IS 0020
Program Design and Software Tools

File Processing, Standard Template Library

Lecture 12

July 19, 2004

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.

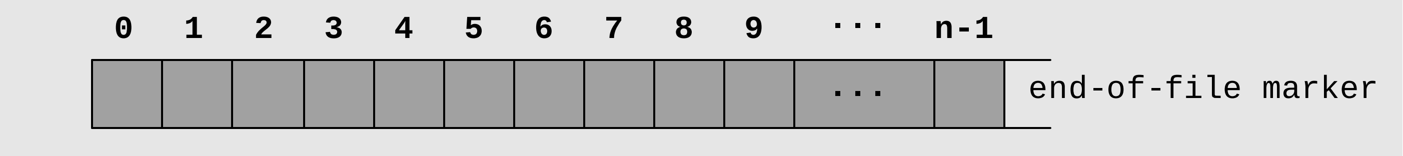
The Data Hierarchy

- From smallest to largest
  - Bit (binary digit)
    - 1 or 0
    - 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`**)
- 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.

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.

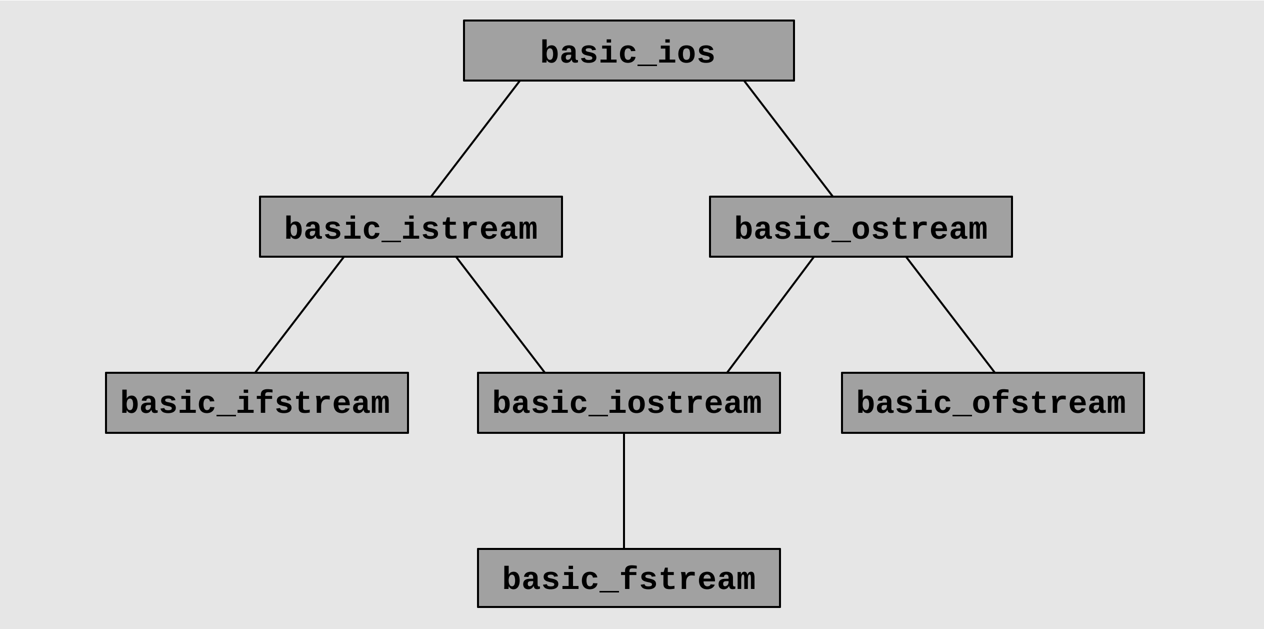
Files and Streams

- 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 : **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 **`failbit`** or **`badbit`** set, otherwise nonzero
      - **`failbit`** set when EOF found
    - **`while ( cin >> myVariable )`**
      - Implicitly converts **`cin`** to pointer
      - Loops until EOF
  - Writing to file (just like **`cout`**)
    - **`outClientFile << myVariable`**
  - Closing file
    - **`outClientFile.close()`**
    - Automatically closed when destructor called

© 2003 Prentice Hall, Inc. All rights reserved.

fig14_04.cpp
(1 of 2)

```
// 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.

© 2003 Prentice Hall, Inc.
All rights reserved.

fig14_04.cpp
(2 of 2)

```

   cout << "Enter the account, name, and balance." << endl
        << "Enter end-of-file t

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

   // read account, name and balance from cin, then place in file
   while ( cin >> account >> name >> balance ) {
      outClientFile << account << ' ' << name << ' ' << balance
                    << endl;
      cout << "? ";

   } // end while

   return 0;  // ofstream destructor closes file

} // 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.

fig14_07.cpp
(2 of 3)

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

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

   // 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.

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

fig14_07.cpp
(3 of 3)

fig14_07.cpp
output (1 of 1)

```

// 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;

} // 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
- To find pointer location
  - **`tellg`** and **`tellp`**
  - **`location = fileObject.tellg()`**

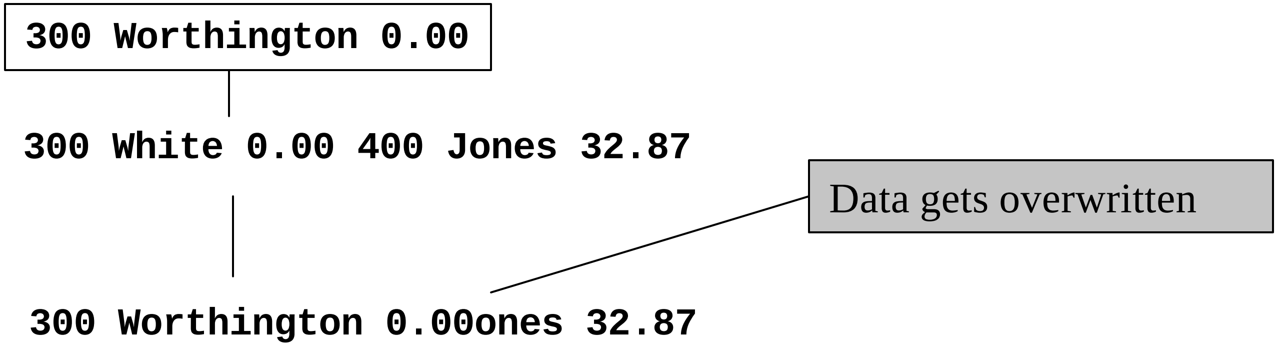
© 2003 Prentice Hall, Inc. All rights reserved.

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.

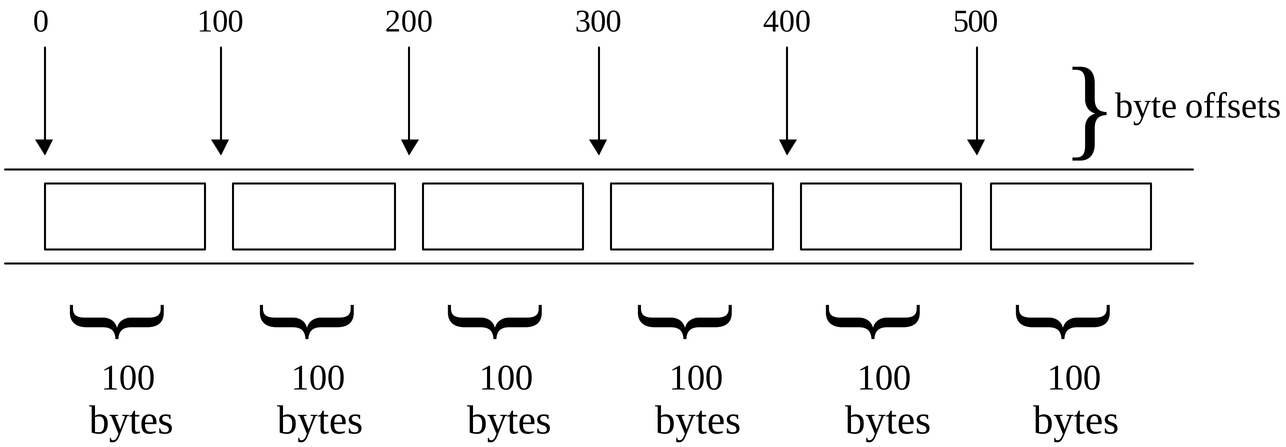
Random-Access Files

- 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
- Usually write entire **struct** or object to file

© 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)
- **`read`** - similar to **`write`**
  - Reads raw bytes from file into memory
  - `inFile.read( reinterpret_cast<char *>( &number ),`

    ```
    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 *`**

© 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.

Introduction to the Standard Template Library (STL)

- STL
  - Powerful, template-based components
    - Containers: template data structures
    - Iterators: like pointers, access elements of containers
    - Algorithms: data manipulation, searching, sorting, etc.
  - Object- oriented programming: reuse, reuse, reuse
  - Only an introduction to STL, a huge class library

© 2003 Prentice Hall, Inc. All rights reserved.

Introduction to Containers

- Three types of containers
  - Sequence containers: **`vector; deque; list`**
    - Linear data structures (vectors, linked lists)
    - First-class container
  - Associative containers: **`set; multiset; map; multimap`**
    - Non-linear, can find elements quickly
    - Key/value pairs
    - First-class container
  - Container adapters:**`stack; queue; priority_queue`**
    - Near containers
    - Similar to containers, with reduced functionality
- Containers have some common functions

© 2003 Prentice Hall, Inc. All rights reserved.

Common STL Member Functions (Fig. 21.2)

- Member functions for all containers
  - Default constructor, copy constructor, destructor
  - **`empty`**
  - **`max_size`**, **`size`**
  - **`= < <= > >= == !=`**
  - **`swap`**
- Functions for first-class containers
  - **`begin`**, **`end`**
  - **`rbegin`**, **`rend`**
  - **`erase`**, **`clear`**

© 2003 Prentice Hall, Inc. All rights reserved.

Common STL typedefs (Fig. 21.4)

- **`typedef`**s for first-class containers
  - **`value_type`**
  - **`reference`**
  - **`const_reference`**
  - **`pointer`**
  - **`iterator`**
  - **`const_iterator`**
  - **`reverse_iterator`**
  - **`const_reverse_iterator`**
  - **`difference_type`**
  - **`size_type`**

© 2003 Prentice Hall, Inc. All rights reserved.

Introduction to Iterators

- Iterators similar to pointers
  - Point to first element in a container
  - Iterator operators same for all containers
    - **`*`** dereferences
    - **`++`** points to next element
    - **`begin()`** returns iterator to first element
    - **`end()`** returns iterator to last element
  - Use iterators with sequences (ranges)
    - Containers
    - Input sequences: **`istream_iterator`**
    - Output sequences: **`ostream_iterator`**

© 2003 Prentice Hall, Inc. All rights reserved.

Iterators

- Usage
  - **std::istream_iterator< int > inputInt(cin)**
    - Can read input from **cin**
    - ***inputInt:** Dereference to read first **int** from **cin**
    - **++inputInt:** Go to next **int** in stream
  - **std::ostream_iterator< int > outputInt(cout)**
    - Can output **int**s to **cout**
    - ***outputInt = 7:** Outputs **7** to **cout**
    - **++outputInt:** Advances iterator so we can output next **int**
  - Example

```
int number1 = *inputInt;
++inputInt
int number1 = *inputInt;
cout << "The sum is: ";
*outputInt = number1 + number2;
```

© 2003 Prentice Hall, Inc. All rights reserved.

Iterator Categories (Fig. 21.6)

- Input
  - Read elements from container, can only move forward
- Output
  - Write elements to container, only forward
- Forward
  - Combines input and output, retains position
- Bidirectional
  - Like forward, but can move backwards as well
  - Multi-pass (can pass through sequence twice)
- Random access
  - Like bidirectional, but can also jump to any element

© 2003 Prentice Hall, Inc. All rights reserved.

Iterator Types Supported (Fig. 21.8)

- Sequence containers
  - **`vector`**: random access
  - **`deque`**: random access
  - **`list`**: bidirectional
- Associative containers (all bidirectional)
  - **`set`**
  - **`multiset`**
  - **`Map`**
  - **`multimap`**
- Container adapters (no iterators supported)
  - **`stack`**
  - **`queue`**
  - **`priority_queue`**

© 2003 Prentice Hall, Inc. All rights reserved.

Iterator Operations (Fig. 21.10)

- All
  - **`++p`**, **`p++`**
- Input iterators
  - **`*p (to use as rvalue)`**
  - **`p = p1`**
  - **`p == p1`**, **`p != p1`**
- Output iterators
  - **`*p`**
  - **`p = p1`**
- Forward iterators
  - Have functionality of input and output iterators
- Bidirectional
  - **`--p, p--`**
- Random access
  - **`p + i, p += i`**
  - **`p - i, p -= i`**
  - **`p[i]`**
  - **`p < p1, p <= p1`**
  - **`p > p1, p >= p1`**

© 2003 Prentice Hall, Inc. All rights reserved.

Introduction to Algorithms

- STL has algorithms used generically across containers
  - Operate on elements indirectly via iterators
  - Often operate on sequences of elements
    - Defined by pairs of iterators
    - First and last element
  - Algorithms often return iterators
    - **`find()`**
    - Returns iterator to element, or **`end()`** if not found
  - Premade algorithms save programmers time and effort

© 2003 Prentice Hall, Inc. All rights reserved.

vector Sequence Container

- **`vector`**
  - Has random access iterators
  - Data structure with contiguous memory locations
    - Access elements with **`[]`**
  - Use when data must be sorted and easily accessible
- When memory exhausted
  - Allocates larger, contiguous area of memory
  - Copies itself there
  - Deallocates old memory
- Declarations
  - **`std::vector <`*`type`*`> v;`**
    - ***`type`***: **`int`**, **`float`**, etc.

© 2003 Prentice Hall, Inc. All rights reserved.

vector Sequence Container

- Iterators
  - **`std::vector<`*`type`*`>::const_iterator iterVar;`**
    - **`const_iterator`** cannot modify elements (read)
  - **`std::vector<`*`type`*`>::reverse_iterator iterVar;`**
    - Visits elements in reverse order (end to beginning)
    - Use **`rbegin`** to get starting point
    - Use **`rend`** to get ending point
- **`vector`** functions
  - **`v.push_back(value)`**
    - Add element to end (found in all sequence containers).
  - **`v.size()`**
    - Current size of vector
  - **`v.capacity()`**
    - How much vector can hold before reallocating memory
    - Reallocation doubles size
  - **`vector<`*`type`*`> v(a, a + SIZE)`**
    - Creates **`vector v`** with elements from array **`a`** up to (not including) **`a + SIZE`**

© 2003 Prentice Hall, Inc. All rights reserved.

vector Sequence Container

- **`vector`** functions
  - **`v.insert(`***iterator***`,`** *value* **`)`**
    - Inserts *value* before location of *iterator*
  - **`v.insert(`***iterator***`,`** *array* **`,`** *array + SIZE***`)`**
    - Inserts array elements (up to, but not including *array + SIZE)* into vector
  - **`v.erase( iterator )`**
    - Remove element from container
  - **`v.erase( iter1, iter2 )`**
    - Remove elements starting from **`iter1`** and up to (not including) **`iter2`**
  - **`v.clear()`**
    - Erases entire container
- **`vector`** functions operations
  - **`v.front(), v.back()`**
    - Return first and last element
  - **`v.[elementNumber] = value;`**
    - Assign **`value`** to an element
  - **`v.at[elementNumber] = value;`**
    - As above, with range checking
    - **`out_of_bounds`** exception

© 2003 Prentice Hall, Inc. All rights reserved.

vector Sequence Container

- **`ostream_iterator`**
  - **`std::ostream_iterator< `*`type`*` > `*`Name`*`( outputStream, separator );`**
    - ***`type`***: outputs values of a certain type
    - **`outputStream`**: iterator output location
    - **`separator`**: character separating outputs
- Example
  - **`std::ostream_iterator< int > output( cout, " " );`**
  - **`std::copy( iterator1, iterator2, output );`**
    - Copies elements from **`iterator1`** up to (not including) **`iterator2`** to output, an **`ostream_iterator`**

© 2003 Prentice Hall, Inc. All rights reserved.

list Sequence Container

- **`list`** container : Header **`<list>`**
  - Efficient insertion/deletion anywhere in container
  - Doubly-linked list (two pointers per node)
  - Bidirectional iterators
  - **`std::list<`** ***`type`*** **`>`** ***`name`*** **`;`**
- **`list`** functions for object **`t`**
  - **`t.sort()`**
    - Sorts in ascending order
  - **`t.splice(iterator, otherObject );`**
    - Inserts values from **`otherObject`** before **`iterator`**
  - **`t.merge( otherObject )`**
    - Removes **`otherObject`** and inserts it into **`t`**, sorted
  - **`t.unique()`**
    - Removes duplicate elements
  - **`t.swap(otherObject);`**
    - Exchange contents
  - **`t.assign(iterator1, iterator2)`**
    - Replaces contents with elements in range of iterators
  - **`t.remove(value)`**
    - Erases all instances of **`value`**

© 2003 Prentice Hall, Inc. All rights reserved.

deque Sequence Container

- **`deque`** ("deek"): double-ended queue
  - Header **`<deque>`**
  - Indexed access using **`[]`**
  - Efficient insertion/deletion in front and back
  - Non-contiguous memory: has "smarter" iterators
- Same basic operations as **`vector`**
  - Also has
    - **`push_front`** (insert at front of **`deque`**)
    - **`pop_front`** (delete from front)

© 2003 Prentice Hall, Inc. All rights reserved.

Associative Containers

- Associative containers
  - Direct access to store/retrieve elements
  - Uses keys (search keys)
  - 4 types: **`multiset`**, **`set`**, **`multimap`** and **`map`**
    - Keys in sorted order
    - **`multiset`** and **`multimap`** allow duplicate keys
    - **`multimap`** and **`map`** have keys and associated values
    - **`multiset`** and **`set`** only have values

© 2003 Prentice Hall, Inc. All rights reserved.

multiset Associative Container

- **`multiset`**
  - Header **`<set>`**
  - Fast storage, retrieval of keys (no values)
  - Allows duplicates
  - Bidirectional iterators
- Ordering of elements
  - Done by comparator function object
    - Used when creating multiset
  - For integer multiset
    - **`less<int>`** comparator function object
    - **`multiset< int, std::less<int> > myObject;`**
    - Elements will be sorted in ascending order

© 2003 Prentice Hall, Inc. All rights reserved.

21.3.1 multiset Associative Container

- Multiset functions
  - **`ms.insert(`*`value`*`)`**
    - Inserts value into multiset
  - **`ms.count(`*`value`*`)`**
    - Returns number of occurrences of ***`value`***
  - **`ms.find(`*`value`*`)`**
    - Returns iterator to first instance of ***`value`***
  - **`ms.lower_bound(`*`value`*`)`**
    - Returns iterator to first location of ***`value`***
  - **`ms.upper_bound(`*`value`*`)`**
    - Returns iterator to location after last occurrence of ***`value`***
- Class **`pair`**
  - Manipulate pairs of values
  - **`Pair`** objects contain **`first`** and **`second`**
    - **`const_iterators`**
  - For a **`pair`** object **`q`**

    `q = ms.equal_range(`*`value`*`)`
    - Sets **`first`** and **`second`** to **`lower_bound`** and **`upper_bound`** for a given ***`value`***

© 2003 Prentice Hall, Inc. All rights reserved.

fig21_19.cpp
(1 of 3)

```
// Fig. 21.19: fig21_19.cpp
// Testing Standard Library class multiset
#include <iostream>

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

#include <set>  // multiset class-template definition

// define short name for multiset type used in this p
typedef std::multiset< int, std::less< int > > ims;

#include <algorithm>  // copy algorithm

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };

   ims intMultiset;  // ims is typedef for "integer multiset"
   std::ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
        << " values of 15 in the multiset\n";

```

`typedef`s help clarify program. This declares an integer multiset that stores values in ascending order.

© 2003 Prentice Hall, Inc.
All rights reserved.

fig21_19.cpp
(2 of 3)

```
   intMultiset.insert( 15 );  // insert 15 in intMultiset
   intMultiset.insert( 15 );  // insert 15 in intMultiset

   cout << "After inserts, there are "
        << intMultiset.count( 15 )
        << " values of 15 in the multiset\n\n";

   // iterator that cannot be used to change e
   ims::const_iterator result;

   // find 15 in intMultiset; find returns iterator
   result = intMultiset.find( 15 );

   if ( result != intMultiset.end() ) // if iterator not at end
      cout << "Found value 15\n";     // found search value 15

   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );

   if ( result == intMultiset.end() )    // will be true hence
      cout << "Did not find value 20\n"; // did not find 20

   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );

   cout << "\nAfter insert, intMultiset contains:\n";
   std::copy( intMultiset.begin(), intMultiset.end(), output );

```

Use member function **`find`**.

© 2003 Prentice Hall, Inc.
All rights reserved.

Outline

```
   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: "
        << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: "
        << *( intMultiset.upper_bound( 22 ) );

   // p represents pair of const_iterators
   std::pair< ims::const_iterator, ims::const_it

   // use equal_range to determine lower and upp
   // of 22 in intMultiset
   p = intMultiset.equal_range( 22 );

   cout << "\n\nequal_range of 22:"
        << "\n   Lower bound: " << *( p.first )
        << "\n   Upper bound: " << *( p.second );

   cout << endl;

   return 0;

} // end main
```

Use a **`pair`** object to get the lower and upper bound for **`22`**.

fig21_19.cpp
(3 of 3)

© 2003 Prentice Hall, Inc.
All rights reserved.

```
There are currently 0 values of 15 in the multiset
After inserts, there are 2 values of 15 in the multiset

Found value 15
Did not find value 20

After insert, intMultiset contains:
1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22
Upper bound of 22: 30

equal_range of 22:
   Lower bound: 22
   Upper bound: 30
```

Outline

fig21_19.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

21.3.2 set Associative Container

- **`Set:`** Header **`<set>`**
  - Implementation identical to **`multiset`**
  - Unique keys: Duplicates ignored and not inserted
  - Supports bidirectional iterators (but not random access)
  - **`std::set<`** ***`type`*****`, std::less<`*****`type`*****`> >`** ***`name`*****`;`**
- **`Multimap:`** Header **`<map>`**
  - Fast storage and retrieval of keys and associated values
    - Has key/value pairs
  - Duplicate keys allowed (multiple values for a single key)
    - One-to-many relationship
    - I.e., one student can take many courses
  - Insert **`pair`** objects (with a key and value)
  - Bidirectional iterators

© 2003 Prentice Hall, Inc. All rights reserved.

21.3.3 multimap Associative Container

- Example

```
std::multimap< int, double, std::less< int > > mmapObject;
```

  - Key type **int**
  - Value type **double**
  - Sorted in ascending order
    - Use **typedef** to simplify code

```
typedef std::multimap<int, double, std::less<int>> mmid;
mmid mmapObject;
mmapObject.insert( mmid::value_type( 1, 3.4 ) );
```

  - Inserts key **1** with value **3.4**
  - **mmid::value_type** creates a **pair** object

© 2003 Prentice Hall, Inc. All rights reserved.

fig21_21.cpp

```
// Fig. 21.21: fig21_21.cpp
// Standard library class multimap test program.
#include <iostream>

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

#include <map>  // map class-template definition

// define short name for multimap type used in this program
typedef std::multimap< int, double, std::less< int > > mmid;

int main()
{
   mmid pairs;

   cout << "There are currently " << pairs.count( 15 )
        << " pairs with key 15 in the multimap\n";

   // insert two value_type objects in pairs
   pairs.insert( mmid::value_type( 15, 2.7 ) );
   pairs.insert( mmid::value_type( 15, 99.3 ) );

   cout << "After inserts, there are "
        << pairs.count( 15 )
        << " pairs with key 15\n\n";
```

Definition for a **`multimap`** that maps integer keys to double values.

Create multimap and insert key-value pairs.

© 2003 Prentice Hall, Inc.
All rights reserved.

fig21_21.cpp
(2 of 2)

```

   // insert five value_type objects in pairs
   pairs.insert( mmid::value_type( 30, 111.11 ) );
   pairs.insert( mmid::value_type( 10, 22.22 ) );
   pairs.insert( mmid::value_type( 25, 33.333 ) );
   pairs.insert( mmid::value_type( 20, 9.345 ) );
   pairs.insert( mmid::value_type( 5, 77.54 ) );

   cout << "Multimap pairs contains:\nKe

   // use const_iterator to walk through elements of pairs
   for ( mmid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   cout << endl;

   return 0;

} // end main
```

Use iterator to print entire **multimap**.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
There are currently 0 pairs with key 15 in the multimap
After inserts, there are 2 pairs with key 15

Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      2.7
15      99.3
20      9.345
25      33.333
30      111.11
```

fig21_21.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

21.3.4 map Associative Container

- **`map`**
  - Header **`<map>`**
  - Like **`multimap`**, but only unique key/value pairs
    - One-to-one mapping (duplicates ignored)
  - Use **`[]`** to access values
  - Example: for **`map`** object **`m`**

    ```
    m[30] = 4000.21;
    ```

    - Sets the value of key 30 to **`4000.21`**
  - If subscript not in **`map`**, creates new key/value pair
- Type declaration
  - **`std::map< int, double, std::less< int > >;`**

© 2003 Prentice Hall, Inc. All rights reserved.

fig21_22.cpp
(1 of 2)

```
// Fig. 21.22: fig21_22.cpp
// Standard library class map test program.
#include <iostream>

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

#include <map>  // map class-template defi

// define short name for map type used in this program
typedef std::map< int, double, std::less< int > > mid;

int main()
{
   mid pairs;

   // insert eight value_type objects in pairs
   pairs.insert( mid::value_type( 15, 2.7 ) );
   pairs.insert( mid::value_type( 30, 111.11 ) );
   pairs.insert( mid::value_type( 5, 1010.1 ) );
   pairs.insert( mid::value_type( 10, 22.22 ) );
   pairs.insert( mid::value_type( 25, 33.333 ) );
   pairs.insert( mid::value_type( 5, 77.54 ) ); // dupe ignored
   pairs.insert( mid::value_type( 20, 9.345 ) );
   pairs.insert( mid::value_type( 15, 99.3 ) ); // dupe ignored

```

Again, use **`typedef`**s to simplify declaration.

Duplicate keys ignored.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
   cout << "pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( mid::const_iterator iter = pairs.begin();
         iter != pairs.end(); ++iter )
      cout << iter->first << '\t'
           << iter->second << '\n';

   // use subscript operator to change val
   pairs[ 25 ] = 9999.99;

   // use subscript operator insert value for key 40
   pairs[ 40 ] = 8765.43;

   cout << "\nAfter subscript operations, pairs contains:"
        << "\nKey\tValue\n";

   for ( mid::const_iterator iter2 = pairs.begin();
         iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t'
           << iter2->second << '\n';

   cout << endl;

   return 0;

} // end main
```

Can use subscript operator to add or change key-value pairs.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       33.333
30       111.11

After subscript operations, pairs contains:
Key      Value
5        1010.1
10       22.22
15       2.7
20       9.345
25       9999.99
30       111.11
40       8765.43
```

fig21_22.cpp
output (1 of 1)

© 2003 Prentice Hall, Inc.
All rights reserved.

21.4 Container Adapters

- Container adapters
  - **`stack`**, **`queue`** and **`priority_queue`**
  - Not first class containers
    - Do not support iterators
    - Do not provide actual data structure
  - Programmer can select implementation
  - Member functions **`push`** and **`pop`**
- **`stack`**
  - Header **`<stack>`**
  - Insertions and deletions at one end
  - Last-in, first-out (LIFO) data structure
  - Can use **`vector`**, **`list`**, or **`deque`** (default)
  - Declarations

```
stack<type, vector<type> > myStack;
stack<type, list<type> > myOtherStack;
stack<type> anotherStack; // default deque
```

    - **`vector`**, **`list`**
      - Implementation of **`stack`** (default **`deque`**)
      - Does not change behavior, just performance (**`deque`** and **`vector`** fastest)

© 2003 Prentice Hall, Inc. All rights reserved.

21.5 Algorithms

- Before STL
  - Class libraries incompatible among vendors
  - Algorithms built into container classes
- STL separates containers and algorithms
  - Easier to add new algorithms
  - More efficient, avoids **`virtual`** function calls
  - **`<algorithm>`**

© 2003 Prentice Hall, Inc. All rights reserved.

21.5.6 Basic Searching and Sorting Algorithms

- **`find(iter1, iter2, value)`**
  - Returns iterator to first instance of **`value`** (in range)
- **`find_if(iter1, iter2, function)`**
  - Like **`find`**
  - Returns iterator when **`function`** returns **`true`**
- **`sort(iter1, iter2)`**
  - Sorts elements in ascending order
- **`binary_search(iter1, iter2, value)`**
  - Searches ascending sorted list for value
  - Uses binary search

© 2003 Prentice Hall, Inc. All rights reserved.

21.7 Function Objects

- Function objects (**`<functional>`**)
  - Contain functions invoked using operator**`()`**

STL function objects	Type
`divides< T >`	arithmetic
`equal_to< T >`	relational
`greater< T >`	relational
`greater_equal< T >`	relational
`less< T >`	relational
`less_equal< T >`	relational
`logical_and< T >`	logical
`logical_not< T >`	logical
`logical_or< T >`	logical
`minus< T >`	arithmetic
`modulus< T >`	arithmetic
`negate< T >`	arithmetic
`not_equal_to< T >`	relational
`plus< T >`	arithmetic
`multiplies< T >`	arithmetic

© 2003 Prentice Hall, Inc. All rights reserved.

```
// Fig. 21.42: fig21_42.cpp
// Demonstrating function objects.
#include <iostream>

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

#include <vector>      // vector class-template definition
#include <algorithm>   // copy algorithm
#include <numeric>     // accumulate algorithm
#include <functional>  // binary_function definition

// binary function adds square of its second argument a
// running total in its first argument, then returns sum
int sumSquares( int total, int value )
{
   return total + value * value;

} // end function sumSquares

```

Create a function to be used with **`accumulate`**.

fig21_42.cpp
(1 of 4)

© 2003 Prentice Hall, Inc.
All rights reserved.

fig21_42.cpp

```
// binary function class template defines overloaded operator()
// that adds suare of its second argument and running total in
// its first argument, then returns sum
template< class T >
class SumSquaresClass : public std::binary_function< T, T, T > {

public:

   // add square of value to total and return result
   const T operator()( const T &total, const T &value )
   {
      return total + value * value;

   } // end function operator()

}; // end class SumSquaresClass

```

Create a function object (it can also encapsulate data). Overload `operator()`.

© 2003 Prentice Hall, Inc.
All rights reserved.

```
int main()
{
   const int SIZE = 10;
   int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

   std::vector< int > integers( array, array + SIZE );

   std::ostream_iterator< int > output( cout, " " );

   int result = 0;

   cout << "vector v contains:\n";
   std::copy( integers.begin(), integers.end(), output );

   // calculate sum of squares of elements of vector integers
   // using binary function sumSquares
   result = std::accumulate( integers.begin(), integers.end(),
      0, sumSquares );

   cout << "\n\nSum of squares of elements in integers using "
        << "binary\nfunction sumSquares: " << result;

```

fig21_42.cpp
(3 of 4)

`accumulate` initially passes **`0`** as the first argument, with the first element as the second. It then uses the return value as the first argument, and iterates through the other elements.

© 2003 Prentice Hall, Inc.
All rights reserved.

fig21_42.cpp
(4 of 4)

fig21_42.cpp
output (1 of 1)

```
   // calculate sum of squares of elements of vector integers
   // using binary-function object
   result = std::accumulate( integers.begin(), integers.end(),
      0, SumSquaresClass< int >() );

   cout << "\n\nSum of squares of elements in integers using "
        << "binary\nfunction object of type "
        << "SumSquaresClass< int >: " << result << endl;

   return 0;

} // end main
```

Use **accumulate** with a function object.

```
vector v contains:
1 2 3 4 5 6 7 8 9 10

Sum of squares of elements in integers using binary
function sumSquares: 385

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385
```

© 2003 Prentice Hall, Inc.
All rights reserved.