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
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
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
Exception-Handling Overview

- C++ code
  ```cpp
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
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
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
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
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
// Fig. 13.1: fig13_01.cpp
// A simple exception-handling example that checks for
// divide-by-zero exceptions.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <exception>

using std::exception;

// DivideByZeroException objects should be thrown by functions
// upon detecting division-by-zero exceptions
class DivideByZeroException : public runtime_error {
 public:
  // constructor specifies default error message
  DivideByZeroException::DivideByZeroException()
    : exception( "attempted to divide by zero" ) {}
};  // end class DivideByZeroException
// perform division and throw DivideByZeroException object if
divide-by-zero exception occurs

double quotient( int numerator, int denominator )
{
    // throw DivideByZeroException if trying to divide by zero
    if ( denominator == 0 )
        throw DivideByZeroException(); // terminate function

    // return division result
    return static_cast< double >( numerator ) / denominator;
}

int main()
{
    int number1; // user-specified numerator
    int number2; // user-specified denominator
    double result; // result of division

    cout << "Enter two integers (end-of-file to end): ";

// enable user to enter two integers to divide
while ( cin >> number1 >> number2 ) {

    // try block contains code that might throw exception
    // and code that should not execute if an exception occurs
    try {
        result = quotient( number1, number2 );
        cout << "The quotient is: " << result << endl;
    }

} // end try

// exception handler handles a divide-by-zero exception
catch ( DivideByZeroException &divideByZeroException ) {
    cout << "Exception occurred: "
         << divideByZeroException.what() << endl;
}

} // end while

return 0; // terminate normally

} // 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.
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
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
// Fig. 13.2: fig13_02.cpp
// Demonstrating exception rethrowing.
#include <iostream>

using std::cout;
using std::endl;

#include <exception>

using std::exception;

// throw, catch and rethrow exception
void throwException() {
    // throw exception and catch it immediately
    try {
        cout << "  Function throwException throws an exception\n";
        throw exception(); // generate exception
    }
    // end try

    // handle exception
    catch ( exception &caughtException ) {
        cout << "  Exception handled in function throwException"
             << "\n Function throwException rethrows exception";
        throw; // rethrow exception for further processing
    }
    // end catch
} // end throwException
```cpp
30    cout << "This also should not print\n";
31
32 } // end function throwException
33
34 int main()
35 {
36    // throw exception
37    try {
38        cout << "\nmain invokes function throwException\n";
39        throwException();
40        cout << "This should not print\n"
41    } // end try
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.
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
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
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`
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
// Fig. 13.3: fig13_03.cpp
// Demonstrating stack unwinding.
#include <iostream>

using std::cout;
using std::endl;

#include <stdexcept>

using std::runtime_error;

// function3 throws run-time error
void function3() throw ( runtime_error )
{
    throw runtime_error( "runtime_error in function3" ); // fourth
}

// function2 invokes function3
void function2() throw ( runtime_error )
{
    function3(); // third
}
// function1 invokes function2
void function1() throw ( runtime_error )
{
    function2(); // second
}

// demonstrate stack unwinding
int main()
{
    // invoke function1
    try {
        function1(); // first
    } // end try

    // handle run-time error
    catch ( runtime_error &error ) // fifth
    {
        cout << "Exception occurred: " << error.what() << endl;
    } // end catch

    return 0;
} // end main

Exception occurred: runtime_error in function3
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
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
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
// Fig. 13.4: fig13_04.cpp
// Demonstrating pre-standard new returning 0 when memory
// is not allocated.
#include <iostream>

using std::cout;

int main()
{
    double *ptr[50];

    // allocate memory for ptr
    for (int i = 0; i < 50; i++) {
        ptr[i] = new double[500000];

        // new returns 0 on failure to allocate
        if (ptr[i] == 0) {
            cout << "Memory allocation failed for ptr[ "
                 << i << " ]\n";
            break;
        }
    }
}

Demonstrating new that returns 0 on allocation failure.
// successful memory allocation
else
    cout << "Allocated 5000000 doubles in ptr[ "
    << i << " ]\n";
}

return 0;

} // 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 ]
// Fig. 13.5: fig13_05.cpp
// Demonstrating standard new throwing bad_alloc when memory
// cannot be allocated.
#include <iostream>

using std::cout;
using std::endl;

#include <new> // standard operator new

using std::bad_alloc;

int main()
{
    double *ptr[50];
    // attempt to allocate memory
    try {
        // allocate memory for ptr[ i ]; new throws bad_alloc
        // on failure
        for (int i = 0; i < 50; i++) {
            ptr[i] = new double[5000000];
            cout << "Allocated 5000000 doubles in ptr[ " << i << " ]\n";
        }
    }
    // end try

    // Demonstrating new that throws an exception.
}
// handle bad_alloc exception

try {
    cout << "Exception occurred: " << memoryAllocationException.what() << endl;
}

return 0;

}  // 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
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
// Fig. 13.6: fig13_06.cpp
// Demonstrating set_new_handler.
#include <iostream>

using std::cout;
using std::cerr;

#include <new>     // standard operator new and set_new_handler
using std::set_new_handler;

#include <cstdlib> // abort function prototype

void customNewHandler()
{
    cerr << "customNewHandler was called";
    abort();
}

// using set_new_handler to handle failed memory allocation
int main()
{
    double *ptr[ 50 ];

    The custom handler must take no arguments and return void.
// specify that customNewHandler should be called on failed memory allocation
set_new_handler( customNewHandler );

// allocate memory for ptr[ i ]; customNewHandler will be called on failed memory allocation
for ( int i = 0; i < 50; i++ ) {
    ptr[ i ] = new double[ 5000000 ];
    cout << "Allocated 5000000 doubles in ptr[ "
        << i << " ]\n";
}

return 0;

} // 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 ]
customNewHandler was called
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
    ```cpp
    auto_ptr< MyClass > newPointer( new MyClass() );
    ```
  - `newPointer` points to dynamically allocated object
// Fig. 13.7: fig13_07.cpp
// Demonstrating auto_ptr.
#include <iostream>

using std::cout;
using std::endl;

#include <memory>
using std::auto_ptr; // auto_ptr class definition

class Integer {

public:

    // Integer constructor
    Integer( int i = 0 )
    : value( i )
    {
        cout << "Constructor for Integer " << value << endl;
    }

}; // end Integer constructor
// Integer destructor
~Integer()
{
    cout << "Destructor for Integer " << value << endl;
}

// function to set Integer
void setInteger( int i )
{
    value = i;
}

// function to return Integer
int getInteger() const
{
    return value;
}

private:
    int value;
}; // end class Integer
```cpp
// use auto_ptr to manipulate Integer object
int main()
{
    cout << "Creating an auto_ptr object that\n        " << "points to an Integer\n";

    // "aim" auto_ptr at Integer object
    auto_ptr< Integer > ptrToInteger( new Integer( 7 ) );

    cout << "\nUsing the auto_ptr to manipulate the Integer\n";

    // use auto_ptr to set Integer value
    ptrToInteger->setInteger( 99 );

    // use auto_ptr to get Integer value
    cout << "Integer after setInteger: "
        << ( *ptrToInteger ).getInteger()
        << "\n\nTerminating program" << endl;

    return 0;
}  // 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.
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
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
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
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)
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…
The Data Hierarchy

File

<table>
<thead>
<tr>
<th>Sally</th>
<th>Black</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom</td>
<td>Blue</td>
</tr>
<tr>
<td>Judy</td>
<td>Green</td>
</tr>
<tr>
<td>Iris</td>
<td>Orange</td>
</tr>
<tr>
<td>Randy</td>
<td>Red</td>
</tr>
</tbody>
</table>

Record

| Judy | Green |

Field

| Judy |

Byte (ASCII character) J

1 Bit

01001010
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
Files and Streams

• To perform file processing
  – Include `<iostream>` and `<fstream>`
  – Class templates
    • `basic_ifstream` (input)
    • `basic_ofstream` (output)
    • `basic_fstream` (I/O)
  – typedefs for specializations that allow `char` I/O
    • `ifstream` (char input)
    • `ofstream` (char output)
    • `fstream` (char I/O)
Files and Streams

• Opening files
  – Create objects from template
  – Derive from stream classes
    • Can use stream methods from Ch. 12
    • put, get, peek, etc.
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);
    ```
Creating a Sequential-Access File

- **File-open modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ios::app</td>
<td>Write all output to the end of the file.</td>
</tr>
<tr>
<td>ios::ate</td>
<td>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.</td>
</tr>
<tr>
<td>ios::in</td>
<td>Open a file for input.</td>
</tr>
<tr>
<td>ios::out</td>
<td>Open a file for output.</td>
</tr>
<tr>
<td>ios::trunc</td>
<td>Discard the file’s contents if it exists (this is also the default action for <code>ios::out</code>)</td>
</tr>
<tr>
<td>ios::binary</td>
<td>Open a file for binary (i.e., non-text) input or output.</td>
</tr>
</tbody>
</table>

- **`ofstream`** opened for output by default
  - `ofstream outClientFile( "clients.dat", ios::out );`
  - `ofstream outClientFile( "clients.dat");`
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
Creating a Sequential-Access File

• Operations
  – Writing to file (just like cout)
    • outClientFile << myVariable
  – Closing file
    • outClientFile.close()
    • Automatically closed when destructor called
// Fig. 14.4: fig14_04.cpp
// Create a sequential file.
#include <iostream>

using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::endl;

#include <fstream>
using std::ofstream;

#include <cstdlib> // exit prototype

int main()
{
    // ofstream constructor opens file
    ofstream outClientFile( "clients.dat", ios::out );

    // exit program if unable to create file
    if ( !outClientFile ) { // overloaded ! operator
        cerr << "File could not be opened" << endl;
        exit( 1 );
    }
} // 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.
```cpp
28  cout << "Enter the account, name, and balance." << endl
   << "Enter end-of-file to end input."
29  >> account >> name >> balance;
30  int account;
31  char name[30];
32  double balance;
33  // read account, name and balance from cin, then place in file
34  while ( cin >> account >> name >> balance ) {
35      outClientFile << account << ' ' << name << ' ' << balance
36          << endl;
37      cout << '? ';
38  }
39  // end while
40  return 0; // ofstream destructor closes file
41  // 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()`.
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
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)
// Fig. 14.7: fig14_07.cpp
// Reading and printing a sequential file.
#include <iostream>

using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::endl;
using std::left;
using std::right;
using std::fixed;
using std::showpoint;

#include <fstream>

using std::ifstream;

#include <iomanip>

using std::setw;
using std::setprecision;

#include <cstdlib> // exit prototype

void outputLine( int, const char * const, double );
```cpp
int main()
{
    // ifstream constructor opens the file
    ifstream inClientFile( "clients.dat", ios::in );

    // exit program if ifstream could not open file
    if ( !inClientFile ) {
        cerr << "File could not be opened" << endl;
        exit( 1 );
    }

    int account;
    char name[ 30 ];
    double balance;

    cout << left << setw( 10 ) << "Account" << setw( 13 ) << "Name" << "Balance" << endl
         << fixed << showpoint;

    // display each record in file
    while ( inClientFile >> account >> name >> balance )
        outputLine( account, name, balance );

    return 0; // ifstream destructor closes the file
} // end main
```

Open and test file for input. Read from file until EOF found.
// display single record from file
void outputLine( int account, const char * const name,
    double balance )
{
    cout << left << setw( 10 ) << account << setw( 13 ) << name
        << setw( 7 ) << setprecision( 2 ) << right << balance
        << endl;
}

Account   Name         Balance
100       Jones          24.98
200       Doe           345.67
300       White           0.00
400       Stone         -42.16
500       Rich          224.62
Reading Data from a Sequential-Access File

• File position pointers
  – Number of next byte to read/write
  – Functions to reposition pointer
    • \texttt{seekg} (seek get for \texttt{istream} class)
    • \texttt{seekp} (seek put for \texttt{ostream} class)
    • Classes have "get" and "put" pointers
  – \texttt{seekg} and \texttt{seekp} take \textit{offset} and \textit{direction}
    • Offset: number of bytes relative to direction
    • Direction (\texttt{ios::beg} default)
      – \texttt{ios::beg} - relative to beginning of stream
      – \texttt{ios::cur} - relative to current position
      – \texttt{ios::end} - relative to end
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 n-th 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
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
// Fig. 14.8: fig14_08.cpp
// Credit-inquiry program.
#include <iostream>

using std::cout;
using std::cin;
using std::ios;
using std::cerr;
using std::endl;
using std::fixed;
using std::showpoint;
using std::left;
using std::right;

#include <fstream>

using std::ifstream;

#include <iomanip>

using std::setw;
using std::setprecision;

#include <cstdlib>
enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };  
int getRequest();  
bool shouldDisplay( int, double );  
void outputLine( int, const char * const, double );  

int main()  
{"  
    // ifstream constructor opens the file  
    ifstream inClientFile( "clients.dat", ios::in );  

    // exit program if ifstream could not open file  
    if ( !inClientFile ) {  
        cerr << "File could not be opened" << endl;  
        exit( 1 );  
    } // end if  

    int request;  
    int account;  
    char name[ 30 ];  
    double balance;  

    // get user's request (e.g., zero, credit or debit balance)  
    request = getRequest();  
"
// process user's request
while ( request != END ) {

    switch ( request ) {

        case ZERO_BALANCE:
            cout << "
Accounts with zero balances:
";
            break;

        case CREDIT_BALANCE:
            cout << "
Accounts with credit balances:
";
            break;

        case DEBIT_BALANCE:
            cout << "
Accounts with debit balances:
";
            break;

    } // end switch
} // end while
// read account, name and balance from file
inClientFile >> account >> name >> balance;

// display file contents (until eof)
while ( !inClientFile.eof() ) {
    // display record
    if ( shouldDisplay( request, balance ) )
        outputLine( account, name, balance );

    // read account, name and balance from file
    inClientFile >> account >> name >> balance;
} // end inner while

inClientFile.clear(); // reset eof for next input
inClientFile.seekg( 0 ); // move to beginning of file
request = getRequest(); // get additional request from user

} // end outer while

cout << "End of run." << endl;
return 0; // ifstream destructor closes the file

} // end main
// obtain request from user
int getRequest()
{
    int request;

    // display request options
    cout << "\nEnter request" << endl
        << " 1 - List accounts with zero balances" << endl
        << " 2 - List accounts with credit balances" << endl
        << " 3 - List accounts with debit balances" << endl
        << " 4 - End of run" << fixed << showpoint;

    // input user request
    do {
        cout << "\n? ";
        cin >> request;
    } while ( request < ZERO_BALANCE && request > END );

    return request;
} // end function getRequest
// determine whether to display given record
bool shouldDisplay( int type, double balance )
{
    // determine whether to display credit balances
    if ( type == CREDIT_BALANCE && balance < 0 )
        return true;

    // determine whether to display debit balances
    if ( type == DEBIT_BALANCE && balance > 0 )
        return true;

    // determine whether to display zero balances
    if ( type == ZERO_BALANCE && balance == 0 )
        return true;

    return false;
}

// display single record from file
void outputLine( int account, const char * const name,
                 double balance )
{
    cout << left << setw( 10 ) << account << setw( 13 ) << name
        << setw( 7 ) << setprecision( 2 ) << right << balance
        << endl;
}

© 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
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.
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
  – Formatted text different from internal representation
  – Problem can be avoided, but awkward
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
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

```
byte offsets

100 bytes 100 bytes 100 bytes 100 bytes 100 bytes 100 bytes
```
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 *
Creating a Random-Access File

• Example

```cpp
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
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
Class `ClientData` stores the information for each person. 100 blank `ClientData` objects will be written to a file.
// accessor functions for firstName
void setFirstName( string );
string getFirstName() const;

// accessor functions for balance
void setBalance( double );
double getBalance() const;

private:
int accountNumber;
char lastName[ 15 ];
char firstName[ 10 ];
double balance;

}; // end class ClientData

#endif
// Fig. 14.11: ClientData.cpp
// Class ClientData stores customer's credit information.
#include <iostream>

using std::string;

#include <cstring>
#include "clientData.h"

// default ClientData constructor
ClientData::ClientData( int accountNumberValue,
    string lastNameValue, string firstNameValue,
    double balanceValue )
{
    setAccountNumber( accountNumberValue );
    setLastName( lastNameValue );
    setFirstName( firstNameValue );
    setBalance( balanceValue );
}

// get account-number value
int ClientData::getAccountNumber() const
{
    return accountNumber;
}
// set account-number value
void ClientData::setAccountNumber( int accountNumberValue )
{
    accountNumber = accountNumberValue;
}

// get last-name value
string ClientData::getLastName() const
{
    return lastName;
}

// set last-name value
void ClientData::setLastName( string lastNameString )
{
    const char *lastNameValue = lastNameString.data();
    int length = strlen( lastNameValue );
    length = ( length < 15 ? length : 14 );
    strncpy( lastName, lastNameValue, length );
    lastName[length] = '\0';
54 */
55 } // end function setLastName
56
57 // get first-name value
58 string ClientData::firstName() 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     // append new-line character to firstName
73     firstName[ length ] = '\0';
74 }
75 } // end function setFirstName
76
77
// get balance value
double ClientData::getBalance() const
{
    return balance;
}

// set balance value
void ClientData::setBalance( double balanceValue )
{
    balance = balanceValue;
}


// Fig. 14.12: fig14_12.cpp
// Creating a randomly accessed file.
#include <iostream>

using std::cerr;
using std::endl;
using std::ios;

#include <fstream>

using std::ofstream;

#include <cstdlib>
#include "clientData.h"  // ClientData class definition

int main()
{
    ofstream outCredit( "credit.dat", ios::binary );
    // exit program if ofstream could not open file
    if ( !outCredit ) {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    }
    // end if
} // end main
// create ClientData with no information
ClientData blankClient;

// output 100 blank records to file
for (int i = 0; i < 100; i++)
    outCredit.write(
        reinterpret_cast<const char*>( &blankClient ),
        sizeof( ClientData ) );

return 0;

} // end main
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)```
// Fig. 14.13: fig14_13.cpp
// Writing to a random access file.
#include <iostream>

using std::cerr;
using std::endl;
using std::cout;
using std::cin;
using std::ios;

#include <iomanip>

using std::setw;

#include <fstream>

using std::ofstream;

#include <cstdlib>

#include "clientData.h" // ClientData class definition
```cpp
int main() {
    int accountNumber;
    char lastName[15];
    char firstName[10];
    double balance;

    ofstream outCredit( "credit.dat", ios::binary );

    // exit program if ofstream cannot open file
    if ( !outCredit ) {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    }

    cout << "Enter account number "
        << "(1 to 100, 0 to end input)\n? ";

    // require user to specify account number
    ClientData client;
    cin >> accountNumber;
    client.setAccountNumber( accountNumber );
```
// user enters information, which is copied into file
while ( client.getAccountNumber() > 0 &&
        client.getAccountNumber() <= 100 ) {

    // user enters last name, first name and balance
    cout << "Enter lastname, firstname, balance\n? ";
    cin >> setw( 15 ) >> lastName;
    cin >> setw( 10 ) >> firstName;
    cin >> balance;

    // set record lastName, firstName and balance values
    client.setLastName( lastName );
    client.setFirstName( firstName );
    client.setBalance( balance );

    // seek position in file of user-specified record
    outCredit.seekp( ( client.getAccountNumber() - 1 ) *
                     sizeof( ClientData ) );

    // write user-specified information in file
    outCredit.write( reinterpret_cast< const char * >( &client ),
                     sizeof( ClientData ) );
// enable user to specify another account number
cout << "Enter account number\n? ";
cin >> accountNumber;
client.setAccountNumber( accountNumber );

} // end while

return 0;

} // end main
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.
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
// Fig. 14.14: fig14_14.cpp
// Reading a random access file.
#include <iostream>

using std::cout;
using std::endl;
using std::ios;
using std::cerr;
using std::left;
using std::right;
using std::fixed;
using std::showpoint;

#include <iomanip>
using std::setprecision;
using std::setw;

#include <fstream>  // ifstream prototype
using std::ifstream;
using std::ostream;

#include <cstdlib>  // exit prototype
#include "clientData.h"  // ClientData class definition
void outputLine( ostream&, const ClientData& );

int main()
{
    ifstream inCredit( "credit.dat", ios::in );
    // exit program if ifstream cannot open file
    if ( !inCredit ) {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    }

    cout << left << setw( 10 ) << "Account" << setw( 16 ) << "Last Name" << setw( 11 ) << "First Name" << left << setw( 10 ) << right << "Balance" << endl;

    ClientData client; // create record

    // read first record from file
    inCredit.read( reinterpret_cast< char* >( &client ),
                   sizeof( ClientData ) );

    // Read \texttt{sizeof(ClientData)} bytes and put into object \texttt{client}. This may be an empty record.
// read all records from file
while ( inCredit && !inCredit.eof() ) {

    // display record
    if ( client.getAccountNumber() != 0 )
        outputLine( cout, client );

    // read next from file
    inCredit.read( reinterpret_cast< char * >( &client ),
                   sizeof( ClientData ) );

} // end while

return 0;

} // end main

// display single record
void outputLine( ostream &output, const ClientData &record ) {
    output << left << setw( 10 ) << record.getAccountNumber() <<
        setw( 16 ) << record.getLastName().data() <<
        setw( 11 ) << record.getFirstName().data() <<
        setw( 10 ) << setprecision( 2 ) << right << fixed <<
        showpoint << record.getBalance() << endl;

} // 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).
<table>
<thead>
<tr>
<th>Account</th>
<th>Last Name</th>
<th>First Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Brown</td>
<td>Nancy</td>
<td>-24.54</td>
</tr>
<tr>
<td>33</td>
<td>Dunn</td>
<td>Stacey</td>
<td>314.33</td>
</tr>
<tr>
<td>37</td>
<td>Barker</td>
<td>Doug</td>
<td>0.00</td>
</tr>
<tr>
<td>88</td>
<td>Smith</td>
<td>Dave</td>
<td>258.34</td>
</tr>
<tr>
<td>96</td>
<td>Stone</td>
<td>Sam</td>
<td>34.98</td>
</tr>
</tbody>
</table>
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`

<table>
<thead>
<tr>
<th>Account</th>
<th>Last Name</th>
<th>First Name</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>Brown</td>
<td>Nancy</td>
<td>-24.54</td>
</tr>
<tr>
<td>33</td>
<td>Dunn</td>
<td>Stacey</td>
<td>314.33</td>
</tr>
<tr>
<td>37</td>
<td>Barker</td>
<td>Doug</td>
<td>0.00</td>
</tr>
<tr>
<td>88</td>
<td>Smith</td>
<td>Dave</td>
<td>258.34</td>
</tr>
<tr>
<td>96</td>
<td>Stone</td>
<td>Sam</td>
<td>34.98</td>
</tr>
</tbody>
</table>

- 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
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
    ```
    ifstream inOutCredit( "credit.dat", ios::in | ios::out );
    ```
// Fig. 14.15: fig14_15.cpp
// This program reads a random access file sequentially, updates
// data previously written to the file, creates data to be placed
// in the file, and deletes data previously in the file.
#include <iostream>

using std::cout;
using std::cerr;
using std::cin;
using std::endl;
using std::ios;
using std::left;
using std::right;
using std::fixed;
using std::showpoint;

#include <fstream>
using std::ofstream;
using std::ostream;
using std::fstream;

#include <iomanip>
using std::setw;
using std::setprecision;

#include <cstdlib> // exit prototype
#include "clientData.h" // ClientData class definition
int enterChoice();
void printRecord( fstream& );
void updateRecord( fstream& );
void newRecord( fstream& );
void deleteRecord( fstream& );
void outputLine( ostream&, const ClientData & );
int getAccount( const char * const );

enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };

int main()
{
    // open file for reading and writing
    fstream inOutCredit( "credit.dat", ios::in | ios::out );

    // exit program if fstream cannot open file
    if ( !inOutCredit ) {
        cerr << "File could not be opened." << endl;
        exit ( 1 );
    }

    // end if
} // end if

Open file for reading and writing (fstream object needed).
int choice;

// enable user to specify action
while ( ( choice = enterChoice() ) != END ) {
    switch ( choice ) {
        // create text file from record file
        case PRINT:
            printRecord( inOutCredit );
            break;

        // update record
        case UPDATE:
            updateRecord( inOutCredit );
            break;

        // create record
        case NEW:
            newRecord( inOutCredit );
            break;

        // delete existing record
        case DELETE:
            deleteRecord( inOutCredit );
            break;
    }
```cpp
int choice;

// enable user to specify action
while ( ( choice = enterChoice() ) != END ) {
    switch ( choice ) {
        // create text file from record file
        case PRINT:
            printRecord( inOutCredit );
            break;

        // update record
        case UPDATE:
            updateRecord( inOutCredit );
            break;

        // create record
        case NEW:
            newRecord( inOutCredit );
            break;

        // delete existing record
        case DELETE:
            deleteRecord( inOutCredit );
            break;
    }
}
```

Displays menu and returns user's choice.
// display error if user does not select valid choice
default:
    cerr << "Incorrect choice" << endl;
    break;

} // end switch

inOutCredit.clear(); // reset end-of-file indicator

} // end while

return 0;

} // end main

// enable user to input menu choice
int enterChoice()
{
    // display available options
    cout << "Enter your choice" << endl
         << "1 - store a formatted text file of accounts" << endl
         << "   called \"print.txt\" for printing" << endl
         << "2 - update an account" << endl
         << "3 - add a new account" << endl
         << "4 - delete an account" << endl
         << "5 - end program\n? ";
Output to `print.txt`. First, print the header for the table.

```cpp
int menuChoice;
cin >> menuChoice; // receive choice from user
return menuChoice;

// create formatted text file for printing
void printRecord( fstream &readFromFile )
{
    // create text file
    ofstream outPrintFile( "print.txt", ios::out );

    // exit program if ofstream cannot create file
    if ( !outPrintFile ) {
        cerr << "File could not be created." << endl;
        exit( 1 );
    }

    outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
        << "Last Name" << setw( 11 ) << "First Name" << right
        << setw( 10 ) << "Balance" << endl;
```
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`.
// update balance in record
void updateRecord( fstream &updateFile )
{
    // obtain number of account to update
    int accountNumber = getAccount( "Enter account to update" );

    // move file-position pointer to correct record in file
    updateFile.seekg( (accountNumber - 1) * sizeof( ClientData ) );

    // read first record from file
    ClientData client;
    updateFile.read( reinterpret_cast< char * >( &client ), sizeof( ClientData ) );

    // update record
    if ( client.getAccountNumber() != 0 ) {
        outputLine( cout, client );

        // request user to specify transaction
        cout << "Enter charge (+) or payment (-): ";
        double transaction; // charge or payment
        cin >> transaction;

        // update record balance
        double oldBalance = client.getBalance();
        client.setBalance( oldBalance + transaction );
        outputLine( cout, client );
    }
}

This is fstream (I/O) because we must read the old balance, update it, and write the new balance.
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 #" << accountNumber
197        << " has no information." << endl;
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.
// read record from file
ClientData client;
insertInFile.read( reinterpret_cast< char * >( &client ),
    sizeof( ClientData ) );

// create record, if record does not previously exist
if ( client.getAccountNumber() == 0 ) {
    char lastName[ 15 ];
    char firstName[ 10 ];
    double balance;

    // user enters last name, first name and balance
    cout << "Enter lastname, firstname, balance\n? ";
    cin >> setw( 15 ) >> lastName;
    cin >> setw( 10 ) >> firstName;
    cin >> balance;

    // use values to populate account values
    client.setLastName( lastName );
    client.setFirstName( firstName );
    client.setBalance( balance );
    client.setAccountNumber( accountNumber );
// move file-position pointer to correct record in file
insertInFile.seekp((accountNumber - 1) * sizeof(ClientData));

// insert record in file
insertInFile.write(
    reinterpret_cast<const char *>(&client),
    sizeof(ClientData));

} // end if

// display error if account previously exists
else
cerr << "Account #" << accountNumber
    << " already contains information." << endl;

} // end function newRecord

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( (accountNumber - 1) * sizeof( ClientData ) );
261
262     // read record from file
263     ClientData client;
264     deleteFromFile.read( reinterpret_cast< char * >( &client ), sizeof( ClientData ) );
265
266     // delete record, if record exists in file
267     if ( client.getAccountNumber() != 0 ) {
268         ClientData blankClient;
269
270         // move file-position pointer to correct record in file
271         deleteFromFile.seekp( (accountNumber - 1) * sizeof( ClientData ) );
272         fstream because we read to check if the account exists. If it does, we write blank data (erase it). If it does not exist, there is no need to delete it.
// replace existing record with blank record
deleteFromFile.write(
    reinterpret_cast< const char * >( &blankClient ),
    sizeof( ClientData ) );

    cout << "Account #" << accountNumber << " deleted.\n";

} // end if

// display error if record does not exist
else
    cerr << "Account #" << accountNumber << " is empty.\n";

} // end deleteRecord

// display single record
void outputLine( ostream &output, const ClientData &record )
{
    output << left << setw( 10 ) << record.getAccountNumber() << setw( 16 ) << record.getLastName().data() << setw( 11 ) << record.getFirstName().data() << setw( 10 ) << setprecision( 2 ) << right << fixed << showpoint << record.getBalance() << endl;

} // end function outputLine

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