Input/output with files
- Open a file
- Closing a file
- Text files
- Checking state flags
- get and put stream positioning
  - tellg() and tellp()
  - seekg() and seekp()
- Binary files
- Buffers and Synchronization

Input/output with files

C++ provides the following classes to perform output and input of characters to/from files:

c++提供了以下类来执行文件的字符输出和输入:

- **`ofstream`**: Stream class to write on files
- **`ifstream`**: Stream class to read from files
- **`fstream`**: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream` and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

这些类直接或间接地派生自istream和ostream类。我们已经使用了这些类类型的对象:cin是istream类的对象，cout是ostream类的对象。因此，我们已经使用了与文件流相关的类。事实上，我们可以像使用cin和cout一样使用文件流，唯一的区别是我们必须将这些流与物理文件关联起来。让我们来看一个例子:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

这段代码创建了一个名为example.txt的文件，并以与使用cout相同的方式将句子插入其中，但使用的是文件流myfile。

But let's go step by step:

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a *stream* (i.e., an object of one of these classes; in the previous example, this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

通常对这些类中的一个对象执行的第一个操作是将它与实际文件关联起来。这个过程称为打开文件。在程序中，打开的文件由流(即其中一个类的对象;在前面的示例中，这是myfile)，并且在此流对象上执行的任何输入或输出操作都将应用于与其关联的物理文件。

In order to open a file with a stream object we use its member function `open`:

为了用stream对象打开文件，我们使用它的成员函数open:

```
open (filename, mode);
```

Where `filename` is a string representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

其中filename是一个表示要打开的文件名的字符串，mode是一个可选参数，由以下标志组合而成:

`ios::in`	**Open for input operations.**
`ios::out`	Open for output operations.
`ios::binary`	Open in binary mode.
`ios::ate`	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
`ios::app`	All output operations are performed at the end of the file, appending the content to the current content of the file.
`ios::trunc`	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file `example.bin` in binary mode to add data we could do it by the following call to member function `open`:

所有这些标志都可以使用按位运算符OR(|)进行组合。例如，如果我们想以二进制模式打开example.bin文件来添加数据，可以通过以下调用成员函数open来实现:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each of the `open` member functions of classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

ofstream、ifstream和fstream类的每个open成员函数都有一个默认模式，在没有第二个参数的情况下打开文件时使用:

class	default mode parameter
`ofstream`	ios::out
`ifstream`	ios::in
`fstream`	ios::in | ios::out

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open` member function (the flags are combined).

对于ifstream和ofstream类，ios::in和ios::out会自动分别假定，即使不包含它们的模式被作为第二个参数传递给open成员函数(标志被组合)。

For `fstream`, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

对于fstream，只有在调用函数时没有为模式参数指定任何值时才应用默认值。如果函数是用该参数中的任何值调用的，则默认模式将被覆盖，而不是组合。

File streams opened in *binary mode* perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

以二进制模式打开的文件流独立于任何格式考虑执行输入和输出操作。非二进制文件被称为文本文件，一些翻译可能会由于某些特殊字符(如换行符和回车符)的格式化而发生。

Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the `open` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conduct the same opening operation in our previous example by writing:

由于在文件流上执行的第一个任务通常是打开文件，所以这三个类包括一个构造函数，该构造函数自动调用open成员函数，并具有与该成员完全相同的参数。因此，我们也可以声明前面的myfile对象，并在前面的例子中执行相同的打开操作，方法如下:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

在一条语句中组合对象构造和流打开。打开文件的两种形式都是有效的和等效的。

To check if a file stream was successful opening a file, you can do it by calling to member `is_open`. This member function returns a `bool` value of `true` in the case that indeed the stream object is associated with an open file, or `false` otherwise:

要检查文件流是否成功打开了文件，可以通过调用成员is_open来实现。如果该stream对象确实与打开的文件相关联，则该成员函数返回bool值true，否则返回false:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function `close`. This member function takes flushes the associated buffers and closes the file:

当我们完成对一个文件的输入和输出操作时，我们应该关闭它，以便操作系统得到通知，其资源再次可用。为此，我们调用流的成员函数' close '。该成员函数接受刷新相关缓冲区并关闭文件:

```
myfile.close();
```

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.

一旦调用了这个成员函数，就可以重用stream对象来打开另一个文件，并且该文件可以再次被其他进程打开。

In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function `close`.

Text files

Text file streams are those where the `ios::binary` flag is not included in their opening mode. These files are designed to store text and thus all values that are input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

文本文件流是那些在其打开模式中不包含' ios::binary '标志的文件流。这些文件被设计用来存储文本，因此从它们输入或输出的所有值都可能遭受一些格式化转换，这些转换不一定对应于它们的文本二进制值。

Writing operations on text files are performed in the same way we operated with `cout`:

对文本文件的写入操作与' cout '的操作方式相同:

```
// writing on a text file
```

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

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Reading from a file can also be performed in the same way that we did with `cin`:

从文件中读取数据也可以用与`cin`相同的方式执行:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << '\n';
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using getline. The value returned by getline is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is `true` if the stream is ready for more operations, and `false` if either the end of the file has been reached or if some other error occurred.

最后一个示例读取文本文件并将其内容打印到屏幕上。我们已经创建了一个while循环，使用getline逐行读取文件。返回的值通过getline流对象本身的引用,当评估一个布尔表达式(如这个while循环)是真的如果准备更多的操作流,和假如果文件的末尾已经达到或者其他一些错误发生。

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a `bool` value):

以下成员函数用于检查流的特定状态(它们都返回bool值):

- `bad()`

  Returns `true` if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.

  如果读或写操作失败，则返回true。例如，在这种情况下，我们试图写入一个文件，没有打开写入或如果我们试图写入的设备没有剩余空间。

- `fail()`

  Returns `true` in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.

  在与bad()相同的情况下返回true，但在发生格式错误的情况下也返回true，例如当我们试图读取一个整数时提取了一个字母字符。

- `eof()`

  Returns `true` if a file open for reading has reached the end.

  如果打开读取的文件已经到达末尾，则返回true。

- `good()`

  It is the most generic state flag: it returns `false` in the same cases in which calling any of the previous functions would return `true`. Note that `good` and `bad` are not exact opposites (`good` checks more state flags at once).

  它是最通用的状态标志:在调用前面的任何函数都会返回true的情况下，它会返回false。注意，好和坏并不是完全相反的(好一次检查更多的州旗)。

The member function `clear()` can be used to reset the state flags.

成员函数clear()可用于重置状态标志。

get and put stream positioning

All i/o streams objects keep internally -at least- one internal position:

所有的i/o流对象在内部保持至少一个内部位置:

`ifstream`, like `istream`, keeps an internal *get position* with the location of the element to be read in the next input operation.

与istream一样，Ifstream保留了一个内部的get位置，该位置是下一个输入操作中要读取的元素的位置。

`ofstream`, like `ostream`, keeps an internal *put position* with the location where the next element has to be written.

Ofstream和ostream一样，保持一个内部放置位置，其中包含必须写入下一个元素的位置。

Finally, `fstream`, keeps both, the *get* and the *put position*, like `iostream`.

最后，fstream保留了get和put的位置，就像iostream一样。

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

这些内部流位置指向流中执行下一个读写操作的位置。这些位置可以通过以下成员函数进行观察和修改:

tellg() and tellp()

These two member functions with no parameters return a value of the member type `streampos`, which is a type representing the current *get position* (in the case of `tellg`) or the *put position* (in the case of `tellp`).

这两个不带参数的成员函数返回一个成员类型streampos的值，该类型表示当前的get位置(在tellg中)或put位置(在tellp中)。

seekg() and seekp()

These functions allow to change the location of the *get* and *put positions*. Both functions are overloaded with two different prototypes. The first form is:

这些函数允许更改获取和放置位置的位置。两个函数都被两个不同的原型重载。第一种形式是:

```
seekg ( position );seekp ( position );
```

Using this prototype, the stream pointer is changed to the absolute position `position` (counting from the beginning of the file). The type for this parameter is `streampos`, which is the same type as returned by functions `tellg` and `tellp`.

使用这个原型，流指针被更改为绝对位置位置(从文件开始计数)。该参数的类型是streampos，与函数tellg和tellp返回的类型相同。

The other form for these functions is:

```
seekg ( offset, direction );seekp ( offset, direction );
```

Using this prototype, the *get* or *put position* is set to an offset value relative to some specific point determined by the parameter `direction`. `offset` is of type `streamoff`. And `direction` is of type `seekdir`, which is an *enumerated type*that determines the point from where offset is counted from, and that can take any of the following values:

使用这个原型，get或put位置被设置为相对于由参数方向确定的特定点的偏移值。Offset是流线类型。而direction的类型是seekdir，这是一种枚举类型，确定从何处计算偏移量，它可以取以下任意值:

`ios::beg`	**offset counted from the beginning of the stream**
`ios::cur`	offset counted from the current position
`ios::end`	offset counted from the end of the stream

The following example uses the member functions we have just seen to obtain the size of a file:

下面的例子使用我们刚刚看到的成员函数来获取文件的大小:

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  streampos begin,end;
  ifstream myfile ("example.bin", ios::binary);
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

Notice the type we have used for variables `begin` and `end`:

注意我们对变量' begin '和' end '使用的类型:

```
streampos size;
```

`streampos` is a specific type used for buffer and file positioning and is the type returned by `file.tellg()`. Values of this type can safely be subtracted from other values of the same type, and can also be converted to an integer type large enough to contain the size of the file.

Streampos是用于缓冲区和文件定位的特定类型，是file.tellg()返回的类型。这种类型的值可以安全地从相同类型的其他值中减去，也可以转换为大到足以包含文件大小的整数类型。

These stream positioning functions use two particular types: `streampos` and `streamoff`. These types are also defined as member types of the stream class:

这些流定位函数使用两种特殊类型:streampos和streamoff。这些类型也被定义为流类的成员类型:

Type	Member type	Description
`streampos`	`ios::pos_type`	Defined as `fpos<mbstate_t>`. It can be converted to/from `streamoff` and can be added or subtracted values of these types.
`streamoff`	`ios::off_type`	It is an alias of one of the fundamental integral types (such as `int` or `long long`).

Each of the member types above is an alias of its non-member equivalent (they are the exact same type). It does not matter which one is used. The member types are more generic, because they are the same on all stream objects (even on streams using exotic types of characters), but the non-member types are widely used in existing code for historical reasons.

上面的每个成员类型都是其等效的非成员的别名(它们是完全相同的类型)。使用哪一个并不重要。成员类型更泛型，因为它们在所有流对象上都是相同的(甚至在使用特殊字符类型的流上也是如此)，但由于历史原因，非成员类型在现有代码中被广泛使用。

Binary files

For binary files, reading and writing data with the extraction and insertion operators (`<<` and `>>`) and functions like `getline` is not efficient, since we do not need to format any data and data is likely not formatted in lines.

对于二进制文件，使用提取和插入操作符(<<和>>)以及getline这样的函数读写数据是没有效率的，因为我们不需要格式化任何数据，而且数据很可能不会在行中进行格式化。

File streams include two member functions specifically designed to read and write binary data sequentially: `write` and `read`. The first one (`write`) is a member function of `ostream` (inherited by `ofstream`). And `read` is a member function of `istream` (inherited by `ifstream`). Objects of class `fstream` have both. Their prototypes are:

文件流包括两个专门用于顺序读写二进制数据的成员函数:写入和读取。第一个(write)是ostream的成员函数(由ofstream继承)。read是istream的成员函数(由ifstream继承)。fstream类的对象两者都有。他们的原型:

`write ( memory_block, size );read ( memory_block, size );`

Where `memory_block` is of type `char*` (pointer to `char`), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken. The `size` parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.

其中memory_block的类型是char*(指向char的指针)，并表示一个字节数组的地址，读取的数据元素存储在该数组中，或从该数组获取要写入的数据元素。size参数是一个整型值，指定从内存块读取或写入的字符数。

```
// reading an entire binary file
#include <iostream>
```

```cpp
#include <fstream>
using namespace std;

int main () {
  streampos size;
  char * memblock;

  ifstream file ("example.bin", ios::in|ios::binary|ios::ate);
  if (file.is_open())
  {
    size = file.tellg();
    memblock = new char [size];
    file.seekg (0, ios::beg);
    file.read (memblock, size);
    file.close();

    cout << "the entire file content is in memory";

    delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

In this example, the entire file is read and stored in a memory block. Let's examine how this is done:

在本例中，整个文件被读取并存储在内存块中。让我们来看看这是如何做到的:

First, the file is open with the `ios::ate` flag, which means that the get pointer will be positioned at the end of the file. This way, when we call to member `tellg()`, we will directly obtain the size of the file.

首先，文件以' ios::ate '标志打开，这意味着get指针将被定位在文件的末尾。这样，当我们调用成员' tellg() '时，我们将直接获得文件的大小。

Once we have obtained the size of the file, we request the allocation of a memory block large enough to hold the entire file:

一旦我们获得了文件的大小，我们就请求分配一个足够容纳整个文件的内存块:

```cpp
memblock = new char[size];
```

Right after that, we proceed to set the *get position* at the beginning of the file (remember that we opened the file with this pointer at the end), then we read the entire file, and finally close it:

在那之后，我们继续在文件的开始处设置get位置(记住，我们在文件的末尾用这个指针打开了文件)，然后我们读取整个文件，最后关闭它:

```
file.seekg (0, ios::beg);
file.read (memblock, size);
file.close();
```

At this point we could operate with the data obtained from the file. But our program simply announces that the content of the file is in memory and then finishes.

此时，我们可以使用从文件中获得的数据进行操作。但我们的程序只是宣布文件的内容在内存中，然后结束。

Buffers and Synchronization

When we operate with file streams, these are associated to an internal buffer object of type `streambuf`. This buffer object may represent a memory block that acts as an intermediary between the stream and the physical file. For example, with an `ofstream`, each time the member function `put` (which writes a single character) is called, the character may be inserted in this intermediate buffer instead of being written directly to the physical file with which the stream is associated.

当我们操作文件流时，它们与类型为streambuf的内部缓冲区对象相关联。这个缓冲区对象可能代表一个充当流和物理文件之间的中介的内存块。例如，对于ofstream，每次调用成员函数put(写入单个字符)时，字符可能会被插入这个中间缓冲区，而不是直接写入与流相关联的物理文件。

The operating system may also define other layers of buffering for reading and writing to files.

操作系统还可以定义用于读写文件的其他缓冲层。

When the buffer is flushed, all the data contained in it is written to the physical medium (if it is an output stream). This process is called *synchronization* and takes place under any of the following circumstances:

当缓冲区被刷新时，其中包含的所有数据都被写入物理介质(如果它是一个输出流)。这个过程被称为同步，并在下列任何一种情况下发生:

- **When the file is closed:** before closing a file, all buffers that have not yet been flushed are synchronized and all pending data is written or read to the physical medium.

  当文件关闭时:在关闭文件之前，所有尚未刷新的缓冲区被同步，所有挂起的数据被写入或读到物理介质。
- **When the buffer is full:** Buffers have a certain size. When the buffer is full it is automatically synchronized.

  当缓冲区满时:缓冲区有一定的大小。当缓冲区满时，它将自动同步。
- **Explicitly, with manipulators:** When certain manipulators are used on streams, an explicit synchronization takes place. These manipulators are: `flush` and `endl`.

  显式地，使用操纵符:当在流上使用某些操纵符时，会发生显式的同步。这些操纵符是:flush和endl。

- **Explicitly, with member function sync():** Calling the stream's member function `sync()` causes an immediate synchronization. This function returns an `int` value equal to `-1` if the stream has no associated buffer or in case of failure. Otherwise (if the stream buffer was successfully synchronized) it returns `0`.

  显式地，使用成员函数sync():调用流的成员函数sync()会导致立即同步。如果流没有相关的缓冲区或在失败的情况下，该函数返回一个等于-1的int值。否则(如果流缓冲区已成功同步)它将返回0。