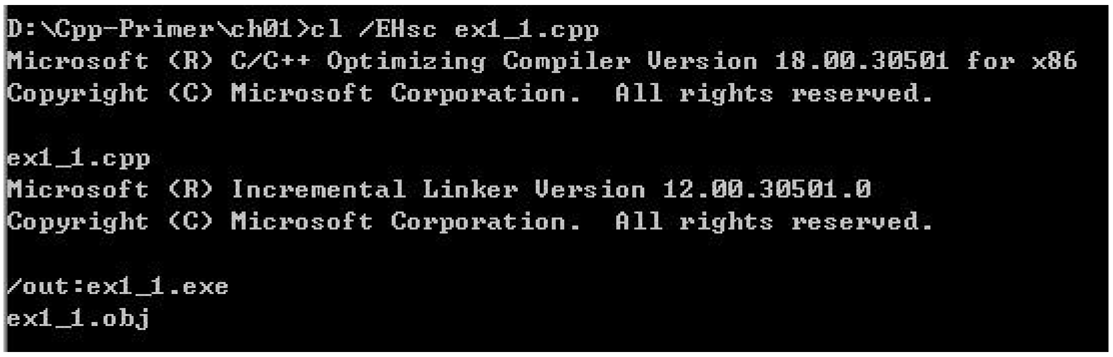
Chapter 1

Exercise 1.1

Review the documentation for your compiler and determine what file naming convention it uses. Compile and run the main program from page 2.

Windows

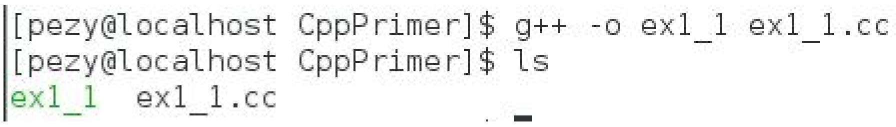


```
D:\Cpp-Primer\ch01>cl /EHsc ex1_1.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ex1_1.cpp
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ex1_1.exe
ex1_1.obj
```

Linux

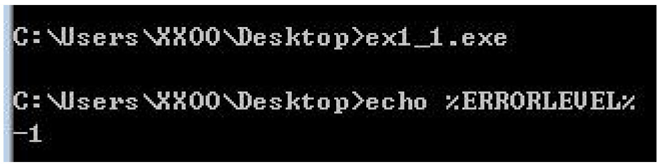


```
[pezy@localhost CppPrimer]$ g++ -o ex1_1 ex1_1.cc
[pezy@localhost CppPrimer]$ ls
ex1_1  ex1_1.cc
```

Exercise 1.2

Exercise 1.2: Change the program to return -1. A return value of -1 is often treated as an indicator that the program failed. Recompile and rerun your program to see how your system treats a failure indicator from main.

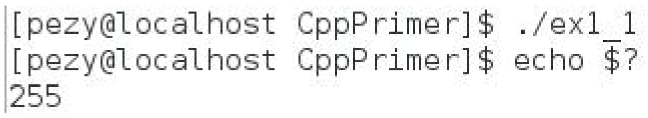
Windows



```
C:\Users\XX00\Desktop>ex1_1.exe

C:\Users\XX00\Desktop>echo %ERRORLEVEL%
-1
```

Linux



```
[pezy@localhost CppPrimer]$ ./ex1_1
[pezy@localhost CppPrimer]$ echo $?
255
```

255? why? please look at this

Exercise 1.3

Write a program to print Hello, World on the standard output.

```
#include <iostream>

int main()
{
    std::cout << "Hello, World" << std::endl;
    return 0;
}
```

Exercise 1.4

Our program used the addition operator, +, to add two numbers. Write a program that uses the multiplication operator, *, to print the product instead.

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The product of " << v1 << " and " << v2
    << " is " << v1 * v2 << std::endl;
    return 0;
}
```

Exercise 1.5

We wrote the output in one large statement. Rewrite the program to use a separate statement to print each operand.

```
#include <iostream>

int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The product of ";
    std::cout << v1;
    std::cout << " and ";
    std::cout << v2;
    std::cout << " is ";
    std::cout << v1 * v2;
    std::cout << std::endl;
    return 0;
}
```

Exercise 1.6

Explain whether the following program fragment is legal.

It's illegal.

[Error] expected primary-expression before '<<' token

Fixed it: remove the spare semicolons.

```
std::cout << "The sum of " << v1
          << " and " << v2
          << " is " << v1 + v2 << std::endl;
```

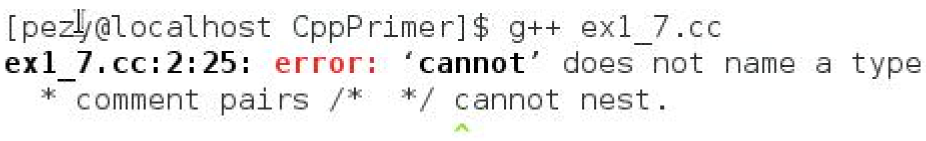
Exercise 1.7

Compile a program that has incorrectly nested comments.

Example:

```
/*
* comment pairs /* */ cannot nest.
* "cannot nest" is considered source code,
* as is the rest of the program
*/
int main()
{
    return 0;
}
```

Compiled result(g++):



```
[pezy@localhost CppPrimer]$ g++ ex1_7.cc
ex1_7.cc:2:25: error: 'cannot' does not name a type
  * comment pairs /*  */ cannot nest.
                         ^
```

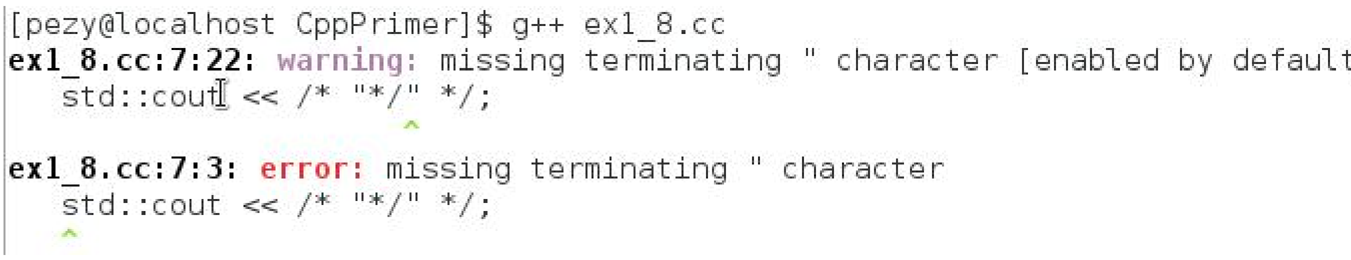
Exercise 1.8

Indicate which, if any, of the following output statements are legal:

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

After you’ve predicted what will happen, test your answers by compiling a program with each of these statements. Correct any errors you encounter.

Compiled result(g++):



```
[pezy@localhost CppPrimer]$ g++ ex1_8.cc
ex1_8.cc:7:22: warning: missing terminating " character [enabled by default
   std::cout << /* "*/" */;
                      ^

ex1_8.cc:7:3: error: missing terminating " character
   std::cout << /* "*/" */;
   ^
```

Corrected? just added a quote:

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */";
std::cout << /* "*/" /* "/*" */;
```

Output:

```
/**/ */ /*
```

Exercise 1.9

Exercise 1.10

Exercise 1.11

Exercise 1.12

What does the following for loop do? What is the final value of sum?

```
int sum = 0;
```

```
for (int i = -100; i <= 100; ++i)
sum += i;
```

the loop sums the numbers from -100 to 100. the final value of sum is zero.

Exercise 1.13

Rewrite the exercises from § 1.4.1 (p. 13) using for loops.

Ex1.9:

```
#include <iostream>

int main()
{
    int sum = 0;
    for (int i=50; i<=100; ++i)
            sum += i;

    std::cout << "the sum is: " << sum << std::endl;

    return 0;
}
```

Ex1.10:

```
#include <iostream>

int main()
{
    for (int i=10; i>=0; --i)
        std::cout << i << std::endl;

    return 0;
}
```

Ex1.11:

```
#include <iostream>

int main()
{
    int val_small = 0, val_big = 0;
    std::cout << "please input two integers:";
    std::cin >> val_small >> val_big;

    if (val_small > val_big)
    {
        int tmp = val_small;
        val_small = val_big;
        val_big = tmp;
    }

    for (int i=val_small; i<=val_big; ++i)
        std::cout << i << std::endl;

    return 0;
}
```

Exercise 1.14

Compare and contrast the loops that used a for with those using a while. Are there advantages or disadvantages to using either form?

If you need a pattern which is using a variable in a condition and incrementing that variable in the body. You should use `for` loop. Else the `while` loop is more simple.

Want to know more? look at this

Exercise 1.15

Write programs that contain the common errors discussed in the box on page 16. Familiarize yourself with the messages the compiler generates.

JUST READ IT!

Exercise 1.16

Write your own version of a program that prints the sum of a set of integers read from cin.

Many people confused about this exercise, such as this and this.

In my opinion, the exercise aim to write the program without "**END-OF-FILE**".

BUT, the code in first link is not correct.

The following are my own version:

```
#include <iostream>

int main()
{
    int limit = 0, sum = 0, value = 0;
    std::cout << "How many integers would you like to enter?";
    std::cin >> limit;

    // assume we don't know what is EOF(End-Of-File).
    while (std::cin >> value && (--limit != 0))
        sum += value;

    std::cout << sum + value << std::endl;

    return 0;
}
```

Watch out for "sum + value" in the `cout` line.

Exercise 1.17

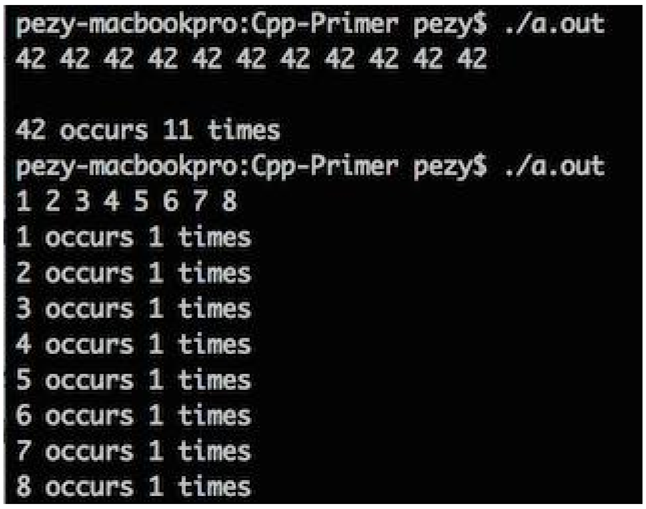
What happens in the program presented in this section if the input values are all equal? What if there are no duplicated values?

If the input values are all equal, it will print a line which shows the count of the number you input.

If there are no duplicated values, when different values input, a new line will be printed if you click `Enter`.

Exercise 1.18

Compile and run the program from this section giving it only equal values as input. Run it again giving it values in which no number is repeated.



```
pezy-macbookpro:Cpp-Primer pezy$ ./a.out
42 42 42 42 42 42 42 42 42 42 42

42 occurs 11 times
pezy-macbookpro:Cpp-Primer pezy$ ./a.out
1 2 3 4 5 6 7 8
1 occurs 1 times
2 occurs 1 times
3 occurs 1 times
4 occurs 1 times
5 occurs 1 times
6 occurs 1 times
7 occurs 1 times
8 occurs 1 times
```

Exercise 1.19

Revise the program you wrote for the exercises in § 1.4.1 (p. 13) that printed a range of numbers so that it handles input in which the first number is smaller than the second.

Yes, we should use `if` to judge which is bigger.

review this code

Exercise 1.20

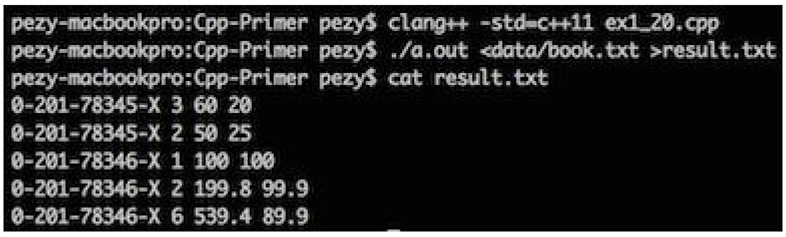
http://www.informit.com/title/032174113 contains a copy of Sales_item.h in the Chapter 1 code directory. Copy that file to your working directory. Use it to write a program that reads a set of book sales transactions, writing each transaction to the standard output.

Here is the code.

You need to enable C++11 support in your compiler. With GCC and Clang, this can be done with the `-std=c++11` option.

(Never say it again.)

How to test it? use the `book.txt` in `data` folder. And do it like this:



```
pezy-macbookpro:Cpp-Primer pezy$ clang++ -std=c++11 ex1_20.cpp
pezy-macbookpro:Cpp-Primer pezy$ ./a.out <data/book.txt >result.txt
pezy-macbookpro:Cpp-Primer pezy$ cat result.txt
0-201-78345-X 3 60 20
0-201-78345-X 2 50 25
0-201-78346-X 1 100 100
0-201-78346-X 2 199.8 99.9
0-201-78346-X 6 539.4 89.9
```

Exercise 1.21

Write a program that reads two Sales_item objects that have the same ISBN and produces their sum.

The program should check whether the objects have the same ISBN.(Have a look at 1.5.2, surprise!)

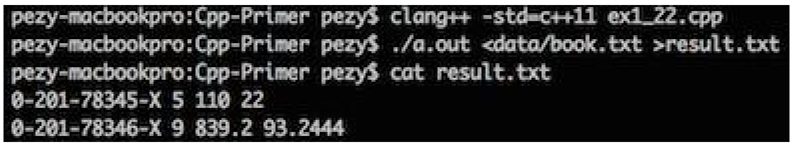
Code

Exercise 1.22

Write a program that reads several transactions for the same ISBN. Write the sum of all the transactions that were read.

Tips: this program will appear in the section 1.6.

Here is the code.



```
pezy-macbookpro:Cpp-Primer pezy$ clang++ -std=c++11 ex1_22.cpp
pezy-macbookpro:Cpp-Primer pezy$ ./a.out <data/book.txt >result.txt
pezy-macbookpro:Cpp-Primer pezy$ cat result.txt
0-201-78345-X 5 110 22
0-201-78346-X 9 839.2 93.2444
```

Exercise 1.23

Write a program that reads several transactions and counts how many transactions occur for each ISBN.

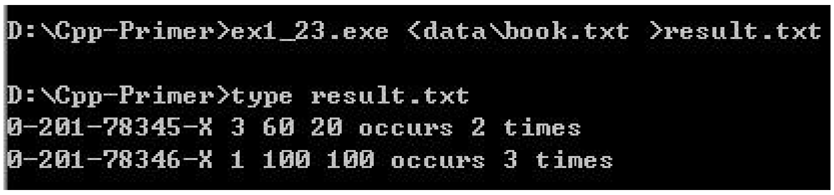
Tip: please review the 1.4.4.

Here is the code.

Exercise 1.24

Test the previous program by giving multiple transactions representing multiple ISBNs. The records for each ISBN should be grouped together.

You can use data/book.txt as the records.



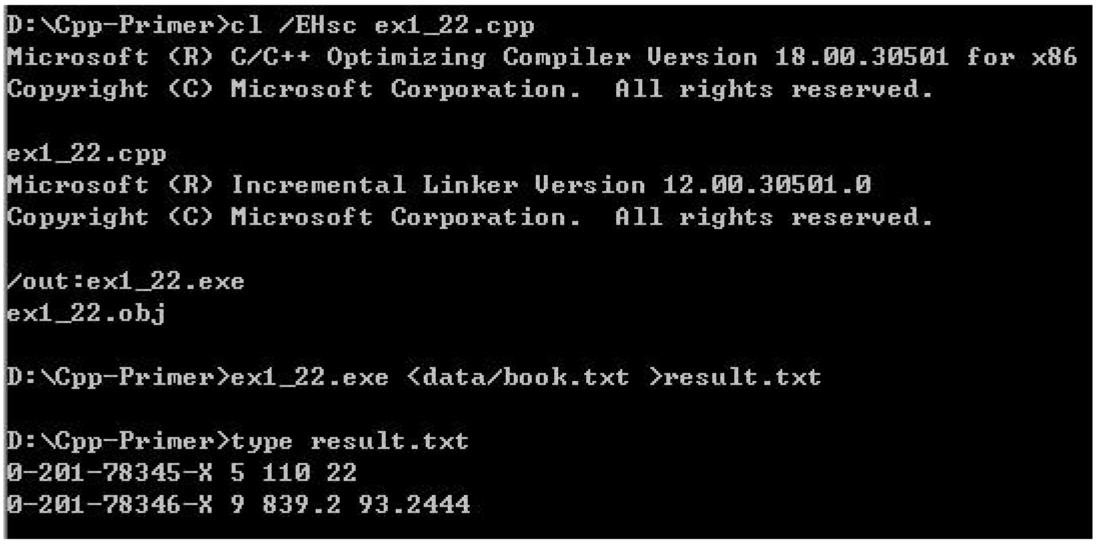
```
D:\Cpp-Primer>ex1_23.exe <data\book.txt >result.txt

D:\Cpp-Primer>type result.txt
0-201-78345-X 3 60 20 occurs 2 times
0-201-78346-X 1 100 100 occurs 3 times
```

Exercise 1.25

Using the Sales_item.h header from the Web site, compile and execute the bookstore program presented in this section.

It is the same as Exercise 1.22.



```
D:\Cpp-Primer>cl /EHsc ex1_22.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 18.00.30501 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ex1_22.cpp
Microsoft (R) Incremental Linker Version 12.00.30501.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ex1_22.exe
ex1_22.obj

D:\Cpp-Primer>ex1_22.exe <data/book.txt >result.txt

D:\Cpp-Primer>type result.txt
0-201-78345-X 5 110 22
0-201-78346-X 9 839.2 93.2444
```

Chapter 2

Exercise 2.1

What are the differences between int, long, long long, and short? Between an unsigned and a signed type? Between a float and a double?

C++ guarantees short and int is **at least** 16 bits, long **at least** 32 bits, long long **at least** 64 bits.

The signed can represent positive numbers, negative numbers and zero, while unsigned can only represent numbers no less than zero.

The C and C++ standards do not specify the representation of float, double and long double. It is possible that all three implemented as IEEE double-precision. Nevertheless, for most architectures (gcc, MSVC; x86, x64, ARM) float is indeed a IEEE **single-precision** floating point number (binary32), and double is a IEEE **double-precision** floating point number (binary64).

Usage:

> Use int for integer arithmetic. short is usually too small and, in practice, long often has the same size as int. If your data values are larger than the minimum guaranteed size of an int, then use long long. (In a word: short <**int** < long < long long)
>
> Use an unsigned type when you know that the values cannot be negative. (In a word: no negative, unsigned.)

Use double for floating-point computations; float usually does not have enough precision, and the cost of double-precision calculations versus single-precision is negligible. In fact, on some machines, double-precision operations are faster than single. The precision offered by long double usually is unnecessary and often entails considerable run-time cost. (In a word: float < **double** < long double)

Reference:

- What are the criteria for choosing between short / int / long data types?
- Difference between float and double
- Advice: Deciding which Type to Use(This book.)

Exercise 2.2

To calculate a mortgage payment, what types would you use for the rate, principal, and payment? Explain why you selected each type.

use `double`, or also `float`.

The rate most like that: 4.50 % per year. The principal most like that: $854.36 The payment most like that: $1,142.36

Reference:

- mortgage-calculator
- What's in a Mortgage Payment?

Exercise 2.3

What output will the following code produce?

```
unsigned u = 10, u2 = 42;
std::cout << u2 - u << std::endl;
std::cout << u - u2 << std::endl;
int i = 10, i2 = 42;
std::cout << i2 - i << std::endl;
std::cout << i - i2 << std::endl;
std::cout << i - u << std::endl;
std::cout << u - i << std::endl;
```

Output(g++ 4.8):

```
32
4294967264
32
-32
0
0
```

Exercise 2.4

Write a program to check whether your predictions were correct. If not, study this section until you understand what the problem is.

Here is the code, please test it in your computer.

Exercise 2.5

Determine the type of each of the following literals. Explain the differences among the literals in each of the four examples:

(a) 'a', L'a', "a", L"a"

(b) 10, 10u, 10L, 10uL, 012, 0xC

(c) 3.14, 3.14f, 3.14L

(d) 10, 10u, 10., 10e-2

(a): character literal, wide character literal, string literal, string wide character literal.

(b): decimal, unsigned decimal, long decimal, unsigned long decimal, octal, hexadecimal.

(c): double, float, long double.

(d): decimal, unsigned decimal, double, double.

Exercise 2.6

What, if any, are the differences between the following definitions:

```
int month = 9, day = 7;
int month = 09, day = 07;
```

The first line's integer is decimal.

The second line:

1. int month = 09 is invalid, cause octal don't have digit 9.

2. day is octal.

Exercise 2.7

What values do these literals represent? What type does each have?

> (a) "Who goes with F\145rgus?\012"
>
> (b) 3.14e1L
>
> (c) 1024f
>
> (d) 3.14L

(a): Who goes with Fergus?(new line) "string"

(b): 31.4 "long double"

(c): 1024 "float"

(d): 3.14 "long double"

Reference:

> [ASCII Table]

Exercise 2.8

Using escape sequences, write a program to print 2M followed by a newline.

Modify the program to print 2, then a tab, then an M, followed by a newline.

```
#include <iostream>

int main()
{
    std::cout << 2 << "\115\012";
    std::cout << 2 << "\t\115\012";

    return 0;
}
```

Exercise 2.9

Explain the following definitions. For those that are illegal, explain what’s wrong and how to correct it.

> (a) std::cin >> int input_value;
>
> (b) int i = { 3.14 };
>
> (c) double salary = wage = 9999.99;
>
> (d) int i = 3.14;

(a): error: expected '(' for function-style cast or type construction.

```
int input_value = 0;
std::cin >> input_value;
```

(b):---when you compile the code without the argument "-std=c++11", you will get the warning below: warning: implicit conversion from 'double' to 'int' changes value from 3.14 to 3. ---when you compile the code using "-std=c+11", you will get a error below: error: type 'double' cannot be narrowed to 'int' in initializer list ---conclusion: Obviously, list initialization becomes strict in c++11.

```
double i = { 3.14 };
```

(c): --if you declared 'wage' before, it's right. Otherwise, you'll get a error: error: use of undeclared identifier 'wage'

```
double wage;
double salary = wage = 9999.99;
```

(d): ok: but value will be truncated.

```
double i = 3.14;
```

Exercise 2.10

What are the initial values, if any, of each of the following variables?

```
std::string global_str;
int global_int;
int main()
{
    int local_int;
    std::string local_str;
}
```

`global_str` is global variable, so the value is empty string. `global_int` is global variable, so the value is zero.`local_int` is a local variable which is not uninitialized, so it has a undefined value. `local_str` is also a local variable which is not uninitialized, but it has a value that is defined by the class. So it is empty string. PS: please read P44 in the English version, P40 in Chinese version to get more. The note: Uninitialized objects of built-in type defined inside a function body have a undefined value. Objects of class type that we do not explicitly inititalize have a value that is defined by class.

Exercise 2.11

Explain whether each of the following is a declaration or a definition:

> (a) extern int ix = 1024;
>
> (b) int iy;
>
> (c) extern int iz;

```
(a): definition.
(b): definition.
(c): declaration.
```

Exercise 2.12

Which, if any, of the following names are invalid?

> (a) int double = 3.14;
>
> (b) int _;
>
> (c) int catch-22;
>
> (d) int 1_or_2 = 1;
>
> (e) double Double = 3.14;

`a`, `c`, `d` are invalid.

Exercise 2.13

What is the value of j in the following program?

```
int i = 42;
int main()
{
    int i = 100;
    int j = i;
}
```

100. local object named reused hides global reused.

Exercise 2.14

Is the following program legal? If so, what values are printed?

```
int i = 100, sum = 0;
for (int i = 0; i != 10; ++i)
    sum += i;
std::cout << i << " " << sum << std::endl;
```

Yes.It is legal.Printed: 100, 45.

Exercise 2.15

Which of the following definitions, if any, are invalid? Why?

(a) int ival = 1.01;

(b) int &rval1 = 1.01;

(c) int &rval2 = ival;

(d) int &rval3;

```
(a): valid.
(b): invalid. initializer must be an object.
(c): valid.
(d): invalid. a reference must be initialized.
```

Exercise 2.16

Which, if any, of the following assignments are invalid? If they are valid, explain what they do.

```
int i = 0, &r1 = i; double d = 0, &r2 = d;
```

(a) r2 = 3.14159;

(b) r2 = r1;

(c) i = r2;

(d) r1 = d;

```
(a): valid. let d equal 3.14159.
(b): valid. automatic convert will happen.
(c): valid. but value will be truncated.
(d): valid. but value will be truncated.
```

Exercise 2.17

What does the following code print?

```
int i, &ri = i;
i = 5; ri = 10;
std::cout << i << " " << ri << std::endl;
```

10, 10

Exercise 2.18

Write code to change the value of a pointer. Write code to change the value to which the pointer points.

```
int a = 0, b = 1;
int *p1 = &a, *p2 = p1;

// change the value of a pointer.
p1 = &b;
```

```
// change the value to which the pointer points
*p2 = b;
```

Exercise 2.19

Explain the key differences between pointers and references.

definition:

the pointer is "points to" any other type.

the reference is "another name" of an **object**.

key difference:

1. a reference is another name of an **already existing** object. a pointer is an object in its **own right**.
2. Once initialized, a reference remains **bound to** its initial object. There is **no way** to rebind a reference to refer to a different object. a pointer can be **assigned** and **copied**.
3. a reference always get the object to which the reference was initially bound. a single pointer can point to **several different objects** over its lifetime.
4. a reference must be initialized. a pointer need **not be** initialized at the time it is defined.

Usage advise:

Look at here

Exercise 2.20

What does the following program do?

```
int i = 42;
int *p1 = &i; *p1 = *p1 * *p1;
```

p1 pointer to i, i's value changed to 1764(42*42)

Exercise 2.21

Explain each of the following definitions. Indicate whether any are illegal and, if so, why.

```
int i = 0;
```

(a) double* dp = &i;

(b) int *ip = i;

(c) int *p = &i;

```
(a): illegal, cannot initialize a variable of type 'double *' with an
         rvalue of type 'int *'
(b): illegal, cannot initialize a variable of type 'int *' with an lvalue
         of type 'int'
(c): legal.
```

Exercise 2.22

Assuming p is a pointer to int, explain the following code:

```
if (p) // ...
if (*p) // ...
```

if (p) // whether p is nullptr?

if (*p) // whether the value pointed by p is zero?

Exercise 2.23

Given a pointer p, can you determine whether p points to a valid object? If so, how? If not, why not?

No. Because more information needed to determine whether the pointer is valid or not.

Exercise 2.24

Why is the initialization of p legal but that of lp illegal?

```
int i =42;
void *p=&i;
long *lp=&i;
```

Because the type `void*` is a special pointer type that can hold the address of any object. But we cannot initialize a variable of type `long *` with an rvalue of type `int *`

Exercise 2.25

Determine the types and values of each of the following variables.

(a) int* ip, i, &r = i;

(b) int i, *ip = 0;

(c) int* ip, ip2;

```
(a): ip is a pointer to int, i is an int, r is a reference to int i.
(b): ip is a valid, null pointer, and i is an int.
(c): ip is a pointer to int, and ip2 is an int.
```

Exercise 2.26

Which of the following are legal? For those that are illegal, explain why.

```
const int buf;           // illegal, buf is uninitialized const.
int cnt = 0;             // legal.
const int sz = cnt; // legal.
++cnt; ++sz;             // illegal, attempt to write to const object(sz).
```

Exercise 2.27

Which of the following initializations are legal? Explain why.

```
int i = -1, &r = 0;              // illegal, r must refer to an object.
int *const p2 = &i2;             // legal.
const int i = -1, &r = 0;      // legal.
const int *const p3 = &i2;    // legal.
const int *p1 = &i2;             // legal
const int &const r2;             // illegal, r2 is a reference that cannot be const.
const int i2 = i, &r = i;      // legal.
```

Exercise 2.28

Explain the following definitions. Identify any that are illegal.

```
int i, *const cp;            // illegal, cp must initialize.
int *p1, *const p2;          // illegal, p2 must initialize.
const int ic, &r = ic;    // illegal, ic must initialize.
const int *const p3;        // illegal, p3 must initialize.
const int *p;                    // legal. a pointer to const int.
```

Exercise 2.29

Uing the variables in the previous exercise, which of the following assignments are legal? Explain why.

```
i = ic;         // legal.
p1 = p3;        // illegal. p3 is a pointer to const int.
p1 = &ic;       // illegal. ic is a const int.
p3 = &ic;       // illegal. p3 is a const pointer.
p2 = p1;        // illegal. p2 is a const pointer.
ic = *p3;       // illegal. ic is a const int.
```

Exercise 2.30

For each of the following declarations indicate whether the object being declared has top-level or low-level const.

```
const int v2 = 0; int v1 = v2;
int *p1 = &v1, &r1 = v1;
const int *p2 = &v2, *const p3 = &i, &r2 = v2;
```

v2 is top-level const, p2 is low-level const. p3: right-most const is top-level, left-most is low-level. r2 is low-level const.

Exercise 2.31

Given the declarations in the previous exercise determine whether the following assignments are legal. Explain how the top-level or low-level const applies in each case.

```
r1 = v2; // legal, top-level const in v2 is ignored.
p1 = p2; // illegal, p2 has a low-level const but p1 doesn't.
p2 = p1; // legal, we can convert int* to const int*.
```

```
p1 = p3; // illegal, p3 has a low-level const but p1 doesn't.
p2 = p3; // legal, p2 has the same low-level const qualification as p3.
```

Exercise 2.32

Is the following code legal or not? If not, how might you make it legal?

```
int null = 0, *p = null;
```

illegal.

```
int null = 0, *p = nullptr;
```

Exercise 2.33

Using the variable definitions from this section, determine what happens in each of these assignments:

```
a=42; // set 42 to int a.
b=42; // set 42 to int b.
c=42; // set 42 to int c.
d=42; // ERROR, d is an int *. correct: *d = 42;
e=42; // ERROR, e is an const int *. correct: e = &c;
g=42; // ERROR, g is a const int& that is bound to ci.
```

Exercise 2.34

Write a program containing the variables and assignments from the previous exercise. Print the variables before and after the assignments to check whether your predictions in the previous exercise were correct. If not, study the examples until you can convince yourself you know ￼￼what led you to the wrong conclusion.

Here is the code.

Exercise 2.35

Determine the types deduced in each of the following definitions. Once you've figured out the types, write a program to see whether you were correct.

```
const int i = 42;
auto j = i; const auto &k = i; auto *p = &i; const auto j2 = i, &k2 = i;
```

j is int. k is const int&. p is const int *. j2 is const int. k2 is const int&.

Here is the code.

Exercise 2.36

In the following code, determine the type of each variable and the value each variable has when the code finishes:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype((b)) d = a;
++c;
++d;
```

`c` is an int, `d` is a reference of `a`. all their value are `4`.

Exercise 2.37

Assignment is an example of an expression that yields a reference type. The type is a reference to the type of the left-hand operand. That is, if i is an int, then the type of the expression i = x is int&. Using that knowledge, determine the type and value of each variable in this code:

```
int a = 3, b = 4;
decltype(a) c = a;
decltype(a = b) d = a;
```

c is an int, d is a reference of int. the value: a=3, b=4, c=3, d=3

Exercise 2.38

Describe the differences in type deduction between decltype and auto. Give an example of an expression where auto and decltype will deduce the same type and an example where they will deduce differing types.

The way decltype handles top-level const and references differs **subtly** from the way auto does. Another important difference between decltype and auto is that the deduction done by decltype depends on the **form** of its given expression.

so the key of difference is **subtly** and **form**.

```
int i = 0, &r = i;
// same
auto a = i;
decltype(i) b = i;
// different
auto c = r;
decltype(r) d = i;
```

More? Look at here and here

Exercise 2.39

Compile the following program to see what happens when you forget the semicolon after a class definition. Remember the message for future reference.

```
struct Foo { /* empty    */ } // Note: no semicolon
int main()
{
return 0;
}
```

Error message: [Error] expected ';' after struct definition

Exercise 2.40

Write your own version of the Sales_data class.

just added some your own define. like this:

```
struct Sale_data
{
    std::string bookNo;
    std::string bookName;
    unsigned units_sold = 0;
    double revenue = 0.0;
    double price = 0.0;
    //...
}
```

Exercise 2.41

Use your Sales_data class to rewrite the exercises in § 1.5.1(p. 22), § 1.5.2(p. 24), and § 1.6(p. 25). For now, you should define your Sales_data class in the same file as your main function.

1.5.1

```
#include <iostream>
#include <string>

struct Sale_data
{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

int main()
{
    Sale_data book;
    double price;
    std::cin >> book.bookNo >> book.units_sold >> price;
    book.revenue = book.units_sold * price;
    std::cout << book.bookNo << " " << book.units_sold << " " << book.revenue << " " << price;

    return 0;
}
```

1.5.2

```
#include <iostream>
#include <string>

struct Sale_data
{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

int main()
{
    Sale_data book1, book2;
    double price1, price2;
    std::cin >> book1.bookNo >> book1.units_sold >> price1;
    std::cin >> book2.bookNo >> book2.units_sold >> price2;
    book1.revenue = book1.units_sold * price1;
    book2.revenue = book2.units_sold * price2;
    if (book1.bookNo == book2.bookNo)
    {
        unsigned totalCnt = book1.units_sold + book2.units_sold;
        double totalRevenue = book1.revenue + book2.revenue;
        std::cout << book1.bookNo << " " << totalCnt << " " << totalRevenue << " ";
        if (totalCnt != 0)
            std::cout << totalRevenue/totalCnt << std::endl;
        else
            std::cout << "(no sales)" << std::endl;

        return 0;
    }
    else
    {
        std::cerr << "Data must refer to same ISBN" << std::endl;
        return -1;    // indicate failure
    }
}
```

1.6

so ugly as you see.

```
#include <iostream>
#include <string>

struct Sale_data
{
    std::string bookNo;
    unsigned units_sold = 0;
    double revenue = 0.0;
};

int main()
{
    Sale_data total;
    double totalPrice;
    if (std::cin >> total.bookNo >> total.units_sold >> totalPrice)
    {
        total.revenue = total.units_sold * totalPrice;

        Sale_data trans;
        double transPrice;
        while (std::cin >> trans.bookNo >> trans.units_sold >> transPrice)
        {
            trans.revenue = trans.units_sold * transPrice;

            if (total.bookNo == trans.bookNo)
            {
                total.units_sold += trans.units_sold;
                total.revenue += trans.revenue;
            }
            else
            {
                std::cout << total.bookNo << " " << total.units_sold << " " << total.revenue << " ";
                if (total.units_sold != 0)
                    std::cout << total.revenue/total.units_sold << std::endl;
                else
                    std::cout << "(no sales)" << std::endl;

                total.bookNo = trans.bookNo;
                total.units_sold = trans.units_sold;
                total.revenue = trans.revenue;
            }
        }

        std::cout << total.bookNo << " " << total.units_sold << " " << total.revenue << " ";
        if (total.units_sold != 0)
            std::cout << total.revenue/total.units_sold << std::endl;
        else
            std::cout << "(no sales)" << std::endl;

        return 0;
    }
    else
    {
        std::cerr << "No data?!" << std::endl;
        return -1;    // indicate failure
    }
}
```

Exercise 2.42

Write your own version of the Sales_data.h header and use it to rewrite the exercise from § 2.6.2(p. 76)

You can add some function in your header file. Look at here.

rewrite the exercise:

- 1.5.1. Code
- 1.5.2. Code
- 1.6. Code

Chapter 3

Exercise 3.1 : part1 | part2

Exercise 3.2 : part1 | part2

Exercise 3.3

Explain how whitespace characters are handled in the string input operator and in the getline function.

The getline function takes an input stream and a string. This function reads the given stream up to and including the first newline and stores what it read—not including the newline—in its string argument. After getline sees a newline, even if it is the first character in the input, it stops reading and returns. If the first character in the input is a newline, then the resulting string is the empty string. `getline` function whitespace handling, do not ignore the beginning of the line blank characters read characters until it encounters a line break, read to termination and discard newline (line breaks removed from the input stream but is not stored in the string object).

Read more

Exercise 3.4 : part1 | part2

Exercise 3.5 : part1 | part2

Exercise 3.6

Exercise 3.7

What would happen if you define the loop control variable in the previous exercise as type char? Predict the results and then change your program to use a char to see if you were right.

No different.

```
auto& c : str
```

We use `auto` to let the compiler determine the type of `c`. which in this case will be `char&`.

Exercise 3.8

Exercise 3.9

What does the following program do? Is it valid? If not, why not?

```
string s;
cout << s[0] << endl;
```

Try to get the first element of the `string`. It is invalid, cause this is **undefined behavior**.

Exercise 3.10

Exercise 3.11

Is the following range for legal? If so, what is the type of c?

```
const string s = "Keep out!";
for (auto &c : s){/*... */}
```

When you don't change `c`'s value, it's legal, else it's illegal.

For example:

```
cout << c;    // legal.
c = 'X';        // illegal.
```

The type of `c` is `const char&`. read-only variable is not assignable.

Exercise 3.12

Which, if any, of the following vector definitions are in error? For those that are legal, explain what the definition does. For those that are not legal, explain why they are illegal.

```
vector<vector<int>> ivec;               // legal(c++11), vectors.
vector<string> svec = ivec;             // illegal, different type.
vector<string> svec(10, "null");    // legal, vector have 10 strings: "null".
```

Exercise 3.13

How many elements are there in each of the following vectors? What are the values of the elements?

```
vector<int> v1;                  // size:0,    no values.
vector<int> v2(10);          // size:10, value:0
vector<int> v3(10, 42); // size:10, value:42
vector<int> v4{10};          // size:1,     value:10
vector<int> v5{10, 42}; // size:2,     value:10, 42
vector<string> v6{10};    // size:10, value:""
vector<string> v7{10, "hi"};    // size:10, value:"hi"
```

Exercise 3.14

Exercise 3.15

Exercise 3.16

Exercise 3.17

Exercise 3.18

Exercise 3.19

Exercise 3.20

Exercise 3.21

Exercise 3.22

Exercise 3.23

Exercise 3.24

Exercise 3.25

Exercise 3.26

In the binary search program on page 112, why did we write `mid=beg+(end-beg)/2;` instead of `mid=(beg+end) /2;`?

Because the iterator of vector don't define the `+` operator **between the two iterators.** `beg + end` is illegal.

We can only use the subtraction between the two iterators.

Exercise 3.27

Assuming txt_size is a function that takes no arguments and returns an int value, which of the following definitions are illegal? Explain why.

```
unsigned buf_size = 1024;

int ia[buf_size];      // illegal, The dimension value must be a constant expression.
int ia[4 * 7 - 14]; // legal
int ia[txt_size()]; // illegal, The dimension value must be a constant expression.
char st[11] = "fundamental";    // illegal, the string's size is 12.
```

Exercise 3.28

What are the values in the following arrays?

```
string sa[10];
int ia[10];

int main() {
    string sa2[10];
    int ia2[10];
}
```

please see 2.2.1. Variable Definitions -> Default Initialization.

`std::string` isn't a build-in type. The initializer will set it empty. `ia` and `ia2` are build-type. But `ia` isn't in the function body, so it will be initialized to **zero**. `ia2` is in the function body. so it's value is **undefined**.

You can also use gdb to debug the value when the code is running.

Exercise 3.29:

List some of the drawbacks of using an array instead of a vector.

1. can't add elements to an array.
2. vector is better supported bt std.

Exercise 3.30

Identify the indexing errors in the following code:

```
constexpr size_t array_size = 10;
int ia[array_size];
for (size_t ix = 1; ix <= array_size; ++ix)
        ia[ix] = ix;
```

The size of ia is 10, so the index of value should less than 10. ix **cannot** equal the array_size.

Exercise 3.31

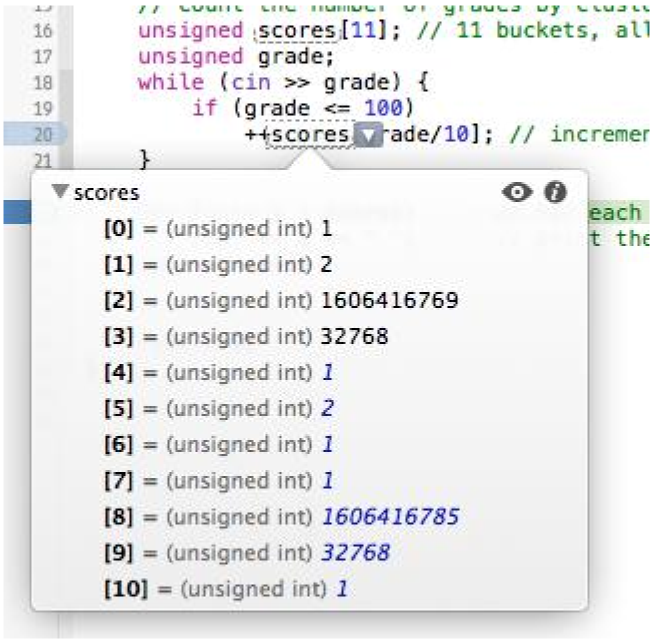
Exercise 3.32

Exercise 3.33

What would happen if we did not initialize the scores array in the program on page 116?

If we did not initialize the scores array. the array is undefined. the value will be Unknown.

Look like this:



```
    // count the number of grades by clust
    unsigned scores[11]; // 11 buckets, all
    unsigned grade;
    while (cin >> grade) {
        if (grade <= 100)
            ++scores[grade/10]; // incremen
    }
```

```
scores
    [0] = (unsigned int) 1
    [1] = (unsigned int) 2
    [2] = (unsigned int) 1606416769
    [3] = (unsigned int) 32768
    [4] = (unsigned int) 1
    [5] = (unsigned int) 2
    [6] = (unsigned int) 1
    [7] = (unsigned int) 1
    [8] = (unsigned int) 1606416785
    [9] = (unsigned int) 32768
    [10] = (unsigned int) 1
```

Exercise 3.34

Given that p1 and p2 point to elements in the same array, what does the following code do? Are there values of p1 or p2 that make this code illegal?

```
p1 += p2 - p1;
```

we assume p1 and p2 point to an array arr. so `p1 = &arr[0];` and `p2 = &arr[0]`. p2 - p1 is the distance of arr[0] to arr[0], and must be zero. so `p1 += 0;` can not change the p1's point.

`p1 += p2 - p1;` same as `p1 = p2;`. If p2 and p1 are legal, this code always legal.

Exercise 3.35

Exercise 3.36

Exercise 3.37

What does the following program do?

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char *cp = ca;
while (*cp) {
        cout << *cp << endl;
        ++cp;
}
```

Print all the elements of the array.

WARNING!!!!

When we use a string, the compiler put it in the section `.rodata`, the code uses C-style character string without adding a '\0' in the end of `ca`.

So, when we code like this:

```
const char ca[] = {'h', 'e', 'l', 'l', 'o'};
const char s[] = "world";
const char *cp = ca;
while (*cp) {
cout << *cp;
++cp;
}
```

The code will print "helloworld" when you run it. because the character list in the `.rodata` like this:

```
h e l l o w o r l d \0
```

`While(*cp)` judge weather *cp is 0 or not. when *cp is not 0, it will print the character until 0.

When you change the code like this:

```
const char ca[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

the character list in the `.rodata`:

```
h e l l o \0 w o r l d \0
```

The program will run correctly. So when using C-style character string, be careful!!

---

see `.rodata`, you can use this command:

```
hexdump -C a.out
```

Exercise 3.38

In this section, we noted that it was not only illegal but meaningless to try to add two pointers. Why would adding two pointers be meaningless?

Because Subtracting two points gives a logically explainable result - the offset in memory between two points. Similarly, you can subtract or add an integral number to/from a pointer, which means "move the pointer up or down". Adding a pointer to a pointer is something which is hard to explain. The result is meaningless.

References:

- Why can't I add pointers

Exercise 3.39

Chapter 4

Exercise 4.1

What is the value returned by 5 + 10 * 20/2?

105

Exercise 4.2

Using Table 4.12 (p. 166), parenthesize the following expressions to indicate the order in which the operands are grouped:

```
* vec.begin() //=> *(vec.begin())
* vec.begin() + 1 //=> (*(vec.begin())) + 1
```

Exercise 4.3

Order of evaluation for most of the binary operators is left undefined to give the compiler opportunities for optimization. This strategy presents a trade-off between efficient code generation and potential pitfalls in the use of the language by the programmer. Do you consider that an acceptable trade-off? Why or why not?

Yes, I think it necessary to hold the trade-off. Because the speed is always the biggest advantage of C++. Sometimes, we need the compiler's features for efficient work. But if you are not a expert. I have to advice you do not touch the undefined behaviors.

For an instance, `cout << i << ++i <<endl` should never appear in your code.

Exercise 4.4

Parenthesize the following expression to show how it is evaluated. Test your answer by compiling the expression (without parentheses) and printing its result.

```
12 / 3 * 4 + 5 * 15 + 24 % 4 / 2
// parenthesize
((12/3)*4) + (5*15) + ((24%4)/2)
// 16 + 75 + 0 = 91
// print: 91
```

Exercise 4.5

Determine the result of the following expressions.

```
-30 * 3 + 21 / 5    // -90+4 = -86
-30 + 3 * 21 / 5    // -30+63/5 = -30+12 = -18
30 / 3 * 21 % 5      // 10*21%5 = 210%5 = 0
-30 / 3 * 21 % 4    // -10*21%4 = -210%4 = -2
```

Exercise 4.6

Write an expression to determine whether an int value is even or odd.

```
i%2 == 0 ? "even" : "odd"
```

Exercise 4.7

What does overflow mean? Show three expressions that will overflow.

```
short svalue = 32767; ++svalue; // -32768
unsigned uivalue = 0; --uivalue;    // 4294967295
unsigned short usvalue = 65535; ++usvalue;    // 0
```

Exercise 4.8

Explain when operands are evaluated in the logical AND, logical OR, and equality operators.

logical `AND` : `true` only if both its operands evaluated to `true`; logical `OR` : `true` if either of its operands evaluated to`true`; equality operators : `true` only if its operands are equal.

Exercise 4.9

Explain the behavior of the condition in the following if:

```
const char *cp = "Hello World";
if (cp && *cp)
```

cp is a pointer to `const char *`, and it's not a nullptr. true.

`*cp` is a const char: 'H', and it is explicit a nonzero value. true.

true && true = true.

Exercise 4.10

Write the condition for a while loop that would read ints from the standard input and stop when the value read is equal to 42.

```
int i = 0;
while(cin >> i && i != 42)
```

Exercise 4.11

Write an expression that tests four values, a, b, c, and d, and ensures that a is greater than b, which is greater than c, which is greater than d.

```
a>b && b>c && c>d
```

Exercise 4.12

Assuming `i`, `j`, and `k` are all ints, explain what `i != j < k` means.

According to Operator precedence, `i != j < k` is same as `i != (j < k)`.

The condition group j and k to the < operator. The bool result of that expression is the right hand operand of the != operator. That is i (int) is compared to the true/false result of the first comparison! To accomplish the test we intended, we can rewrite the expression as follows:

```
i != j && j < k
```

---

reference

It is usually a bad idea to use the boolean literals true and false as operands in a comparison. These literals should be used only to compare to an object of type bool.

Exercise 4.13

What are the values of i and d after each assignment?

```
int i;     double d;
d = i = 3.5; // i = 3, d = 3.0
i = d = 3.5; // d = 3.5, i = 3
```

Exercise 4.14

Explain what happens in each of the if tests:

```
if (42 = i)     // complie error: expression is not assignable
if (i = 42)     // true.
```

Exercise 4.15

The following assignment is illegal. Why? How would you correct it?

```
double dval; int ival; int *pi;
dval = ival = pi = 0;
// pi is a pointer to int.
// can not assign to 'int' from type 'int *'
// correct it:
dval = ival = 0;
pi = 0;
```

Exercise 4.16

Although the following are legal, they probably do not behave as the programmer expects. Why? Rewrite the expressions as you think they should be.

```
if (p = getPtr() != 0)
if (i = 1024)
// why? always true. use an assigment as a condition.
// correct it
if ((p=getPtr()) != 0)
if (i == 1024)
```

Exercise 4.17

Explain the difference between prefix and postfix increment.

The postfix operators increment(or decrement) the operand but yield a copy of the original, unchanged value as its result.

The prefix operators return the object itself as an **lvalue**.

The postfix operators return a copy of the object's original value as an **rvalue**.

Exercise 4.18

What would happen if the while loop on page 148 that prints the elements from a vector used the prefix increment operator?

It will print from the second element and will dereference the v.end() at last.(It's undefined and very dangerous)

Exercise 4.19

Given that ptr points to an int, that vec is a vector, and that ival is an int, explain the behavior of each of these expressions. Which, if any, are likely to be incorrect? Why? How might each be corrected?

```
ptr != 0 && *ptr++    // check ptr is not a nullptr. and check the pointer value.
ival++ && ival // check ival and ival+1 whether equal zero.
vec[ival++] <= vec[ival] // incorrect. It is an **undefined behavior.**
// correct:
vec[ival] <= vec[ival+1]
```

Exercise 4.20

Assuming that iter is a vector::iterator, indicate which, if any, of the following expressions are legal. Explain the behavior of the legal expressions and why those that aren’t legal are in error.

```
*iter++;    // return *iter, then ++iter.
(*iter)++;    // illegal, *iter is a string, cannot increment value.
*iter.empty() // illegal, iter should use '->' to indicate whether empty.
iter->empty();    // indicate the iter' value whether empty.
```

```
++*iter;              // illegal, string have not increment.
iter++->empty();    // return iter->empty(), then ++iter.
```

Exercise 4.21

Exercise 4.22

Exercise 4.23

The following expression fails to compile due to operator precedence. Using Table 4.12 (p. 166), explain why it fails. How would you fix it?

```
string s = "word";
string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;
```

Operator Precedence: ?: < + Fix it:

```
string pl = s + (s[s.size() - 1] == 's' ? "" : "s") ;
```

Exercise 4.24

Our program that distinguished between high pass, pass, and fail depended on the fact that the conditional operator is right associative. Describe how that operator would be evaluated if the operator were left associative.

if the operator were left associative.

```
finalgrade = (grade > 90) ? "high pass" : (grade < 60) ? "fail" : "pass";
```

would same as :

```
finalgrade = ((grade > 90) ? "high pass" : (grade < 60)) ? "fail" : "pass";
```

if `grade > 90`, first conditional operator's result is `high pass`. so the finalgrade is always fail. It's contradictory obviously.

Exercise 4.25

What is the value of ~'q' << 6 on a machine with 32-bit ints and 8 bit chars, that uses Latin-1 character set in which 'q' has the bit pattern 01110001?

The final value in decimal representation is `-7296`.

The bitwise NOT operator (`~`) yields us the Ones' Complement of `0000 0000 0000 0000 0000 0000 0111 0001`, which is`1111 1111 1111 1111 1111 1111 1000 1110`. The value of `1111 1111 1111 1111 1111 1111 1000 1110` in decimal form is`-114`. This may come as a surprise to some as the unsigned value of said binary sequence is `4294967182`. The most significant bit (the left-most bit, commonly referred to as the sign bit) is however "turned on", or `1`, which signifies a negation operation on that particular bit. The value of that particular bit is then `-2147483648`.

We then shift the bits `6` digits to the left, which yields us `1111 1111 1111 1111 1110 0011 1000 0000`. Overflowing bits were discarded. The decimal representation of the binary sequence is `-7296`.

Exercise 4.26

In our grading example in this section, what would happen if we used unsigned int as the type for quiz1?

no different in most situation. unsigned int have the same size as unsigned long on most machine. But the second one could make sure that it have **at least 32 bits** on any machine.

Exercise 4.27

What is the result of each of these expressions?

```
unsigned long ul1 = 3, ul2 = 7;
ul1 & ul2 // == 3
ul1 | ul2 // == 7
ul1 && ul2 // == true
ul1 || ul2 // == ture
```

Exercise 4.28

Exercise 4.29

Predict the output of the following code and explain your reasoning. Now run the program. Is the output what you expected? If not, figure out why.

```
int x[10];      int *p = x;
cout << sizeof(x)/sizeof(*x) << endl;
cout << sizeof(p)/sizeof(*p) << endl;
```

The first result is 10. It returns the number of elements in x. But the second result depends on your machine. It would be 2 on the 64-bit machine and 1 on the 32-bit machine. Because of the size of pointer is different on various machines.

reference: Why the size of a pointer is 4bytes in C++

Exercise 4.30

Using Table 4.12 (p. 166), parenthesize the following expressions to match the default evaluation:

```
sizeof x + y           // (sizeof x)+y . "sizeof" has higher precedence than binary `+`.
sizeof p->mem[i]    // sizeof(p->mem[i])
sizeof a < b           // sizeof(a) < b
sizeof f()             //If `f()` returns `void`, this statement is undefined, otherwise it returns the size of return
type.
```

---

reference: sizeof operator

Exercise 4.31

The program in this section used the prefix increment and decrement operators. Explain why we used prefix and not postfix. What changes would have to be made to use the postfix versions? Rewrite the program using postfix operators.

p~~ostfix will copy itself as return, then increment or decrement. prefix will increment or decrement first, and return itself. so prefix is more effective in this program.(reduce one copy space.)~~

We use prefix and not postfix, just because of the Advice: Use Postfix Operators only When Necessary on §4.5. Increment and Decrement Operators.

Advice: Use Postfix Operators only When Necessary

Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version avoids unnecessary work. It increments the value and returns the incremented version.The postfix operator must store the original value so that it can return the unincremented value as its result. If we don't need the unincremented value, there's no need for the extra work done by the postfix operator.

For ints and pointers, the compiler can optimize away this extra work. For more complicated iterator types, this extra work potentially might be more costly. By habitually using the prefix versions, we do not have to worry about whether the performance difference matters. Moreover—and perhaps more importantly—we can express the intent of our programs more directly.

So, it's just a good habits. And there are no changes if we have to be made to use the postfix versions. Rewrite:

```
for(vector<int>::size_type ix = 0; ix != ivec.size(); ix++, cnt--)
        ivec[ix] = cnt;
```

This is not an appropriate example to discuss the difference of prefix and postfix. Look at the section Built-in comma operator on this page.

---

reference: Usage of the Built-in Comma Operator

Exercise 4.32

Explain the following loop.

```
constexpr int size = 5;
int ia[size] = {1,2,3,4,5};
for (int *ptr = ia, ix = 0;
     ix != size && ptr != ia+size;
     ++ix, ++ptr) { /* ... */ }
```

`ptr` and `ix` have the same function. The former use a pointer, and the latter use the index of array. we use the loop to through the array.(just choose one from `ptr` and `ix`)

Exercise 4.33

Using Table 4.12 (p. 166) explain what the following expression does:

```
someValue ? ++x, ++y : --x, --y
```

Because of the most lowest precedence of the comma operator, the expression is same as:

```
(someValue ? ++x, ++y : --x), --y
```

If someValue is true, then `++x`, and the result is `y`, if someValue is false, then `--x`, and the result is `--y`. so it is also same as:

```
someValue ? (++x,y) : (--x,--y);
```

Even though the result has nothing to do with `x`, the evaluation of `someValue` does effect the operation on `x`.

Exercise 4.34

Given the variable definitions in this section, explain what conversions take place in the following expressions: (a) if (fval) (b) dval = fval + ival; (c) dval + ival * cval; Remember that you may need to consider the associativity of the operators.

```
if (fval) // fval converted to bool
dval = fval + ival; // ival converted to fval, then the result of fval add ival converted to double.
dval + ival * cval; // cval converted to int, then that int and ival converted to double.
```

Exercise 4.35

Given the following definitions,

```
char cval; int ival; unsigned int ui; float fval; double dval;
```

identify the implicit type conversions, if any, taking place:

```
cval = 'a' + 3; // 'a' promoted to int, then the result of ('a' + 3)(int) converted to char.
fval = ui - ival * 1.0; // ival converted to double , ui also converted to double. then that double converted(by truncation) to float.
dval = ui * fval; // ui promoted to float. then that float converted to double.
cval = ival + fval + dval;    // ival converted to float, then that float and fval converted to double. At last, that double converted to char(by truncation).
```

Exercise 4.36

Assuming i is an int and d is a double write the expression i *= d so that it does integral, rather than floating-point, multiplication.

```
i *= static_cast<int>(d);
```

Exercise 4.37

Rewrite each of the following old-style casts to use a named cast:

```
int i; double d; const string *ps; char *pc; void *pv;
pv = (void*)ps; // pv = const_cast<string*>(ps); or pv = static_cast<void*>(const_cast<string*>(ps));
i = int(*pc);      // i = static_cast<int>(*pc);
pv = &d;              // pv = static_cast<void*>(&d);
pc = (char*)pv; // pc = reinterpret_cast<char*>(pv);
```

Exercise 4.38

Explain the following expression:

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

j/i is an int(by truncation), then converted to double and assigned to slope.

Chapter 5

Exercise 5.1

What is a null statement? When might you use a null statement?

null statement is the empty statement. like this:

```
; // null statement
```

I might use a null statement when the **language requires a statement but the program's logic does not. For example:

```
// read until we hit end-of-file or find an input equal to sought
while (cin >> s && s != sought)
    ; // null statement.
```

Exercise 5.2

What is a block? When might you might use a block?

block is a (possiby empty) sequence of statements and declarations

surrounded by a pair of curly braces.

I might use a block when the language requires a single statement but the logic

of our program needs more than one. For example:

```
while (val <= 10)
{
        sum += val;
        ++val;
}
```

Exercise 5.3

Use the comma operator (§ 4.10, p. 157) to rewrite the while loop from § 1.4.1

(p. 11) so that it no longer requires a block. Explain whether this rewrite

improves or diminishes the readability of this code.

```
#include <iostream>
int main()
{
        int sum = 0, val = 1;
        while (val <= 10)
                sum += val, ++val;
        std::cout << "Sum of 1 to 10 inclusive is "
                          << sum << std::endl;

        return 0;
}
```

This rewrite diminishes the readability of the code. The comma operator always guarantees the order and discards the front result. But there are no meaning in this example, however, also are incomprehensible.

Exercise 5.4

Explain each of the following examples, and correct anyproblems you detect.

> (a) while (string::iterator iter != s.end()) { /* . . . */ }
>
> (b) while (bool status = find(word)) { /* . . ./ } *if (!status) { /. . .* */ }

(a) iter point at nothing. invalid.

```
std::string::iterator iter = s.begin();
       while (iter != s.end()) { /* . . . */ }
```

(b) The if statement is not in the while's block. so the `status` is invalid. And if find(word) return true, it will go through the while block. we should declare the status before while.

```
bool status;
while ((status = find(word))) {/* ... */}
if (!status) {/* ... */}
```

In fact, the judge `!status` is unnecessary. If the `status=false`, we leave the while, and `!status` is always true.

Exercise 5.5

Exercise 5.6

Exercise 5.7

Correct the errors in each of the following code fragments:

```
(a) if (ival1 != ival2) ival1 = ival2
        else ival1 = ival2 = 0;
(b) if (ival < minval) minval = ival;
        occurs = 1;
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
        if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
(a) if (ival1 != ival2) ival1 = ival2; // lost semicolon.
        else ival1 = ival2 = 0;
(b) if (ival < minval)
        {
                minval = ival;
                occurs = 1;
        }
(c) if (int ival = get_value())
                cout << "ival = " << ival << endl;
        else if (!ival)
                cout << "ival = 0\n";
(d) if (ival == 0)
        ival = get_value();
```

Exercise 5.8

What is a “dangling else”? How are else clauses resolved in C++?

Colloquial term used to refer to the problem of how to process nested if statements in which there are more ifs than elses. In C++, an else is always paired with the closest preceding unmatched if.

Exercise 5.9

Exercise 5.10

Exercise 5.11

Exercise 5.12

Exercise 5.13

Each of the programs in the highlighted text on page 184 contains a common programming error. Identify and correct each error.

```
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
        char ch = next_text();
        switch (ch) {
                case 'a': aCnt++;
                case 'e': eCnt++;
                default: iouCnt++;
        }
(b) unsigned index = some_value();
        switch (index) {
                case 1:
                        int ix = get_value();
                        ivec[ ix ] = index;
                        break;
                default:
                        ix = ivec.size()-1;
                        ivec[ ix ] = index;
        }
(c) unsigned evenCnt = 0, oddCnt = 0;
        int digit = get_num() % 10;
        switch (digit) {
                case 1, 3, 5, 7, 9:
                        oddcnt++;
                        break;
                case 2, 4, 6, 8, 10:
                        evencnt++;
                        break;
        }
(d) unsigned ival=512, jval=1024, kval=4096;
        unsigned bufsize;
        unsigned swt = get_bufCnt();
        switch(swt) {
                case ival:
                        bufsize = ival * sizeof(int);
                        break;
                case jval:
                        bufsize = jval * sizeof(int);
                        break;
                case kval:
                        bufsize = kval * sizeof(int);
                        break;
        }
(a) unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
        char ch = next_text();
        switch (ch) {
                case 'a': aCnt++; break;
                case 'e': eCnt++; break;
                case 'i':
                case 'o':
                case 'u': iouCnt++; break;
        }
(b) unsigned index = some_value();
        int ix;
        switch (index) {
                case 1:
                        ix = get_value();
                        ivec[ ix ] = index;
                        break;
                default:
                        ix = static_cast<int>(ivec.size())-1;
                        ivec[ ix ] = index;
        }
(c) unsigned evenCnt = 0, oddCnt = 0;
        int digit = get_num() % 10;
        switch (digit) {
```

```
        case 1: case 3: case 5: case 7: case 9:
            oddcnt++;
            break;
        case 2: case 4: case 6: case 8: case 0:
            evencnt++;
            break;
    }
(d) const unsigned ival=512, jval=1024, kval=4096;
    unsigned bufsize;
    unsigned swt = get_bufCnt();
    switch(swt) {
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
    }
```

Exercise 5.14

Write a program to read strings from standard input looking for duplicated words. The program should find places in the input where one word is followed immediately by itself. Keep track of the largest number of times a single repetition occurs and which word is repeated. Print the maximum number of duplicates, or else print a message saying that no word was repeated. For example, if the input is

```
how now now now brown cow cow
```

the output should indicate that the word now occurred three times.

concise solution

easy to understand

Exercise 5.15

Explain each of the following loops. Correct any problems you detect.

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ... */ }
        if (ix != sz)
        // . . .
(b) int ix;
        for (ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix, ++sz) { /*...*/ }
(a) int ix;
        for (ix = 0; ix != sz; ++ix)    { /* ... */ }
        if (ix != sz)
        // . . .
(b) int ix;
        for (; ix != sz; ++ix) { /* ... */ }
(c) for (int ix = 0; ix != sz; ++ix) { /*...*/ }
```

Exercise 5.16

The while loop is particularly good at executing while some condition holds; for example, when we need to read values until end-of-file. The for loop is generally thought of as a **step loop**: An index steps through a range of values in a collection. Write an idiomatic use of each loop and then rewrite each using the other loop construct. If you could use only one loop, which would you choose? Why?

```
// while idiomatic
int i;
while ( cin >> i )
        // ...
// same as for
for (int i = 0; cin >> i;)
        // ...

// for idiomatic
for (int i = 0; i != size; ++i)
        // ...

// same as while
int i = 0;
while (i != size)
{
        // ...
        ++i;
}
```

I prefer `for` to `while` in such cases, because it's terse. More importantly, object i won't pollute the external scope after it goes out of the loop. It's a little bit easier to add new code into the external scope, since it reduces the possibility of naming conflicts .That is, a higher maintainability. Of course, this way makes the code a bit harder to read. (@Mooophy)

Exercise 5.17

Exercise 5.18

Explain each of the following loops. Correct any problems you detect.

```
(a) do { // added bracket.
                int v1, v2;
                cout << "Please enter two numbers to sum:" ;
                if (cin >> v1 >> v2)
                        cout << "Sum is: " << v1 + v2 << endl;
        }while (cin);
(b) int ival;
        do {
                // . . .
        } while (ival = get_response()); // should not declared in this scope.
(c) int ival = get_response();
        do {
                ival = get_response();
        } while (ival); // ival is not declared in this scope.
```

Exercise 5.19

Exercise 5.20

Exercise 5.21

Exercise 5.22

The last example in this section that jumped back to begin could be better written using a loop. Rewrite the code to eliminate the goto.

```
// backward jump over an initialized variable definition is okay
begin:
        int sz = get_size();
        if (sz <= 0) {
                goto begin;
        }
```

use for to replace goto:

```
for (int sz = get_size(); sz <=0; sz = get_size())
        ; // should not remove.
```

Exercise 5.23

Exercise 5.24

Exercise 5.25

Chapter 6

Exercise 6.1

Parameters: Local variable declared inside the function parameter list. they are initialized by the **arguments** provided in the each function call.

Arguments: Values supplied in a function call that are used to initialize the function's **parameters**.

Exercise 6.2

```
(a) string f() {
            string s;
            // ...
            return s;
    }
(b) void f2(int i) { /* ... */ }
(c) int calc(int v1, int v2) { /* ... */ }
(d) double square (double x) { return x * x; }
```

Exercise 6.3

```
#include <iostream>

int fact(int val)
{
    if (val == 0 || val == 1) return 1;
```

```
        else return val * fact(val-1);
}

int main()
{
        int j = fact(5);    // j equals 120, i.e., the result of fact(5)
        std::cout << "5! is " << j << std::endl;
        return 0;
}
```

Exercise 6.4

Exercise 6.5

```
template <typename T>
T abs(T i)
{
        return i >= 0 ? i : -i;
}
```

Exercise 6.6

local variable: Variables defined inside a **block**;

parameter: **Local variables** declared inside the **function parameter list**

local static variable: **local static variable (object)** is initialized before the first time execution passes through the object's definition. **Local statics** are not destroyed when a function ends; they are **destroyed when the program terminates.**

```
// example
size_t count_add(int n)            // n is a parameter.
{
        static size_t ctr = 0;        // ctr is a static variable.
        ctr += n;
        return ctr;
}

int main()
{
        for (size_t i = 0; i != 10; ++i)     // i is a local variable.
            cout << count_add(i) << endl;

        return 0;
}
```

Exercise 6.7

```
size_t generate()
{
        static size_t ctr = 0;
        return ctr++;
}
```

Exercise 6.8

Exercise 6.9 fact.cc | factMain.cc

Exercise 6.10

Exercise 6.11

Exercise 6.12

Exercise 6.13

void f(T) pass the argument by value. **nothing the function does to the parameter can affect the argument**. void f(T&) pass a reference, will be **bound to** whatever T object we pass.

Exercise 6.14

a parameter should be a reference type:

```
void reset(int &i)
{
        i = 0;
}
```

a parameter should not be a reference:

```
void print(std::vector<int>::iterator begin, std::vector<int>::iterator end)
{
        for (std::vector<int>::iterator iter = begin; iter != end; ++iter)
                std::cout << *iter << std::endl;
}
```

Exercise 6.15

why is s a reference to const but occurs is a plain reference?

cause the s should not be changed by this function. but occurs's result must be calculated by the function.

Why are these parameters references, but the char parameter c is not?

casue c maybe a temp varable. such as find_char(s, 'a', occurs)

What would happen if we made `s` a plain reference? What if we made `occurs` a reference to const?

`s` could be changed in the function, and `occurs` whould not be changed. so `occurs = 0;` is an error.

Exercise 6.16

```
bool is_empty(const string& s) { return s.empty(); }
```

Since this function doesn't change the argument,"const" shoud be added before string&s,otherwise this function is misleading and can't be used with const string or in a const function.

Exercise 6.17

Not the same. For the first one "const" was used, since no change need to do for the argument. For the second function,"const" can't be used,because the content of the agument should be changed.

Exercise 6.18

(a)

```
bool compare(matrix &m1, matrix &m2){ /.../ }
```

(b)

```
vector<int>::iterator change_val(int, vector<int>::iterator) { /.../ }
```

Exercise 6.19

(a) illegal, only one parameter. (b) legal. (c) legal. (d) legal.

Exercise 6.20

If we can use `const`, just use it. If we make a parameter a plain reference when it could be a reference to `const`, the reference value maybe changed.

Exercise 6.21

Exercise 6.22

Exercise 6.23

Exercise 6.24

Arrays have two special properties that affect how we define and use functions that operate on arrays: We cannot copy an array, and when we use an array it is (usually) **converted to a pointer**.

So we cannot pass an array by value, and when we pass an array to a function, we are actually passing a pointer to the array's first element.

In this question, `const int ia[10]` is actually same as `const int*`, and the size of the array is **irrelevant**. we can pass`const int ia[3]` or `const int ia[255]`, there are no differences. If we want to pass an array which size is ten, we should use reference like that:

```
void print10(const int (&ia)[10]) { /*...*/ }
```

see more discusses at http://stackoverflow.com/questions/26530659/confused-about-array-parameters

Exercise 6.25 && Exercise 6.26

Exercise 6.27

Exercise 6.28

The type of `elem` in the `for` loop is `const std::string&`.

Exercise 6.29

We should use `const reference` as the loop control variable. because the elements in an `initializer_list` are always const values, so we cannot change the value of an element in an `initializer_list`.

Exercise 6.30

Error (Clang):

Non-void function 'str_subrange' should return a value. // error #1

Control may reach end of non-void function. // error #2

Exercise 6.31

when you can find the preexited object that the reference refered.

Exercise 6.32

legal, it gave the values (0 ~ 9) to array `ia`.

Exercise 6.33(Generics Version)

Exercise 6.34

When the recursion termination condition becomes `var != 0`, two situations can happen : case 1 : If the argument is positive, recursion stops at 0. This is exactly what @shbling pointed out. case 2 : if the argument is negative, recursion would never stop. As a result,a stack overflow would occur.

Exercise 6.35

the recursive function will always use `val` as the parameter. *a recursion loop* would happen.

Exercise 6.36

```
string (&func(string (&arrStr)[10]))[10]
```

Exercise 6.37

```
using ArrT = string[10];
ArrT& func1(ArrT& arr);

auto func2(ArrT& arr) -> string(&)[10];

string arrS[10];
decltype(arrS)& func3(ArrT& arr);
```

I pefer the first one. because it is more simpler to me.

Exercise 6.38

```
decltype(arrStr)& arrPtr(int i)
{
          return (i % 2) ? odd : even;
}
```

Exercise 6.39

(a) illegal

(b) illegal

(c) legal

Exercise 6.40

(a) no error

(b) Missing default argument on parameter 'wd', 'bckgrnd'.

Exercise 6.41

(a) illegal. No matching function for call to 'init'.

(b) legal, and match.

(c) legal, but not match. `wd` whould be setting to '*'.

Exercise 6.42

Exercise 6.43

Both two should put in a header. (a) is an inline function. (b) is the declaration of useful function. we always put them in the header.

Exercise 6.44

Exercise 6.45

For example, the function arrPtr in Exercise 6.38 and make_plural in Exercise 6.42 should be defined as inline. But the function func in Exercise 6.4 shouldn't. Cause it just being call once and too many codes in the function.

Exercise 6.46

Yes.

```
constexpr bool isShorter(const string& str1, const string& str2)
{
	return str1.size() < str2.size();
}
```

If you want know more about constexpr function, maybe it is useful to you.

Exercise 6.47

Exercise 6.48

This loop let user input a word all the way until the word is sought.

It isn't a good use of assert. because if user begin to input a word, the cin would be always have content. so the assertwould be always true. It is meaningless. using assert(s == sought) is more better.

Exercise 6.49

candidate function:

Set of functions that are considered when resolving a function call. (all the functions with the name used in the call for which a declaration is in scope at the time of the call.)

viable function:

Subset of the candidate functions that could match a given call. It have the same number of parameters as arguments to the call, and each argument type can be converted to the corresponding parameter type.

Exercise 6.50

(a) illegal. 2.56 match the double, but 42 match the int.

(b) match void f(int).

(c) match void f(int, int).

(d) match void f(double, double = 3.14).

Exercise 6.51

Exercise 6.52

(a) Match through a promotion

(b) Arithmetic type conversion

Exercise 6.53

(a)

```
int calc(int&, int&); // calls lookup(int&)
int calc(const int&, const int&); // calls lookup(const int&)
```

(b)

```
int calc(char*, char*); // calls lookup(char*)
int calc(const char*, const char*); calls lookup(const char *)
```

(c)

illegal. both calls lookup(char*)

Exercise 6.54

```
int func(int a, int b);

using pFunc1 = decltype(func) *;
typedef decltype(func) *pFunc2;
using pFunc3 = int (*)(int a, int b);
using pFunc4 = int(int a, int b);
typedef int(*pFunc5)(int a, int b);
using pFunc6 = decltype(func);

std::vector<pFunc1> vec1;
std::vector<pFunc2> vec2;
std::vector<pFunc3> vec3;
std::vector<pFunc4*> vec4;
std::vector<pFunc5> vec5;
std::vector<pFunc6*> vec6;
```

Exercise 6.55

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }
int multiply(int a, int b) { return a * b; }
int divide(int a, int b) { return b != 0 ? a / b : 0; }
```

Exercise 6.56

```
std::vector<decltype(func) *> vec{add, subtract, multiply, divide};
for (auto f : vec)
		std::cout << f(2, 2) << std::endl;
```

Chapter 7

Exercise 7.1

Exercise 7.2

Exercise 7.3

Exercise 7.4

Exercise 7.5

Exercise 7.6

Exercise 7.7

Exercise 7.8

Define `read`'s Sales_data parameter as plain reference since it's intended to change the `revenue`'s value.

Define `print`'s Sales_data parameter as a reference to const since it isn't intended to change any member's value of this object.

Exercise 7.9

Exercise 7.10

```
if(read(read(cin, data1), data2))
```

we can try to divide it like that:

```
std::istream &firstStep = read(cin, data1);
sdt::istream &secondStep = read(firstStep, data2);
if (secondStep)
```

the condition of the `if` statement would read two Sales_data object at one time.

Exercise 7.11 Header|CPP

Exercise 7.12

Exercise 7.13

Exercise 7.14

```
Sales_data() : units_sold(0) , revenue(0){}
```

Exercise 7.15

Exercise 7.16

There are no restrictions on how often an access specifier may appear.The specified access level remains in effect until the next access specifier or the end of the class body.

The members which are accessible to all parts of the program should define after a public specifier.

The members which are accessible to the member functions of the class but are not accessible to code that uses the class should define after a private specifier.

Exercise 7.17

The only difference between using `class` and using `struct` to define a class is the default access level. (`class` : private, `struct` : public)

Exercise 7.18

encapsulation is the separation of implementation from interface. It hides the implementation details of a type. (In C++, encapsulation is enforced by putting the implementation in the private part of a class)

Important advantages:

- User code cannot inadvertently corrupt the state of an encapsulation object.
- The implementation of an encapsulated class can change over time without requiring changes in user-level code.

Exercise 7.19

public include: constructors, `getName()`, `getAddress()`. private include: `name`, `address`.

the interface should be defined as public, the data shouldn't expose to outside of the class.

Exercise 7.20

`friend` is a mechanism by which a class grants access to its nonpublic members. They have the same rights as members.

Pros:

- the useful functions can refer to class members in the class scope without needing to explicitly prefix them with the class name.

you can access all the nonpublic members conveniently.

sometimes, more readable to the users of class.

Cons:

lessens encapsulation and therefore maintainability.

code verbosity, declarations inside the class, outside the class.

Exercise 7.21

Exercise 7.22

Exercise 7.23 Header|CPP

Exercise 7.24

Exercise 7.25

The class below can rely on it. It goes in *Section 7.1.5*:

..the synthesized versions are unlikely to work correctly for classes that allocate resources that reside outside the class objects themselves.

Moreover, the synthesized versions for copy, assignment, and destruction work correctly for classes that have**vector or string members**.

Hence the class below which used only built-in type and strings can rely on the default version of copy and assignment. (by @Mooophy)

Exercise 7.26 Header|CPP

Exercise 7.27 Class|Test

Exercise 7.28

The second call to `display` couldn't print `#` among the output, cause the call to `set` would change the **temporary copy**, not myScreen.

Exercise 7.29

```
#with '&'
XXXXXXXXXXXXXXXXXXXX#XXXX
XXXXXXXXXXXXXXXXXXXX#XXXX
                    ^^^
# without '&'
XXXXXXXXXXXXXXXXXXXX#XXXX
XXXXXXXXXXXXXXXXXXXXXXXXX
                    ^^^
```

Exercise 7.30

Pros

- more explicit
- less scope for misreading

can use the member function parameter which name is same as the member name.

```
void setAddr(const std::string &addr) {this->addr = addr;}
```

Cons

more to read

sometimes redundant

```
std::string getAddr() const { return this->addr; } // unnecessary
```

Exercise 7.31

Exercise 7.32

Exercise 7.33

[clang]error: unknown type name 'pos'

fixed:

```
Screen::pos Screen::size() const
{
    return height*width;
}
```

Exercise 7.34

There is an error in

```
dummy_fcn(pos height)
                   ^
Unknown type name 'pos'
```

Exercise 7.35

```
typedef string Type;
Type initVal(); // use `string`
class Exercise {
public:
    typedef double Type;
    Type setVal(Type); // use `double`
    Type initVal(); // use `double`
private:
    int val;
};

Type Exercise::setVal(Type parm) {    // first is `string`, second is `double`
    val = parm + initVal();          // Exercise::initVal()
    return val;
}
```

fixed

changed

```
Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}
```

to

```
Exercise::Type Exercise::setVal(Type parm) {
    val = parm + initVal();
    return val;
}
```

and `Exercise::initVal()` should be defined.

Exercise 7.36

In this case, the constructor initializer makes it appear as if `base` is initialized with `i` and then `base` is used to initialize `rem`. However, `base` is initialized first. The effect of this initializer is to initialize `rem` with the undefined value of `base`!

fixd

```
struct X {
    X (int i, int j): base(i), rem(base % j) { }
    int base, rem;
};
```

Exercise 7.37

```
Sales_data first_item(cin);      // use Sales_data(std::istream &is) ; its value are up to your input.

int main() {
    Sales_data next;    // use Sales_data(std::string s = ""); bookNo = "", cnt = 0, revenue = 0.0
    Sales_data last("9-999-99999-9"); // use Sales_data(std::string s = ""); bookNo = "9-999-99999-9", cnt = 0,
revenue = 0.0
}
```

Exercise 7.38

```
Sales_data(std::istream &is = std::cin) { read(is, *this); }
```

Exercise 7.39

illegal. cause the call of overloaded 'Sales_data()' is **ambiguous**.

Exercise 7.40

Such as `Book`

```
class Book {
public:
    Book() = default;
    Book(unsigned no, std::string name, std::string author, std::string pubdate):no_(no), name_(name), author_(author), pubdate_(pubdate) { }
    Book(std::istream &in) { in >> no_ >> name_ >> author_ >> pubdate_; }

private:
    unsigned no_;
    std::string name_;
    std::string author_;
    std::string pubdate_;
};
```

Exercise 7.41 Header|Cpp|Test

Exercise 7.42

```
class Book {
public:
    Book(unsigned no, std::string name, std::string author, std::string pubdate):no_(no), name_(name), author_(author), pubdate_(pubdate) { }
    Book() : Book(0, "", "", "") { }
    Book(std::istream &in) : Book() { in >> no_ >> name_ >> author_ >> pubdate_; }

private:
    unsigned no_;
    std::string name_;
    std::string author_;
    std::string pubdate_;
};
```

Exercise 7.43

Exercise 7.44

illegal, cause there are ten elements, each would be default initialized. But no default initializer for the temporary object.

Exercise 7.45

No problem. cause `C` have the default constructor.

Exercise 7.46

> a) A class must provide at least one constructor. (**untrue**, "The compiler-generated constructor is known as the synthesized default constructor.")
>
> b) A default constructor is a constructor with an empty parameter list. (**untrue**, A default constructor is a constructor that is used if no initializer is supplied.What's more, A constructor that supplies default arguments for all its parameters also defines the default constructor)
>
> c) If there are no meaningful default values for a class, the class should not provide a default constructor. (**untrue**, the class should provide.)
>
> d) If a class does not define a default constructor, the compiler generates one that initializes each data member to the default value of its associated type. (**untrue**, only if our class does not explicitly define **any constructors**, the compiler will implicitly define the default constructor for us.)

Exercise 7.47

Whether the conversion of a `string` to `Sales_data` is desired **depends on how we think our users will use the conversion**. In this case, it might be okay. The `string` in null_book probably represents a nonexistent ISBN.

Benefits:

- prevent the use of a constructor in a context that requires an implicit conversion
- we can define a constructor which is used only with the direct form of initialization

Drawbacks:

- meaningful only on constructors that can be called with a single argument

Exercise 7.48

Both are noting happened.

Exercise 7.49

```
(a) Sales_data &combine(Sales_data); // ok
(b) Sales_data &combine(Sales_data&); // [Error] no matching function for call to 'Sales_data::combine(std::string&)' (`std::string&` can not convert to `Sales_data` type.)
(c) Sales_data &combine(const Sales_data&) const; // The trailing const mark can't be put here, as it forbids any mutation on data members. This comflicts with combine's semantics.
```

Some detailed explanation about problem (b) :It's wrong. Because `combine`'s parameter is a non-const reference , we can't pass a temporary to that parameter. If combine's parameter is a reference to const , we can pass a temporary to that parameter. Like this :Sales_data &combine(const Sales_data&); Here we call the Sales_data combine member function with a string argument. This call is perfectly legal; the compiler automatically creates a Sales_data object from the given string. That newly generated (temporary) Sales_data is passed to combine.(Also you can read C++ Primer Page 295(English Edition))

Exercise 7.50

Exercise 7.51

Such as a function like that:

```
int getSize(const std::vector<int>&);
```

if vector has not defined its single-argument constructor as explicit. we can use the function like:

```
getSize(34);
```

What is this mean? It's very confused.

But the std::string is different. In ordinary, we use std::string to replace const char *(the C language). so when we call a function like that:

```
void setYourName(std::string); // declaration.
setYourName("pezy"); // just fine.
```

it is very natural.

Exercise 7.52

In my opinion ,the aim of the problem is Aggregate Class. Test-makers think that Sales_data is Aggregate Class,soSales_data should have no in-class initializers if we want to initialize the data members of an aggregate class by providing a braced list of member initializers:

FIXED:

```
struct Sales_data {
        std::string bookNo;
        unsigned units_sold;
        double revenue;
};
```

Exercise 7.53

Exercise 7.54

shouldn't, cause a constexpr function must contain exactly one **return** statement.

Exercise 7.55

yes.

An aggregate class whose data members are all of literal type is a literal class.

Exercise 7.56

What is a static class member?

A class member that is **associated with the class**, rather than with individual objects of the class type.

What are the advantages of static members?

each object can no need to store a common data. And if the data is changed, each object can use the new value.

How do they differ from ordinary members?

- a static data member can have **incomplete type**.
- we can use a static member **as a default argument**.

Exercise 7.57

Exercise 7.58

```
static double rate = 6.5;
                      ^
            rate should be a constant expression.

static vector<double> vec(vecSize);
                        ^
            we may not specify an in-class initializer inside parentheses.
```

Fixed:

```
// example.h
class Example {
public:
    static constexpr double rate = 6.5;
    static const int vecSize = 20;
```

```
	static vector<double> vec;
};

// example.C
#include "example.h"
constexpr double Example::rate;
vector<double> Example::vec(Example::vecSize);
```

Chapter 8

Chapter 8. The IO Library

Exercise 8.1:

Write a function that takes and returns an istream&. The function should read the stream until it hits end-of-file. The function should print what it reads to the standard output. Reset the stream so that it is valid before returning the stream.

```
istream& func(istream &is)
{
	std::string buf;
	while (is >> buf)
		std::cout << buf << std::endl;
	is.clear();
	return is;
}
```

Exercise 8.2

Exercise 8.3:

What causes the following while to terminate?

```
while (cin >> i) /*    ...        */
```

putting cin in an error state cause to terminate. such as eofbit, failbit and badbit.

Exercise 8.4

Exercise 8.5

Exercise 8.6

Exercise 8.7

Exercise 8.8

Exercise 8.9

Exercise 8.10

Exercise 8.11

Exercise 8.12:

Why didn’t we use in-class initializers in PersonInfo?

Cause we need a aggregate class here. so it should have no in-class initializers.

Exercise 8.13

Exercise 8.14:

Why did we declare entry and nums as const auto &?

> cause they are all class type, not the built-in type. so **reference** more effective.
>
> output shouldn't change their values. so we added the const.

Chapter 9. Sequential Containers

Exercise 9.1:

Which is the most appropriate—a vector, a deque, or a list—for the following program tasks?Explain the rationale for your choice.If there is no reason to prefer one or another container, explain why not.

(a) Read a fixed number of words, inserting them in the container alphabetically as they are entered. We'll see in the next chapter that associative containers are better suited to this problem.

(b) Read an unknown number of words. Always insert new words at the back. Remove the next value from the front.

(c) Read an unknown number of integers from a file. Sort the numbers and then print them to standard output.

(a) `std::set` is the best. now, we can select `vector` or `deque`, better than `list`, cause we don't need insert or delete elements in the middle.

(b) `deque`. If the program needs to insert or delete elements at the front and the back, but not in the middle, use a deque

(c) `vector`, no need that insert or delete at the front or back. and If your program has lots of small elements and space overhead matters, don't use list or forward_list.

Exercise 9.2:

Define a list that holds elements that are deques that hold ints.

```
std::list<std::deque<int>> ldi;
```

Exercise 9.3:

What are the constraints on the iterators that form iterator ranges?

two iterators, `begin` and `end`:

they refer to elements of the same container.

It is possible to reach end by repeatedly incrementing begin.

Exercise 9.4:

Write a function that takes a pair of iterators to a vector and an int value. Look for that value in the range and return a bool indicating whether it was found.

```
bool find(vector<int>::iterator beg, vector<int>::iterator end, int value)
{
    for (auto iter = beg; iter != end; ++iter)
        if (*iter == value) return true;
    return false;
}
```

Exercise 9.5:

Rewrite the previous program to return an iterator to the requested element. Note that the program must handle the case where the element is not found.

```
vector<int>::iterator find(vector<int>::iterator beg, vector<int>::iterator end, int value)
{
    for (auto iter = beg; iter != end; ++iter)
        if (*iter == value) return iter;
    return end;
}
```

Exercise 9.6:

What is wrong with the following program? How might you correct it?

```
list<int> lst1;
list<int>::iterator iter1 = lst1.begin(), iter2 = lst1.end();
while (iter1 < iter2) /*ERROR: operator< can't be applied to iterator for list*/
```

Fixed:

```
while(iter1 != iter2)
```

note:

operator `<` can be used in `list`,but can't be applied to iterator for `list`.

Exercise 9.7:

What type should be used as the index into a vector of ints?

```
vector<int>::size_type
```

Exercise 9.8:

What type should be used to read elements in a list of strings? To write them?

```
list<string>::iterator || list<string>::const_iterator // read
list<string>::iterator // write
```

Exercise 9.9:

What is the difference between the `begin` and `cbegin` functions?

`cbegin` is a const member that returns the container's **const_iterator** type.

`begin` is nonconst and returns the container's **iterator** type.

Exercise 9.10:

What are the types of the following four objects?

```
vector<int> v1;
const vector<int> v2;
auto it1 = v1.begin(), it2 = v2.begin();
auto it3 = v1.cbegin(), it4 = v2.cbegin();
```

@shidenggui:

The question example codes have an error in `gcc 4.8`:

error: inconsistent deduction for 'auto': '__gnu_cxx::__normal_iterator >' and then '__gnu_cxx::__normal_iterator >' auto it1 = v1.begin(), it2 = v2.begin();

the correct codes should be:

```
auto it1 = v1.begin();
auto it2 = v2.begin(), it3 = v1.cbegin(), it4 = v2.cbegin();
```

`it1` is `vector<int>::iterator`

`it2`,`it3` and `it4` are `vector<int>::const_iterator`

Exercise 9.11:

Show an example of each of the six ways to create and initialize a vector.

Explain what values each vector contains.

```
vector<int> vec;        // 0
vector<int> vec(10);        // 0
vector<int> vec(10,1);    // 1
vector<int> vec{1,2,3,4,5}; // 1,2,3,4,5
vector<int> vec(other_vec); // same as other_vec
vector<int> vec(other_vec.begin(), other_vec.end()); // same as other_vec
```

Exercise 9.12:

Explain the differences between the constructor that takes a container to copy and the constructor that takes two iterators.

Constructor that takes two iterators copies the items between [first, last),e.g.

```
auto data = { 1, 2, 3 };
std::vector<int> vec(data.begin(), data.begin()+1); // vec is {1}
```

Constructor that takes another container copies all items from it. e.g.

```
auto data = { 1, 2, 3 };
std::vector<int> vec(data); //vec is {1,2,3}
```

Exercise 9.13

Exercise 9.14

Exercise 9.15

Exercise 9.16

Exercise 9.17:

Assuming c1 and c2 are containers, what (if any) constraints does the following usage place on the types of c1 and c2?

First, there must be the identical container and same type holded. Second,the type held must support relational operation. (@Mooophy)

Both c1 and c2 are the containers except the unordered associative containers.(@pezy)

Exercise 9.18

Exercise 9.19

Exercise 9.20

Exercise 9.21:

Explain how the loop from page 345 that used the return from insert to add elements to a list would work if we inserted into a vector instead.

It's the same.

The first call to insert takes the string we just read and puts it in front of the element denoted by iter. The value returned by insert is an iterator referring to this new element. We assign that iterator to iter and repeat thewhile, reading another word. As long as there are words to insert, each trip through the while inserts a new element ahead of iter and reassigns to iter the location of the newly inserted element. That element is the (new) first element. Thus, each iteration inserts an element ahead of the first element in the vector.

Exercise 9.22:

Assuming iv is a vector of ints, what is wrong with the following program? How might you correct the problem(s)?

```
vector<int>::iterator iter = iv.begin(), mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*iter == some_val)
        iv.insert(iter, 2 * some_val);
```

Problems:

1. It's a endless loop. iter never equal mid.
2. mid will be invalid after the insert.(see issue 133)

FIXED:

```
// cause the reallocation will lead the iterators and references
// after the insertion point to invalid. Thus, we need to call reserver at first.

vector<int> iv = {0,1,2,3,4,5,6,7,8,9}; // For example.
iv.reserver(25); // make sure that enough

vector<int>::iterator iter = iv.begin(), mid = iv.begin() + iv.size()/2;
while (iter != mid)
    if (*mid == some_val)
        mid = iv.insert(mid, 2 * some_val);
    else
        --mid;
```

The complete test codes, check this.

Exercise 9.23:

In the first program in this section on page 346, what would the values of val, val2, val3, and val4 be if c.size() is 1?

same value that equal to the first element's.

Exercise 9.24

Exercise 9.25:

In the program on page 349 that erased a range of elements, what happens if elem1 and elem2 are equal? What if elem2 or both elem1 and elem2 are the off-the-end iterator?

if elem1 and elem2 are equal, nothing happened.

if elem2 is the off-the-end iterator, it would delete from elem1 to the end.

if both elem1 and elem2 are the off-the-end iterator, nothing happened too.

Exercise 9.26

Exercise 9.27

Exercise 9.28:

Write a function that takes a forward_list and two additional string arguments. The function should find the first string and insert the second immediately following the first. If the first string is not found, then insert the second string at the end of the list.

```
void insert(forward_list<string> &flst, string find, string insrt)
{
        auto prev = flst.before_begin();
        for (auto curr = flst.begin(); curr != flst.end(); prev = curr++)
                if (*curr == find)
                {
                        flst.insert_after(curr, insrt);
                        return;
                }
        flst.insert_after(prev, insrt);
}
```

Exercise 9.29:

Given that vec holds 25 elements, what does vec.resize(100) do? What if we next wrote vec.resize(10)?

```
vec.resize(100);        // adds 75 items to the back of vec. These added items are value initialized.
vec.resize(10);         // erases 90 elements from the back of vec
```

Exercise 9.30:

What, if any, restrictions does using the version of resize that takes a single argument place on the element type?

If the container holds elements of a class type and resize adds elements we **must supply an initializer** or the element type must have a **default constructor**.

Exercise 9.31 use list | use forward_list

Exercise 9.32

Exercise 9.33

Exercise 9.34

Exercise 9.35:

Explain the difference between a vector's capacity and its size.

The **size** of a container is the number of **elements** it already holds;

The **capacity** is how many elements it can hold before more **space** must be allocated.

Exercise 9.36:

Can a container have a capacity less than its size?

cannot.

Exercise 9.37:

Why don't list or array have a capacity member?

list elements does not store contiguously. array has the fixed size, thus cannot added elements to it.

Exercise 9.38

Exercise 9.39:

Explain what the following program fragment does:

```
vector<string> svec;
svec.reserve(1024);          // sets capacity to at least 1024
string word;
while (cin >> word)          // input word continually
        svec.push_back(word);
svec.resize(svec.size()+svec.size()/2); // sets capacity to at least 3/2's size. may do nothing.
```

Exercise 9.40:

If the program in the previous exercise reads 256 words, what is its likely capacity after it is resized? What if it reads 512? 1,000? 1,048?

read	size	capacity
256	384	1024
512	768	1024
1000	1500	2000(clang is 2048)

read	size	capacity
1048	1572	2048

Exercise 9.41

Exercise 9.42:

Given that you want to read a character at a time into a string, and you know that you need to read at least 100 characters, how might you improve the performance of your program?

Use member `reserve(120)` to allocate enough space for this string. (@Mooophy)

Exercise 9.43

Exercise 9.44

Exercise 9.45

Exercise 9.46

Exercise 9.47 find_first_of | find_first_not_of

Exercise 9.48:

Given the definitions of name and numbers on page 365, what does numbers.find(name) return?

string::npos

Exercise 9.49

Exercise 9.50

Exercise 9.51

Exercise 9.52

Chapter 10. Generic Algorithms

Exercise 10.1 and 10.2

Exercise 10.3 and 10.4

Exercise 10.5

Exercise 10.6

Exercise 10.7

Exercise 10.8:

We said that algorithms do not change the size of the containers over which they operate. Why doesn't the use of back_inserter invalidate this claim?

Cause the back_inserter is a **insert iterator**, what iterator adaptor that generates an iterator that **uses a container operation** to add elements to a given container.

the algorithms don't change the size, but the iterator can change it by using the container operation.

Exercise 10.9

Exercise 10.10:

Why do you think the algorithms don't change the size of containers?

@Mooophy: The aim of this design is to separate the algorithms and the operation provided by member function.

@pezy: Cause the library algorithms operate on **iterators**, **not containers**. Thus, an algorithm **cannot (directly)** add or remove elements.

Exercise 10.11

Exercise 10.12

Exercise 10.13

Exercise 10.14:

Write a lambda that takes two ints and returns their sum.

```
auto add = [](int lhs, int rhs){return lhs + rhs;};
```

Exercise 10.15:

Write a lambda that captures an int from its enclosing function and takes an int parameter. The lambda should return the sum of the captured int and the int parameter.

```
int i = 42;
auto add = [i](int num){return i + num;};
```

Exercise 10.16

Exercise 10.17

Exercise 10.18 and 10.19

Exercise 10.20 and 10.21

Exercise 10.22

Exercise 10.23:

How many arguments does bind take?

Assuming the function to be bound have n parameters, bind take $n+1$ parameters. The additional one is for the function to be bound itself.

Exercise 10.24

Exercise 10.25

Exercise 10.26:

Explain the differences among the three kinds of insert iterators.

> back_inserter uses push_back.
>
> front_inserter uses push_front.
>
> insert uses insert >This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator.

Exercise 10.27

Exercise 10.28

Exercise 10.29

Exercise 10.30

Exercise 10.31

Exercise 10.32

Exercise 10.33

Exercise 10.34 ~ 10.37

Exercise 10.38:

List the five iterator categories and the operations that each supports.

Input iterators : ==, !=, ++, *, ->

Output iterators : ++, *

Forward iterators : ==, !=, ++, *, ->

Bidirectional iterators : ==, !=, ++, --, *, ->

Random-access iterators : ==, !=, <, <=, >, >=, ++, --, +, +=, -, -=, -(two

iterators), *, ->, iter[n]== * (iter + n)

Exercise 10.39:

What kind of iterator does a list have? What about a vector?

list have the **Bidirectional iterators.** vector have the **Random-access iterators.**

Exercise 10.40:

What kinds of iterators do you think copy requires? What about reverse or unique?

copy : first and second are Input iterators, last is Output iterators.

reverse : Bidirectional iterators.

unique : Forward iterators.

Exercise 10.41:

Based only on the algorithm and argument names, describe the operation that the each of the following library algorithms performs:

```
replace(beg, end, old_val, new_val); // replace the old_elements in the input range as new_elements.
replace_if(beg, end, pred, new_val); // replace the elements in the input range which pred is true as
new_elements.
```

```
replace_copy(beg, end, dest, old_val, new_val); // copy the new_elements which is old_elements in the input range into dest.
replace_copy_if(beg, end, dest, pred, new_val); // copy the new_elements which pred is true in the input range into dest.
```

Exercise 10.42

Chapter 11. Associative Containers

Exercise 11.1:

Describe the differences between a map and a vector.

A map is a collection of key-value pairs. we can get a value **lookup by key** efficiently.

A vector is a collection of objects, and every object has an **associated index**, which gives access to that object.

Exercise 11.2:

Give an example of when each of list, vector, deque, map, and set might be most useful.

- list : a to-do list. always need insert or delete the elements anywhere.

> vector : save some important associated data, always need query elements by index.
>
> deque : message handle. FIFO.
>
> map : dictionary.
>
> set : bad_checks.

Exercise 11.3 and 11.4

Exercise 11.5:

Explain the difference between a map and a set. When might you use one or the other?

> `set` : the element type is the **key type**.
>
> `map` : we should use a key-value pair, such as `{key, value}` to indicate that the items together from one element in the map.

I use `set` when i just need to store the `key`, In other hand, I would like use `map` when i need to store a key-value pair.

Exercise 11.6:

Explain the difference between a set and a list. When might you use one or the other?

set is unique and order, but list is neither. using which one depend on whether the elements are unique and order to store.

Exercise 11.7

Exercise 11.8

Exercise 11.9 and 11.10

Exercise 11.11

Exercise 11.12 and 11.13

Exercise 11.14

Exercise 11.15:

What are the mapped_type, key_type, and value_type of a map from int to vector?

> mapped_type : vector
>
> key_type : int
>
> value_type : std::pair>

Exercise 11.16:

Using a map iterator write an expression that assigns a value to an element.

```
std::map<int, std::string> map;
map[25] = "Alan";
std::map<int, std::string>::iterator it = map.begin();
it->second = "Wang";
```

Exercise 11.17:

Assuming c is a multiset of strings and v is a vector of strings, explain the following calls. Indicate whether each call is legal:

```
copy(v.begin(), v.end(), inserter(c, c.end())); // legal
copy(v.begin(), v.end(), back_inserter(c)); // illegal, no `push_back` in `set`.
copy(c.begin(), c.end(), inserter(v, v.end())); // legal.
copy(c.begin(), c.end(), back_inserter(v)); // legal.
```

Exercise 11.18

Exercise 11.19:

Define a variable that you initialize by calling begin() on the multiset named bookstore from 11.2.2 (p. 425). Write the variable's type without using auto or decltype.

```
using compareType = bool (*)(const Sales_data &lhs, const Sales_data &rhs);
std::multiset<Sales_data, compareType> bookstore(compareIsbn);
std::multiset<Sales_data, compareType>::iterator c_it = bookstore.begin();
```

Exercise 11.20

Exercise 11.21:

Assuming word_count is a map from string to size_t and word is a string, explain the following loop:

```
while (cin >> word)
        ++word_count.insert({word, 0}).first->second;
++ (word_count.insert({word, 0}).first->second)
```

Exercise 11.22:

Given a map>, write the types used as an argument and as the return value for the version of insert that inserts one element.

```
std::pair<std::string, std::vector<int>>         // argument
std::pair<std::map<std::string, std::vector<int>>::iterator, bool> // return
```

Exercise 11.23

Exercise 11.24 ~ 11.26

Exercise 11.27 ~ 11.30

Exercise 11.31

Exercise 11.32

Exercise 11.33

Exercise 11.34:

What would happen if we used the subscript operator instead of find in the transform function?

In gcc 4.8.3, will report error:

```
error: passing ‘const std::map<std::basic_string<char>, std::basic_string<char> >’ as ‘this’ argument of
‘std::map<_Key, _Tp, _Compare, _Alloc>::mapped_type& std::map<_Key, _Tp, _Compare,
_Alloc>::operator[](const key_type&) [with _Key = std::basic_string<char>; _Tp = std::basic_string<char>;
_Compare = std::less<std::basic_string<char> >; _Alloc = std::allocator<std::pair<const
std::basic_string<char>, std::basic_string<char> > >; std::map<_Key, _Tp, _Compare, _Alloc>::mapped_type
= std::basic_string<char>; std::map<_Key, _Tp, _Compare, _Alloc>::key_type = std::basic_string<char>]’
discards qualifiers [-fpermissive]
         auto key = m[s];
                     ^
```

Because std::map's operator is not declared as **const**,but m is declared as a reference to std::map with **const**.If insert new pair,it will cause error.

Exercise 11.35:

In buildMap, what effect, if any, would there be from rewriting `trans_map[key] = value.substr(1);` as `trans_map.insert({key, value.substr(1)})`?

> use subscript operator: if a word does appear multiple times, our loops will put the **last** corresponding phrase into trans_map

use insert: if a word does appear multiple times, our loops will put

the **first** corresponding phrase into trans_map

Exercise 11.36:

Our program does no checking on the validity of either input file. In particular, it assumes that the rules in the transformation file are all sensible. What would happen if a line in that file has a key, one space, and then the end of the line? Predict the behavior and then check it against your version of the program.

we added a file that name "word_transformation_bad.txt" to folder data. the file only has a key, one space.

the program of 11.33 don't influenced by that.

Exercise 11.37:

What are the advantages of an unordered container as compared to the ordered version of that container? What are the advantages of the ordered version?

the advantages of an unordered container:

- useful when we have a key type for which there is no obvious ordering relationship among the elements
- useful for applications in which the cost of maintaining the elements in order is prohibitive

the advantages of the ordered version:

- Iterators for the ordered containers access elements in order by key
- we can directly define an ordered container that uses a our own class types for its key type.

Exercise 11.38

Chapter 12. Dynamic Memory

Exercise 12.1:

How many elements do b1 and b2 have at the end of this code?

```
StrBlob b1;
{
    StrBlob b2 = {"a", "an", "the"};
    b1 = b2;
    b2.push_back("about");
}
```

b2 is destroyed, but the elements in b2 must not be destroyed.

so b1 and b2 both have 4 elements.

Exercise 12.2

StrBlob | TEST

Exercise 12.3:

Does this class need const versions of push_back and pop_back? If so, add them. If not, why aren't they needed?

You can certainly do this if you want to, but there doesn't seem to be any logical reason. The compiler doesn't complain because this doesn't modify data (which is a pointer) but rather the thing data points to, which is perfectly legal to do with a const pointer. by David Schwartz.

Discussion over this exercise on Stack Overflow

Discussion over this exercise more on douban(chinese)

Exercise 12.4:

In our check function we didn't check whether i was greater than zero. Why is it okay to omit that check?

Because the type of i is std::vector<std::string>::size_type which is an unsigned.When any argument less than 0 is passed in, it will convert to a number greater than 0. In short std::vector<std::string>::size_type will ensure it is a positive number or 0.

Exercise 12.5:

We did not make the constructor that takes an initializer_list explicit (7.5.4, p. 296). Discuss the pros and cons of this design choice.

@Mooophy:

keyword explicit prevents automatic conversion from an initializer_list to StrBlob. This design choice would easy to use but hard to debug.

@pezy:

Pros

> The compiler will not use this constructor **in an automatic conversion**.
>
> We can realize clearly which class we have used.

Cons

> We always uses the constructor to construct **a temporary StrBlob object**.
>
> cannot use the copy form of initialization with an explicit constructor. not easy to use.

Exercise 12.6

Exercise 12.7

Exercise 12.8:

Explain what if anything is wrong with the following function.

```
bool b() {
	int* p = new int;
	// ...
	return p;
}
```

The p will convert to a bool ,which means that the dynamic memory allocated has no chance to be freed. As a result, memory leakage will occur.

Exercise 12.9:

Explain what happens in the following code:

```
int *q = new int(42), *r = new int(100);
r = q;
auto q2 = make_shared<int>(42), r2 = make_shared<int>(100);
r2 = q2;
```

- to q and r:

Memory leakage happens. Because after `r = q` was executed, no pointer points to the int `r` had pointed to. It implies that no chance to free the memory for it.

- to q2 and r2:

Exercise 12.10

Exercise 12.11

Exercise 12.12

Exercise 12.13

Exercise 12.14

Exercise 12.15

Exercise 12.16

Exercise 12.17 and 12.18

Exercise 12.19 Header|Implementation

Exercise 12.20

Exercise 12.21:

We could have written StrBlobPtr’s deref member as follows:

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

Which version do you think is better and why?

the origin version is better. cause it's more **readability** and **easier to debug**.

Exercise 12.22 Header|Implementation

Exercise 12.23

Exercise 12.24

Exercise 12.25:

Given the following new expression, how would you delete pa?

```
int *pa = new int[10];
delete [] pa;
```

Exercise 12.26

Exercise 12.27 Header|Implementation|Test

Exercise 12.28

Exercise 12.29:

We could have written the loop to manage the interaction with the user as a do while (5.4.4, p. 189) loop. Rewrite the loop to use a do while. Explain which version you prefer and why.

```
do {
	std::cout << "enter word to look for, or q to quit: ";
	string s;
	if (!(std::cin >> s) || s == "q") break;
	print(std::cout, tq.query(s)) << std::endl;
} while ( true );
```

I prefer the `do while`, cause the process according with our logic.

Exercise 12.30 Header|Implementation|Test

Exercise 12.31:

What difference(s) would it make if we used a vector instead of a set to hold the line numbers? Which approach is better? Why?

The `vector` can not ensure no duplicates. Hence, in terms of this programme `set` is a better option.

Exercise 12.32 Header|Implementation

Exercise 12.33 Header|Implementation

Chapter 13. Copy Control

Exercise 13.1:

What is a copy constructor? When is it used?

A copy constructor is a constructor which first parameter is a **reference** to the class type and any additional parameters have **default values**.

When copy initialization happens and that copy initialization requires either the copy constructor or the move constructor.

- Define variables using an =
- Pass an object as an argument to a parameter of nonreference type
- Return an object from a function that has a nonreference return type
- Brace initialize the elements in an array or the members of an aggregate class
- Some class types also use copy initialization for the objects they allocate.

Exercise 13.2:

Explain why the following declaration is illegal:

```
Sales_data::Sales_data(Sales_data rhs);
```

If declaration like that, the call would never succeed to call the copy constructor, `Sales_data rhs` is an argument to a parameter, thus, we'd need to use the copy constructor to copy the argument, but to copy the argument, we'd need to call the copy constructor, and so on indefinitely.

Exercise 13.3:

What happens when we copy a `StrBlob`? What about `StrBlobPtrs`?

```
// added a public member function to StrBlob and StrBlobPrts
long count() {
	return data.use_count(); // and wptr.use_count();
}

// test codes in main()
```

```
StrBlob str({"hello", "world"});
std::cout << "before: " << str.count() << std::endl; // 1
StrBlob str_cp(str);
std::cout << "after: " << str.count() << std::endl;    // 2

ConstStrBlobPtr p(str);
std::cout << "before: " << p.count() << std::endl; // 2
ConstStrBlobPtr p_cp(p);
std::cout << "after: " << p.count() << std::endl; // 2
```

when we copy a `StrBlob`, the `shared_ptr` member's use_count add one.

when we copy a `StrBlobPrts`, the `weak_ptr` member's use_count isn't changed.(cause the count belongs to`shared_ptr`)

Exercise 13.4:

Assuming Point is a class type with a public copy constructor, identify each use of the copy constructor in this program fragment:

```
Point global;
Point foo_bar(Point arg) // 1
{
        Point local = arg, *heap = new Point(global); // 2, 3
        *heap = local;
        Point pa[ 4 ] = { local, *heap }; // 4, 5
        return *heap; // 6
}
```

Exercise 13.5

Exercise 13.6:

What is a copy-assignment operator? When is this operator used? What does the synthesized copy-assignment operator do? When is it synthesized?

The copy-assignment operator is function named `operator=`.

This operator is used when assignment occurred.

The synthesized copy-assignment operator assigns each nonstatic member of the right-hand object to corresponding member of the left-hand object using the copy-assignment operator for the type of that member.

It is synthesized when the class does not define its own.

Exercise 13.7:

What happens when we assign one StrBlob to another? What about StrBlobPtrs?

In both cases, shallow copy will happen. All pointers point to the same address. The `use_count` changed the same as 13.3.

Exercise 13.8

Exercise 13.9:

What is a destructor? What does the synthesized destructor do? When is a destructor synthesized?

The destructor is a member function with the name of the class prefixed by a tilde(~).

As with the copy constructor and the copy-assignment operator, for some classes, the synthesized destructor is defined to disallow objects of the type from being destoryed. Otherwise, the synthesized destructor has an empty function body.

The compiler defines a synthesized destructor for any class that does not define its own destructor.

Exercise 13.10:

What happens when a StrBlob object is destroyed? What about a StrBlobPtr?

When a `StrBlob` object destroyed, the `use_count` of the dynamic object will decrement. It will be freed if no `shared_ptr`to that dynamic object.

When a `StrBlobPter` object is destroyed the object dynamically allocated will not be freed.

Exercise 13.11

Exercise 13.12:

How many destructor calls occur in the following code fragment?

```
bool fcn(const Sales_data *trans, Sales_data accum)
```

```
{
        Sales_data item1(*trans), item2(accum);
        return item1.isbn() != item2.isbn();
}
```

3 times. There are accum, item1 and item2.

Exercise 13.13

Exercise 13.14:

Assume that numbered is a class with a default constructor that generates a unique serial number for each object, which is stored in a data member named mysn. Assuming numbered uses the synthesized copy- control members and given the following function:

```
void f (numbered s) { cout << s.mysn << endl; }
```

what output does the following code produce?

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

Three identical numbers.

Exercise 13.15:

Assume numbered has a copy constructor that generates a new serial number. Does that change the output of the calls in the previous exercise? If so, why? What output gets generated?

Yes, the output will change. cause we don't use the synthesized copy-control members rather than own defined.The output will be three different numbers.

Exercise 13.16:

What if the parameter in f were const numbered&? Does that change the output? If so, why? What output gets generated?

Yes, the output will change. cause the function f haven't any copy operators. Thus, the output are the same when pass the each object to f.

Exercise 13.17

Write versions of numbered and f corresponding to the previous three exercises and check whether you correctly predicted the output.

For 13.14 | For 13.15 | For 13.16

Exercise 13.18 .h | .cpp

Exercise 13.19

Exercise 13.20:

Explain what happens when we copy, assign, or destroy objects of our TextQuery and QueryResult classes from § 12.3 (p. 484).

The member (smart pointer and container) will be copied.

Exercise 13.21:

Do you think the TextQuery and QueryResult classes need to define their own versions of the copy-control members? If so, why? If not, why not? Implement whichever copy-control operations you think these classes require.

(@Mooophy) No copy-control members needed.

Because, all these classes are using smart pointers to manage dynamic memory which can be freed automatically by calling synthesized destructors. The objects of these classes should share the same dynamic memory.Hence no user-defined version needed as well.

```
TextQuery(const TextQuery&) = delete;
TextQuery& operator=(const TextQuery) = delete;

QueryResult(const QueryResult&) = delete;
QueryResult& operator=(const QueryResult) = delete;
```

Exercise 13.22

Exercise 13.23:

Compare the copy-control members that you wrote for the solutions to the previous section’s exercises to the code presented here. Be sure you understand the differences, if any, between your code and ours.

Check 13.22.

Exercise 13.24:

What would happen if the version of HasPtr in this section didn’t define a destructor? What if HasPtr didn’t define the copy constructor?

If HasPtr didn't define a destructor, memory leak will happened. If HasPtr didn't define the copy constructor, when assignment happened, just points copied, the string witch ps points haven't been copied.

Exercise 13.25:

Assume we want to define a version of StrBlob that acts like a value. Also assume that we want to continue to use a shared_ptr so that our StrBlobPtr class can still use a weak_ptr to the vector. Your revised class will need a copy constructor and copy-assignment operator but will not need a destructor. Explain what the copy constructor and copy-assignment operators must do. Explain why the class does not need a destructor.

Copy constructor and copy-assignment operator should dynamically allocate memory for its own , rather than share the object with the right hand operand.

StrBlob is using smart pointers which can be managed with synthesized destructor, If an object of StrBlob is out of scope, the destructor for std::shared_ptr will be called automatically to free the memory dynamically allocated when the use_count goes to 0.

Exercise 13.26 hpp | cpp

Exercise 13.27

Exercise 13.28 hpp | cpp

Exercise 13.29:

Explain why the calls to swap inside swap(HasPtr&, HasPtr&) do not cause a recursion loop.

swap(lhs.ps, rhs.ps); feed the version : swap(std::string*, std::string*) and swap(lhs.i, rhs.i); feed the version : swap(int, int). Both them can't call swap(HasPtr&, HasPtr&). Thus, the calls don't cause a recursion loop.

Exercise 13.30

Exercise 13.31

Exercise 13.32:

Would the pointerlike version of HasPtr benefit from defining a swap function? If so, what is the benefit? If not, why not?

@Mooophy:

Essentially, the specific avoiding memory allocation is the reason why it improve performance. As for the pointerlike version, no dynamic memory allocation anyway. Thus, a specific version for it will not improve the performance.

Exercise 13.33:

Why is the parameter to the save and remove members of Message a Folder&? Why didn't we define that parameter as Folder? Or const Folder&?

Because these operations must also update the given Folder. Updating a Folder is a job that the Folder class controls through its addMsg and remMsg members, which will add or remove a pointer to a given Message, respectively.

Exercise 13.34 hpp | cpp

Exercise 13.35:

What would happen if Message used the synthesized versions of the copy-control members?

some existing Folders will out of sync with the Message after assignment.

Exercise 13.36 hpp | cpp

Exercise 13.37 hpp | cpp

Exercise 13.38:

We did not use copy and swap to define the Message assignment operator. Why do you suppose this is so?

@Mooophy The copy and swap is an elegant way when working with dynamicly allocated memory. In the Message class ,noing is allocated dynamically. Thus using this idiom makes no sense and will make it more complicated to implement due to the pointers that point back.

@pezy In this case, `swap` function is special. It will be clear two `Message`'s folders , then swap members, and added themselves to each folders.
But, `Message` assignment operator just clear itself, and copy the members, and added itself to each folders. The `rhs` don't need to clear and add to folders. So, if using copy and swap to define, it will be very inefficiency.

Exercise 13.39 hpp | cpp

Exercise 13.40 hpp | cpp

Exercise 13.41:

Why did we use postfix increment in the call to construct inside push_back?

What would happen if it used the prefix increment?

```
|a|b|c|d|f|.............|
^           ^              ^
elements    first_free     cap

// if use alloc.construct(first_free++, "g");
|a|b|c|d|f|g|............|
^             ^            ^
elements      first_free   cap

// if use alloc.construct(++first_free, "g");
|a|b|c|d|f|.|g|...........|
^           ^ ^             ^
elements    | first_free    cap
            |
    "unconstructed"
```

Exercise 13.42:

Test your StrVec class by using it in place of the vector in your TextQuery and

QueryResult classes (12.3, p. 484).

StrVec : hpp | cpp

TextQuery and QueryResult : hpp | cpp

Text : ex13_42.cpp

Exercise 13.43:

Rewrite the free member to use for_each and a lambda (10.3.2, p. 388) in place of the for loop to destroy the elements. Which implementation do you prefer, and why?

Rewrite

```
for_each(elements, first_free, [this](std::string &rhs){ alloc.destroy(&rhs); });
```

@Mooophy: The new version is better. Compared to the old one, it doesn't need to worry about the order and decrement.So more straightforward and handy. The only thing to do for using this approach is to add "&" to build the pointers to string pointers.

Exercise 13.44:

Write a class named String that is a simplified version of the library string class. Your class should have at least a default constructor and a constructor that takes a pointer to a C-style string. Use an allocator to allocate memory that your String class uses.

hpp | cpp | Test

more information to see A trivial String class that designed for write-on-paper in an interview

Exercise 13.45:

Distinguish between an rvalue reference and an lvalue reference.

Definition：

lvalue reference: reference that can bind to **an lvalue**. (Regular reference)

rvalue reference: reference **to an object that is about to be destroyed**.

We can bind an rvalue reference to expression that require conversion, to literals, or to expressions that return an rvalue, but we cannot directly bind an rvalue reference to an lvalue.

```
int i = 42;
int &r = i; // lvalue reference
int &&rr = i; // rvalue reference (Error: i is a lvalue)
int &r2 = i*42; // lvalue reference (Error: i*42 is a rvalue)
const int &r3 = i*42; // reference to const (bind to a rvalue)
int &&rr2 = i*42; // rvalue reference
```

lvalue : functions that return lvalue references, assignment, subscript, dereference, and prefix increment/decrement operator.

rvalue / const reference : functions that return a nonreferences, arithmetic, relational bitwise, postfix increment/decrement operators.

Exercise 13.46:

Which kind of reference can be bound to the following initializers?

```
int f();
vector<int> vi(100);
int&& r1 = f();
int& r2 = vi[0];
int& r3 = r1;
int&& r4 = vi[0] * f();
```

Exercise 13.47 hpp | cpp

Exercise 13.48

Exercise 13.49:

Add a move constructor and move-assignment operator to your StrVec, String, and Message classes.

- StrVec: hpp | cpp
- String: hpp | cpp
- Message:hpp | cpp

Exercise 13.50:

Put print statements in the move operations in your String class and rerun the program from exercise 13.48 in 13.6.1 (p. 534) that used a vector to see when the copies are avoided.

```
String baz()
{
    String ret("world");
    return ret; // first avoided
}

String s5 = baz(); // second avoided
```

Exercise 13.51:

Although unique_ptrs cannot be copied, in 12.1.5 (p. 471) we wrote a clone function that returned a unique_ptr by value. Explain why that function is legal and how it works.

In the second assignment, we assign from the result of a call to getVec. That expression is an rvalue. In this case, both assignment operators are viable—we can bind the result of getVec to either operator’s parameter. Calling the copy-assignment operator requires a conversion to const, whereas StrVec&& is an exact match. Hence, the second assignment uses the move-assignment operator.

```
unique_ptr<int> clone(int p) {
    // ok: explicitly create a unique_ptr<int> from int*
    return unique_ptr<int>(new int(p));
}
```

the result of a call to clone is an **rvalue**, so it uses the move-assignment operator rather than copy-assignment operator. Thus, it is legal and can pretty work.

Exercise 13.52:

Explain in detail what happens in the assignments of the HasPtr objects on page 541. In particular, describe step by step what happens to values of hp, hp2, and of the rhs parameter in the HasPtr assignment operator.

rhs parameter is nonreference, which means the parameter is **copy initialized**. Depending on the type of the argument, copy initialization uses either the *copy constructor* or the *move constructor*.

lvalues are copied and rvalues are moved.

Thus, in `hp = hp2;`, `hp2` is an lvalue, copy constructor used to copy `hp2`. In `hp = std::move(hp2);`, move constructor moves `hp2`.

Exercise 13.53:

As a matter of low-level efficiency, the `HasPtr` assignment operator is not ideal. Explain why. Implement a copy-assignment and move-assignment operator for `HasPtr` and compare the operations executed in your new move-assignment operator versus the copy-and-swap version.

nothing to say, just see the versus codes:

hpp | cpp | Test

see more information at this question && answer.

Exercise 13.54:

What would happen if we defined a `HasPtr` move-assignment operator but did not change the copy-and-swap operator? Write code to test your answer.

```
error: ambiguous overload for 'operator=' (operand types are 'HasPtr' and
'std::remove_reference<HasPtr&>::type {aka HasPtr}')
hp1 = std::move(*pH);
^
```

Exercise 13.55:

Add an rvalue reference version of push_back to your StrBlob.

```
void push_back(string &&s) { data->push_back(std::move(s)); }
```

Exercise 13.56:

What would happen if we defined sorted as:

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

recursion and stack overflow.

Exercise 13.57:

What if we defined sorted as:

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

ok, it will call the move version.

Exercise 13.58:

Write versions of class Foo with print statements in their sorted functions to test your answers to the previous two exercises.

Exercise 13.58

Chapter 14. Overloaded Operations and Conversions

Exercise 14.1:

In what ways does an overloaded operator differ from a built-in operator? In what ways are overloaded operators the same as the built-in operators?

Differ 1. We can call an overloaded operator function directly. 2. An overloaded operator function must either be a member of a class or have at least one parameter of class type. 3. A few operators guarantee the order in which operands are evaluated. These overloaded versions of these operators do not preserve order of evaluation and/or short-circuit evaluation, it is usually a bad idea to overload them.

In particular, the operand-evaluation guarantees of the logical `AND`, logical `OR`, and comma operators are not preserved, Moreover, overloaded versions of `&&` or `||` operators do not preserve short-circuit evaluation properties of the built-in operators. Both operands are always evaluated.

Same

> An overloaded operator has the same precedence and associativity as the corresponding built-in operator.

Exercise 14.2:

Write declarations for the overloaded input, output, addition, and compound-assignment operators for Sales_data.

hpp | cpp

Exercise 14.3:

Both string and vector define an overloaded == that can be used to compare objects of those types. Assuming svec1 and svec2 are vectors that hold strings, identify which version of == is applied in each of the following expressions:

> (a) "cobble" == "stone"
>
> (b) svec1[0] == svec2[0]
>
> (c) svec1 == svec2
>
> (d) "svec1[0] == "stone"

(a) neither. (b) string (c) vector (d) string

Reference

> Why does the following not invoke the overloaded operator== (const String &, const String &)? “cobble” == “stone”

Exercise 14.4:

Explain how to decide whether the following should be class members:

(a) %

(b) %=

(c) ++

(d) ->

(e) <<

(f) &&

(g) ==

(h) ()

(a) symmetric operator. Hence, non-member

(b) changing state of objects. Hence, member

(c) changing state of objects. Hence, member

(d) = -> must be member

(e) non-member

(f) symetric , non-member

(g) symetric , non-member

(h) = -> must be member

Exercise 14.5:

In exercise 7.40 from 7.5.1 (p. 291) you wrote a sketch of one of the following classes. Decide what, if any, overloaded operators your class should provide.

Such as `Book`

hpp | cpp | test

Exercise 14.6:

Define an output operator for your `Sales_data` class.

see Exercise 14.2.

Exercise 14.7:

Define an output operator for you `String` class you wrote for the exercises in 13.5 (p. 531).

hpp | cpp | Test

Exercise 14.8:

Define an output operator for the class you chose in exercise 7.40 from 7.5.1 (p. 291).

see Exercise 14.5

Exercise 14.9:

Define an input operator for your Sales_data class.

see Exercise 14.2.

Exercise 14.10:

Describe the behaviour of the Sales_data input operator if given the following input:

> (a) 0-201-99999-9 10 24.95
>
> (b) 10 24.95 0-210-99999-9

> (a) correct format.
>
> (b) ilegal input. But 0-210-99999-9 will be converted to a float stored in this object. As a result, the data inside will be a wrong one. Output: 10 24 22.8 0.95

check Test

Exercise 14.11:

What, if anything, is wrong with the following Sales_data input operator? What would happen if we gave this operator the data in the previous exercise?

```
istream& operator>>(istream& in, Sales_data& s)
{
        double price;
        in >> s.bookNo >> s.units_sold >> price;
        s.revenue = s.units_sold * price;
        return in;
}
```

no input check. nothing happend.

Exercise 14.12:

Define an input operator for the class you used in exercise 7.40 from 7.5.1 (p. 291). Be sure the operator handles input errors.

see Exercise 14.5

Exercise 14.13:

Which other arithmetic operators (Table 4.1 (p. 139)), if any, do you think Sales_data ought to support? Define any you think the class should include.

no others.

Exercise 14.14:

Why do you think it is more efficient to define `operator+` to call `operator+=` rather than the other way around?

Discussing on SO.

Exercise 14.15:

Should the class you chose for exercise 7.40 from 7.5.1 (p. 291) define any of the arithmetic operators? If so, implement them. If not, explain why not.

hpp | cpp | Test

Exercise 14.16:

Define equality and inequality operators for your StrBlob (12.1.1, p. 456), StrBlobPtr (12.1.6, p. 474), StrVec(13.5, p.526), and String (13.5, p. 531) classes.

- StrBlob & StrBlobPtr: hpp | cpp | Test
- StrVec: hpp | cpp | Test
- String: hpp | cpp | Test

Exercise 14.17:

Should the class you chose for exercise 7.40 from 7.5.1(p. 291) define the equality operators? If so, implement them. If not, explain why not.

yes.see Exercise 14.15

Exercise 14.18:

Define relational operators for your StrBlob, StrBlobPtr, StrVec, and String classes.

StrBlob & StrBlobPtr: hpp | cpp | Test

StrVec: hpp | cpp | Test

String: hpp | cpp | Test

Exercise 14.19:

Should the class you chose for exercise 7.40 from 7.5.1 (p. 291) define the relational operators? If so, implement them. If not, explain why not.

yes.see Exercise 14.15

Exercise 14.20:

Define the addition and compound-assignment operators for your Sales_data class.

see Exercise 14.2.

Exercise 14.21:

Write the Sales_data operators so that + does the actual addition and += calls +. Discuss the disadvantages of this approach compared to the way these operators were defined in 14.3 (p. 560) and 14.4 (p.564).

```
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
	Sales_data old_data = *this;
```

```
    *this = old_data + rhs;
    return *this;
}

Sales_data operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum;
    sum.units_sold = lhs.units_sold + rhs.units_sold;
    sum.revenue = lhs.revenue + rhs.revenue;
    return sum;
}
```

Disadvantages: Both `+` and `+=`, uses an temporary object of `Sales_data`. But it is no need for that.

Exercise 14.22:

Define a version of the assignment operator that can assign a `string` representing an ISBN to a `Sales_data`.

hpp | cpp | Test

Exercise 14.23:

Define an initializer_list assignment operator for your version of the `StrVec` class.

hpp | cpp | Test

Exercise 14.24:

Decide whether the class you used in exercise 7.40 from 7.5.1 (p. 291) needs a copy- and move-assignment operator. If so, define those operators.

hpp | cpp | Test

Exercise 14.25:

Implement any other assignment operators your class should define. Explain which types should be used as operands and why.

see Exercise 14.24

Exercise 14.26:

Define subscript operators for your StrVec, String, StrBlob, and StrBlobPtr classes.

- StrBlob & StrBlobPtr: hpp | cpp | Test
- StrVec: hpp | cpp | Test
- String: hpp | cpp | Test

Exercise 14.27:

Add increment and decrement operators to your StrBlobPtr class.

hpp | cpp | Test

Exercise 14.28:

Define addition and subtraction for StrBlobPtr so that these operators implement pointer arithmetic (3.5.3, p. 119).

see Exercise 14.27

Exercise 14.29:

We did not define a `const` version of the increment and decrement operators. Why not?

Because `++` and `--` change the state of the object. Hence ,it's meaningless to do so.

Exercise 14.30:

Add dereference and arrow operators to your `StrBlobPtr` class and to the `ConstStrBlobPtr` class that you defined in exercise 12.22 from 12.1.6 (p. 476). Note that the operators in `constStrBlobPtr` must return `const` references because the `data` member in `constStrBlobPtr` points to a `const vector`.

hpp | cpp | Test

Exercise 14.31:

Our StrBlobPtr class does not define the copy constructor, assignment operator, or a destructor. Why is that okay?

Applying the Rule of 3/5: There is no dynamic allocation to deal with, so the synthesized destructor is enough. Moreover, no unique is needed. Hence, the synthesized ones can handle all the corresponding operations.

Exercise 14.32:

Define a class that holds a pointer to a StrBlobPtr. Define the overloaded arrow operator for that class.

hpp | cpp

Exercise 14.33:

How many operands may an overloaded function-call operator take?

An overloaded operator function has the same number of parameters as the operator has operands. Hence the maximum value should be around 256. (question on SO)

Exercise 14.34:

Define a function-object class to perform an if-then-else operation: The call operator for this class should take three parameters. It should test its first parameter and if that test succeeds, it should return its second parameter; otherwise, it should return its third parameter.

```
struct Test {
    int operator()(bool b, int iA, int iB) {
        return b ? iA : iB;
    }
};
```

Exercise 14.35:

Write a class like PrintString that reads a line of input from an istream and returns a string representing what was read. If the read fails, return the empty string.

Test

Exercise 14.36:

Use the class from the previous exercise to read the standard input, storing each line as an element in a vector.

Test

Exercise 14.37:

Write a class that tests whether two values are equal. Use that object and the library algorithms to write a program to replace all instances of a given value in a sequence.

Test

Exercise 14.38:

Write a class that tests whether the length of a given string matches a given bound. Use that object to write a program to report how many words in an input file are of sizes 1 through 10 inclusive.

BoundTest

Exercise 14.39:

Revise the previous program to report the count of words that are sizes 1 through 9 and 10 or more.

see Exercise 14.38

Exercise 14.40:

Rewrite the `biggies` function from 10.3.2 (p. 391) to use function-object classes in place of lambdas.

Test

Exercise 14.41:

Why do you suppose the new standard added lambdas? Explain when you would use a lambda and when you would write a class instead.

IMO, lambda is quite handy to use. Lambda can be used when the functor is not used frequently nor complicated, whereas functor is supposed to call more times than lambda or quite complicated to implement as a lambda.

Exercise 14.42:

Using library function objects and adaptors, define an expression to

(a) Count the number of values that are greater than 1024

(b) Find the first string that is not equal to pooh

(c) Multiply all values by 2

```
std::count_if(ivec.cbegin(), ivec.cend(), std::bind(std::greater<int>(), _1, 1024));
std::find_if(svec.cbegin(), svec.cend(), std::bind(std::not_equal_to<std::string>(), _1, "pooh"));
std::transform(ivec.begin(), ivec.end(), ivec.begin(), std::bind(std::multiplies<int>(), _1, 2));
```

Test

Exercise 14.43:

Using library function objects, determine whether a given int value is divisible by any element in a container of ints.

ex14_43.cpp

Exercise 14.44:

Write your own version of a simple desk calculator that can handle binary operations.

ex14_44.cpp

Exercise 14.45:

Write conversion operators to convert a Sales_data to string and to double. What values do you think these operators should return?

hpp | cpp | Test

Exercise 14.46:

Explain whether defining these Sales_data conversion operators is a good idea and whether they should be explicit.

It's a bad idea to do so, because these conversion is misleading.explicit should be added to prevent implicit conversion.

Exercise 14.47:

Explain the difference between these two conversion operators:

```
struct Integral {
        operator const int();      // meaningless, it will be ignored by compiler.
        operator int() const;      // promising that this operator will not change the state of the obj
};
```

Exercise 14.48:

Determine whether the class you used in exercise 7.40 from 7.5.1 (p. 291) should have a conversion to bool. If so, explain why, and explain whether the operator should be explicit. If not, explain why not.

A conversion to bool can be useful for the class Date. But it must be an explicit one to prevent any automatic conversion.

Exercise 14.49:

Regardless of whether it is a good idea to do so, define a conversion to bool for the class from the previous exercise.

hpp | cpp | Test

Exercise 14.50:

Show the possible class-type conversion sequences for the initializations of ex1 and ex2. Explain whether the initializations are legal or not.

```
struct LongDouble {
        LongDouble(double = 0.0);
        operator double();
        operator float();
};
LongDouble ldObj;
int ex1 = ldObj;        // error ambiguous: double or float?
float ex2 = ldObj;    // legal
```

Exercise 14.51:

Show the conversion sequences (if any) needed to call each version of calc and explain why the best viable function is selected.

```
void calc(int);
void calc(LongDouble);
double dval;
```

```
calc(dval); // which calc?
```

best viable function: `void calc(int)`. cause class-type conversion is the lowest ranked.

review the order:

1. exact match
2. const conversion
3. promotion
4. arithmetic or pointer conversion
5. class-type conversion

Exercise 14.52:

Which `operator+`, if any, is selected for each of the addition expressions? List the candidate functions, the viable functions, and the type conversions on the arguments for each viable function:

```
struct LongDouble {
	// member operator+ for illustration purposes; + is usually a nonmember LongDouble operator+(const SmallInt&); // 1
	// other members as in 14.9.2 (p. 587)
};
LongDouble operator+(LongDouble&, double); // 2
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

`ld = si + ld;` is ambiguous. `ld = ld + si` can use both 1 and 2, but 1 is more exactly. (in the 2, SmallInt need to convert to `double`)

Exercise 14.53:

Given the definition of SmallInt on page 588, determine whether the following addition expression is legal. If so, what addition operator is used? If not, how might you change the code to make it legal?

```
SmallInt s1;
double d = s1 + 3.14;
```

ambiguous.

Fixed:

```
SmallInt s1;
double d = s1 + SmallInt(3.14);
```

Chapter 15

ex15.1.2.3	rename and clear	10 months
ex15.11	remove system("pause")	
ex15.12.13.14	rename and clear	
ex15.15.16.17	add error information	
ex15.18.19.20	rename and clear	
ex15.21.22	rename and clear	
ex15.26	Update bulk_quote.h	

ex15.27	rename and clear
ex15.28.29	rename and clear
ex15.30	Modify test data
ex15.34.35.36.38	rename and clear
ex15.39.40	rename and clear
ex15.42_b	solution for exercise 15.42 (b)
ex15.42_c	a solution for exercise 15.42 (c)
ex15.5.6	fixed :#154
ex15.7	rename and clear
ex15.8.9.10	rename and clear
ex15.23.cpp	rename and clear
ex15.24.25.cpp	rename and clear
ex15.31.32.33.cpp	rename and clear

Chapter 16

..

ex16.1.2.3	fix exercise 16.3 16.4
ex16.12.13	Update blobptr.h
ex16.14.15	A solution for exercise 16.15
ex16.16	rename and clear
ex16.17.18	Fix spelling error
ex16.19.20	Add an ostream parameter
ex16.21.22	rename and clear
ex16.24	rename and clear
ex16.25.26	rename and clear
ex16.28	rename and clear
ex16.29	rename and clear
ex16.32.33.34.35.36	rename and clear
ex16.37.38.39	rename and clear
ex16.4	fix exercise 16.3 16.4
ex16.40	rename and clear
ex16.41	rename and clear
ex16.42.43.44.45.46	rename and clear

ex16.47	rename and clear
ex16.48	rename and clear
ex16.49.50	rename and clear
ex16.5	Use range for instead of for
ex16.51.52	rename and clear
ex16.53.54.55	rename and clear
ex16.56.57	rename and clear
ex16.58.59	rename and clear
ex16.6	rename and clear
ex16.60.61	rename and clear
ex16.62	rename and clear
ex16.63.64	rename and clear
ex16.65.66.67	rename and clear
ex16.7.8	rename and clear
ex16.9.10.11	rename and clear

Chapter 17

..	
ex17.1.2	rename and clear
ex17.10	rename and clear
ex17.11.12.13	Template functions update and mark.
ex17.14.15.16	exercise 17.19.20
ex17.17.18	exercise 17.19.20
ex17.19.20	exercise 17.19.20
ex17.21	Fix character mistake
ex17.28.29.30	rename and clear
ex17.3	Beautify the code.
ex17.4.5.6.7.8	A solution for exercise 17.6
ex17.9	rename and clear
ex_33.cpp	Create ex_33.cpp

Chapter 18

..

ex18.1.2.3	Update ex18.1.2.3.cpp
ex18.12.13.14	ex18.12.13.14
ex18.15.16.17	ex18.15.16.17
README.md	Add Exercise 18.2

README.md

Exercise 18.1

What is the type of the exception object in the following throws?

(a)range_error r("error"); throw r; **(b)**exception *p = &r; throw *P;

What would happen if the throw in **(b)** were written as throw p?

```
The type of the exception object in (a) is range_error which is used to report range errors in internal
computations.
The type of the exception object in (b) is exception.
If the "throw" in (b) were written as "throw p", there will be a runtime error.
```

Exercise 18.2

Explain what happens if an exception occurs at the indicated point:

```
void exercise(int *b, int *e)
{
    vector<int> v(b,e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
}
```

The space "p" points will not be free. There will be a memory leak.

Exercise 18.3