different object.

```
// Function circle(): Compute circumference and area.
void circle( const double& r, double& u, double& f)
{
   const double pi = 3.1415926536;
   u = 2 * pi * r;
   f = pi * r * r;
}
```

Exercise 3

```
// ------------------------------------------------------
// swap.cpp
// Definition and call of the function swap().
// 1. version: parameters with pointer type,
// 2. version: parameters with reference type.
// ------------------------------------------------------
#include <iostream>
using namespace std;

void swap( float*, float*);      // Prototypes of swap()
void swap( float&, float&);

int main()
{
   float x = 11.1F;
   float y = 22.2F;

   cout << "x and y before swapping:   "
        << x << "   " << y << endl;

   swap( &x, &y);            // Call pointer version.

   cout << "x and y after 1. swapping: "
        << x << "   " << y << endl;

   swap( x, y);              // Call reference version.

   cout << "x and y after 2. swapping: "
        << x << "   " << y << endl;

   return 0;
}

void swap(float *p1, float *p2)     // Pointer version
{
  float temp;                       // Temporary variable

   temp = *p1;                      // Above call points p1
   *p1  = *p2;                      // to x and p2 to y.
   *p2  = temp;
}
```

■ USING OBJECTS

Sample program

```
// account_t.cpp
// Uses objects of class Account.
// ------------------------------------------------

#include "Account.h"

int main()
{
   Account current1, current2;

   current1.init("Cheers, Mary", 123456, -1200.99);
   current1.display();

//   current1.balance += 100; // Error: private member

   current2 = current1;          // ok: Assignment of
                                 // objects is possible.
   current2.display();           // ok

                                 // New values for current2
   current2.init("Jones, Tom", 3512347, 199.40);

   current2.display();
                                   // To use a reference:
   Account& mtr = current1;        // mtr is an alias name
                                   // for object current1.
   mtr.display();                  // mtr can be used just
                                   // as object current1.
   return 0;
}
```

☐ Class Member Access Operator

An application program that manipulates the objects of a class can access only the `public` members of those objects. To do so, it uses the *class member access operator* (in short: *dot operator*).

Syntax: `object.member`

Where `member` is a data member or a method.

Example:
```
Account current;
current.init("Jones, Tom",1234567,-1200.99);
```

The expression `current.init` represents the `public` method `init` of the `Account` class. This method is called with three arguments for `current`.

The `init()` call cannot be replaced by direct assignments.

Example:
```
current.name = "Dylan, Bob";   // error
current.nr   = 1234567;        // private
current.balance = 1200.49;     // members
```

Access to the `private` members of an object is not permissible outside the class. It is therefore impossible to display single members of the `Account` class on screen.

Example:
```
cout << current.balance;          // Error
current.display();                // ok
```

The method `display()` displays all the data members of `current`. A method such as `display()` can only be called for one object. The statement

```
display();
```

would result in an error message, since there is no global function called `display()`. What data would the function have to display?

☐ Assigning Objects

The assignment operator `=` is the only operator that is defined for all classes by default. However, the source and target objects must both belong to the same class. The assignment is performed to assign the individual data members of the source object to the corresponding members of the target object.

Example:
```
Account current1, current2;
current2.init("Marley, Bob",350123, 1000.0);
current1 = current2;
```

This copies the data members of `current2` to the corresponding members of `current1`.

■ structs

Sample program

```
// structs.cpp
// Defines and uses a struct.
// ------------------------------------------------
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
struct Representative  // Defining struct Representative
{
  string name;         // Name of a representative.
  double sales;        // Sales per month.
};
inline void print( const Representative& v)
{
   cout << fixed << setprecision(2)
        << left  << setw(20) << v.name
        << right << setw(10) << v.sales << endl;
}
int main()
{
   Representative rita, john;
   rita.name    = "Strom, Rita";
   rita.sales   = 37000.37;
   john.name    = "Quick, John";
   john.sales = 23001.23;

   rita.sales += 1700.11;             // More Sales
   cout << "  Representative            Sales\n"
        << "-------------------------------" << endl;
   print( rita);
   print( john);
   cout << "\nTotal of sales: "
        << rita.sales + john.sales << endl;
   Representative *ptr = &john;          // Pointer ptr.
                                         // Who gets the
   if( john.sales < rita.sales)          // most sales?
     ptr = &rita;
   cout << "\nSalesman of the month: "
        << ptr->name << endl;    // Representative's name
                                 // pointed to by ptr.
   return 0;
}
```

A class typically contains multiple methods that fulfill simple tasks, such as reading or updating data members. This is the only way to ensure data encapsulation and class functionality.

However, continually calling "short" methods can impact a program's runtime. In fact, saving a re-entry address and jumping to the called function and back into the calling function can take more time than executing the function itself. To avoid this overhead, you can define `inline` methods in a way similar to defining `inline` global functions.

☐ Explicit and Implicit `inline` Methods

Methods can be explicitly or implicitly defined as `inline`. In the first case the method is declared within the class, just like any other method. You simply need to place the `inline` keyword before the method name in the function header when defining the method.

Example:

```
inline void Account::display()
{
    . . .
}
```

Since the compiler must have access to the code block of an `inline` function, the `inline` function should be defined in the header containing the class definition.

Short methods can be defined within the class. Methods of this type are known as implicit `inline` methods, although the `inline` keyword is not used.

Example:

```
                    // Within class Account:
bool isPositive(){ return state > 0; }
```

☐ Constructors and Destructors with `inline` Definitions

Constructors and destructors are special methods belonging to a class and, as such, can be defined as `inline`. This point is illustrated by the new definition of the `Account` class opposite. The constructor and the destructor are both implicit `inline`. The constructor has a default value for each argument, which means that we also have a default constructor. You can now define objects without supplying an initialization list.

Example:

```
Account temp;
```

Although we did not explicitly supply values here, the object `temp` was correctly initialized by the default constructor we defined.

☐ Accessing `const` Objects

If you define an object as `const`, the program can only read the object. As mentioned earlier, the object must be initialized when you define it for this reason.

Example: `const Account inv("YMCA, FL", 5555, 5000.0);`

The object `inv` cannot be modified at a later stage. This also means that methods such as `setName()` cannot be called for this object. However, methods such as `getName` or `display()` will be similarly unavailable although they only perform read access with the data members.

The reason for this is that the compiler cannot decide whether a method performs write operations or only read operations with data members unless additional information is supplied.

☐ Read-Only Methods

Methods that perform only read operations and that you need to call for constant objects must be identified as read-only. To identify a method as read-only, append the `const` keyword in the method declaration and in the function header for the method definition.

Example: `unsigned long getNr() const;`

This declares the `getNr()` method as a *read-only method* that can be used for constant objects.

Example: `cout << "Account number: " << inv.getNr();`

Of course, this does not prevent you from calling a read-only method for a non-constant object.

The compiler issues an error message if a read-only method tries to modify a data member. This also occurs when a read-only method calls another method that is not defined as `const`.

☐ `const` and Non-`const` Versions of a Method

Since the `const` keyword is part of the method's signature, you can define two versions of the method: a read-only version, which will be called for constant objects by default, and a normal version, which will be called for non-`const` objects.

■ STANDARD METHODS

Sample program

```
// stdMeth.cpp
// Using standard methods.
// ------------------------------------------------
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

class CD
{  private:
     string interpret, title;
     long   seconds;                  // Time duration of a song
   public:
     CD( const string i="", const string& t="", long s = 0L)
     {  interpret = i;    title = t;    seconds = s;
     }
     const string getInterpret() const{ return interpret; }
     const string& getTitle() const    { return title; }
     long  getSeconds() const          { return seconds; }
};
// Generate objects of class CD and output it in tabular form
void printLine( CD cd) ;             // A row of the table
int main()
{
   CD cd1( "Mister X", "Let's dance", 30*60 + 41),
      cd2( "New Guitars", "Flamenco Collection", 2772 ),
      cd3 = cd1,                     // Copy constructor!
      cd4;                           // Default constructor.
      cd4 = cd2;                     // Assignment!
   string line( 70,'-');   line += '\n';
   cout << line << left
        << setw(20) << "Interpreter" << setw(30) << "Title"
        << "Length (Min:Sec)\n" << line << endl;
   printLine(cd3);                   // Call by value ==>
   printLine(cd4);                   // Copy constructor!
   return 0;
}
void printLine( CD cd)
{   cout << left   << setw(20) << cd.getInterpret()
                   << setw(30) << cd.getTitle()
         << right << setw(5)  << cd.getSeconds() / 60
         << ':'   << setw(2)  << cd.getSeconds() % 60 << endl;
}
```

```
// ------------------------------------------------------
// article_t.cpp
// Tests the Article class.
// ------------------------------------------------------
#include "Article.h"      // Definition of the class
#include <iostream>
#include <string>
using namespace std;

void test();

// -- Creates and destroys objects of Article class --
Article Article1( 1111,"volley ball", 59.9);
int main()
{
   cout << "\nThe first statement in main().\n" << endl;
   Article Article2( 2222,"gym-shoes", 199.99);
   Article1.print();
   Article2.print();
   Article& shoes = Article2;          // Another name
   shoes.setNr( 2232);
   shoes.setName("Jogging-shoes");
   shoes.setSP( shoes.getSP() - 50.0);

   cout << "\nThe new values of the shoes object:\n";
   shoes.print();
   cout << "\nThe first call to test()." << endl;
   test();
   cout << "\nThe second call to test()." << endl;
   test();
   cout << "\nThe last statement in main().\n" << endl;
   return 0;
}

void test()
{
   Article shirt( 3333, "T-Shirt", 29.9);
   shirt.print();
   static Article net( 4444, "volley ball net", 99.0);
   net.print();
   cout << "\nLast statement in function test()"
        << endl;
}
```

```
inline bool Date::isLess( const Date& d) const
{
   if( year != d.year)          return year < d.year;
   else if( month != d.month)   return month < d.month;
   else                         return day < d.day;
}

inline bool isLeapYear( int year)
{
   return (year%4 == 0 && year%100 != 0) || year%400 == 0;
}
#endif   //  _DATE_


// ------------------------------------------------------
// Date.cpp
// Implements those methods of Date class,
// which are not defined inline.
// ------------------------------------------------------
#include "Date.h"                // Class definition
#include <iostream>
#include <sstream>
#include <iomanip>
#include <string>
#include <ctime>
using namespace std;

// ------------------------------------------------------
void Date::setDate()     // Get the present date and
{                        // assign it to the data members.
   struct tm *dur;           // Pointer to struct tm.
   time_t sec;               // For seconds.

   time(&sec);               // Get the present time.
   dur = localtime(&sec);    // Initialize a struct of
                             // type tm and return a
                             // pointer to it.
   day   = (short) dur->tm_mday;
   month = (short) dur->tm_mon + 1;
   year  = (short) dur->tm_year + 1900;
}
```

```
// ------------------------------------------------
// article.cpp
// Methods of Article, which are not defined as inline.
// Constructor and destructor output when called.
// ------------------------------------------------

#include "article.h"            // Definition of the class

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

// Defining the static data member:
int Article::countObj = 0;          // Number of objects

// Defining the constructor and destructor:

Article::Article( long nr, const string& name, double sp)
{
   setNr(nr);   setName(name);   setSp(sp);
   ++countObj;
   cout << "An article \"" << name
        << "\" is created.\n"
        << "This is the " << countObj << ". article!"
        << endl;
}
// Defining the copy constructor:
Article::Article( const Article& art)
:nr(art.nr), name(art.name), sp(art.sp)
{
   ++countObj;
   cout << "A copy of the article \"" << name
        << "\" is generated.\n"
        << "This is the " << countObj << ". article!"
        << endl;
}

Article::~Article()
{
   cout << "The article \"" << name
        << "\" is destroyed.\n"
        << "There are still " << --countObj << " articles!"
        << endl;
}

// The method print() outputs an article.
void Article::print()
{
    // As before! Compare to the solutions of chapter 14.
}
```

■ 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;
}
```

solutions

■ SOLUTIONS

Exercise 1

```
// ------------------------------------------------
// 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 << "\nSorting Integers \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;
       }
     }
   }
```

```
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 != 'Q')
  {
    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;
```

■ ARRAYS AND POINTERS (1)

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'
        << 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!
```

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`.

☐ 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
{                               // of str without '\0'.
   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.
```

```
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.
```

■ 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;
       exit( 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;
}
```

■ 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;
}
```

exercises

■ EXERCISES

For exercise 1

Possible calls to the program `fcopy`:

`fcopy file1 file2`

A file, `file1`, is copied to `file2`. If `file2` already exists, it is overwritten.

`fcopy file1`

A file, `file1`, is copied to standard output, that is, to the screen if standard output has not been redirected.

`fcopy`

For calls without arguments, the source and destination files are entered in a user dialog.

More details on the istream class method `read()`

If `is` is a file stream that references a file opened for reading, the following call

Example:
```
char buf[1024];
is.read(buf, 1024);
```

transfers the next 1024 bytes from file to the buffer `buf`. Provided that no error occurs, no less than 1024 bytes will be copied unless end-of-file is reached. In this case the `fail` and `eof` bits are set. The last block of bytes to be read also has to be written to the destination file. The method `gcount()` returns the number of bytes transferred by the last read operation.

Example:
```
int nread = is.gcount();     // Number of bytes
                             // in last read op.
```

Exercise 4

The program `TelList`, which was written as an exercise for Chapter 16, needs to be modified to allow telephone lists to be saved in a file.

To allow this, first add the data members and methods detailed on the opposite page to `TelList`. The string `filename` is used to store the name of the file in use. The dirty flag is raised to indicate that the phone list has been changed but not saved. You will need to modify the existing methods `append()` and `erase()` to provide this functionality.

The strings in the phone list must be saved as C strings in a binary file, allowing for entries that contain several lines.

Add the following items to the application program menu:

```
O = Open a file
```

Read a phone list previously stored in a file.

```
W = Save
```

Save the current phone list in a file.

```
U = Save as . . .
```

Save the current phone list in a new file.

Choosing one of these menu items calls one of the following methods as applicable: `load()`, `save()` or `saveAs()`. These methods return `true` for a successful action and `false` otherwise. The user must be able to supply a file name for the `save()` method, as the list may not have been read from a file previously.

If the phone list has been modified but not saved, the user should be prompted to save the current phone list before opening another file or terminating the program.

```
bool TelList::append( const string& name,
                      const string& telNr)
{
    if( count < MAX                   // Any space
        && name.length() > 1          // minimum 2 characters
        && search(name) == PSEUDO)    // does not exist
    {
      v[count].name  = name;
      v[count].telNr = telNr;
      ++count;
      dirty = true;
      return true;
    }
    return false;
}

bool TelList::erase( const string& key )
{
   int i = search(key);
   if( i != PSEUDO)
   {
      if( i != count-1 )              // Copy the last element
         v[i] = v[count-1];           // to position i.
     --count;
     dirty = true;
     return true;
   }
   return false;
}
// ------------------------------------------------
// Methods search(), print(), getNewEntries()
// are unchanged (refer to solutions of chapter 16).
// ------------------------------------------------

// Methods for loading and saving the telephone list.

bool TelList::load()
{
   cout << "\n--- Load the telephone list "
        << "from a file. ---" << "\nFile: ";

   string file;                       // Input file name.
   cin.sync(); cin.clear();           // No previous input
   getline( cin, file);
   if( file.empty())
   {
      cerr << "No filename declared!" << endl;
      return false;
   }
```

```
inline void go_on()
{
   cout << "\n\nGo on with return! ";
   cin.sync();  cin.clear();        // No previous input
   while( cin.get() != '\n')
       ;
}
int menu();                 // Enter a command
char askForSave();          // Prompt user to save.
char header[] =
"\n\n          * * * * *  Telephone List  * * * * *\n\n";
TelList myFriends;         // A telephone list

int main()
{
  int action = 0;           // Command
  string name;              // Read a name
  while( action != 'E')
  {
    action = menu();
    cls();  cout << header << endl;
    switch( action)
    {
// ------------------------------------------------------
//   case 'S':  case 'F':  case 'A':  case 'D':
//   unchanged (refer to the solutions of chapter 16).
// ------------------------------------------------------
     case 'O':                          // To open a file
        if(myFriends.isDirty() && askForSave() == 'y')
           myFriends.save();
        if( myFriends.load())
           cout << "Telephone list read from file "
                << myFriends.getFilename() <<"!"
                << endl;
        else
           cerr << "Telephone list not read!"
                << endl;
        go_on();
        break;
     case 'U':                          // Save as ...
        if( myFriends.saveAs())
          cout << "Telephone list has been saved in file: "
               << myFriends.getFilename() << " !" <<endl;
        else
           cerr << "Telephone list not saved!" << endl;
        go_on();
        break;
```

■ OPERATOR FUNCTIONS (1)

Operators < and ++ for class DayTime

```
// DayTime.h
// The class DayTime containing operators < and ++ .
// ------------------------------------------------
#ifndef _DAYTIME_
#define _DAYTIME_
class DayTime
{
  private:
     short hour, minute, second;
     bool overflow;
  public:
    DayTime( int h = 0, int m = 0, int s = 0);
    bool setTime(int hour, int minute, int second = 0);
    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 operator<( const DayTime& t) const // compare
    {                                       // *this and t
      return  asSeconds() < t.asSeconds();
    }
    DayTime& operator++()           // Increment seconds
    {
       ++second;                    // and handle overflow.
      return  *this;
    }
    void print() const;
};
#endif   // _DAYTIME_
```


Calling the Operator <

```
#include "DayTime.h"
  . . .
  DayTime depart1( 11, 11, 11), depart2(12,0,0);
  . . .
  if( depart1 < depart2 )
    cout << "\nThe 1st plane takes off earlier!" << endl;
  . . .
```

☐ Naming Operator Functions

To overload an operator, you just define an appropriate *operator function*. The operator function describes the actions to be performed by the operator. The name of an operator function must begin with the `operator` keyword followed by the operator symbol.

Example: `operator+`

This is the name of the operator function for the + operator.

An operator function can be defined as a global function or as a class method. Generally, operator functions are defined as methods, especially in the case of unary operators. However, it can make sense to define an operator function globally. This point will be illustrated later.

☐ Operator Functions as Methods

If you define the operator function for a *binary* operator as a method, the left operand will always be an object of the class in question. The operator function is called for this object. The second, right operand is passed as an argument to the method. The method thus has a single parameter.

Example: `bool operator<( const DayTime& t) const;`

In this case the lesser than operator is overloaded to compare two `DayTime` objects. It replaces the method `isLess()`, which was formerly defined for this class.

The prefix operator `++` has been overloaded in the example on the opposite page to illustrate overloading unary operators. The corresponding operator function in this class has no parameters. The function is called if the object `a` in the expression `++a` is an object of class `DayTime`.

☐ Calling an Operator Function

The example opposite compares two times of day:

Example: `depart1 < depart2`

The compiler will attempt to locate an applicable operator function for this expression and then call the function. The expression is thus equivalent to

```
depart1.operator<( depart2)
```

Although somewhat uncommon, you can call an operator function explicitly. The previous function call is therefore technically correct.

Programs that use operators are easier to encode and read. However, you should be aware of the fact that an operator function should perform a similar operation to the corresponding operator for the fundamental type. Any other use can lead to confusion.

☐ Calling Operator Functions

The following expressions are valid for the operators in the Euro class.

Example:
```
Euro  wholesale(15,30), retail,
      profit(7,50), discount(1,75);
retail = wholesale + profit;
// Call:  wholesale.operator+( profit)
retail -= discount;
// Call:  retail.operator-=( discount)
retail += Euro( 1.49);
// Call:  retail.operator+=( Euro(1.49))
```

These expressions contain only Euro type objects, for which operator functions have been defined. However, you can also add or subtract int or double types. This is made possible by the Euro constructors, which create Euro objects from int or double types. This allows a function that expects a Euro value as argument to process int or double values.

As the program opposite shows, the statement

Example:
```
retail += 0.99;
```

is valid. The compiler attempts to locate an operator function that is defined for both the Euro object and the double type for +=. Since there is no operator function with these characteristics, the compiler converts the double value to Euro and calls the existing operator function for euros.

☐ Symmetry of Operands

The available constructors also allow you to call the operator functions of + and – with int or double type arguments.

Example:
```
retail = wholesale + 10;        // ok
wholesale = retail - 7.99;      // ok
```

The first statement is equivalent to

```
retail = wholesale.operator+( Euro(10));
```

But the following statement is invalid!

Example:
```
retail = 10 + wholesale;      // wrong!
```

Since the operator function was defined as a method, the left operand must be a class object. Thus, you cannot simply exchange the operands of the operator +. However, if you want to convert both operands, you will need global definitions for the operator functions.

■ FRIEND FUNCTIONS

Class Euro with friend functions

```
// Euro.h
// The class Euro with operator functions
// declared as friend functions.
// ------------------------------------------------
#ifndef _EURO_H_
#define _EURO_H_
// ....
class Euro
{
  private:
    long  data;              // Euros * 100 + Cents
   public:
    // Constructors and other methods as before.
    // Operators - (unary), +=, -= as before.
    // Division Euro / double :
    Euro operator/( double x)          // Division *this/x
    {                                  // =  *this * (1/x)
       return (*this * (1.0/x));
    }
    // Global friend functions
    friend Euro operator+( const Euro& e1, const Euro& e2);
    friend Euro operator-( const Euro& e1, const Euro& e2);
    friend Euro operator*( const Euro& e, double x)
    {
       Euro temp( ((double)e.data/100.0) * x) ;
       return temp;
    }
    friend Euro operator*( double x, const Euro& e)
    {
       return e * x;
    }
};
// Addition:
inline Euro operator+( const Euro& e1, const Euro& e2)
{
    Euro temp;   temp.data = e1.data + e2.data;
    return  temp;
}
// Subtraction:
inline Euro operator-( const Euro& e1, const Euro& e2)
{
    Euro temp;   temp.data = e1.data - e2.data;
    return  temp;
}
#endif   // _EURO_H_
```

When outputting a `Euro` class object, `price`, on screen, the following output statement causes a compiler error:

Example: `cout << price;`

`cout` can only send objects to standard output if an output function has been defined for the *type* in question—and this, of course, is not the case for user-defined classes.

However, the compiler can process the previous statement if it can locate a suitable operator function, `operator<<()`. To allow for the previous statement, you therefore need to define a corresponding function.

☐ Overloading the << Operator

In the previous example, the left operand of `<<` is the object `cout`, which belongs to the `ostream` class. Since the standard class `ostream` should not be modified, it is necessary to define a global operator function with two parameters. The right operand is a `Euro` class object. Thus the following prototype applies for the operator function:

Prototype: `ostream& operator<<(ostream& os, const Euro& e);`

The return value of the operator function is a reference to `ostream`. This allows for normal concatenation of operators.

Example: `cout << price << endl;`

☐ Overloading the >> Operator

The `>>` operator is overloaded for input to allow for the following statements.

Example:
```
cout << "Enter the price in Euros: "
cin >> price;
```

The second statement causes the following call:

```
operator>>( cin, price);
```

As `cin` is an object of the standard `istream` class, the first parameter of the operator function is declared as a reference to `istream`. The second parameter is again a reference to `Euro`.

The header file `Euro.h` contains only the declarations of `<<` and `>>`. To allow these functions to access the private members of the `Euro` class, you can add a `friend` declaration within the class. However, this is not necessary for the current example.

Exercise 1

The < and ++ operators for the sample class `DayTime` were overloaded at the beginning of this chapter. Now modify the class as follows:

- Overload the relational operators

  `<` `>` `<=` `>=` `==` and `!=`

  and the shift operators

  `>>` and `<<` for input and output

  using global operator functions. You can define these `inline` in the header file.
- Then overload both the prefix and postfix versions of the `++` and `--` operators. The operator functions are methods of the class. The `--` operator decrements the time by one second. The time is not decremented after reaching 0:0:0.
- Write a main function that executes all the overloaded operators and displays their results.

Exercise 2

You are to develop a class that represents fractions and performs typical arithmetic operations with them.

- Use a header file called `fraction.h` to define the `Fraction` class with a numerator and a denominator of type `long`. The constructor has two parameters of type `long`: the first parameter (numerator) contains the default value 0, and the second parameter (denominator) contains the value 1. Declare operator functions as methods for `-` (unary), `++` and `--` (prefix only), `+=`, `-=`, `*=`, and `/=`. The operator functions of the binary operators `+`, `-`, `*`, `/` and the input / output operators `<<`, `>>` are to be declared as `friend` functions of the `Fraction` class.
- Implement the constructor for the `Fraction` class to obtain a positive value for the denominator at all times. If the denominator assumes a value of 0, issue an error message and terminate the program. Then write the operator functions. The formulae for arithmetic operations are shown opposite.
- Then write a `main` function that calls all the operators in the `Fraction` class as a test application. Output both the operands and the results.

Exercise

Enhance the numerical class `Fraction`, which you know from the last chapter, to convert both `double` values to fractions and fractions to `double`. In addition, fractions should be rounded after arithmetic operations.

- First declare the `simplify()` method for the `Fraction` class and insert the definition on the opposite page in your source code. The method computes the largest common divisor of numerator and denominator. The numerator and the denominator are then divided by this value.
- Add an appropriate call to the `simplify()` function to all operator functions (except `++` and `--`).
- Then add a conversion constructor with a `double` type parameter to the class.

  Example: `Fraction b(0.5); // yields the fraction 1/2`

  `Double` values should be converted to fractions with an accuracy of three decimal places. The following technique should suffice for numbers below one million. Multiply the `double` value by `1000` and add `0.5` for rounding. Assign the result to the numerator. Set the value of the denominator to `1000`. Then proceed to simplify the fraction.
- You now have a conversion constructor for `long` and `double` types. To allow for conversion of `int` values to fractions, you must write your own conversion constructor for `int`!
- Now modify the class to allow conversion of a fraction to a `double` type number. Define the appropriate conversion function `inline`.

Use the function `main()` to test various type conversions. More specifically, use assignments and arithmetic functions to do so. Also compute the sum of a fraction and a floating-point number.

Output the operands and the results on screen.

```
// ---------------------------------------------------------
// Fraction.cpp
// Defines methods and friend functions
// that are not inline.
// ---------------------------------------------------------

#include <iostream.h>
#include <stdlib.h>
#include "Fraction.h"

// Constructors:
Fraction::Fraction(long z, long n)
{
   // Unchanged! Same as in Chapter 19.
}

Fraction::Fraction( double x)
{
   x *= 1000.0;
   x += (x >= 0.0) ? 0.5 : -0.5;   // Round the 4th digit.
   numerator = (long)x;
   denominator = 1000;
   simplify();
}

Fraction operator+(const Fraction& a, const Fraction& b )
{
   Fraction temp;

   temp.denominator = a.denominator * b.denominator;
   temp.numerator = a.numerator*b.denominator
                  + b.numerator * a.denominator;
   temp.simplify();
   return temp;
}

// The functions
//  operator-()    operator<<()     operator>>()
// are left unchanged.

// The functions
//  operator*()   and   operator/()
// are completed by a call to temp.simplify()
// just like the function operator+().
//

// The code of method Fraction::simplify(), as
// specified in the exercise, should be here.
```

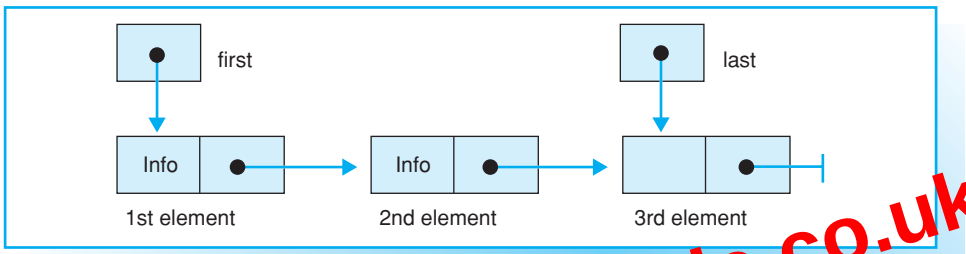
■ DYNAMIC STORAGE ALLOCATION FOR CLASSES

Sample program

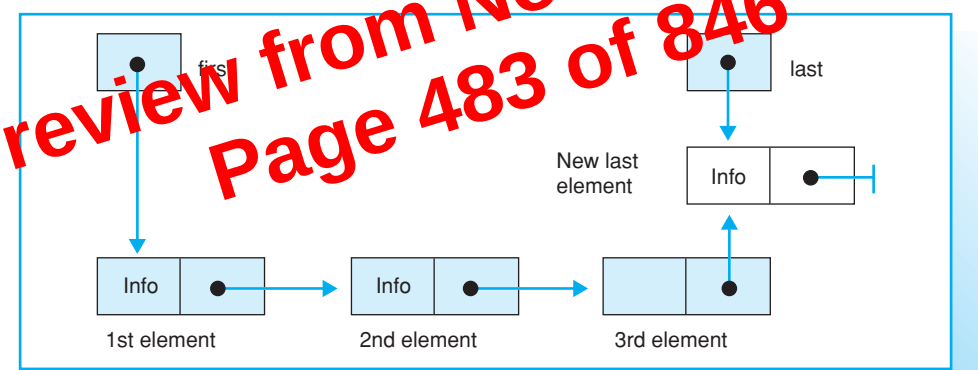
```
// DynObj.cpp
// The operators new and delete for classes.
// ------------------------------------------------
#include "account.h"
#include <iostream>
using namespace std;

Account *clone( const Account* pK);  // Create a copy
                                     // dynamically.
int main()
{
   cout << "Dynamically created objects.\n" << endl;

   // To allocate storage:
   Account *ptrA1, *ptrA2, *ptrA3;

   ptrA1 = new Account;              // With default constructor
   ptrA1->display();                 // Show default values.

   ptrA1->setNr(302010);             // Set the other
   ptrA1->setName("Tang, Ming");     // values by access
   ptrA1->setStand(2345.87);         // methods.
   ptrA1->display();                 // Show new values.

   // Use the constructor with three arguments:
   ptrA2 = new Account("Xiang, Zhang", 7531357, 999.99);
   ptrA2->display();                 // Display new account.

   ptrA3 = clone( ptrA1);            // Pointer to a dyna-
                                     // mically created copy.
   cout << "Copy of the first account: " << endl;
   ptrA3->display();                 // Display the copy.

   delete ptrA1;                     // Release memory
   delete ptrA2;
   delete ptrA3;

  return 0;
}

Account *clone( const Account* pK)   // Create a copy
{                                    // dynamically.
    return new Account(*pK);
}
```

■ APPLICATION: LINKED LISTS

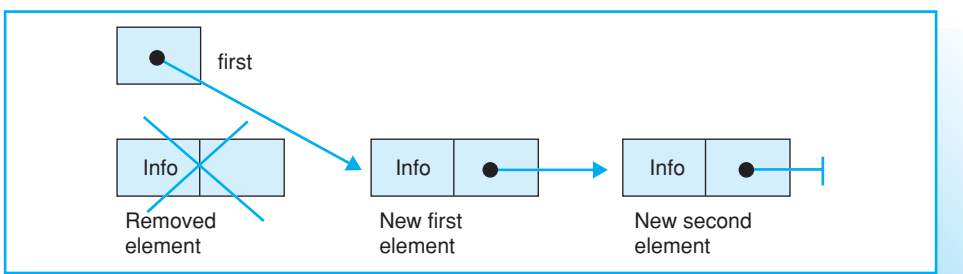
A simple linked list



Appending a list element



Deleting a list element



■ REPRESENTING A LINKED LIST

Classes of header file `List.h`

```
// List.h
// Defines the classes ListEl and List.
// ---------------------------------------------------
#ifndef _LISTE_H_
#define _LISTE_H_
#include "Date.h"          // Class Date from Chapter 14
#include <iostream>
#include <iomanip>
using namespace std;

class ListEl                    // A list element.
{
  private:
    Date   date;                // Date
    double amount;              // Amount of money
    ListEl* next;               // Pointer to successor

  public:
    ListEl( Date   d = Date(1,1,1), double  b = 0.0,
            ListEl* p = NULL)
            : date(d), amount(b), next(p) {}
    // Access methods:
    // getDate(), setDate(), getAmount(), setAmount()
    ListEl* getNext()  const { return next; }
    friend class List;
};

// ---------------------------------------------------
// Defining the class List
class List
{
  private:
    ListEl* first, *last;
  public:
    List(){ first = last = NULL; }  // Constructor
    ~List();                        // Destructor
    // Access to the first and last elements:
    ListEl* front() const { return first; }
    ListEl* back()  const { return last; }
    // Append a new element at the end of the list:
    void pushBack(const Date& d, double b);
    // Delete an element at the beginning of the list
    void popFront();
};
#endif  // _LIST_H_
```

Exercise 1

Write a global function called `splice()` that "splices" two `int` arrays together by first allocating memory for a dynamic array with enough room for both `int` arrays, and then copying the elements from both arrays to the new array, as follows:

- first, the elements of the first array are inserted up to a given position,
- then the second array is inserted,
- then the remainder of the first array is appended.

Arguments: The two `int` arrays, their length, and the position at which they are to be spliced.
Return value: A pointer to the new array

Exercise 2

Write a global function called `merge()` that merges two sorted `int` arrays by first allocating memory for a dynamic array with enough room for both `int` arrays and then inserting the elements of both arrays into the new array in sequence.

Arguments: The two `int` arrays and their length.
Return value: A pointer to the new array

To test the function, modify the program used to sort arrays in Exercise 4 of Chapter 17.

Exercise 3

Complete and test the implementation of a linked list found in this chapter.

- First define the access methods shown opposite. Then overload the `<<` operator for the class `ListEl` to allow formatted output of the data in the list elements. You can use the `asString()` in the date class to do so.
- Then implement the destructor for the `List` class. The destructor will release the memory used by all the remaining elements. Make sure that you read the pointer to the successor of each element before destroying it!
- Implement the methods `pushBack()` and `popFront()` used for appending and deleting list elements.
- Overload the operator `<<` in the `List` class to output all the data stored in the list.
- Test the `List` class by inserting and deleting several list elements and repeatedly outputting the list.

solutions

■ SOLUTIONS

Exercise 1

```
// ---------------------------------------------------
// Splice.cpp
// Implements the splice algorithm.
// ---------------------------------------------------
#include <iostream>
#include <iomanip>
#include <cstdlib>          // For srand() and rand()
#include <ctime>            // and for time().
using namespace std;

// Prototype:
int *splice( int v1[], int len1
             int v2[], int len2, int pos);

int main()
{
    cout << "\n  * * * Testing the splice function * * *\n"
         << endl;
    int i, len1 = 10, len2 = 5;
    int *a1 = new int[len1],
        *a2 = new int[len2];
   // Initialize the random number generator
   // with the current time:
    srand( (unsigned)time(NULL));

    for( i=0; i < len1; ++i)     // Initialize the arrays:
       a1[i] = rand();           // with positive and
    for( i=0; i < len2; ++i)
       a2[i] = -rand();          // negative numbers.

   // To output the array:
   cout << "1. array: " << endl;
   for( i = 0; i < len1; ++i)
      cout << setw(12) << a1[i];
   cout << endl;
   cout << "2. array: " << endl;
   for( i = 0; i < len2; ++i)
      cout << setw(12) << a2[i];
   cout << endl;
   cout << "\n At what position do you want to insert "
           "\n the 2nd array  into 1st  array?"
           "\n Possible positions: 0, 1, ..., " << len1
        << " : ";

   int pos;  cin >> pos;
```

☐ Dynamic Members

You can exploit the potential of dynamic memory allocation to leverage existing classes and create data members of variable length. Depending on the amount of data an application program really has to handle, memory is allocated as required while the application is running. In order to do this the class needs a pointer to the dynamically allocated memory that contains the actual data. Data members of this kind are also known as *dynamic members* of a class.

When compiling a program that contains arrays, you will probably not know how many elements the array will need to store. A class designed to represent arrays should take this point into consideration and allow for dynamically defined variable length arrays.

☐ Requirements

In the following section you will be developing a new version of the `FloatArr` class to meet these requirements and additionally allow you to manipulate arrays as easy as fundamental types. For example, a simple assignment should be possible for two objects `v1` and `v2` in the new class.

Example: `v2 = v1;`

The object `v2` itself—and not the programmer—will ensure that enough memory is available to accommodate the array `v1`.

Just as in the case of fundamental types, it should also be possible to use an existing object, `v2`, to initialize a new object, `v3`.

Example:
```
FloatArr v3(v2);
```

Here the object `v3` ensures that enough memory is available to accommodate the array elements of `v2`.

When an object of the `FloatArr` is declared, the user should be able to define the initial length of the array. The statement

Example:
```
FloatArr fArr(100);
```

allocates memory for a maximum of 100 array elements.

The definition of the `FloatArr` class therefore comprises a member that addresses a dynamically allocated array. In addition to this, two `int` variables are required to store the maximum and current number of array elements.

■ CLASSES WITH A DYNAMIC MEMBER

First version of class `FloatArr`

```
// floatArr.h : Dynamic array of floats.
// -------------------------------------------------
#ifndef _FLOATARR_
#define _FLOATARR_
class FloatArr
{
   private:
     float* arrPtr;     // Dynamic member
     int max;           // Maximum quantity without
                        // reallocation of new storage.
     int cnt;           // Number of array elements
   public:
     FloatArr( int = 5 );      // Constructor
     FloatArr( int, float val);
     ~FloatArr();              // Destructor
     int length() const { return cnt; }
     float& operator[](int i);   // Subscript operator.
     float operator[](int i) const;
     bool append(float val);     // Append value val.
     bool remove(int pos);       // Delete position pos.
};
#endif   // _FLOATARR_
```

Creating objects with dynamic members

```
#include "floatArr.h"
#include <iostream>
using namespace std;
int main()
{
   FloatArr v(10);         // Array v of 10 float values
   FloatArr w(20, 1.0F);   // To initialize array w of
                           // 20 float values with 1.0.
   v.append( 0.5F);
   cout << " Current number of elements in v: "
        << v.length() << endl;                  //   1
   cout << " Current number of elements in w: "
        << w.length() << endl;                  //  20
   return 0;
}
```

The next question you need to ask when designing a class to represent arrays is what methods are necessary and useful. You can enhance `FloatArr` class step by step by optimizing existing methods or adding new methods.

The first version of the `FloatArr` class comprises a few basic methods, which are introduced and discussed in the following section.

☐ Constructors

It should be possible to create an object of the `FloatArr` class with a given length and store a `float` value in the object, if needed. A constructor that expects an `int` value as an argument is declared for this purpose.

```
FloatArr(int n = 256);
```

The number `256` is the default argument for the length of the array. This provides for a default constructor that creates an array with 256 empty array elements.

An additional constructor

```
FloatArr( int n, int val );
```

allows you to define an array where the given value is stored in each array element. In this case you need to state the length of the array.

Example: `FloatArr arr( 100, 0.0F));`

This statement initializes the 100 elements in the array with a value of `0.0`.

☐ Additional Methods

The `length()` method allows you to query the number of elements in the array. `arr.length()` returns a value of `100` for the array `arr`.

You can overload the subscript operator `[]` to access individual array elements.

Example: `arr[i] = 15.0F;`

The index `i` must lie within the range `0` to `cnt-1`.

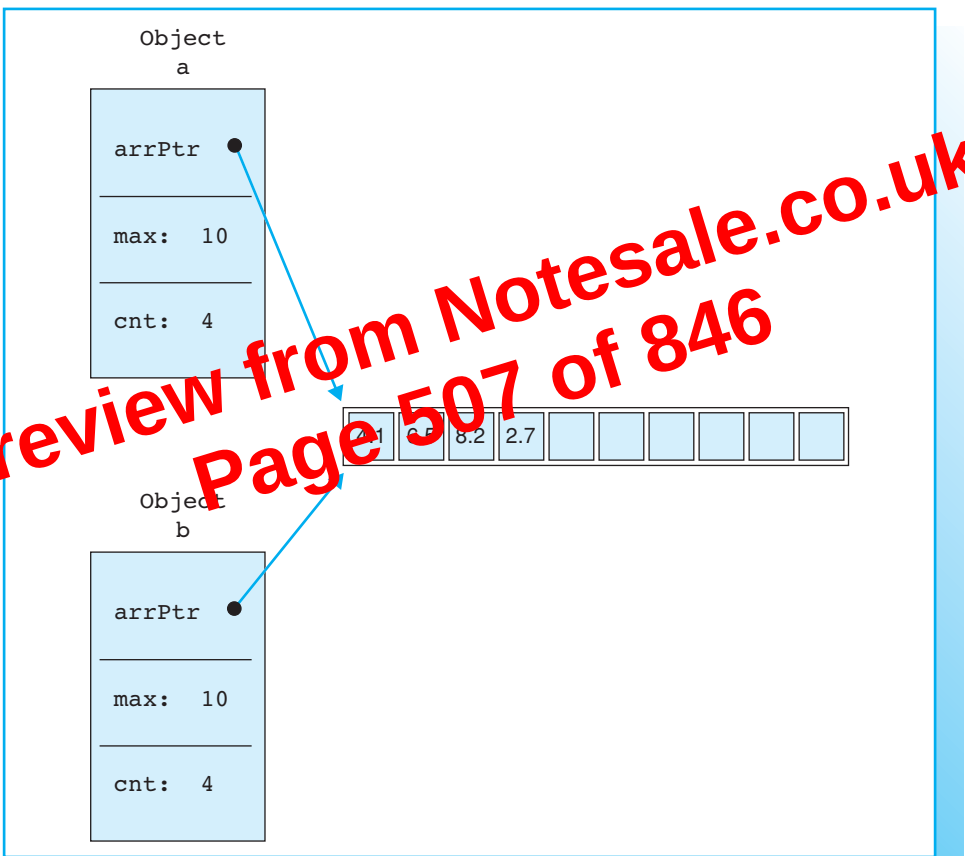
The `append()` method can be used to append a value to the array. The number of elements is then incremented by one.

When you call the `remove()` method it does exactly the opposite of `append()`—deleting the element at the stated position. This reduces the current count by one, provided a valid position was stated.

■ COPY CONSTRUCTOR

Effect of the standard copy constructor

```
FloatArr b(a);          // Creates a copy of a.
```



A self-defined copy constructor for class `FloatArr`

```
//floatArr.cpp: Implementing the methods.
// ------------------------------------------------------
FloatArr::FloatArr(const FloatArr& src)
{
   max = src.max;      cnt = src.cnt;
   arrPtr = new float[max];

   for( int i = 0; i < cnt; i++ )
      arrPtr[i] = src.arrPtr[i];
}
```

Each class comprises four implicitly defined default methods, which you can replace with your own definitions:

- the default constructor and the destructor
- the copy constructor and the standard assignment

In contrast to initialization by means of the copy constructor, which takes place when an object is defined, an assignment always requires an existing object. Multiple assignments, which modify an object, are possible.

☐ Default Assignment

Given that `v1` and `v2` are two `FloatArr` class objects, the following assignment is valid:

Example:
```
v1 = v2;                    // Possible, but risky
```

Default assignment is performed member by member. The data members of `v2` are copied to the corresponding data members of `v1` just like the copy constructor would copy them. However, this technique is not suitable for classes with dynamic members. This would simply point the pointers belonging to different objects at the same dynamic allocated memory. In addition, memory previously addressed by a pointer of the target object will be unreferenced after the assignment.

☐ Overloading the Assignment Operator

In other words, you need to overload the default assignment for classes containing dynamic members. Generally speaking, if you need to define a copy constructor, you will also need to define an assignment.

The operator function for the assignment must perform the following tasks:

- release the memory referenced by the dynamic members
- allocate sufficient memory and copy the source object's data to that memory.

The operator function is implemented as a class method and returns a reference to the target object allowing multiple assignments. The prototype of the operator function for the `FloatArr` class is thus defined as follows:

```
FloatArr& FloatArr::operator=( const FloatArr& src)
```

When implementing the operator function you must avoid self assignment, which would read memory areas that have already been released.

```
// ------------------------------------------------------
// FloatArr.cpp
// Implements the methods of FloatArr.
// ------------------------------------------------------

#include "floatArr.h"

//  Constructors, destructor, assignment,
//  and subscript operator unchanged.

//  ---  The new functions  ---

// Private auxiliary function to enlarge the array.
void FloatArr::expand( int new)
{
   if( newMax == max)
      return;
   max = newMax;
   if( newMax < cnt)
      cnt = newMax;
   float *temp = new float[newMax];
   for( int i = 0; i < cnt; ++i)
      temp[i] = arrPtr[i];

   delete[] arrPtr;
   arrPtr = temp;
}

// Append floating-point number or an array of floats.
void FloatArr::append( float val)
{
   if( cnt+1 > max)
       expand( cnt+1);

   arrPtr[cnt++] = val;
}

void FloatArr::append( const FloatArr& v)
{
   if( cnt + v.cnt > max)
       expand( cnt + v.cnt);

   int count = v.cnt;            // Necessary if v == *this

   for( int i=0; i < count; ++i)
     arrPtr[cnt++] = v.arrPtr[i];
}
```

```
// Insert a float or an array of floats
bool FloatArr::insert( float val, int pos)
{
   return insert( FloatArr(1,val), pos);
}

bool FloatArr::insert( const FloatArr& v, int pos )
{
   if( pos < 0 || pos >= cnt)
      return false;              // Invalid position

   if( max < cnt + v.cnt)
      expand(cnt + v.cnt);
   int i;
   for( i = cnt-1; i >= pos; --i)     // Shift up
      arrPtr[i+v.cnt] = arrPtr[i];    // starting at pos
   for( i = 0; i < v.cnt; ++i)        // Fill gap
      arrPtr[i+pos] = v.arrPtr[i];
   cnt = cnt + v.cnt;
   return true;
}

// To delete
bool FloatArr::remove(int pos)
{
   if( pos >= 0 && pos < cnt)
   {
      for( int i = pos; i < cnt-1; ++i)
         arrPtr[i] = arrPtr[i+1];
      --cnt;
      return true;
   }
   else
      return false;
}

// Output the array
ostream& operator<<( ostream& os, const FloatArr& v)
{
   int w = os.width();             // Save field width.
   for( float *p = v.arrPtr; p < v.arrPtr + v.cnt; ++p)
   {
      os.width(w);    os << *p;
   }
   return os;
}
```

When you define a derived class, the base class, the additional data members and methods, and the access control to the base class are defined.

The opposite page shows a schematic definition of a derived class, `C`. The `C` class inherits the `B` class, which is defined in the `public` section following the colon. The `private` and `public` sections contain additional members of the `C` class.

☐ Access to Public Members in the Base Class

Access privileges to the base class `B` are designated by the `public` keyword that precedes the `B`. In other words,

- all the `public` members in base class `B` are publicly available in the derived class `C`.

This kind of inheritance ports the public interface of the base class to the derived class where it is extended by additional declarations. Thus, objects of the derived class can call the `public` methods of the base class. A `public` base class, therefore, implements the *is* relationship; this is quite common.

There are somewhat less common cases where access to the members of the base class needs to be restricted or prohibited. Only the methods of class `C` can still access the `public` members of `B`, but not the users of that class. You can use `private` or `protected` derivation to achieve this (these techniques will be discussed later).

☐ Access to Private Members of the Base Class

The `private` members of the base class are protected in all cases. That is,

- the methods of the derived class cannot access the `private` members of the base class.

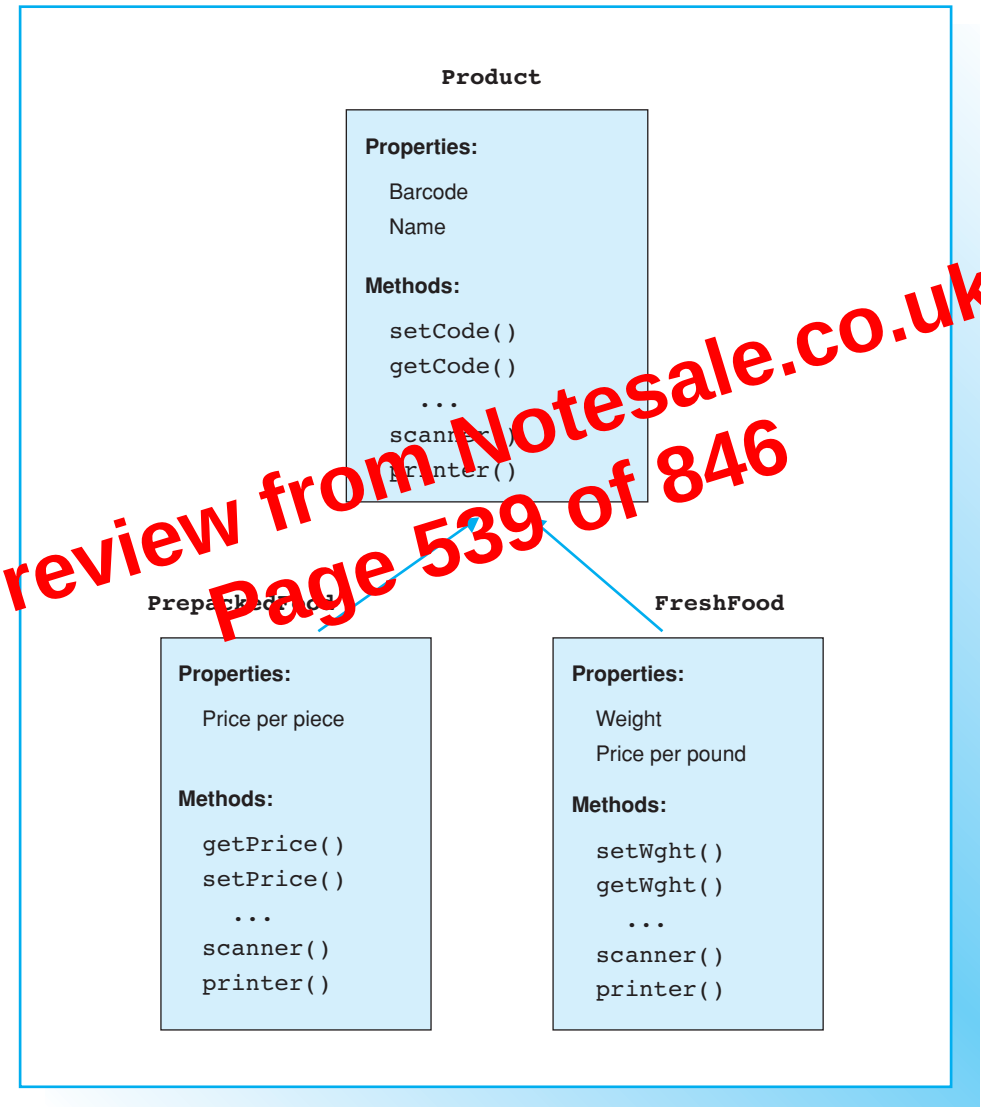
Imagine the consequences if this were not so: you would be able to hack access to the base class by simply defining a derived class, thus undermining any protection offered by data encapsulation.

☐ Direct and Indirect Base Classes

The derived class `C` can itself be a base class for a further class, `D`. This allows for class hierarchies. Class `B` then becomes an indirect base class for class `D`.

In the graphic on the opposite page, the arrow ↑ means directly derived from. That is, class `D` is a direct derivation of class `C` and an indirect derivation of `B`.

Exercise 3



```
// ------------------------------------------------------
// car.cpp
// Implements the methods of Car, PassCar, and Truck
// ------------------------------------------------------
#include "car.h"
// ------------------------------------------------------
// The methods of base class Car:
Car::Car( long n, const string& prod)
{
    cout << "Creating an object of type Car." << endl;
    nr = n;   producer = prod;
}
Car::~Car()
{
    cout << "Destroying an object of type Car" << endl;
}
void Car::display() const
{
    cout << "\n---------- Car ---------- "
         << "\nCar number: " << nr
         << "\nProduced by " << producer
         << endl;
}

// ------------------------------------------------------
// The methods of the derived class PassCar:
PassCar::PassCar(const string& tp, bool sd, int n,
         const string& hs)
    : Car( n, hs), PassCarTyp( tp ), sunRoof( sd )
{
   cout << "I create an object of type PassCar." << endl;
}
PassCar::~PassCar()
{
    cout << "\nDestroying an object of type PassCar"
         << endl;
}

void PassCar::display( void) const
{
   Car::display();            // Base class method
   cout << "Type:          " << passCarType
        << "\nSunroof:       ";
   if(sunRoof)
       cout << "yes "<< endl;
   else
       cout << "no " << endl;
}
```

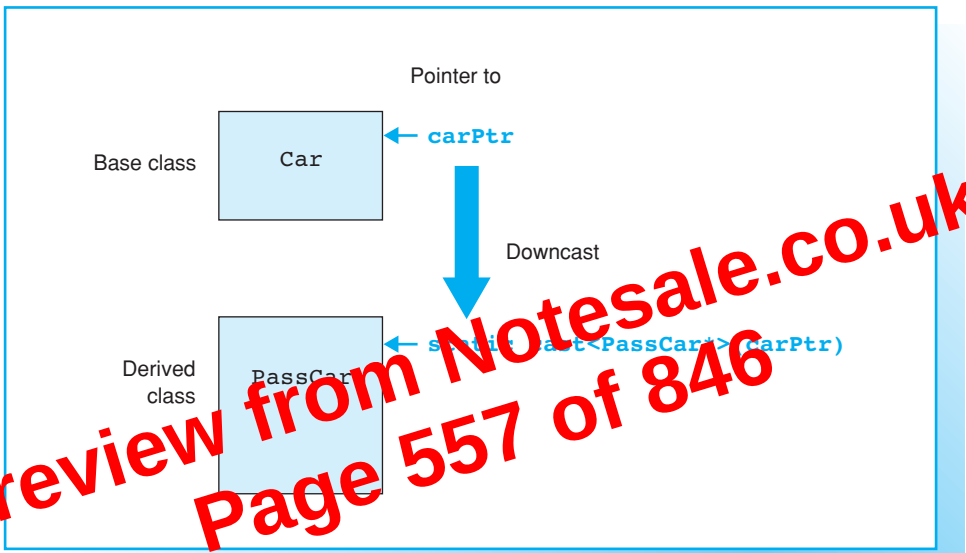
```
// ------------------------------------------------------
// The methods of the derived class Truck:
Truck::Truck( int a, double t, int n, const string& hs)
        : Car( n, hs), axles(a), tons(t)
{
   cout << "Creating an object of type Truck." << endl;
}
Truck::~Truck()
{
    cout << "\nDestroying an object of type Truck\n";
}
void Truck::display() const
{
    Car::display();
    cout <<   "Axles:       " << axles
         << "\nCapacity:    " << tons << " long tons\n";
}

// ------------------------------------------------------
// CarTest.cpp : Tests the base class Car and
//               the derived classes PassCar and Truck.
// ------------------------------------------------------
#include "car.h"
int main()
{
    Truck toy(5, 7.5, 1111, "Volvo");
    toy.display();
    char c;
    cout << "\nDo you want to create an object of type "
         << " PassCar? (y/n) ";  cin >> c;
    if( c == 'y' || c == 'Y')
    {
       const PassCar beetle("Beetle", false, 3421, "VW");
       beetle.display();
    }

    cout << "\nDo you want to create an object "
         << " of type car? (y/n) ";   cin >> c;
    if( c == 'y' || c == 'Y')
    {
       const Car oldy(3421, "Rolls Royce");
       oldy.display();
    }
    return 0;
}
```

```
  public:
    PrepackedFood(double p = 0.0,long b = 0L,
                  const string& s = "")
    : Product(b, s), pce_price(p)
    {}
    void   setPrice(double p){ pce_price = p;}
    double getPrice()const   { return pce_price; }
    void scanner()
    {   Product::scanner();
        cout << "Price per piece:    "; cin >> pce_price;
    }
    void printer() const
    {  Product::printer();
       cout << fixed << setprecision(2)
            << "Price per piece:    " << pce_price << endl;
    }
};

class FreshFood : public Product
{
  private:
    double wght;
    double lbs_price;

  public:
    FreshFood(double g = 0.0, double p = 0.0,
                long b = 0L, const string& s = "")
    : Product(b, s), wght(g), lbs_price(p) {}
    void   setWght(double g) { wght = g;}
    double getWght()const    { return wght; }
    void   setPrice(double p) { lbs_price = p;}
    double getPrice()const    { return lbs_price; }
    void scanner()
    {   Product::scanner();
        cout << "Weight(lbs):  "; cin >> wght;
        cout << "Price/lbs:    "; cin >> lbs_price;
        cin.sync(); cin.clear();
    }
    void printer() const
    {
       Product::printer();
       cout << fixed << setprecision(2)
            << "Price per Lbs:   " << lbs_price
            << "\nWeight:         " << wght
            << "\nTotal:          " << lbs_price * wght
            << endl;
    }
};
#endif
```

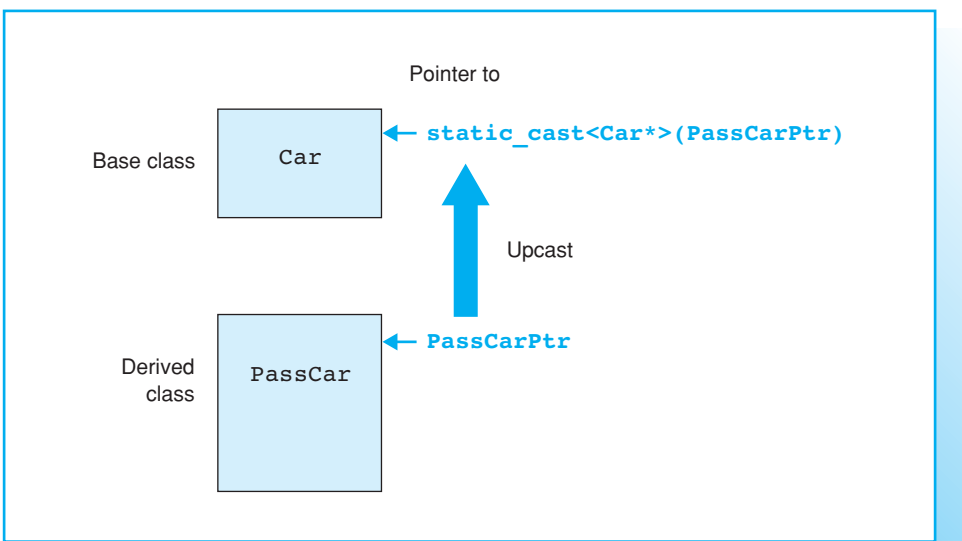
```
// ------------------------------------------------------
// product_t.cpp
// Tests classes Product, PrepackedFood, and FreshFood.
// ------------------------------------------------------

#include "product.h"

int main()
{
   Product p1(12345L, "Flour"), p2;

   p1.printer();              // Output the first product

   p2.setName("Sugar");       // Set the data members
   p2.setCode(543221);

   p2.printer();              // Output the second product

                              // Prepacked products:
   PrepackedFood pf1(0.49, 23456, "Salt"), pf2;

   pf1.printer();             // Output the first
                              // prepacked product
   cout << "\n Input data of a prepacked product: ";
   pf2.scanner();             // Input and output
   pf2.printer();             // data of 2nd product

   FreshFood pu1(1.5, 1.69, 98765, "Grapes"), pu2;

   pu1.printer();             // Output first item
                              // fresh food
   cout <<"\n Input data for a prepacked product: ";
   pu2.scanner();             // Input and output
   pu2.printer();             // data of 2nd product.

   cout << "\n-------------------------------"
        << "\n-------------------------------"
        << "\nAgain in detail: \n"
        << fixed << setprecision(2)
        << "\nBarcode:       " << pu2.getCode()
        << "\nName:          " << pu2.getName()
        << "\nPrice per Lbs: " << pu2.getPrice()
        << "\nWeight:        " << pu2.getWght()
        << "\nEnd price:     " << pu2.getPrice()
                                 * pu2.getWght()
        << endl;

   return 0;
}
```

■ EXPLICIT TYPE CONVERSIONS

Downcast



Upcast



☐ Upcasts and Downcasts

Type conversions that walk up a class hierarchy, or *upcasts*, are always possible and safe. Upcasting is performed implicitly for this reason.

Type conversions that involve walking down the tree, or *downcasts*, can only be performed explicitly by means of a cast construction. The cast operator `(type)`, which was available in C, or the `static_cast< >` operator are available for this task, and are equivalent in this case.

☐ Explicit Cast Constructions

Given that `cabrio` is again an object of the derived class `PassCar`, the following statements

Example:
```
Car* carPtr = &cabrio;
( (PassCar*) carPtr )->display();
```

first point the base class pointer `carPtr` to the `cabrio` object. `carPtr` is then cast as a pointer to the derived class. This allows you to access the `display()` method of the derived class `PassCar` via the pointer. Parentheses are necessary in this case as the member access operator `->` has a higher precedence than the cast operator `(type)`.

The operator `static_cast< >` conforms to the following

Syntax:
```
static_cast<type>(expression)
```

and converts the expression to the target type `type`. The previous example is thus equivalent to

Example:
```
static_cast<PassCar*>(carPtr)->display();
```

No parentheses are required here as the operators `static_cast<>` and `->` are of equal precedence. They are read from left to right.

After downcasting a pointer or a reference, the entire public interface of the derived class is accessible.

☐ Downcast Safety Issues

Type conversions from top to bottom need to be performed with great care. Downcasting is only safe when the object referenced by the base class pointer really is a derived class type. This also applies to references to base classes.

To allow safe downcasting C++ introduces the concept of *dynamic casting*. This technique is available for polymorphic classes and will be introduced in the next chapter.

Dynamically created objects in a class hierarchy are normally handled by a base class pointer. When such an object reaches the end of its lifetime, the memory occupied by the object must be released by a `delete` statement.

Example:
```
Car *carPtr;
carPtr = new PassCar("500",false,21,"Geo");
. . .
delete carPtr;
```

☐ Destructor Calls

When memory is released, the destructor for an object is automatically called. If multiple constructors were called to create the object, the corresponding destructors are called in reverse order. What does this mean for objects in derived classes? The destructor of the derived class is called first and then the destructor of the base class executed.

If you use a base class pointer to manage an object, the appropriate virtual methods of the derived class are called. However, non-virtual methods will always execute the base class version.

In the previous example, only the base class destructor for `Car` was executed. As the `PassCar` destructor is not called, neither is the destructor called for the data member `passCarType`, which is additionally defined in the derived class. The data member `passCarType` is a `string`, however, and occupies dynamically allocated memory—this memory will not be released.

If multiple objects are created dynamically in the derived class, a dangerous situation occurs. More and more unreferenced memory blocks will clutter up the main memory without you being able to reallocate them—this can seriously impact your program's response and even lead to external memory being swapped in.

☐ Virtual Destructors

This issue can be solved simply by declaring virtual destructors. The opposite page shows how you would define a virtual destructor for the `Car` class. Just like any other virtual method, the appropriate version of the destructor will be executed. The destructors from any direct or indirect base class then follow.

A class used as a base class for other classes should always have a virtual destructor defined. Even if the base class does not need a destructor itself, it should at least contain a dummy destructor, that is, a destructor with an empty function body.

☐ Static Binding

When a non-virtual method is called, the address of the function is known at time of compilation. The address is inserted directly into the machine code. This is also referred to as *static* or *early binding*.

If a virtual method is called via an object's name, the appropriate version of this method is also known at time of compilation. So this is also a case of early binding.

☐ Dynamic Binding

However, if a virtual method is called by a pointer or reference, the function that will be executed when the program is run is unknown at time of compilation. The statement

Example: `carPtr->display();`

could execute different versions of the `display()` method, depending on the object currently referenced by the pointer.

The compiler is therefore forced to create machine code that does not form an association with a particular function until the program is run. This is referred to as *late* or *dynamic binding*.

☐ VMT

Dynamic binding is supported internally by *virtual method tables* (or VMT for short). A VMT is created for each class with at least one virtual method—that is, an array with the addresses of the virtual methods in the current class.

Each object in a polymorphic class contains a VMT pointer, that is, a hidden pointer to the VMT of the corresponding class. Dynamic binding causes the virtual function call to be executed in two steps:

1. The pointer to the VMT in the referenced object is read.
2. The address of the virtual method is read in the VMT.

In comparison with static binding, dynamic binding does have the disadvantage that VMTs occupy memory. Moreover, program response can be impacted by indirect addressing of virtual methods.

However, this is a small price to pay for the benefits. Dynamic binding allows you to enhance compiled source code without having access to the source code. This is particularly important when you consider commercial class libraries, from which a user can derive his or her own classes and virtual function versions.

```
// Insert a truck:
bool CityCar::insert( int a, double t,
                        long n, const string& prod)
{
   if( cnt < 100)
   {
       vp[cnt++] = new Truck( a, t, n, prod);
       return true;
    }
    else
        return false;
}

void CityCar::display() const
{
    cin.sync(); cin.clear();     // No previous input
    for(int i=0; i < cnt; ++i)
    {
       vp[i]->display();
       if((i+1)%4 == 0)  cin.get();
    }
}

// ------------------------------------------------
// city_t.cpp : Test the CityCar class
// ------------------------------------------------
#include "city.h"
char menu(void);
void getPassCar(string&, bool&, long&, string&);
void getTruck(int&, double&, long&, string&);

int main()
{
   CityCar carExpress;
   string tp, prod;  bool   sr;
   int    a;    long   n;  double t;

   // Two cars are already present:
   carExpress.insert(6, 9.5, 54321, "Ford");
   carExpress.insert("A-class", true, 54320, "Mercedes");
   char choice;
   do
   {   choice = menu();
       switch( choice )
       {
         case 'Q':
         case 'q': cout << "Bye Bye!"  << endl;
                   break;
```

```
            case 'P':
            case 'p': getPassCar(tp, sr, n, prod);
                      carExpress.insert(tp, sr, n, prod  );
                      break;
            case 'T':
            case 't': getTruck(a, t, n, prod);
                      carExpress.insert(a, t, n, prod);
                      break;
            case 'D':
            case 'd': carExpress.display();
                      cin.get();
                      break;
            default:  cout << "\a";     // Beep
                      break;
         }
   }while( choice != 'Q'  && choice != 'q' );
   return 0;
}

char menu()                              // Input a command.
{
   cout << "\n * * * Car Rental Management * * *\n\n"
   char c;
   cout <<   "\n           P = Add a passenger car "
        <<   "\n           T = Add a truck "
        <<   "\n           D = Display all cars "
        <<   "\n           Q = Quit the program "
        << "\n\nYour choice: ";
   cin >> c;
   return c;
}

void getPassCar(string& tp, bool& sr, long& n,string& prod)
{
   char c;
   cin.sync(); cin.clear();
   cout << "\nEnter data for passenger car:" << endl;
   cout << "Car type:           "; getline(cin, tp);
   cout << "Sun roof (y/n):     "; cin >> c;
   if(c == 'y' || c == 'Y')
       sr = true;
   else
       sr = false;
   cout << "Car number:       "; cin >> n;
   cin.sync();
   cout << "Producer:        "; getline(cin, prod);
   cin.sync(); cin.clear();
}
```

```
class DerivedEl : public BaseEl
{
   private:
     string rem;

   public:
     DerivedEl(Cell* suc = NULL,
               const string& s="", const string& b="")
     : BaseEl(suc, s), rem(b){ }
     // Access methods:
     void    setRem(const string& b){ rem = b; }
     const string& getRem() const { return rem; }
     void display() const
     {
         BaseEl::display();
         cout << "Remark:     " << rem << endl;
     }
};
#endif
```



```
// ------------------------------------------------------
// List.h : Define the class InhomList
// ------------------------------------------------------
#ifndef _LIST_H_
#define _LIST_H_
#include "cell.h"
class InhomList
{
  private:
    Cell* first;
  protected:
    Cell* getPrev(const string& s);
    Cell* getPos( const string& s);
    void  insertAfter(const string& s, Cell* prev);
    void  insertAfter(const string& s,const string& b,
                      Cell* prev);
    void  erasePos(Cell* pos);
  public:
     InhomList(){ first = NULL; }
     InhomList(const InhomList& src);
     ~InhomList();
     InhomList& operator=( const InhomList& src);
     void  insert(const string& n);
     void  insert(const string& n, const string& b);
     void  erase(const string& s);
     void displayAll() const;
};
#endif
```

```
// ------------------------------------------------------
// List.cpp : The methods of class InhomList
// ------------------------------------------------------
#include "List.h"
#include <typeinfo>

// Copy constructor:
InhomList::InhomList(const InhomList& src)
{
   // Append the elements from src to the empty list.
   first = NULL;
   Cell *pEl = src.first;
   for(  ; pEl != NULL;  pEl = pEl->getNext() )
       if(typeid(*pEl) == typeid(DerivedEl))
           insert(dynamic_cast<DerivedEl*>(pEl)->getName(),
                  dynamic_cast<DerivedEl*>(pEl)->getRem());
       else
           insert(dynamic_cast<BaseEl*>(pEl)->getName());
}

// Assignment:
InhomList& InhomList::operator=(const InhomList& src)
{
   // To free storage for all elements:
   Cell *pEl = first,
        *next = NULL;
   while( pEl != NULL )
   {
      next = pEl->getNext();
      delete pEl;
      pEl = next;
   }

   first = NULL;                // Empty list

   // Copy the elements from src to the empty list.
   pEl = src.first;

   for(  ; pEl != NULL;  pEl = pEl->getNext() )
      if(typeid(*pEl) == typeid(DerivedEl))
           insert(dynamic_cast<DerivedEl*>(pEl)->getName(),
                  dynamic_cast<DerivedEl*>(pEl)->getRem());
      else
           insert(dynamic_cast<BaseEl*>(pEl)->getName());

   return *this;
}
```

```
void InhomList::displayAll() const
{
   Cell* pEl = first;
   while(pEl != NULL)
   {
       pEl->display();
       pEl = pEl->getNext();
   }
}

// ------------------------------------------------------
// List_t.cpp : Tests the sorted inhomogeneous list
// ------------------------------------------------------

#include "List.h"

int main()
{
   InhomList liste1;

   cout << "\nTo test inserting. " << endl;

   liste1.insert("Billy, Max");
   liste1.insert("Cheers, Rita", "always merry");
   liste1.insert("Quick, John", "topfit");
   liste1.insert("Banderas, Antonio");

   liste1.displayAll(); cin.get();

   cout << "\nTo test deleting. " << endl;

   liste1.erase("Banderas, Antonio");
   liste1.erase("Quick, John");
   liste1.erase("Cheers, Rita");

   liste1.displayAll(); cin.get();

   cout << "\n---------------------------------"
        << "\nGenerate a copy and insert an element. "
        << endl;

   InhomList liste2(liste1),     // Copy constructor
             liste3;             // and an empty list.

   liste2.insert("Chipper, Peter", "in good temper");
   liste3 = liste2;              // Assignment
   cout << "\nAfter the assignment:  " << endl;
   liste3.displayAll();

   return 0;
}
```

This page intentionally left blank

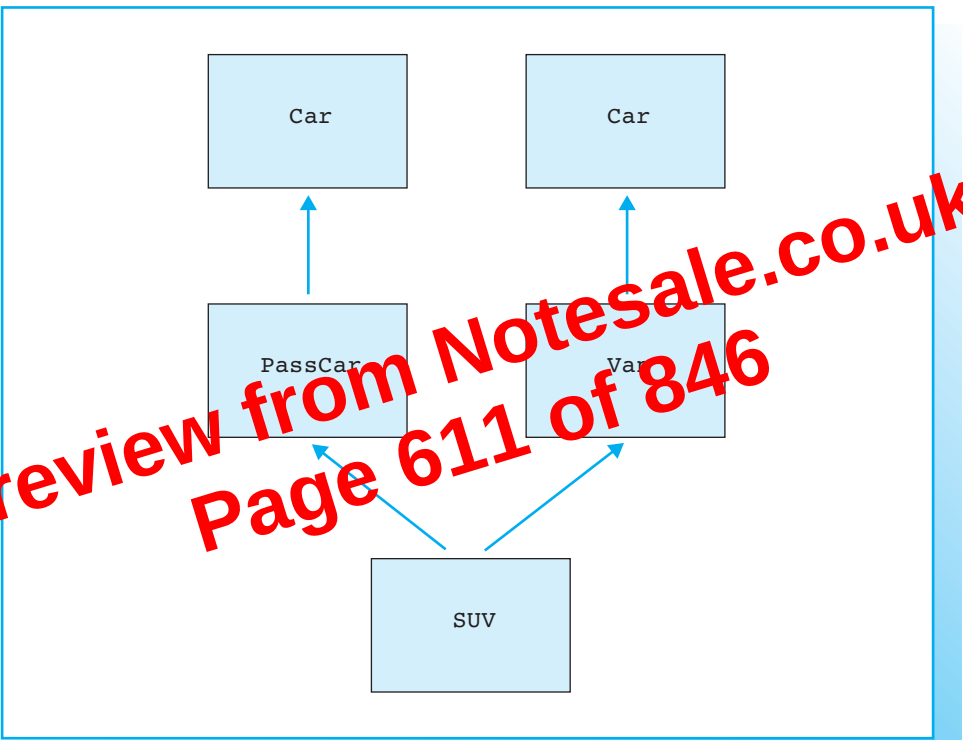
chapter 27

Multiple Inheritance

This chapter describes how new classes are created by multiple inheritance and explains their uses. Besides introducing you to creating and destroying objects in multiply-derived classes, virtual base classes are depicted to avoid ambiguity in multiple inheritance.

■ MULTIPLE INDIRECT BASE CLASSES

The multiple indirect base class `Car`



Definition scheme of class `SUV`

```
class SUV : public PassCar, public Van
{
   // Here are additional methods and data members
};
```

Constructor Calls in Virtual Base Classes

When an object is created for a multiply-derived class, the constructors of the base classes are called first. However, if there is one virtual base class in the class hierarchy, the virtual base class constructor is executed *before* a constructor of a non-virtual base class is called.

NOTE

The constructors of the virtual base classes are called first, followed by the constructors of non-virtual base classes in the order defined in the inheritance graph.

The constructor of the virtual base class nearest the top of the inheritance graph is executed first. This does not necessarily mean the top level of the class hierarchy, since a virtual base class can be derived from a non-virtual base class.

In our example with the multiply derived class `SUV` (Sport Utility Vehicle) the constructor for the virtual base class `Car` is called first, followed by the direct base classes `PassCar` and `Van`, and last but not least the constructor of the `SUV` class.

Base Initializers

You may be wondering what arguments are used to call the constructor of a virtual base class. A base initializer of the directly-derived class or any other derivation could be responsible. The following applies:

NOTE

The constructor of a virtual base class is called with the arguments stated for the base initializer of the *last* class to be derived, i.e. class at the bottom end of the inheritance graph.

The example opposite shows `SUV` containing a constructor with *one* base initializer. Its arguments are passed to the constructor of the virtual base class `Car`.

For the purpose of initialization, it does not matter whether a class derived directly from `Car` contains a base initializer or not. Base initializers for virtual indirect base classes defined in the constructor of a direct base class are ignored. If the base classes `PassCar` and `Van` also contained base initializers for the virtual base class `Car`, these would be ignored too.

If the constructor for the last derived class does not contain a base initializer, the default constructor is executed for each virtual base class. Whatever happens, a default constructor must then exist in every virtual base class! Thus, base initializers that happen to exist in base classes are also ignored.

■ EXCEPTION HANDLERS

Syntax of `try` and `catch` blocks

```
try
{

   // Exceptions thrown by this block will be
   // caught by the exception handlers,
   // which are defined next.

}
catch( Type1 exc1)
{
   // Type1 exceptions are handled here.
}
[ catch( Type2 exc2)
 {
    // Type2 exceptions are handled here.
 }
   . . .              //etc.
]
[ catch( ... )
 {
    // All other exceptions are handled here.
 }]
```


NOTE

The brackets `[...]` in a syntax description indicate that the enclosed section is optional.

■ THROWING AND CATCHING EXCEPTIONS

Demonstration program

```
// calc_err.cpp:  Tests the function calc(),
//                which throws exceptions.
// ------------------------------------------------
#include <iostream>
#include <string>
using namespace std;

double calc( int a, int b );

int main()
{
    int x, y;
    double res;
    bool flag = false;
    do
    {
      try                                    // try block
      {
        cout << "Enter two positive integers: ";
        cin >> x >> y;
        res = calc( x, y);
        cout << x << "/" << y << " = " << res << endl;
        flag = true;          // Then to leave the loop.
      }
      catch( string& s)                      // 1st catch block
      {
        cerr << s << endl;
      }
      catch( Error& )                        // 2nd catch block
      {
        cerr << "Division by 0! " << endl;
      }
      catch(...)                            // 3rd catch block
      {
        cerr << "Unexpected exception! \n";
        exit(1);
      }
   }while( !flag);

    // continued ...
    return 0;
}
```


NOTE

As the `Error` class contains no data members, the corresponding `catch` block declares only the type of exception, and no parameters. This avoids a compiler warning since the parameter is not used.

exercises

■ EXERCISES

Exercise 1: Error messages of the exception handler

The first exception handler's message:

```
Error in reading:
Invalid index: ...
```


The second exception handler's message:

```
Error in writing:
Invalid index: ...
```

```
    Fraction& operator+=(const Fraction& a)
    {
       numerator = a.numerator * denominator
                 + numerator * a.denominator;
       denominator *= a.denominator;
       return *this;
    }

    Fraction& operator-=(const Fraction& a)
    {
       *this += (-a);
       return *this;
    }

    Fraction& operator++()
    {
       numerator += denominator;
       return *this;
    }

    Fraction& operator--()
    {
       numerator -= denominator;
       return *this;
    }
  friend Fraction operator+(const Fraction&,
                            const Fraction&);
  friend Fraction operator-(const Fraction&,
                            const Fraction&);
  friend Fraction operator*(const Fraction&,
                            const Fraction&);
  friend Fraction operator/(const Fraction&,
                            const Fraction&)
                          throw(Fraction::DivisionByZero);

  friend ostream& operator<<(ostream&, const Fraction&);
  friend istream& operator>>(istream& is, Fraction& a)
                          throw(Fraction::DivisionByZero);
};

#endif
```

☐ Random File Access

So far we have only looked at sequential file access. If you need access to specific information in such a file, you have to walk through the file from top to tail, and new records are always appended at the end of the file.

Random file access gives you the option of reading and writing information directly at a pre-defined position. To be able to do this, you need to change the current file position explicitly, that is, you need to point the `get`/`put` pointer to the next byte to be manipulated. After pointing the pointer, you can revert to using the read and write operations that you are already familiar with.

☐ Open Modes

One prerequisite of random file access is that the position of the records in the file can be precisely identified. This implies opening the file in binary mode to avoid having to transfer additional escape characters to the file.

Example:
```
ios::openmode mode = ios::in | ios::out |
                     ios::app | ios::binary;
fstream fstr("account.fle", mode);
```

This statement opens the file `"Account.fle"` in binary mode for reading and *appending at end-of-file*. The file will be created if it did not previously exist. Random read access to the file is possible, but for write operations new records will be appended at the end of the file.

To enable random read and write access to a file, the file can be opened as follows:

Example:
```
ios::openmode mode = ios::in | ios::out |
                     ios::binary;
fstream fstr("account.fle", mode);
```

However, this technique can only be used for existing files. If the file does not exist, you can use the `ios::trunc` flag to create it.

The section "File State" discusses your error handling options if a file, such as `"account.fle"` cannot be found.

State Flags

A file stream can assume various states, for example, when it reaches the end of a file and cannot continue reading. A file operation can also fail if a file cannot be opened, or if a block is not transferred correctly.

The `ios` class uses *state flags* to define the various states a file can assume. Each state flag corresponds to a single bit in a *status-word*, which is represented by the `iostate` type in the `ios` class. The following state flags exist:

- `ios::eofbit` end of file reached
- `ios::failbit` last read or write operation failed
- `ios::badbit` an irrecoverable error occurred
- `ios::goodbit` the stream is ok, e.g. no other state flag is set

The "flag" `ios::goodbit` is an exception to the rule since it is not represented by a single bit, but by the value 0 if no other flag has been set. In other words a status-word has the value `ios::goodbit` if everything is fine!

Discovering and Changing the State

There are multiple methods for discovering and modifying the status of a stream. A method exists for each state flag; these are `eof()`, `fail()`, `bad()`, and `good()`. They return `true` when the corresponding flag has been raised. This means you can discover the end of a file with the following statement:

Example: `if( fstr.eof() ) ...`

The status-word of a stream can be read using the `rdstate()` method. Individual flags can then be queried by a simple comparison:

Example: `if( myfile.rdstate() == ios::badbit ). . .`

The `clear()` method is available for clearing the status-word. If you call `clear()` without any arguments, all the state flags are cleared. An argument of the `iostate` type passed to `clear()` automatically becomes the new status-word for the stream.

The `IndexFile` Class

The `IndexFile` class, which uses a file to represent an index, is defined opposite. The constructor for this class uses the `clear()` method to reset the `fail` bit after an invalid attempt to open a non-existent file. A new file can then be created.

The `IndexFile` class comprises methods for inserting, seeking, and retrieving index entries, which we will be implementing later in this chapter.

■ PERSISTENCE OF POLYMORPHIC OBJECTS

```
// account.h :  Defines the classes
//              Account, DepAcc, and SavAcc
//              with virtual read and write methods.
// -------------------------------------------------
// . . .
enum TypeId { ACCOUNT, DEP_ACC, SAV_ACC };
class Account
{
  private:   // Data members: as previously defined.
  public:    // Constructor, access methods ...
    virtual TypeId getTypeId() const { return ACCOUNT; }
    virtual ostream& write(ostream& fs) const;
    virtual istream& read(istream& fs);
};
class DepAcc : public Account
{      // Data members, constructor, . . .
    TypeId getTypeId() const { return DEP_ACC; }
    ostream& write(ostream& fs) const;
    istream& read(istream& fs);
};
class SavAcc : public Account
{       // Data members, constructor, . . .
    TypeId getTypeId() const { return SAV_ACC; }
    ostream& write(ostream& fs) const;
    istream& read(istream& fs);
};
```


The methods `read()` and `write()` of class `DepAcc`

```
// account.cpp:  Implements the methods.
// -------------------------------------------------------
#include "account.h"
ostream& DepAcc::write(ostream& os) const
{
   if(!Account::write(os))
      return os;
   os.write((char*)&limit, sizeof(limit) );
   os.write((char*)&deb, sizeof(deb) );
   return os;
}
istream& DepAcc::read(istream& is)
{
   if(!Account::read(is))
      return is;
   is.read((char*)&limit, sizeof(limit) );
   is.read((char*)&deb, sizeof(deb));
   return is;
}
// . . .
```

☐ 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

■ EXERCISES

Exercise 1

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 1

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.

```
    // 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);
```

```
// ---- 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
    }
```

```
// ------------------------------------------------------
// 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;
}
```

```
    kde.setNr(10L); kde.setName("Peter");
    hash.insert( kde );

    kde.setNr(17L); kde.setName("Alexa");
    hash.insert( kde );

    kde.setNr(21L); kde.setName("Peter");
    hash.insert( kde );

    kde.setNr(15L); kde.setName("Jeany");
    hash.insert( kde );
    cout << "\nInsertion complete: " << endl;

    hash.display();

    unsigned long key;
    cout << "Key? ";  cin >> key;

    HashEntry temp = hash.retrieve(key);
    if(temp.getNr() != 0L)
       temp.display();
    else
       cout << "Key " << key
            << " not found" << endl;
  }
  catch(OpenError& err)
  {
     cerr << "Error in opening the file:"
          << err.getName() << endl;
     exit(1);
  }



  catch(WriteError& err)
  {
     cerr << "Error writing to file: "
          << err.getName() << endl;
     exit(1);
  }
  catch(ReadError& err)
  {
     cerr << "Error reading from file: "
          << err.getName() << endl;
     exit(1);
  }

  return 0;
}
```

chapter 30

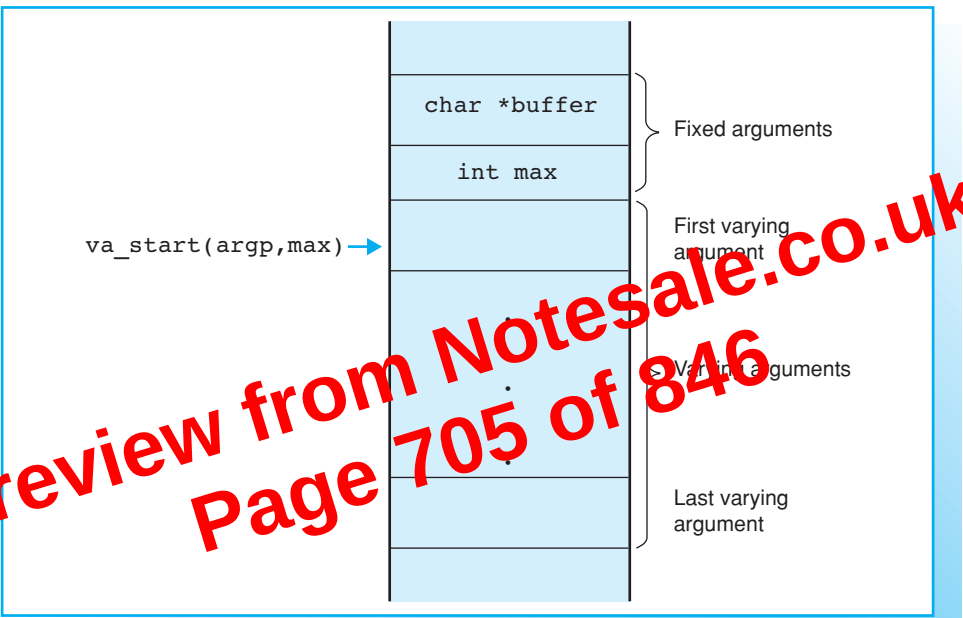
More about Pointers

This chapter describes advanced uses of pointers. These include pointers to pointers, functions with a variable number of arguments, and pointers to functions.

An application that defines a class used to represent dynamic matrices is introduced.

■ VARIABLE NUMBER OF ARGUMENTS

Fixed and varying arguments on the stack



Scheme of a function with varying arguments

```
#include <stdarg.h>

int func( char *buffer, int max, ... )
{
   va_list argptr;      // Declares argument pointer.
   long arg3;
    . . .
   va_start( argptr, max);          // Initialization.
   arg3 = va_arg( argptr, long );   // Read arguments.

   // To use argument arg3.
    . . . .

   va_end(argptr);      // Set argument pointer to NULL.
}
```

■ APPLICATION: DYNAMIC MATRICES

Class Matrix

```
// matrix.h:  Representing dynamic matrices.
// ------------------------------------------------
#include <stdexcept>
#include <iostream>
using namespace std;

class Row
{
   double *ro;   int size;
 public:
  Row( int s) { size = s; ro = new double[s]; }
  ~Row(){ delete[]ro; }

  double& operator[](int i) throw(out_of_range)
  {if(i < 0 || i >=size)
     throw out_of_range("Column index: Out of Range\n");
   else
     return ro[i];
  }
};

class Matrix
{
    Row **mat;             // Pointer to array of rows
    int lines, cols;       // Number of rows and columns
  public:
   Matrix(int ro , int co )
   { lines = ro; cols = co;
     mat = new Row*[lines];
     for(int i=0; i < lines; i++)
         mat[i] = new Row(cols);
   }

   ~Matrix()
   {   for(int i=0; i < lines; i++)
           delete mat[i];
       delete[] mat;
   }
   int  getLines() const { return lines; }
   int  getCols() const { return cols; }
   Row& operator[](int i) throw(out_of_range)
   { if(i < 0 || i > cols)
      throw out_of_range("Row index: Out of Range\n");
     else
      return *mat[i];
   }
};
```

Exercise 4

```
// -- ----------------------------------------------------
// matrix.h :  Represents dynamic matrices
// ----------------------------------------------------
#ifndef _MATRIX_H_
#define _MATRIX_H_

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

class Row
{
    double *ro;
    int size;
  public:
    Row( int s) { size = s; ro = new double[s] ; }
    ~Row(){ delete[] ro ; }

    double& operator[](int i)
    {
      if(i < 0 || i > size)
        throw out_of_range("Row index: Out of Range\n");
      return ro[i];
    }
    const double& operator[](int i)  const
    {
      if(i < 0 || i > size)
        throw out_of_range("Row index: Out of Range\n");
      return ro[i];
    }
};

class Matrix
{
  private:
    Row **mat;              // Pointer to array of rows
    int lines, cols;        // Number of rows and columns

  public:
    Matrix(int ro , int co)
    {
      lines = ro; cols = co;
      mat = new Row*[lines];
      for(int i=0; i < lines; i++)
         mat[i] = new Row(cols);
    }
    Matrix:: Matrix( int z, int s, double val);
```

```
     Matrix( const Matrix& );
    ~Matrix()
    {  for(int i=0; i < lines; i++)
           delete mat[i];
       delete[] mat;
    }
    int  getLines() const { return lines; }
    int  getCols()  const { return cols; }

    Row& operator[](int i)
    {
      if(i < 0 || i > cols)
         throw out_of_range("Row index: Out of Range\n");
      return *mat[i];
    }

    const Row& operator[](int i) const
    {
      if(i < 0 || i > cols)
         throw out_of_range("Row index: Out of Range\n");
      return *mat[i];
    }
                                        // Assignments:
    Matrix& operator=( const Matrix& );
    Matrix& operator+=( const Matrix& );
};
#endif

// ------------------------------------------------------
// matrix.cpp : Defines methods of class Matrix
// ------------------------------------------------------
#include "matrix.h"

Matrix:: Matrix( int ro, int co, double val)
{
    lines = ro; cols = co;
    mat = new Row*[lines];          // Array of pointers to
                                    // arrays of rows
    int i, j;
    for(i=0; i < lines; i++)        // Arrays of rows:
    {
       mat[i] = new Row(cols);      // Allocate memory
       for(j = 0; j < cols; ++j)
          (*this)[i][j] = val;     // and copy values.
    }
}
```

```
// ----------------------------------------------------------
// matrix_t.cpp : Tests dynamic matrices
// ----------------------------------------------------------
#include "matrix.h"
void display( Matrix& m);         // Output a matrix.
int main()
{
    Matrix m(4,5);
    try
    {
      int i,j;
      for( i=0; i < m.getLines(); i++)
        for( j=0; j < m.getCols(); j++)
          m[i][j] = (double)i + j/ 100.0;

      cout << "Matrix created" << endl;
      display(m);

      Matrix cop(m);
      cout << "Copy generated." << endl;
      display(cop);

      cop += m;
      cout << "Compute the sum:" << endl;
      display(cop);

      Matrix m1(4, 5, 0.0);
      cout << "Initializing a matrix with 0:" << endl;
      display(m1);
      m = m1;
      cout << "Matrix assigned:" << endl;
      display(m);
    }
    catch(out_of_range& err)
    {  cerr << err.what() << endl;    exit(1);  }
    return 0;
}

void display( Matrix& m)
{
    for(int i=0; i < m.getLines(); i++)
    {
      for(int j=0; j < m.getCols(); j++)
         cout << m[i][j] << "   ";
      cout << endl;
    }
    cin.get();
}
```

■ BITWISE OPERATORS

"True or False" table for bitwise operators

Bitwise AND	Result	Bitwise inclusive OR	Result
0 & 0	0	0 | 0	0
0 & 1	0	0 | 1	1
1 & 0	0	1 | 0	1
1 & 1	1	1 | 1	1
0 ^ 0	0	0	1
0 ^ 1	1	~1	
1 ^ 1	0		

Examples

```
unsigned int a, b, c;

a = 5;
b = 12;
c = a & b;
c = a | b;
c = a ^ b;
c = ~a;
```

Bit pattern
0 0 . . . . . 0 0 1 0 1
0 0 . . . . . 0 1 1 0 0
0 0 . . . . . 0 0 1 0 0
0 0 . . . . . 0 1 1 0 1
0 0 . . . . . 0 1 0 0 1
1 1 . . . . . 1 1 0 1 0

☐ Deleting Bits

The *bitwise AND operator* is normally used to delete specific bits. A so-called *mask* is used to determine which bits to delete.

Example: `c = c & 0x7F;`

In the mask `0x7F` the seven least significant bits are set to `1`, and all significant bits are set to `0`. This means that all the bits in `c`, with the exception of the least significant bits, are deleted. These bits are left unchanged.

The variable `c` can be of any integral type. If the variable occupies more than one byte, the significant bits in the mask, `0x7F`, are padded with `0` bits when integral promotion is performed.

☐ Setting and Inverting Bits

You can use the *bitwise OR operator* `|` to set specific bits. The example on the opposite page shows how to change the case of a letter. In ASCII code, the only difference between a lowercase and an uppercase letter is the fifth bit.

Finally, you can use the *bitwise exclusive OR operator* `^` to invert specific bits. Each `0`-bit is set to `1` and each `1`-bit is deleted if the corresponding bit in the mask has a value of `1`.

Example: `c = c ^ 0xAA;`

The bit pattern for `0xAA` is `10101010`. Every second bit in the least significant eight bits of `c` is therefore inverted.

It is worthy of note that you can perform double inversion using the same mask to restore the original bit pattern, that is, `(x ^ MASK) ^ MASK` restores the value `x`.

The following overview demonstrates the effect of a statement for an integral expression `x` and any given mask, `MASK`:

- `x & MASK` deletes all bits that have a value of 0 in `MASK`
- `x | MASK` sets all bits that have a value of 1 in `MASK`
- `x ^ MASK` inverts all bits that have a value of 0 in `MASK`.

The other bits are left unchanged.

■ SPECIALIZATION

Function template `min()`

```
template <class T>
T min( T x, T y)
{
   return( (x < y) ? x : y)
}
```

Specializing the function template for C strings

```
#include <cstring>

const char* min( const char* s1, const char* s2 )
{
   return( (strcmp(s1, s2) < 0 ) ? s1: s2 );
}
```


☐ ANSI specialization

The ANSI standard does not differ between template functions and "normal" functions. The definition of a function template and a function with the same name, which can be generated by the function template, causes the compiler to output an error message (ex. "duplicate definition ...").

That is why the ANSI standard provides its own syntax for defining specializations:

```
#include <cstring>

template<>
const char* min( const char* s1, const char* s2 )
{
   return( (strcmp(s1, s2) < 0 ) ? s1: s2 );
}
```

```
template <class T>
void display(T* vp, int len)
{
   cout << "\n\nThe array: " << endl;
   for(int i = 0; i < len; i++)
   {
      cout << vp[i] << "  ";
      if( (i+1)%10 == 0)
          cout << endl;
   }
   cout << endl; cin.get();
}

// Two arrays for testing:
short  sv[5] = { 7, 9, 2, 4, 1};
double dv[5] = { 5.7, 3.5, 2.1, 9.4, 4.8};

int main()
{
   cout << "\nInstantiation for type short: " << endl;
   display(sv, 5);
   insertionSort(sv, 5);
   cout << "\nAfter sorting: ";
   display(sv, 5);

   short key;
   cout << "\nArray element? ";  cin >> key; cin.sync();
   int pos = interpolSearch(key, sv, 5);
   if( pos != -1)
       cout << "\nfound!" << endl, cin.get();
   else
       cout << "\nnot found!" << endl, cin.get();
   // ------------------------------------------------
   cout << "\nInstantiation for type double: " << endl;
   display(dv, 5);

   insertionSort(dv, 5);
   cout << "\nAfter sorting: ";
   display(dv, 5);

   double dkey;
   cout << "\nArray element? "; cin >> dkey; cin.sync();
   pos = interpolSearch(dkey, dv, 5);
   if( pos != -1)
       cout << "\nfound!" << endl, cin.get();
   else
       cout << "\nnot found!" << endl, cin.get();

   return 0;
}
```

☐ Insertion Methods

The following methods are defined in the container classes `vector`, `deque`, and `list`

- `push_back()` insert at end
- `insert()` insert after a given position.

Additionally, the following method is available in the `list` and `deque` classes

- `push_front()` insert at beginning.

This method is not defined in the `vector` class.

The `insert()` method is overloaded in various versions, allowing you to insert a single object, multiple copies of an object, or object copies from another container. Given two containers `v` and `w` the following

Example: `w.insert(--w.begin(), v.begin(), v.end());`

inserts the objects from container `v` in front of all the other objects in `w`. A container can of course be assigned to another container of the same type. The assignment operator is overloaded for containers to allow this operation.

☐ Runtime Behavior

The `push_back()` and `push_front()` methods are preferable on account of their constant runtime. Insertion of *one* object with the `insert()` method also has a constant runtime in the `list` class. However, this is linear in the `vector` and `deque` classes, that is, the time increases proportionally to the number of objects in the container.

This dissimilar runtime behavior for methods can be ascribed to the implementation of various container classes. Normally, containers of the `list` type are represented by double linked lists in which each element possesses a pointer to the preceding and following element. This allows for extremely quick inserting at a given position.

The container classes `vector` and `deque` are represented as arrays. Inserting in the middle means shifting the objects in the container to make place for the new object. Therefore the runtime will increase proportionally with the number of objects the container holds.

☐ Insertion in Adapter Classes

There is only one insertion method for adapter classes: `push()`. In stacks and queues, `push()` appends an object with a constant runtime. Insertion of objects into priority queues depends on the priority of the object and the runtime is linear.

■ ASSOCIATIVE CONTAINERS

Container classes

Container Class	Representing
`set< class T, class Compare = less<T>, class Allocator = allocator<T> >`	collections of objects with unique keys
`multiset< class T, class Compare = less<T>, class Allocator = allocator<T> >`	collections of objects with equivalent keys, i.e. possibly multiple copies of the same key value
`map< class Key, class T, class Compare = less<T>, class Allocator = allocator<T> >`	collections of objects/key pairs where the keys are unique
`multimap< class Key, class T, class Compare = less<T>, class Allocator = allocator<T> >`	collections of objects/key pairs with possibly equivalent keys

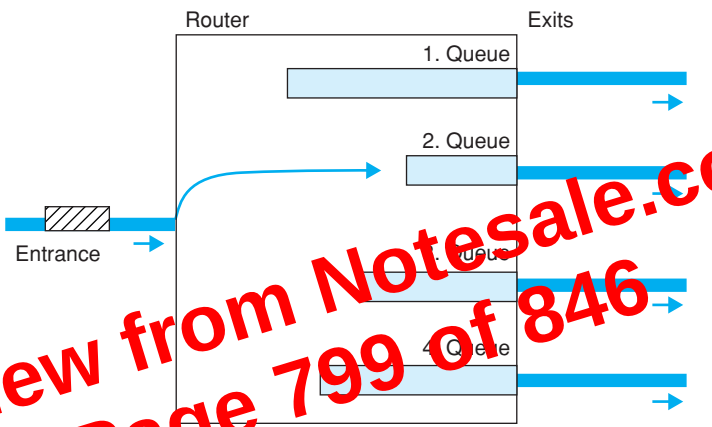
Associative containers and header files

Container Class	Header File
`set< T, Compare, Allocator >`	`<set>`
`multiset<T, Compare, Allocator >`	`<set>`
`map< Key, T, Compare, Allocator >`	`<map>`
`multimap< Key, T, Compare, Allocator >`	`<map>`

exercise

■ EXERCISE

Hot potato algorithm



Test output

```
9 queues have been created.
The queues will now be filled
using the hot potato algorithm.
Some elements of randomly selected
queues are removed.
Output the queues:
1.queue: 28  88  70  60   6
2.queue: 64   6  54   1
3.queue:  2  88  64  30  66  29  11  74  49  41
4.queue: 17  25
5.queue: 96  97  47  27  71  34  87  58
6.queue: 77  82  54
7.queue: 35  65  23  40   5  83  92
8.queue: 32  23  54
9.queue: 28  55  54  73  28  82  21  99
```

You can also use two's complement to compute the absolute value of a negative number. Two's complement for `-4` yields a value of `4`.

Sign bits are not required for `unsigned` types. The bit can then be used to represent further positive numbers, doubling the range of positive numbers that can be represented.

The following table contains the binary formats of signed and unsigned integral 8 bit values:

Binary	Signed decimal	Unsigned decimal
0000 0000	0	0
0000 0001	1	1
0000 0010	2	2
0000 0011	3	3
.	.	.
.	.	.
0111 1101	125	125
0111 1110	126	126
0111 1111	127	127
1000 0000	−128	128
1000 0001	−127	129
.	.	.
.	.	.
1111 1100	−4	252
1111 1101	−3	253
1111 1110	−2	254
1111 1111	−1	255

If the bit-pattern of a negative number is interpreted as an `unsigned` number, the value of the number changes. The bit-pattern `1111 1100` of the number −4 will thus yield the following `unsigned` value:

$$0*2^0 + 0*2^1 + 1*2^2 + 1*2^3 + 1*2^4 + 1*2^5 + 1*2^6 + 1*2^7$$

that is, the decimal number `252`.

Representing Floating-point Numbers

To represent a given floating-point number, `x`, the number is first broken down into a sign, `v`, a mantissa, `m`, and a power, `exp`, with a base of `2`:

$$x = v * m * 2^{exp}$$

■ LITERATURE

International Standard ISO/IEC 14882, *Programming Languages*—C++; published by American National Standards Institute, New York NY, 1998.

International Standard ISO/IEC 9899:1999(E), *Programming Languages*—C; published by ISO Copyright Office, Case postale 56, CH-1211, Geneva 20, 1999.

Stroustrup, Bjarne, *The C++ Programming Language*, Addison Wesley, 2000.

Josuttis, Nicolai, *The C++ Standard Library*, Addison Wesley, 1999.

This page intentionally left blank

index

Note: Italicized page locators indicate figures.

J

K

X

Z