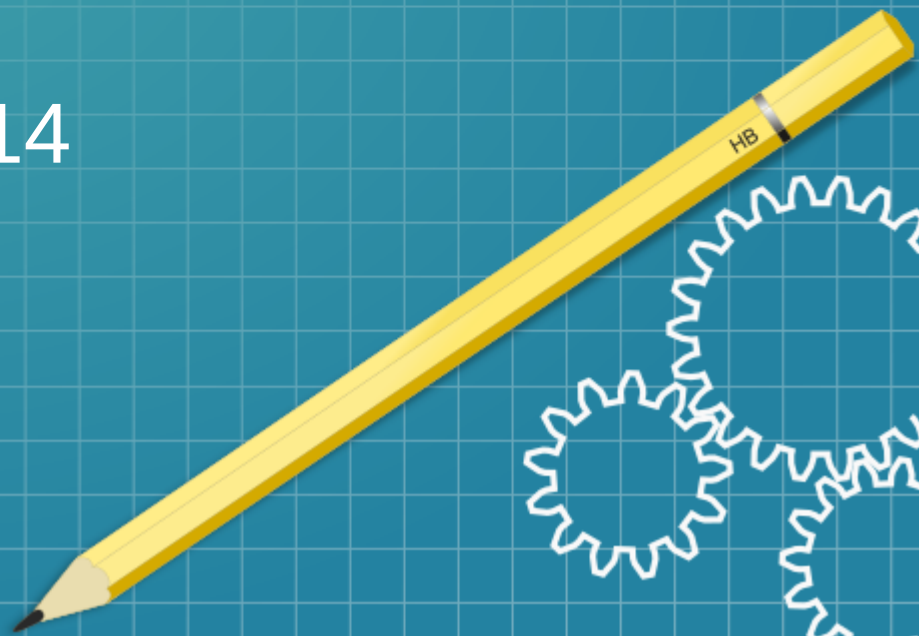
File Processing (Part 2)

Chapter 14



Some bits and pieces from Chapter 13

```
if (!cin) {
    // process invalid input stream
}
```

Some bits and pieces from Chapter 13

```
if (!cin) {
    // process invalid input stream
}
```

`operator!` member function, inherited into the stream classes from `basic_ios`, returns *true* if the `badbit` or `failbit` are *false*

Some bits and pieces from Chapter 13

```
if (!cin) {
    // process invalid input stream
}
```

`operator!` member function, inherited into the stream classes from `basic_ios`, returns *true* if the `badbit` or `failbit` are *false*

```
while (cin >> a){
 // process valid input
}
```

Some bits and pieces from Chapter 13

```
if (!cin) {
    // process invalid input stream
}
```

`operator!` member function, inherited into the stream classes from `basic_ios`, returns *true* if the `badbit` or `failbit` are *false*

```
while (cin >> a){
 // process valid input
}
```

`operator bool` member function, added in C++ 11, returns *false* if the `badbit` or `failbit` are *true*

Random Access Files

So far we created and read *sequential files*. Searched them for information, stored information in them.

Sequential files are not good for *instant-access applications*, in which a particular record must be located immediately.

Random Access Files

So far we created and read *sequential files*. Searched them for information, stored information in them.

Sequential files are not good for *instant-access applications*, in which a particular record must be located immediately.

Examples of *instant – access applications*:
- airline reservation systems
- banking systems
- point-of-sale systems
- automated teller machines
- other kinds of transaction-processing systems

- all of them require rapid access to specific data

Random Access Files

So far we created and read *sequential files*. Searched them for information, stored information in them.

Sequential files are not good for *instant-access applications*, in which a particular record must be located immediately.

Examples of *instant – access applications*:
- airline reservation systems
- banking systems
- point-of-sale systems
- automated teller machines
- other kinds of transaction-processing systems

- all of them require rapid access to specific data

Random-access files provide such an access: the individual records ca be accessed directly and quickly.

Random Access Files

C++ does not impose structure on a file.

The application that wants to use random-access files must create them.

Random Access Files

C++ does not impose structure on a file.

The application that wants to use random-access files must create them.

A variety of techniques can be used.

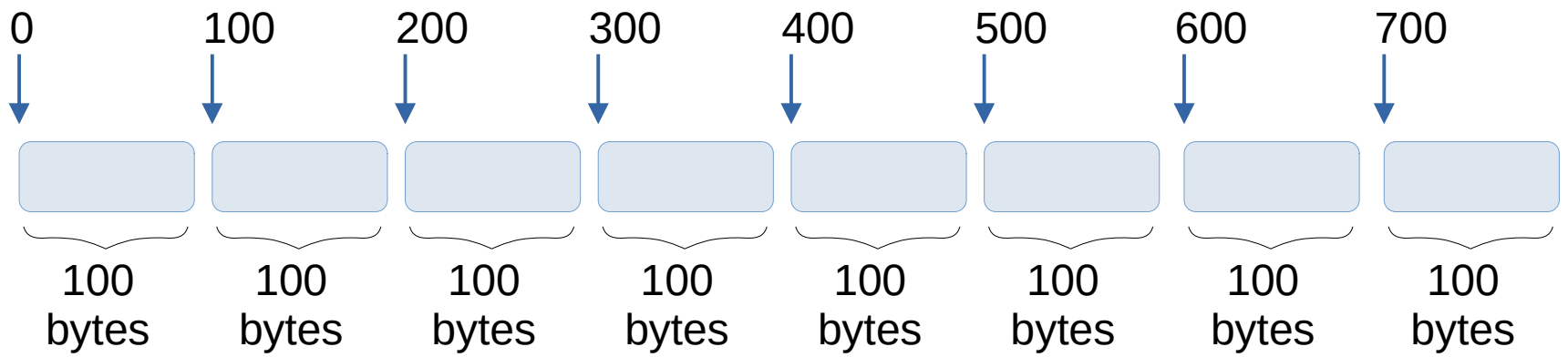
Random Access Files

C++ does not impose structure on a file.

The application that wants to use random-access files must create them.

A variety of techniques can be used.

An easy one: require all records to be of the *same fixed length*.



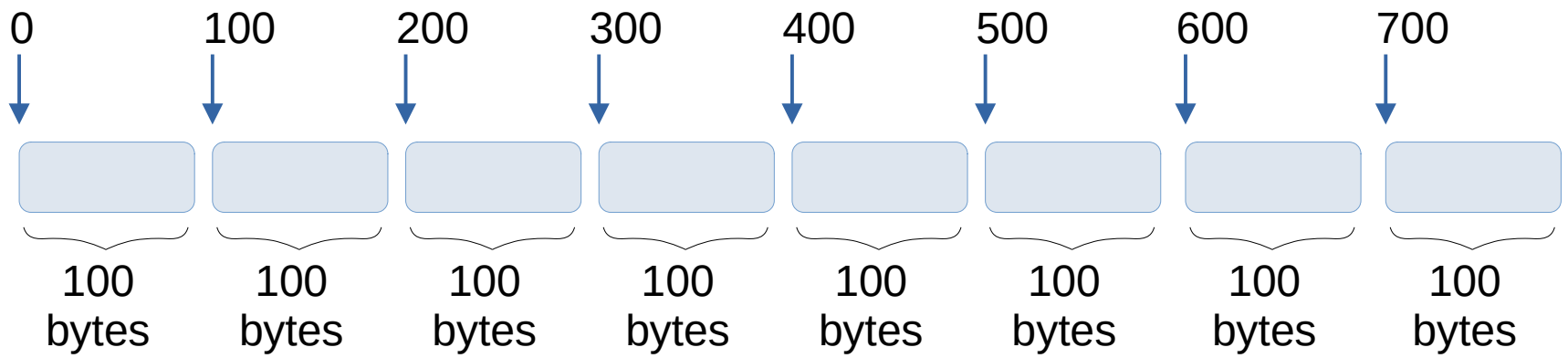
Random Access Files

C++ does not impose structure on a file.

The application that wants to use random-access files must create them.

A variety of techniques can be used.

An easy one: require all records to be of the *same fixed length*.



Data can be *inserted* into a file without disturbing other data in the file. Data stored can be *updated/deleted* without rewriting the entire file.

We will use:

`write` (member function of `ostream`) that provides unformatted output; it inserts the first `n` characters of the array pointed to by `s` into the stream.

```
ostream& write(const char* s, streamsize n);
```

This function simply copies a block of data, without checking its contents: the array may contain null characters, which are also copied without stopping the copying process.

We will use:

`read` (member function of `istream`) that provides unformatted input; it reads `n` input bytes into a built-in array of chars;

```
istream& read (char* s, streamsize n);
```

This function simply copies a block of data, without checking its contents nor appending a null character at the end. If the input sequence runs out of characters to extract (i.e., the end-of-file is reached) before n characters have been successfully read, the array pointed to by s contains all the characters read until that point, and both the `eofbit` and `failbit` flags are set for the stream.

We will use:

`read` (member function of `istream`) that provides unformatted input; it reads `n` input bytes into a built-in array of chars;

```
istream& read (char* s, streamsize n);
```

This function simply copies a block of data, without checking its contents nor appending a null character at the end. If the input sequence runs out of characters to extract (i.e., the end-of-file is reached) before n characters have been successfully read, the array pointed to by s contains all the characters read until that point, and both the `eofbit` and `failbit` flags are set for the stream.

Let's see an example that outputs three integer values into a file, and then retrieves the second number from it:

see writingAndReadingFilesForRandomAccess.cpp

File Position Pointers will be used as well:

`istream` and `ostream` provide member functions:

`seekg` : seek get; sets the *position* of the next character to be extracted from the input stream.

```
istream& seekg(streampos pos);
```

`seekp` : seek put; sets the *position* where the next character is to be inserted into the output stream

```
ostream& seekp(streampos pos);
```

Both functions reposition the *file-position pointer*.

Each `istream` object has a *get pointer*, the byte number in the file from which the *next input* to occur.
Each `ostream` object has a *put pointer*, the byte number in the file at which the next *output* should be placed.

Bank Accounts

We discussed *bank accounts* at the previous meeting.

Consider the following idea:

- create instances of `class Account` (needs to be defined)
- these objects can be made of fixed size (implementation decision)
- store them in a file
- retrieve them from a file using *random-access*

Bank Accounts

We discussed *bank accounts* at the previous meeting.

Consider the following idea:

- create instances of `class Account` (needs to be defined)
- these objects can be made of fixed size (implementation decision)
- store them in a file
- retrieve them from a file using *random-access*

see Account.h, Account.cpp, StoringClientData.cpp

Object Serialization

When we output an object into a file, its data attributes are output, not the member functions.

We “lose” object’s type information as well.

So if a program reads information from a file, it needs to know what type objects are “stored” there.

Our random-access files are not portable, because the size of the Account object is platform dependent.

Object Serialization

Object serialization allows us to represent objects in a *platform-independent manner* as a sequence of bytes that include the *object's data* as well as information about the *object's type* and the *types of data stored in the object*.

After such a *serialized object* is written to a file, it can be read from the file and *deserialized*, i.e. the type information and bytes that represent the object and its data can be used to *recreate* the object in memory.

C++ (up to C++ 14) doesn't provide a built-in serialization mechanism.

There are third-party and open-source C++ libraries that support object serialization.

HW assignment

1) Exercise 14.11
(feel free to reduce the number of records to 10)
2) to be posted after the next lecture

Self-Study:
Chapter 14, Self-Review Exercises

Optional (for self-development):
Sections 13.6.4