IS 0020
Program Design and Software Tools

Exception Handling, File Processing

Lecture 11

March 30, 2004

Introduction

- Exceptions
  - Indicates problem occurred in program
  - Not common
    - An "exception" to a program that usually works
- Exception Handling
  - Resolve exceptions
  - Program may be able to continue
    - Controlled termination
  - Write fault-tolerant programs

© 2003 Prentice Hall, Inc. All rights reserved.

Exception-Handling Overview

- Consider pseudocode

  Perform a task

  If the preceding task did not execute correctly

          Perform error processing

  Perform next task

  If the preceding task did not execute correctly

          Perform error processing

- Mixing logic and error handling
  - Can make program difficult to read/debug
  - Exception handling removes error correction from "main line" of program

© 2003 Prentice Hall, Inc. All rights reserved.

Exception-Handling Overview

- Exception handling
  - For synchronous errors (divide by zero, null pointer)
    - Cannot handle asynchronous errors (independent of program)
    - Disk I/O, mouse, keyboard, network messages
  - Easy to handle errors
- Terminology
  - Function that has error *throws an exception*
  - *Exception handler* (if it exists) can deal with problem
    - *Catches* and *handles* exception
  - If no exception handler, *uncaught* exception
    - Could terminate program

© 2003 Prentice Hall, Inc. All rights reserved.

Exception-Handling Overview

- C++ code

```
try {
    code that may raise exception
}
catch (exceptionType){
    code to handle exception
}
```

  - **`try`** block encloses code that may raise exception
  - One or more **`catch`** blocks follow
    - Catch and handle exception, if appropriate
    - Take parameter; if named, can access exception object

© 2003 Prentice Hall, Inc. All rights reserved.

Exception-Handling Overview

- Throw point
  - Location in **`try`** block where exception occurred
  - If exception handled
    - Program skips remainder of **`try`** block
    - Resumes after **`catch`** blocks
  - If not handled
    - Function terminates
    - Looks for enclosing **`catch`** block (stack unwinding, 13.8)
- If no exception
  - Program skips **`catch`** blocks

© 2003 Prentice Hall, Inc. All rights reserved.

Other Error-Handling Techniques

- Ignore exception
  - Typical for personal (not commercial) software
  - Program may fail
- Abort program
  - Usually appropriate
  - Not appropriate for mission-critical software
- Set error indicators
  - Unfortunately, may not test for these when necessary
- Test for error condition
  - Call exit (**`<cstdlib>`**) and pass error code

© 2003 Prentice Hall, Inc. All rights reserved.

Other Error-Handling Techniques

- **`setjump`** and **`longjump`**
  - **`<csetjmp>`**
  - Jump from deeply nested function to call error handler
  - Can be dangerous
- Dedicated error handling
  - **`new`** can have a special handler

© 2003 Prentice Hall, Inc. All rights reserved.

Simple Exception-Handling Example: Divide by Zero

- Keyword **`throw`**
  - Throws an exception
    - Use when error occurs
  - Can throw almost anything (exception object, integer, etc.)
    - **`throw myObject;`**
    - **`throw 5;`**
- Exception objects
  - Base class **`runtime_error`** (**`<stdexcept>`**)
  - Constructor can take a string (to describe exception)
  - Member function **`what()`** returns that string

© 2003 Prentice Hall, Inc. All rights reserved.

Simple Exception-Handling Example: Divide by Zero

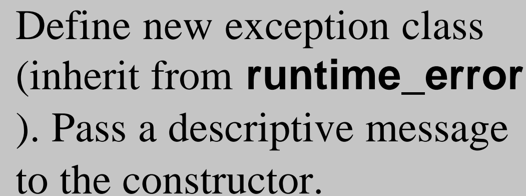
- Upcoming example
  - Handle divide-by-zero errors
  - Define new exception class
    - **`DivideByZeroException`**
    - Inherit from **`runtime_error`**
  - In division function
    - Test denominator
    - If zero, throw exception (**`throw object`**)
  - In **`try`** block
    - Attempt to divide
    - Have enclosing **`catch`** block
      - Catch **`DivideByZeroException`** objects

© 2003 Prentice Hall, Inc. All rights reserved.

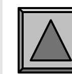
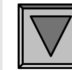
Outline

fig13_01.cpp
(1 of 3)

```
1   // Fig. 13.1: fig13_01.cpp
2   // A simple exception-handling example that checks for
3   // divide-by-zero exceptions.
4   #include <iostream>
5
6   using std::cout;
7   using std::cin;
8   using std::endl;
9
10  #include <exception>
11
12  using std::exception;
13
14  // DivideByZeroException objects should be thrown by functions
15  // upon detecting division-by-zero exceptions
16  class DivideByZeroException : public runtime_error {
17
18  public:
19
20     // constructor specifies default error message
21     DivideByZeroException::DivideByZeroException()
22        : exception( "attempted to divide by zero" ) {}
23
24  };  // end class DivideByZeroException
25
```



© 2003 Prentice Hall, Inc.
All rights reserved.

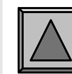
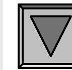




```
26 // perform division and throw DivideByZeroException object if
27 // divide-by-zero exception occurs
28 double quotient( int numerator, int denominator )
29 {
30    // throw DivideByZeroException if trying to divide by zero
31    if ( denominator == 0 )
32       throw DivideByZeroException(); // terminate function
33
34    // return division result
35    return static_cast< double >( numerator ) / denominator;
36
37 }  // end function quotient
38
39 int main()
40 {
41    int number1;    // user-specified numerator
42    int number2;    // user-specified denominator
43    double result;  // result of division
44
45    cout << "Enter two integers (end-of-file to end): ";
46
```

If the denominator is zero, **throw** a **DivideByZeroException** object.

fig13_01.cpp

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

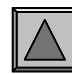
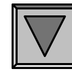
fig13_01.cpp
(3 of 3)

```
47     // enable user to enter two integers to divide
48     while ( cin >> number1 >> number2 ) {
49
50        // try block contains code that might throw exception
51        // and code that should not execute if an exception occurs
52        try {
53           result = quotient( number1, number2 );
54           cout << "The quotient is: " << result << endl;
55
56        } // end try
57
58        // exception handler handles a divide-by-zero exception
59        catch ( DivideByZeroException &divideByZeroException ) {
60           cout << "Exception occurred: "
61                << divideByZeroException.what() << endl;
62
63        } // end catch
64
65        cout << "\nEnter two intege
66
67     }  // end while
68
69     cout << endl;
70
71     return 0;  // terminate normal
72
73  }  // end main
```

Notice the structure of the **`try`** and **`catch`** blocks. The **`catch`** block can catch **`DivideByZeroException`** objects, and print an error message. If no exception occurs, the **`catch`** block is skipped.

Member function **`what`** returns the string describing the exception.

© 2003 Prentice Hall, Inc.
All rights reserved.


```
Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

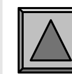
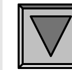
fig13_01.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Rethrowing an Exception

- Rethrowing exceptions
  - Use when exception handler cannot process exception
    - Can still rethrow if handler did some processing
  - Can rethrow exception to another handler
    - Goes to next enclosing **`try`** block
    - Corresponding **`catch`** blocks try to handle
- To rethrow
  - Use statement "**`throw;`**"
    - No arguments
    - Terminates function

© 2003 Prentice Hall, Inc. All rights reserved.

Outline

fig13_02.cpp
(1 of 2)

```
1  // Fig. 13.2: fig13_02.cpp
2  // Demonstrating exception rethrowing.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <exception>
9
10 using std::exception;
11
12 // throw, catch and rethrow exception
13 void throwException()
14 {
15    // throw exception and catch it immediately
16    try {
17       cout << "  Function throwException throws an exception\n";
18       throw exception();  // generate exception
19
20    } // end try
21
22    // handle exception
23    catch ( exception &caughtException ) {
24       cout << "  Exception handled in function throwException"
25            << "\n  Function throwException rethrows exception";
26
27       throw;  // rethrow exception for further processing
28
29    } // end catch
```

Exception handler generates a default exception (base class **`exception`**). It immediately catches and rethrows it (note use of **`throw;`**).

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

fig13_02.cpp
(2 of 2)

```
30
31      cout << "This also should not print\n";
32
33   }  // end function throwException
34
35   int main()
36   {
37      // throw exception
38      try {
39         cout << "\nmain invokes function throwException\n";
40         throwException();
41         cout << "This should not print\n";
42
43      } // end try
44
45      // handle exception
46      catch ( exception &caughtException ) {
47         cout << "\n\nException handled in main\n";
48
49      } // end catch
50
51      cout << "Program control continues after catch in main\n";
52
53      return 0;
54
55   }  // end main
```

This should never be reached, since the **`throw`** immediately exits the function.

`throwException` rethrows an exception to **`main`**. It is caught and handled.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
main invokes function throwException
  Function throwException throws an exception
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

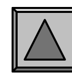
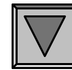



fig13_02.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Exception Specifications

- List of exceptions function can throw
  - Also called throw list

```
int someFunction( double value )
  throw ( ExceptionA, ExceptionB, ExceptionC )
{
   // function body
}
```

  - Can only throw **ExceptionA**, **ExceptionB**, and **ExceptionC** (and derived classes)
    - If throws other type, function **unexpected** called
    - By default, terminates program (more 13.7)
  - If no throw list, can throw any exception
  - If empty throw list, cannot throw any exceptions

© 2003 Prentice Hall, Inc. All rights reserved.

Processing Unexpected Exceptions

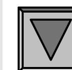
- Function **`unexpected`**
  - Calls function registered with **`set_unexpected`**
    - **`<exception>`**
    - Calls **`terminate`** by default
  - **`set_terminate`**
    - Sets what function **`terminate`** calls
    - By default, calls **`abort`**
      - If redefined, still calls **`abort`** after new function finishes
- Arguments for set functions
  - Pass pointer to function
    - Function must take no arguments
    - Returns **`void`**

© 2003 Prentice Hall, Inc. All rights reserved.

Stack Unwinding

- If exception thrown but not caught
  - Goes to enclosing `try` block
  - Terminates current function
    - Unwinds function call stack
  - Looks for `try`/`catch` that can handle exception
    - If none found, unwinds again
- If exception never caught
  - Calls `terminate`

© 2003 Prentice Hall, Inc. All rights reserved.



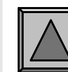
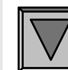
Outline

fig13_03.cpp
(1 of 2)

```
1  // Fig. 13.3: fig13_03.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <stdexcept>
9
10 using std::runtime_error;
11
12 // function3 throws run-time error
13 void function3() throw ( runtime_error )
14 {
15    throw runtime_error( "runtime_error in function3" ); // fourth
16 }
17
18 // function2 invokes function3
19 void function2() throw ( runtime_error )
20 {
21    function3(); // third
22 }
23
```

Note the use of the throw list. Throws a runtime error exception, defined in `<stdexcept>`.

© 2003 Prentice Hall, Inc.
All rights reserved.


```
24 // function1 invokes function2
25 void function1() throw ( runtime_error )
26 {
27    function2(); // second
28 }
29
30 // demonstrate stack unwinding
31 int main()
32 {
33    // invoke function1
34    try {
35       function1(); // first
36
37    } // end try
38
39    // handle run-time error
40    catch ( runtime_error &error ) // fifth
41    {
42       cout << "Exception occurred: " << error.what() << endl;
43
44    } // end catch
45
46    return 0;
47
48 }  // end main
```

function1 calls **function2** which calls **function3**. The exception occurs, and unwinds until an appropriate **try**/**catch** block can be found.

```
Exception occurred: runtime_error in function3
```

© 2003 Prentice Hall, Inc.
All rights reserved.

Constructors, Destructors and Exception Handling

- Error in constructor
  - `new` fails; cannot allocate memory
  - Cannot return a value - how to inform user?
    - Hope user examines object, notices errors
    - Set some global variable
  - Good alternative: throw an exception
    - Destructors automatically called for member objects
    - Called for automatic variables in `try` block
- Can catch exceptions in destructor

© 2003 Prentice Hall, Inc. All rights reserved.

Exceptions and Inheritance

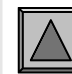
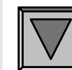
- Exception classes
  - Can be derived from base classes
    - I.e., **`runtime_error`**; **`exception`**
  - If **`catch`** can handle base class, can handle derived classes
    - Polymorphic programming

© 2003 Prentice Hall, Inc. All rights reserved.

Processing new Failures

- When **`new`** fails to get memory
  - Should **`throw`** **`bad_alloc`** exception
    - Defined in **`<new>`**
  - Some compilers have **`new`** return 0
  - Result depends on compiler

© 2003 Prentice Hall, Inc. All rights reserved.

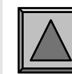
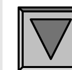




```
1   // Fig. 13.4: fig13_04.cpp
2   // Demonstrating pre-standard new returning 0 when memory
3   // is not allocated.
4   #include <iostream>
5
6   using std::cout;
7
8   int main()
9   {
10     double *ptr[ 50 ];
11
12     // allocate memory for ptr
13     for ( int i = 0; i < 50; i++ ) {
14        ptr[ i ] = new double[ 5000000 ];
15
16        // new returns 0 on failure to alloc
17        if ( ptr[ i ] == 0 ) {
18           cout << "Memory allocation failed for ptr[ "
19                << i << " ]\n";
20
21           break;
22
23        } // end if
24
```

Demonstrating `new` that returns `0` on allocation failure.

fig13_04.cpp
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

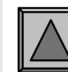
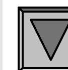

```
25          // successful memory allocation
26          else
27             cout << "Allocated 5000000 doubles in ptr[ "
28                  << i << " ]\n";
29
30       } // end for
31
32       return 0;
33
34    }  // end main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Memory allocation failed for ptr[ 4 ]
```

fig13_04.cpp
(2 of 2)

fig13_04.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

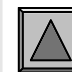
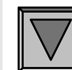
Outline

fig13_05.cpp
(1 of 2)

```
1   // Fig. 13.5: fig13_05.cpp
2   // Demonstrating standard new throwing bad_alloc when memory
3   // cannot be allocated.
4   #include <iostream>
5
6   using std::cout;
7   using std::endl;
8
9   #include <new> // standard operator new
10
11  using std::bad_alloc;
12
13  int main()
14  {
15     double *ptr[ 50 ];
16
17     // attempt to allocate memory
18     try {
19
20        // allocate memory for ptr[ i ]; new throws bad_alloc
21        // on failure
22        for ( int i = 0; i < 50; i++ ) {
23           ptr[ i ] = new double[ 5000000 ];
24           cout << "Allocated 5000000 doubles in ptr[ "
25                << i << " ]\n";
26        }
27
28     } // end try
```

Demonstrating **new** that throws an exception.

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

```
29
30     // handle bad_alloc exception
31     catch ( bad_alloc &memoryAllocationException ) {
32        cout << "Exception occurred: "
33             << memoryAllocationException.what() << endl;
34
35     } // end catch
36
37     return 0;
38
39  }  // end main
```

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
Exception occurred: Allocation Failure
```

fig13_05.cpp
(2 of 2)

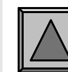
fig13_05.cpp
output (1 of 1)

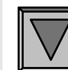
© 2003 Prentice Hall, Inc.
All rights reserved.

Processing new Failures

- **`set_new_handler`**
  - Header **`<new>`**
  - Register function to call when **`new`** fails
  - Takes function pointer to function that
    - Takes no arguments
    - Returns **`void`**
  - Once registered, function called instead of throwing exception

© 2003 Prentice Hall, Inc. All rights reserved.






```
1   // Fig. 13.6: fig13_06.cpp
2   // Demonstrating set_new_handler.
3   #include <iostream>
4
5   using std::cout;
6   using std::cerr;
7
8   #include <new>     // standard operator new and set_new_handler
9
10  using std::set_new_handler;
11
12  #include <cstdlib> // abort :
13
14  void customNewHandler()
15  {
16     cerr << "customNewHandler was called";
17     abort();
18  }
19
20  // using set_new_handler to handle failed memory allocation
21  int main()
22  {
23     double *ptr[ 50 ];
24
```

The custom handler must take no arguments and return **`void`**.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
25    // specify that customNewHandler should be called on failed
26    // memory allocation
27    set_new_handler( customNewHandler );
28
29    // allocate memory for ptr[ i ]; customNewHandler will be
30    // called on failed memory allocation
31    for ( int i = 0; i < 50; i++ ) {
32       ptr[ i ] = new double[ 5000000 ];
33
34       cout << "Allocated 5000000 doubles in ptr[ "
35            << i << " ]\n";
36
37    } // end for
38
39    return 0;
40
41 }  // end main
```

Note call to **set_new_handler**.

```
Allocated 5000000 doubles in ptr[ 0 ]
Allocated 5000000 doubles in ptr[ 1 ]
Allocated 5000000 doubles in ptr[ 2 ]
Allocated 5000000 doubles in ptr[ 3 ]
customNewHandler was called
```


© 2003 Prentice Hall, Inc.
All rights reserved.

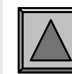
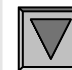
Class auto_ptr and Dynamic Memory Allocation

- Declare pointer, allocate memory with **`new`**
  - What if exception occurs before you can **`delete`** it?
  - Memory leak
- Template class **`auto_ptr`**
  - Header **`<memory>`**
  - Like regular pointers (has **`*`** and **`->`**)
  - When pointer goes out of scope, calls **`delete`**
  - Prevents memory leaks
  - Usage

```
auto_ptr< MyClass > newPointer( new MyClass() );
```

    - **`newPointer`** points to dynamically allocated object

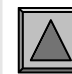
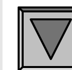
© 2003 Prentice Hall, Inc. All rights reserved.


```
1   // Fig. 13.7: fig13_07.cpp
2   // Demonstrating auto_ptr.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <memory>
9
10  using std::auto_ptr; // auto_ptr class definition
11
12  class Integer {
13
14  public:
15
16     // Integer constructor
17     Integer( int i = 0 )
18        : value( i )
19     {
20        cout << "Constructor for Integer " << value << endl;
21
22     } // end Integer constructor
23
```

fig13_07.cpp
(1 of 3)

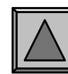
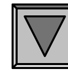
© 2003 Prentice Hall, Inc.
All rights reserved.


```
24     // Integer destructor
25     ~Integer()
26     {
27        cout << "Destructor for Integer " << value << endl;
28
29     } // end Integer destructor
30
31     // function to set Integer
32     void setInteger( int i )
33     {
34        value = i;
35
36     } // end function setInteger
37
38     // function to return Integer
39     int getInteger() const
40     {
41        return value;
42
43     } // end function getInteger
44
45  private:
46     int value;
47
48  };  // end class Integer
49
```

fig13_07.cpp
(2 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.


```
50 // use auto_ptr to manipulate Integer object
51 int main()
52 {
53    cout << "Creating an auto_ptr object that points to an "
54         << "Integer\n";
55
56    // "aim" auto_ptr at Integer object
57    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );
58
59    cout << "\nUsing the auto_ptr to manipulate the Integer\n";
60
61    // use auto_ptr to set Integer value
62    ptrToInteger->setInteger( 99 );
63
64    // use auto_ptr to get Integer value
65    cout << "Integer after setInteger: "
66         << ( *ptrToInteger ).getInteger()
67         << "\n\nTerminating program" << endl;
68
69    return 0;
70
71 }  // end main
```

Create an **`auto_ptr`**. It can be manipulated like a regular pointer.

`delete` not explicitly called, but the **`auto_ptr`** will be destroyed once it leaves scope. Thus, the destructor for class **`Integer`** will be called.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
Creating an auto_ptr object that points to an Integer
Constructor for Integer 7

Using the auto_ptr to manipulate the Integer
Integer after setInteger: 99

Terminating program
Destructor for Integer 99
```

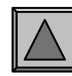
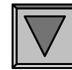



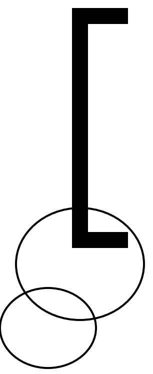
fig13_07.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Standard Library Exception Hierarchy

- Exception hierarchy
  - Base class **`exception`** (**`<exception>`**)
    - Virtual function **`what`**, overridden to provide error messages
  - Sample derived classes
    - **`runtime_error`**, **`logic_error`**
    - **`bad_alloc`**, **`bad_cast`**, **`bad_typeid`**
      - Thrown by **`new`**, **`dynamic_cast`** and **`typeid`**
- To catch all exceptions
  - **`catch(...)`**
  - **`catch( exception AnyException)`**
    - Will not catch user-defined exceptions

© 2003 Prentice Hall, Inc. All rights reserved.



© 2003 Prentice Hall, Inc. All rights reserved.

Introduction

- Storage of data
  - Arrays, variables are temporary
  - Files are permanent
    - Magnetic disk, optical disk, tapes
- In this chapter
  - Create, update, process files
  - Sequential and random access
  - Formatted and raw processing

© 2003 Prentice Hall, Inc. All rights reserved.

The Data Hierarchy

- From smallest to largest
  - Bit (binary digit)
    - 1 or 0
    - Everything in computer ultimately represented as bits
    - Cumbersome for humans to use
    - Character set
      - Digits, letters, symbols used to represent data
      - Every character represented by 1's and 0's
  - Byte: 8 bits
    - Can store a character (**`char`**)
    - Also Unicode for large character sets (**`wchar_t`**)

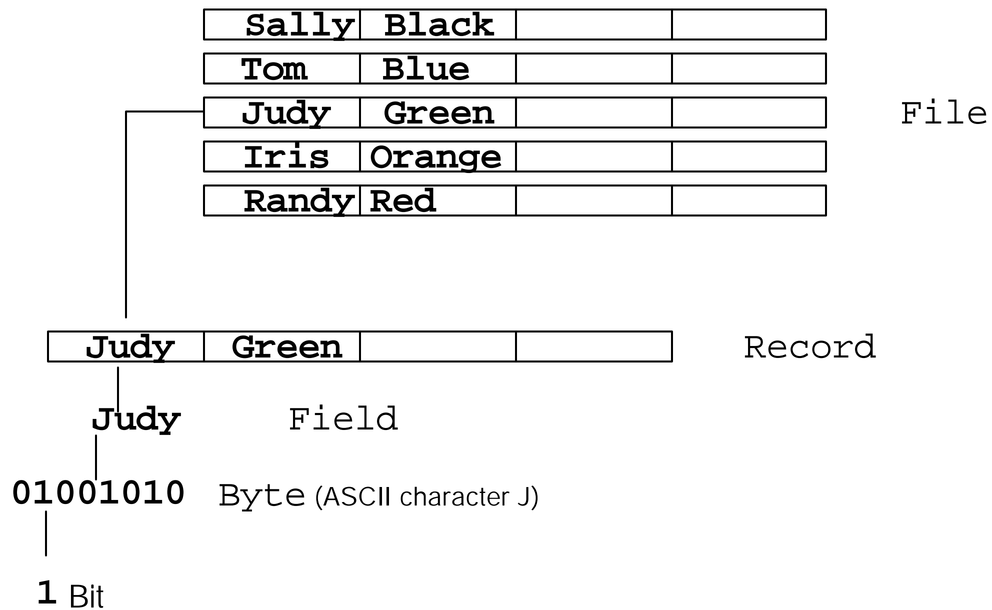
© 2003 Prentice Hall, Inc. All rights reserved.

The Data Hierarchy

- From smallest to largest (continued)
  - Field: group of characters with some meaning
    - Your name
  - Record: group of related fields
    - `struct` or `class` in C++
    - In payroll system, could be name, SS#, address, wage
    - Each field associated with same employee
    - Record key: field used to uniquely identify record
  - File: group of related records
    - Payroll for entire company
    - Sequential file: records stored by key
  - Database: group of related files
    - Payroll, accounts-receivable, inventory…

© 2003 Prentice Hall, Inc. All rights reserved.

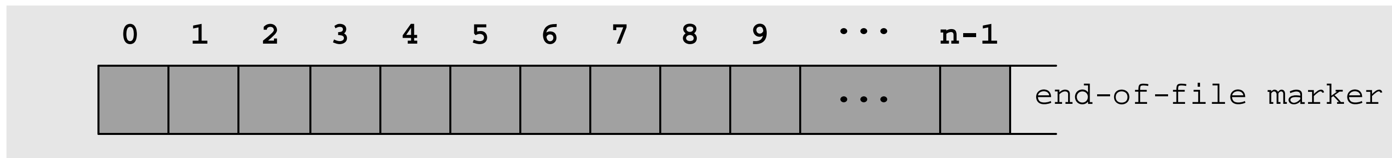
The Data Hierarchy



© 2003 Prentice Hall, Inc. All rights reserved.

Files and Streams

- C++ views file as sequence of bytes
  - Ends with *end-of-file* marker



- When file opened
  - Object created, stream associated with it
  - **`cin`**, **`cout`**, etc. created when **`<iostream>`** included
    - Communication between program and file/device

© 2003 Prentice Hall, Inc. All rights reserved.

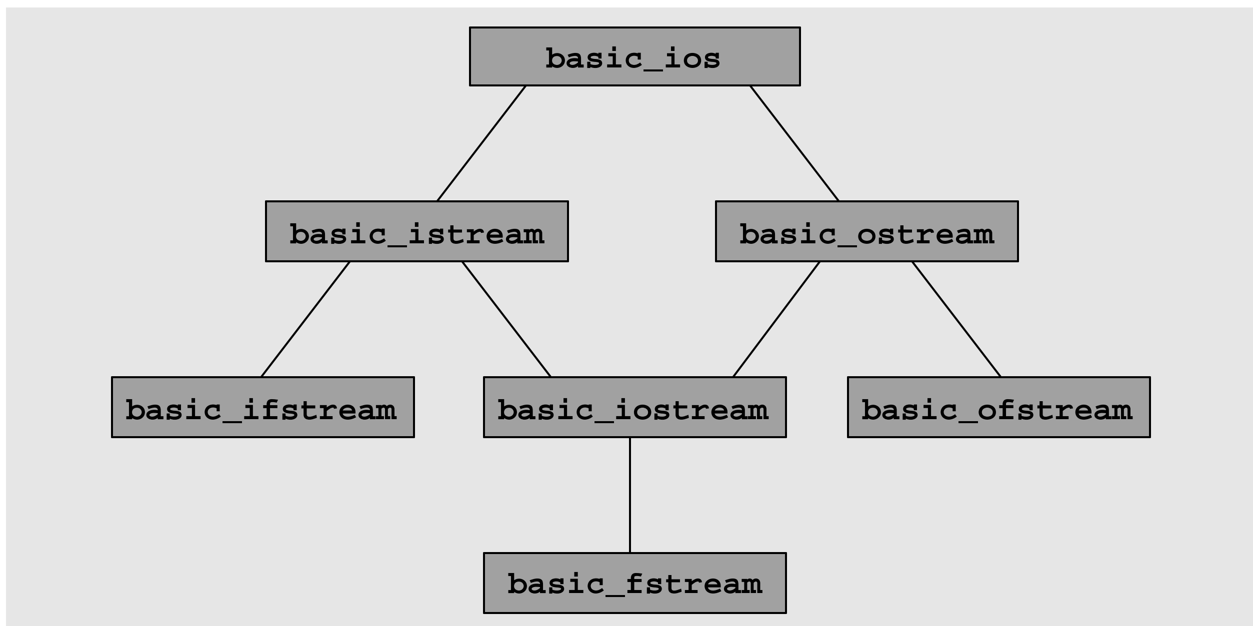
Files and Streams

- To perform file processing
  - Include `<iostream>` and `<fstream>`
  - Class templates
    - **basic_ifstream** (input)
    - **basic_ofstream** (output)
    - **basic_fstream** (I/O)
  - **typedef**s for specializations that allow **char** I/O
    - **ifstream** (**char** input)
    - **ofstream** (**char** output)
    - **fstream** (**char** I/O)

© 2003 Prentice Hall, Inc. All rights reserved.

Files and Streams

- Opening files
  - Create objects from template
  - Derive from stream classes
    - Can use stream methods from Ch. 12
    - **put**, **get**, **peek**, etc.



© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Sequential-Access File

- C++ imposes no structure on file
  - Concept of "record" must be implemented by programmer
- To open file, create objects
  - Creates "line of communication" from object to file
  - Classes
    - **`ifstream`** (input only)
    - **`ofstream`** (output only)
    - **`fstream`** (I/O)
  - Constructors take *file name* and *file-open mode*

```
ofstream outClientFile( "filename", fileOpenMode );
```

  - To attach a file later

```
Ofstream outClientFile;
outClientFile.open( "filename", fileOpenMode);
```

© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Sequential-Access File

- File-open modes

Mode	Description
ios::app	Write all output to the end of the file.
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::in	Open a file for input.
ios::out	Open a file for output.
ios::trunc	Discard the file's contents if it exists (this is also the default action for **ios::out**)
ios::binary	Open a file for binary (i.e., non-text) input or output.

  - **ofstream** opened for output by default
    - **ofstream outClientFile("clients.dat", ios::out);**
    - **ofstream outClientFile("clients.dat");**

© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Sequential-Access File

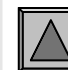
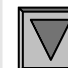
- Operations
  - Overloaded **`operator!`**
    - **`!outClientFile`**
    - Returns nonzero (true) if **`badbit`** or **`failbit`** set
      - Opened non-existent file for reading, wrong permissions
  - Overloaded **`operator void*`**
    - Converts stream object to pointer
    - **`0`** when when **`failbit`** or **`badbit`** set, otherwise nonzero
      - **`failbit`** set when EOF found
    - **`while ( cin >> myVariable )`**
      - Implicitly converts **`cin`** to pointer
      - Loops until EOF

© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Sequential-Access File

- Operations
  - Writing to file (just like **`cout`**)
    - **`outClientFile << myVariable`**
  - Closing file
    - **`outClientFile.close()`**
    - Automatically closed when destructor called

© 2003 Prentice Hall, Inc. All rights reserved.

Outline

fig14_04.cpp
(1 of 2)

```
1   // Fig. 14.4: fig14_04.cpp
2   // Create a sequential file.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::ios;
8   using std::cerr;
9   using std::endl;
10
11  #include <fstream>
12
13  using std::ofstream;
14
15  #include <cstdlib>  // exit prototype
16
17  int main()
18  {
19     // ofstream constructor opens file
20     ofstream outClientFile( "clients.dat", ios::out );
21
22     // exit program if unable to create file
23     if ( !outClientFile ) {  // overloaded ! operator
24        cerr << "File could not be opened" << endl;
25        exit( 1 );
26
27     } // end if
```

Notice the the header files required for file I/O.

`ofstream` object created and used to open file `"clients.dat"`. If the file does not exist, it is created.

`!` operator used to test if the file opened properly.

© 2003 Prentice Hall, Inc.
All rights reserved.

fig14_04.cpp
(2 of 2)

```
28
29     cout << "Enter the account, name, and balance." << endl
30          << "Enter end-of-file t
31
32     int account;
33     char name[ 30 ];
34     double balance;
35
36     // read account, name and balance from cin, then place in file
37     while ( cin >> account >> name >> balance ) {
38        outClientFile << account << ' ' << name << ' ' << balance
39                      << endl;
40        cout << "? ";
41
42     } // end while
43
44     return 0;  // ofstream destructor closes file
45
46  } // end main
```

cin is implicitly converted to a pointer. When EOF is encountered, it returns 0 and the loop stops.

Write data to file like a regular stream.

File closed when destructor called for object. Can be explicitly closed with **close()**.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

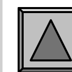
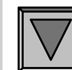



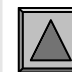
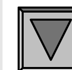
fig14_04.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Reading Data from a Sequential-Access File

- Reading files
  - **`ifstream inClientFile( "filename", ios::in );`**
  - Overloaded **`!`**
    - **`!inClientFile`** tests if file was opened properly
  - **`operator void*`** converts to pointer
    - **`while (inClientFile >> myVariable)`**
    - Stops when EOF found (gets value **`0`**)

© 2003 Prentice Hall, Inc. All rights reserved.

Outline

fig14_07.cpp
(1 of 3)

```
1   // Fig. 14.7: fig14_07.cpp
2   // Reading and printing a sequential file.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::ios;
8   using std::cerr;
9   using std::endl;
10  using std::left;
11  using std::right;
12  using std::fixed;
13  using std::showpoint;
14
15  #include <fstream>
16
17  using std::ifstream;
18
19  #include <iomanip>
20
21  using std::setw;
22  using std::setprecision;
23
24  #include <cstdlib> // exit prototype
25
26  void outputLine( int, const char * const, double );
27
```

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

fig14_07.cpp
(2 of 3)

```
28 int main()
29 {
30    // ifstream constructor opens the file
31    ifstream inClientFile( "clients.dat", ios::in );
32
33    // exit program if ifstream could not open file
34    if ( !inClientFile ) {
35       cerr << "File could not be opened" << endl;
36       exit( 1 );
37
38    } // end if
39
40    int account;
41    char name[ 30 ];
42    double balance;
43
44    cout << left << setw( 10 ) << "Acco
45         << "Name" << "Balance" << endl
46
47    // display each record in file
48    while ( inClientFile >> account >> name >> balance )
49       outputLine( account, name, balance );
50
51    return 0; // ifstream destructor closes the file
52
53 } // end main
```

Open and test file for input.

Read from file until EOF found.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
54
55 // display single record from file
56 void outputLine( int account, const char * const name,
57    double balance )
58 {
59    cout << left << setw( 10 ) << account << setw( 13 ) << name
60         << setw( 7 ) << setprecision( 2 ) << right << balance
61         << endl;
62
63 } // end function outputLine
```

```
Account   Name           Balance
100       Jones            24.98
200       Doe             345.67
300       White             0.00
400       Stone           -42.16
500       Rich            224.62
```


© 2003 Prentice Hall, Inc.
All rights reserved.

Reading Data from a Sequential-Access File

- File position pointers
  - Number of next byte to read/write
  - Functions to reposition pointer
    - **`seekg`** (seek get for **`istream`** class)
    - **`seekp`** (seek put for **`ostream`** class)
    - Classes have "get" and "put" pointers
  - **`seekg`** and **`seekp`** take *offset* and *direction*
    - Offset: number of bytes relative to direction
    - Direction (**`ios::beg`** default)
      - **`ios::beg`** - relative to beginning of stream
      - **`ios::cur`** - relative to current position
      - **`ios::end`** - relative to end

© 2003 Prentice Hall, Inc. All rights reserved.

Reading Data from a Sequential-Access File

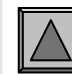
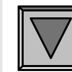
- Examples
  - **`fileObject.seekg(0)`**
    - Goes to front of file (location **`0`**) because **`ios::beg`** is default
  - **`fileObject.seekg(n)`**
    - Goes to nth byte from beginning
  - **`fileObject.seekg(n, ios::cur)`**
    - Goes n bytes forward
  - **`fileObject.seekg(y, ios::end)`**
    - Goes y bytes back from end
  - **`fileObject.seekg(0, ios::cur)`**
    - Goes to last byte
  - **`seekp`** similar

© 2003 Prentice Hall, Inc. All rights reserved.

Reading Data from a Sequential-Access File

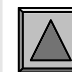
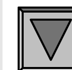
- To find pointer location
  - **`tellg`** and **`tellp`**
  - **`location = fileObject.tellg()`**
- Upcoming example
  - Credit manager program
  - List accounts with zero balance, credit, and debit

© 2003 Prentice Hall, Inc. All rights reserved.

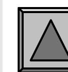
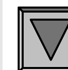

```
1   // Fig. 14.8: fig14_08.cpp
2   // Credit-inquiry program.
3   #include <iostream>
4
5   using std::cout;
6   using std::cin;
7   using std::ios;
8   using std::cerr;
9   using std::endl;
10  using std::fixed;
11  using std::showpoint;
12  using std::left;
13  using std::right;
14
15  #include <fstream>
16
17  using std::ifstream;
18
19  #include <iomanip>
20
21  using std::setw;
22  using std::setprecision;
23
24  #include <cstdlib>
25
```


© 2003 Prentice Hall, Inc.
All rights reserved.

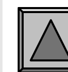
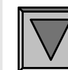

```
26 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE,
27    DEBIT_BALANCE, END };
28 int getRequest();
29 bool shouldDisplay( int, double );
30 void outputLine( int, const char * const, double );
31
32 int main()
33 {
34    // ifstream constructor opens the file
35    ifstream inClientFile( "clients.dat", ios::in );
36
37    // exit program if ifstream could not open file
38    if ( !inClientFile ) {
39       cerr << "File could not be opened" << endl;
40       exit( 1 );
41
42    } // end if
43
44    int request;
45    int account;
46    char name[ 30 ];
47    double balance;
48
49    // get user's request (e.g., zero, credit or debit balance)
50    request = getRequest();
51
```


© 2003 Prentice Hall, Inc.
All rights reserved.


```
52    // process user's request
53    while ( request != END ) {
54
55       switch ( request ) {
56
57          case ZERO_BALANCE:
58             cout << "\nAccounts with zero balances:\n";
59             break;
60
61          case CREDIT_BALANCE:
62             cout << "\nAccounts with credit balances:\n";
63             break;
64
65          case DEBIT_BALANCE:
66             cout << "\nAccounts with debit balances:\n";
67             break;
68
69       } // end switch
70
```

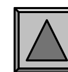
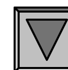
© 2003 Prentice Hall, Inc.
All rights reserved.


```
71          // read account, name and balance from file
72          inClientFile >> account >> name >> balance;
73
74          // display file contents (until eof)
75          while ( !inClientFile.eof() ) {
76
77             // display record
78             if ( shouldDisplay( request, balance ) )
79                outputLine( account, name, balance );
80
81             // read account, name and balance from file
82             inClientFile >> account >> na
83
84          } // end inner while
85
86          inClientFile.clear();    // reset eof for next input
87          inClientFile.seekg( 0 ); // move to beginning of file
88          request = getRequest();  // get additional request from user
89
90       } // end outer while
91
92       cout << "End of run." << endl;
93
94       return 0; // ifstream destructor closes the file
95
96    } // end main
```

Use **clear** to reset eof. Use **seekg** to set file position pointer to beginning of file.

© 2003 Prentice Hall, Inc.
All rights reserved.

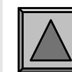
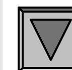



Outline

```
97
98  // obtain request from user
99  int getRequest()
100 {
101    int request;
102
103    // display request options
104    cout << "\nEnter request" << endl
105         << " 1 - List accounts with zero balances" << endl
106         << " 2 - List accounts with credit balances" << endl
107         << " 3 - List accounts with debit balances" << endl
108         << " 4 - End of run" << fixed << showpoint;
109
110    // input user request
111    do {
112       cout << "\n? ";
113       cin >> request;
114
115    } while ( request < ZERO_BALANCE && request > END );
116
117    return request;
118
119 } // end function getRequest
120
```

fig14_08.cpp
(5 of 6)

© 2003 Prentice Hall, Inc.
All rights reserved.


```
121 // determine whether to display given record
122 bool shouldDisplay( int type, double balance )
123 {
124    // determine whether to display credit balances
125    if ( type == CREDIT_BALANCE && balance < 0 )
126       return true;
127
128    // determine whether to display debit balances
129    if ( type == DEBIT_BALANCE && balance > 0 )
130       return true;
131
132    // determine whether to display zero balances
133    if ( type == ZERO_BALANCE && balance == 0 )
134       return true;
135
136    return false;
137
138 } // end function shouldDisplay
139
140 // display single record from file
141 void outputLine( int account, const char * const name,
142    double balance )
143 {
144    cout << left << setw( 10 ) << account << setw( 13 ) << name
145         << setw( 7 ) << setprecision( 2 ) << right << balance
146         << endl;
147
148 } // end function outputLine
```

fig14_08.cpp
(6 of 6)

© 2003 Prentice Hall, Inc.
All rights reserved.

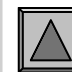
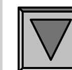
```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 2
Accounts with credit balances:
400       Stone         -42.16
```

fig14_08.cpp
output (1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.


```
Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 3

Accounts with debit balances:
100       Jones           24.98
200       Doe            345.67
500       Rich           224.62

Enter request
 1 - List accounts with zero balances
 2 - List accounts with credit balances
 3 - List accounts with debit balances
 4 - End of run
? 4
End of run.
```

fig14_08.cpp
output (2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Updating Sequential-Access Files

- Updating sequential files
  - Risk overwriting other data
  - Example: change name "White" to "Worthington"
    - Old data

```
300 White 0.00 400 Jones 32.87
```

    - Insert new data

```
300 Worthington 0.00

300 White 0.00 400 Jones 32.87

300 Worthington 0.00ones 32.87
```

Data gets overwritten

  - Formatted text different from internal representation
  - Problem can be avoided, but awkward

© 2003 Prentice Hall, Inc. All rights reserved.

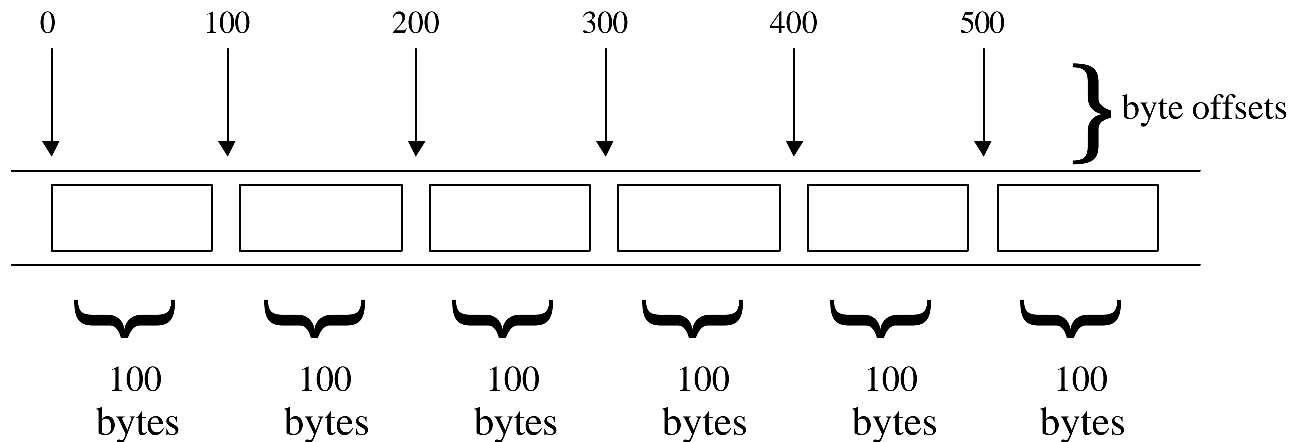
Random-Access Files

- Instant access
  - Want to locate record quickly
    - Airline reservations, ATMs
  - Sequential files must search through each one
- Random-access files are solution
  - Instant access
  - Insert record without destroying other data
  - Update/delete items without changing other data

© 2003 Prentice Hall, Inc. All rights reserved.

Random-Access Files

- C++ imposes no structure on files
  - Programmer must create random-access files
  - Simplest way: fixed-length records
    - Calculate position in file from record size and key



© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Random-Access File

- **"1234567"** (**char ***) vs **1234567** (**int**)
  - **char *** takes 8 bytes (1 for each character + null)
  - **int** takes fixed number of bytes (perhaps 4)
    - 123 same size in bytes as 1234567
- **<<** operator and **write()**
  - **outFile << number**
    - Outputs **number** (**int**) as a **char ***
    - Variable number of bytes
  - **outFile.write(** ***const char *, size*** **);**
    - Outputs raw bytes
    - Takes pointer to memory location, number of bytes to write
      - Copies data directly from memory into file
      - Does not convert to **char ***

© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Random-Access File

- Example

```
outFile.write( reinterpret_cast<const char *>(&number),
   sizeof( number ) );
```

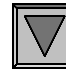
  - **`&number`** is an **`int *`**
    - Convert to **`const char *`** with **`reinterpret_cast`**
  - **`sizeof(number)`**
    - Size of **`number`** (an **`int`**) in bytes
  - **`read`** function similar (more later)
  - Must use **`write`**/**`read`** between compatible machines
    - Only when using raw, unformatted data
  - Use **`ios::binary`** for raw writes/reads

© 2003 Prentice Hall, Inc. All rights reserved.

Creating a Random-Access File

- Usually write entire **`struct`** or object to file
- Problem statement
  - Credit processing program
  - Store at most 100 fixed-length records
  - Record
    - Account number (key)
    - First and last name
    - Balance
  - Account operations
    - Update, create new, delete, list all accounts in a file
- Next: program to create blank 100-record file

© 2003 Prentice Hall, Inc. All rights reserved.




```
1  // Fig. 14.10: clientData.h
2  // Class ClientData definition used in Fig. 14.12-Fig. 14.15.
3  #ifndef CLIENTDATA_H
4  #define CLIENTDATA_H
5
6  #include <iostream>
7
8  using std::string;
9
10 class ClientData {
11
12 public:
13
14    // default ClientData constructor
15    ClientData( int = 0, string = "", string = "", double = 0.0 );
16
17    // accessor functions for accountNumber
18    void setAccountNumber( int );
19    int getAccountNumber() const;
20
21    // accessor functions for lastName
22    void setLastName( string );
23    string getLastName() const;
24
```

Class **ClientData** stores the information for each person. 100 blank **ClientData** objects will be written to a file.

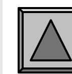
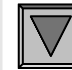
clientData.h
(1 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
25    // accessor functions for firstName
26    void setFirstName( string );
27    string getFirstName() const;
28
29    // accessor functions for balance
30    void setBalance( double );
31    double getBalance() const;
32
33 private:
34    int accountNumber;
35    char lastName[ 15 ];
36    char firstName[ 10 ];
37    double balance;
38
39 }; // end class ClientData
40
41 #endif
```

Put limits on the size of the first and last name. **accountNumber** (an **int**) and **balance** (**double**) are already of a fixed size.

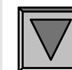
© 2003 Prentice Hall, Inc.
All rights reserved.


```
1   // Fig. 14.11: ClientData.cpp
2   // Class ClientData stores customer's credit information.
3   #include <iostream>
4
5   using std::string;
6
7   #include <cstring>
8   #include "clientData.h"
9
10  // default ClientData constructor
11  ClientData::ClientData( int accountNumberValue,
12     string lastNameValue, string firstNameValue,
13     double balanceValue )
14  {
15     setAccountNumber( accountNumberValue );
16     setLastName( lastNameValue );
17     setFirstName( firstNameValue );
18     setBalance( balanceValue );
19
20  } // end ClientData constructor
21
22  // get account-number value
23  int ClientData::getAccountNumber() const
24  {
25     return accountNumber;
26
27  } // end function getAccountNumber
```

ClientData.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.



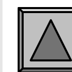
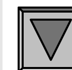

```
28
29  // set account-number value
30  void ClientData::setAccountNumber( int accountNumberValue )
31  {
32     accountNumber = accountNumberValue;
33
34  } // end function setAccountNumber
35
36  // get last-name value
37  string ClientData::getLastName() const
38  {
39     return lastName;
40
41  } // end function getLastName
42
43  // set last-name value
44  void ClientData::setLastName( string lastNameString )
45  {
46     // copy at most 15 characters from string to lastName
47     const char *lastNameValue = lastNameString.data();
48     int length = strlen( lastNameValue );
49     length = ( length < 15 ? length : 14 );
50     strncpy( lastName, lastNameValue, length );
51
52     // append null character to lastName
53     lastName[ length ] = '\0';
```

© 2003 Prentice Hall, Inc.
All rights reserved.

ClientData.cpp
(3 of 4)

```
54
55 } // end function setLastName
56
57 // get first-name value
58 string ClientData::getFirstName() const
59 {
60    return firstName;
61
62 } // end function getFirstName
63
64 // set first-name value
65 void ClientData::setFirstName( string firstNameString )
66 {
67    // copy at most 10 characters from string to firstName
68    const char *firstNameValue = firstNameString.data();
69    int length = strlen( firstNameValue );
70    length = ( length < 10 ? length : 9 );
71    strncpy( firstName, firstNameValue, length );
72
73    // append new-line character to firstName
74    firstName[ length ] = '\0';
75
76 } // end function setFirstName
77
```

© 2003 Prentice Hall, Inc.
All rights reserved.


```
78 // get balance value
79 double ClientData::getBalance() const
80 {
81    return balance;
82
83 } // end function getBalance
84
85 // set balance value
86 void ClientData::setBalance( double balanceValue )
87 {
88    balance = balanceValue;
89
90 } // end function setBalance
```

ClientData.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

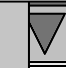
Outline

fig14_12.cpp
(1 of 2)

```
1   // Fig. 14.12: fig14_12.cpp
2   // Creating a randomly accessed file.
3   #include <iostream>
4
5   using std::cerr;
6   using std::endl;
7   using std::ios;
8
9   #include <fstream>
10
11  using std::ofstream;
12
13  #include <cstdlib>
14  #include "clientData.h"  // ClientData class definition
15
16  int main()
17  {
18     ofstream outCredit( "credit.dat", ios::binary );
19
20     // exit program if ofstream could not open file
21     if ( !outCredit ) {
22        cerr << "File could not be opened." << endl;
23        exit( 1 );
24
25     } // end if
```

Open a file for raw writing using an **ofstream** object and **ios::binary**.

© 2003 Prentice Hall, Inc.
All rights reserved.




```
26
27    // create ClientData with no information
28    ClientData blankClient;
29
30    // output 100 blank records to file
31    for ( int i = 0; i < 100; i++ )
32       outCredit.write(
33          reinterpret_cast< const char * >( &blankClient ),
34          sizeof( ClientData ) );
35
36    return 0;
37
38 } // end main
```

Create a blank object. Use **write** to output the raw data to a file (passing a pointer to the object and its size).

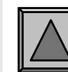
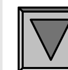
fig14_12.cpp
(2 of 2)

© 2003 Prentice Hall, Inc.
All rights reserved.

Writing Data Randomly to a Random-Access File

- Use **`seekp`** to write to exact location in file
  - Where does the first record begin?
    - Byte 0
  - The second record?
    - Byte 0 + sizeof(object)
  - Any record?
    - (Recordnum - 1) * sizeof(object)

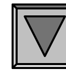
© 2003 Prentice Hall, Inc. All rights reserved.


```
1   // Fig. 14.13: fig14_13.cpp
2   // Writing to a random access file.
3   #include <iostream>
4
5   using std::cerr;
6   using std::endl;
7   using std::cout;
8   using std::cin;
9   using std::ios;
10
11  #include <iomanip>
12
13  using std::setw;
14
15  #include <fstream>
16
17  using std::ofstream;
18
19  #include <cstdlib>
20  #include "clientData.h"  // ClientData class definition
21
```

fig14_13.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.




```
22 int main()
23 {
24    int accountNumber;
25    char lastName[ 15 ];
26    char firstName[ 10 ];
27    double balance;
28
29    ofstream outCredit( "credit.dat", ios::binary );
30
31    // exit program if ofstream cannot open file
32    if ( !outCredit ) {
33       cerr << "File could not be opened." << endl;
34       exit( 1 );
35
36    } // end if
37
38    cout << "Enter account number "
39         << "(1 to 100, 0 to end input)\n? ";
40
41    // require user to specify account number
42    ClientData client;
43    cin >> accountNumber;
44    client.setAccountNumber( accountNumber );
45
```

Open file for raw (binary) writing.

Get account number, put into object. It has not yet been written to file.

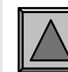
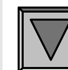
© 2003 Prentice Hall, Inc.
All rights reserved.

```
46     // user enters information, which is copied into file
47     while ( client.getAccountNumber() > 0 &&
48        client.getAccountNumber() <= 100 ) {
49
50        // user enters last name, first name and balance
51        cout << "Enter lastname, firstname, balance\n? ";
52        cin >> setw( 15 ) >> lastName;
53        cin >> setw( 10 ) >> firstName;
54        cin >> balance;
55
56        // set record lastName, firstNam
57        client.setLastName( lastName );
58        client.setFirstName( firstName )
59        client.setBalance( balance );
60
61        // seek position in file of user-specified record
62        outCredit.seekp( ( client.getAcco
63           sizeof( ClientData ) );
64
65        // write user-specified information in file
66        outCredit.write(
67           reinterpret_cast< const char * >( &client ),
68           sizeof( ClientData ) );
69
```

Position **outCredit** to the proper location in the file (based on the account number).

Write **ClientData** object to file at specified position.

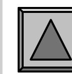
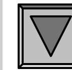
© 2003 Prentice Hall, Inc.
All rights reserved.


```
70        // enable user to specify another account number
71        cout << "Enter account number\n? ";
72        cin >> accountNumber;
73        client.setAccountNumber( accountNumber );
74
75     } // end while
76
77     return 0;
78
79  } // end main
```

fig14_13.cpp
(4 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.


```
Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Notice that accounts can be created in any order.

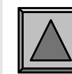
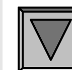
fig14_13.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Reading Data Sequentially from a Random-Access File

- **`read`** - similar to **`write`**
  - Reads raw bytes from file into memory
  - ```
    inFile.read( reinterpret_cast<char *>( &number ),
                                     sizeof( int ) );
    ```
    - **`&number`**: location to store data
    - **`sizeof(int)`**: how many bytes to read
  - Do not use **`inFile >> number`** with raw bytes
    - **`>>`** expects **`char *`**
- Upcoming program
  - Output data from a random-access file
  - Go through each record sequentially
    - If no data (accountNumber == 0) then skip

© 2003 Prentice Hall, Inc. All rights reserved.
    ```


```
1   // Fig. 14.14: fig14_14.cpp
2   // Reading a random access file.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7   using std::ios;
8   using std::cerr;
9   using std::left;
10  using std::right;
11  using std::fixed;
12  using std::showpoint;
13
14  #include <iomanip>
15
16  using std::setprecision;
17  using std::setw;
18
19  #include <fstream>
20
21  using std::ifstream;
22  using std::ostream;
23
24  #include <cstdlib>  // exit protoyype
25  #include "clientData.h"  // ClientData class definition
26
```

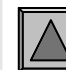

© 2003 Prentice Hall, Inc.
All rights reserved.

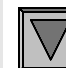
fig14_14.cpp
(2 of 3)

```
27 void outputLine( ostream&, const ClientData & );
28
29 int main()
30 {
31    ifstream inCredit( "credit.dat", ios::in );
32
33    // exit program if ifstream cannot open file
34    if ( !inCredit ) {
35       cerr << "File could not be opened." << endl;
36       exit( 1 );
37
38    } // end if
39
40    cout << left << setw( 10
41         << "Last Name" << se
42         << setw( 10 ) << rig
43
44    ClientData client; // create record
45
46    // read first record from file
47    inCredit.read( reinterpret_cast< char * >( &client ),
48       sizeof( ClientData ) );
49
```

Read **sizeof(ClientData)** bytes and put into object **client**. This may be an empty record.

© 2003 Prentice Hall, Inc.
All rights reserved.





Outline

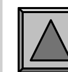
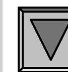
fig14_14.cpp
(3 of 3)

```
50     // read all records from file
51     while ( inCredit && !inCredit.eof() ) {
52
53        // display record
54        if ( client.getAccountNumber() != 0 )
55           outputLine( cout, client );
56
57        // read next from file
58        inCredit.read( reinterpret_cast< char * >( &client ),
59           sizeof( ClientData ) );
60
61     } // end while
62
63     return 0;
64
65  } // end main
66
67  // display single record
68  void outputLine( ostream &output, const ClientData &record )
69  {
70     output << left << setw( 10 ) << record.getAccountNumber()
71            << setw( 16 ) << record.getLastName().data()
72            << setw( 11 ) << record.getFirstName().data()
73            << setw( 10 ) << setprecision( 2 ) << right << fixed
74            << showpoint << record.getBalance() << endl;
75
76  } // end outputLine
```

Loop exits if there is an error reading (**inCredit == 0**) or EOF is found (**inCredit.eof() == 1**)

Output non-empty accounts. Note that **outputLine** takes an **ostream** argument. We could easily output to another file (opened with an **ofstream** object, which derives from **ostream**).

© 2003 Prentice Hall, Inc.
All rights reserved.


```
Account   Last Name        First Name     Balance
29        Brown            Nancy           -24.54
33        Dunn             Stacey          314.33
37        Barker           Doug              0.00
88        Smith            Dave            258.34
96        Stone            Sam              34.98
```

fig14_14.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

Example: A Transaction-Processing Program

- Instant access for bank accounts
  - Use random access file (data in **client.dat**)
- Give user menu
  - Option 1: store accounts to **print.txt**

```
Account    Last Name        First Name    Balance
29         Brown            Nancy          -24.54
33         Dunn             Stacey         314.33
37         Barker           Doug             0.00
88         Smith            Dave           258.34
96         Stone            Sam             34.98
```

  - Option 2: update record

```
Enter account to update (1 - 100): 37
37         Barker           Doug             0.00

Enter charge (+) or payment (-): +87.99
37         Barker           Doug            87.99
```

© 2003 Prentice Hall, Inc. All rights reserved.

Example: A Transaction-Processing Program

- Menu options (continued)
  - Option 3: add new record

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

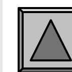
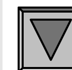
  - Option 4: delete record

```
Enter account to delete (1 - 100): 29
Account #29 deleted.
```

- To open file for reading and writing
  - Use **`fstream`** object
  - "Or" file-open modes together

```
fstream inOutCredit( "credit.dat", ios::in | ios::out );
```

© 2003 Prentice Hall, Inc. All rights reserved.

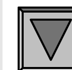

```
1   // Fig. 14.15: fig14_15.cpp
2   // This program reads a random access file sequentially, updates
3   // data previously written to the file, creates data to be placed
4   // in the file, and deletes data previously in the file.
5   #include <iostream>
6
7   using std::cout;
8   using std::cerr;
9   using std::cin;
10  using std::endl;
11  using std::ios;
12  using std::left;
13  using std::right;
14  using std::fixed;
15  using std::showpoint;
16
17  #include <fstream>
18
19  using std::ofstream;
20  using std::ostream;
21  using std::fstream;
22
23  #include <iomanip>
24
25  using std::setw;
26  using std::setprecision;
27
28  #include <cstdlib>       // exit prototype
29  #include "clientData.h"  // ClientData class definition
```


© 2003 Prentice Hall, Inc.
All rights reserved.

```
30
31 int enterChoice();
32 void printRecord( fstream& );
33 void updateRecord( fstream& );
34 void newRecord( fstream& );
35 void deleteRecord( fstream& );
36 void outputLine( ostream&, const ClientData & );
37 int getAccount( const char * const );
38
39 enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
40
41 int main()
42 {
43    // open file for reading and writing
44    fstream inOutCredit( "credit.dat", ios::in | ios::out );
45
46    // exit program if fstream cannot open file
47    if ( !inOutCredit ) {
48       cerr << "File could not be opened." << endl;
49       exit ( 1 );
50
51    } // end if
52
```

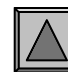
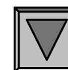
Open file for reading and writing (**fstream** object needed).

© 2003 Prentice Hall, Inc.
All rights reserved.




```
53    int choice;
54
55    // enable user to specify action
56    while ( ( choice = enterChoice() ) != END ) {
57
58       switch ( choice ) {
59
60          // create text file from record file
61          case PRINT:
62             printRecord( inOutCredit );
63             break;
64
65          // update record
66          case UPDATE:
67             updateRecord( inOutCredit );
68             break;
69
70          // create record
71          case NEW:
72             newRecord( inOutCredit );
73             break;
74
75          // delete existing record
76          case DELETE:
77             deleteRecord( inOutCredit );
78             break;
79
```

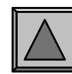
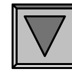

© 2003 Prentice Hall, Inc.
All rights reserved.


```
53      int choice;
54
55      // enable user to specify action
56      while ( ( choice = enterChoice() ) != END ) {
57
58         switch ( choice ) {
59
60            // create text file from record file
61            case PRINT:
62               printRecord( inOutCredit );
63               break;
64
65            // update record
66            case UPDATE:
67               updateRecord( inOutCredit );
68               break;
69
70            // create record
71            case NEW:
72               newRecord( inOutCredit );
73               break;
74
75            // delete existing record
76            case DELETE:
77               deleteRecord( inOutCredit );
78               break;
79
```

Displays menu and returns user's choice.

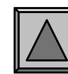
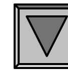
© 2003 Prentice Hall, Inc.
All rights reserved.


```
80          // display error if user does not select valid choice
81          default:
82             cerr << "Incorrect choice" << endl;
83             break;
84
85       } // end switch
86
87       inOutCredit.clear(); // reset end-of-file indicator
88
89    } // end while
90
91    return 0;
92
93 } // end main
94
95 // enable user to input menu choice
96 int enterChoice()
97 {
98    // display available options
99    cout << "\nEnter your choice" << endl
100        << "1 - store a formatted text file of accounts" << endl
101        << "    called \"print.txt\" for printing" << endl
102        << "2 - update an account" << endl
103        << "3 - add a new account" << endl
104        << "4 - delete an account" << endl
105        << "5 - end program\n? ";
```

fig14_15.cpp
(5 of 14)

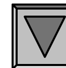
© 2003 Prentice Hall, Inc.
All rights reserved.


```
106
107    int menuChoice;
108    cin >> menuChoice; // receive choice from user
109
110    return menuChoice;
111
112 } // end function enterChoice
113
114 // create formatted text file for printing
115 void printRecord( fstream &readFromFile )
116 {
117    // create text file
118    ofstream outPrintFile( "print.txt", ios::out );
119
120    // exit program if ofstream cannot create file
121    if ( !outPrintFile ) {
122       cerr << "File could not be created." << endl;
123       exit( 1 );
124
125    } // end if
126
127    outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
128        << "Last Name" << setw( 11 ) << "First Name" << right
129        << setw( 10 ) << "Balance" << endl;
130
```

Output to `print.txt`. First, print the header for the table.

© 2003 Prentice Hall, Inc.
All rights reserved.




```
131    // set file-position pointer to beginning of record file
132    readFromFile.seekg( 0 );
133
134    // read first record from record file
135    ClientData client;
136    readFromFile.read( reinterpret_cast< char * >( &client ),
137       sizeof( ClientData ) );
138
139    // copy all records from record file into text file
140    while ( !readFromFile.eof() ) {
141
142       // write single record to text file
143       if ( client.getAccountNumber() != 0 )
144          outputLine( outPrintFile, client );
145
146       // read next record from record file
147       readFromFile.read( reinterpret_cast< char * >( &client ),
148          sizeof( ClientData ) );
149
150    } // end while
151
152 } // end function printRecord
153
```

Go to front of file, read account data, and print record if not empty.

Note that **`outputLine`** takes an **`ostream`** object (base of **`ofstream`**). It can easily print to a file (as in this case) or **`cout`**.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
154 // update balance in record
155 void updateRecord( fstream &updateFile )
156 {
157    // obtain number of account to update
158    int accountNumber = getAccount( "Enter account to update" );
159
160    // move file-position pointer to correct record in file
161    updateFile.seekg(
162       ( accountNumber - 1 ) * sizeo
163
164    // read first record from file
165    ClientData client;
166    updateFile.read( reinterpret_cast< char * >( &client ),
167       sizeof( ClientData ) );
168
169    // update record
170    if ( client.getAccountNumber() != 0 ) {
171       outputLine( cout, client );
172
173       // request user to specify transaction
174       cout << "\nEnter charge (+) or payment (-): ";
175       double transaction; // charge or payment
176       cin >> transaction;
177
178       // update record balance
179       double oldBalance = client.getBalance();
180       client.setBalance( oldBalance + transaction );
181       outputLine( cout, client );
182
```

This is `fstream` (I/O) because we must read the old balance, update it, and write the new balance.

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

fig14_15.cpp
(9 of 14)

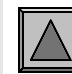
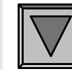
```
183       // move file-position pointer to correct record in file
184       updateFile.seekp(
185          ( accountNumber - 1 ) * sizeof( ClientData ) );
186
187       // write updated record over old record in file
188       updateFile.write(
189          reinterpret_cast< const char * >( &client ),
190          sizeof( ClientData ) );
191
192    } // end if
193
194    // display error if account does not exist
195    else
196       cerr << "Account #" << accountNumb
197          << " has no information." << en
198
199 } // end function updateRecord
200
201 // create and insert record
202 void newRecord( fstream &insertInFile )
203 {
204    // obtain number of account to create
205    int accountNumber = getAccount( "Enter new account number" );
206
207    // move file-position pointer to correct record in file
208    insertInFile.seekg(
209       ( accountNumber - 1 ) * sizeof( ClientData ) );
```

This is `fstream` because we read to see if a non-empty record already exists. If not, we write a new record.

© 2003 Prentice Hall, Inc.
All rights reserved.

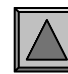
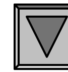
```
210
211    // read record from file
212    ClientData client;
213    insertInFile.read( reinterpret_cast< char * >( &client ),
214       sizeof( ClientData ) );
215
216    // create record, if record does not previously exist
217    if ( client.getAccountNumber() == 0 ) {
218
219       char lastName[ 15 ];
220       char firstName[ 10 ];
221       double balance;
222
223       // user enters last name, first name and balance
224       cout << "Enter lastname, firstname, balance\n? ";
225       cin >> setw( 15 ) >> lastName;
226       cin >> setw( 10 ) >> firstName;
227       cin >> balance;
228
229       // use values to populate account values
230       client.setLastName( lastName );
231       client.setFirstName( firstName );
232       client.setBalance( balance );
233       client.setAccountNumber( accountNumber );
234
```


© 2003 Prentice Hall, Inc.
All rights reserved.


```
235          // move file-position pointer to correct record in file
236          insertInFile.seekp( ( accountNumber - 1 ) *
237             sizeof( ClientData ) );
238
239          // insert record in file
240          insertInFile.write(
241             reinterpret_cast< const char * >( &client ),
242             sizeof( ClientData ) );
243
244       } // end if
245
246       // display error if account previously exists
247       else
248          cerr << "Account #" << accountNumber
249               << " already contains information." << endl;
250
251    } // end function newRecord
252
```

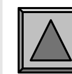
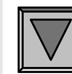

© 2003 Prentice Hall, Inc.
All rights reserved.


```
253 // delete an existing record
254 void deleteRecord( fstream &deleteFromFile )
255 {
256    // obtain number of account to delete
257    int accountNumber = getAccount( "Enter account to delete" );
258
259    // move file-position pointer to correct record in file
260    deleteFromFile.seekg(
261       ( accountNumber - 1 ) * sizeof( ClientData ) );
262
263    // read record from file
264    ClientData client;
265    deleteFromFile.read( reinterpret_cast< char * >( &client ),
266       sizeof( ClientData ) );
267
268    // delete record, if record exists in file
269    if ( client.getAccountNumber() != 0 ) {
270       ClientData blankClient;
271
272       // move file-position pointer to correct record in file
273       deleteFromFile.seekp( ( accountNumber - 1 ) *
274          sizeof( ClientData ) );
275
```

`fstream` because we read to check if the account exits. If it does, we write blank data (erase it). If it does not exist, there is no need to delete it.

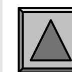
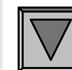
© 2003 Prentice Hall, Inc.
All rights reserved.


```
276        // replace existing record with blank record
277        deleteFromFile.write(
278           reinterpret_cast< const char * >( &blankClient ),
279           sizeof( ClientData ) );
280
281        cout << "Account #" << accountNumber << " deleted.\n";
282
283     } // end if
284
285     // display error if record does not exist
286     else
287        cerr << "Account #" << accountNumber
288
289  } // end deleteRecord
290
291  // display single record
292  void outputLine( ostream &output, const ClientData &record )
293  {
294     output << left << setw( 10 ) << record.getAccountNumber()
295            << setw( 16 ) << record.getLastName().data()
296            << setw( 11 ) << record.getFirstName().data()
297            << setw( 10 ) << setprecision( 2 ) << right << fixed
298            << showpoint << record.getBalance() << endl;
299
300  } // end function outputLine
301
```

outputLine is very flexible, and can output to any **ostream** object (such as a file or **cout**).

© 2003 Prentice Hall, Inc.
All rights reserved.


```
302 // obtain account-number value from user
303 int getAccount( const char * const prompt )
304 {
305    int accountNumber;
306
307    // obtain account-number value
308    do {
309       cout << prompt << " (1 - 100): ";
310       cin >> accountNumber;
311
312    } while ( accountNumber < 1 || accountNumber > 100 );
313
314    return accountNumber;
315
316 } // end function getAccount
```


© 2003 Prentice Hall, Inc.
All rights reserved.

Input/Output of Objects

- I/O of objects
  - Chapter 8 (overloaded **`>>`**)
  - Only object's data transmitted
    - Member functions available internally
  - When objects stored in file, lose type info (class, etc.)
    - Program must know type of object when reading
  - One solution
    - When writing, output object type code before real object
    - When reading, read type code
      - Call proper overloaded function (**`switch`**)

© 2003 Prentice Hall, Inc. All rights reserved.