IS 0020
Program Design and Software Tools

Template, Standard Template Library

Lecture 9

March 22, 2005

Introduction

- Overloaded functions
  - Similar operations but Different types of data
- Function templates
  - Specify entire range of related (overloaded) functions
  - Function-template specializations
  - Identical operations
    - Different types of data
  - Single function template
    - Compiler generates separate object-code functions
  - Unlike Macros they allow Type checking
- Class templates
  - Specify entire range of related classes
    - Class-template specializations

Function Templates

- Function-template definitions
  - Keyword **`template`**
  - List formal type parameters in angle brackets (**`<`** and **`>`**)
    - Each parameter preceded by keyword **`class`** or **`typename`**
      - **`class`** and **`typename`** interchangeable

```
template< class T >
template< typename ElementType >
template< class BorderType, class FillType >
```

    - Specify types of
      - Arguments to function
      - Return type of function
      - Variables within function

Outline

fig11_01.cpp
(1 of 2)

```
// Fig. 11.1: fig11_01.cpp
// Using template functions.
#include <iostream>

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

// function template printArray defi
template< class T >
void printArray( const T *array, const int count )
{
   for ( int i = 0; i < co
      cout << array[ i ] <

   cout << endl;

} // end function prin

int main()
{
   const int aCount = 5;
   const int bCount = 7;
   const int cCount = 6;

```

Function template definition; declare single formal type parameter **`T`**.

`T` is type parameter; use any valid identifier.

If **`T`** is user-defined type, stream-insertion operator must be overloaded for class **`T`**.

Outline

fig11_01.cpp
(2 of 2)

```
   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO";  // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, aCount );

   cout << "Array b contains

   // call double func
   printArray( b, bCou

   cout << "Array c co

   // call character fu
   printArray( c, cCou

   return 0;

} // end main
```

Crea pecialization for printing array

```
void                               , const int count )
{
   f                               + )
   c
} // end function printArray
```

Compiler infers **`T`** is **`double`**; instantiates function-template specialization where **`T`** is

Compiler infers **`T`** is **`char`**; instantiates function-template specialization where **`T`** is **`char`**.

Overloading Function Templates

- Related function-template specializations
  - Same name
    - Compiler uses overloading resolution
- Function template overloading
  - Other function templates with same name
    - Different parameters
  - Non-template functions with same name
    - Different function arguments
  - Compiler performs matching process
    - Tries to find precise match of function name and argument types
    - If fails, function template
      - Generate function-template specialization with precise match

Class Templates

- Stack
  - LIFO (last-in-first-out) structure
- Class templates
  - Generic programming
  - Describe notion of stack generically
    - Instantiate type-specific version
  - Parameterized types
    - Require one or more type parameters
      - Customize “generic class” template to form class-template specialization

Outline

tstack1.h (1 of 4)

```
// Fig. 11.2: tstack1.h
// Stack class template.
#ifndef TSTACK1_H
#define TSTACK1_H

template< class T >
class Stack {

public:
   Stack( int = 10 );  // default constructor (stack size 10)

   // destructor
   ~Stack()
   {
      delete [] stackPtr;

   } // end ~Stack destructor

   bool push( const T& );  // push an element onto the stack
   bool pop( T& );         // pop an element off the stack

```

Specify class-template definition; type parameter **`T`** indicates type of **`Stack`** class to be created.

Function parameters of type **`T`**.

Outline

tstack1.h (2 of 4)

```
   // determine whether Stack is empty
   bool isEmpty() const
   {
      return top == -1;

   } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {
      return top == size - 1;

   } // end function isFull

private:
   int size;     // # of elements in the stack
   int top;      // location of the top element
   T *stackPtr;  // pointer to the stack

}; // end class Stack

```

Array of elements of type **`T`**.

Outline

tstack1.h (3 of 4)

```
// constructor
template< class T >
Stack< T >::Stack( int s )
{
   size = s > 0 ? s : 10;
   top = -1;  // Stack initially empt
   stackPtr = new T[ size ]; // alloc

} // end Stack constructor

// push element onto stack;
// if successful, return true; ot
template< class T >
bool Stack< T >::push( const T &p
{
   if ( !isFull() ) {
      stackPtr[ ++top ] = pushValue;  // place item on Stack
      return true;  // push successful

   } // end if

   return false;  // push unsuccessful

} // end function push

```

Constructor creates array of type **`T`**. For example, compiler generates

`stackPtr = new T[ size ];`

for class-template specialization **`Stack< double >`**.

`T >`

Use binary scope resolution operator (**`::`**) with class-template name (**`Stack< T >`**) to tie definition to class template's scope.

Outline

tstack1.h (4 of 4)

```
// pop element off stack;
// if successful, return true; otherwise, return false
template< class T >
bool Stack< T >::pop( T &popValue )
{
   if ( !isEmpty() ) {
      popValue = stackPtr[ top-- ];  // r
      return true;  // pop successful

   } // end if

   return false;  // pop unsuccessful

} // end function pop

#endif
```

Member function preceded with header

Use binary scope resolution operator (`::`) with class-template name (**`Stack< T >`**) to tie definition to class template's scope.

Outline

fig11_03.cpp
(1 of 3)

```
// Fig. 11.3: fig11_03.cpp
// Stack-class-template test program.
#include <iostream>

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

#include "tstack1.h"  // Stack class template definition

int main()
{
   Stack< double > doubleStack( 5 );
   double doubleValue = 1.1;

   cout << "Pushing elements onto doubleSt

   while ( doubleStack.push( doubleValue ) ) {
      cout << doubleValue << ' ';
      doubleValue += 1.1;

   } // end while

   cout << "\nStack is full. Cannot push " << doubleValue
        << "\n\nPopping elements from doubleStack\n";
```

Link to class template definition.

Instantiate object of class **`Stack< double >`**.

Invoke function **`push`** of class-template specialization **`Stack< double >`**.

Outline

fig11_03.cpp
(2 of 3)

```
   while ( doubleStack.pop( doubleValue ) )
      cout << doubleValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   Stack< int > intStack;
   int intValue = 1;
   cout << "\nPushing elements onto intStack\n";

   while ( intStack.push( intValue ) ) {
      cout << intValue << ' ';
      ++intValue;

   } // end while

   cout << "\nStack is full. Cannot push " << intValue
        << "\n\nPopping elements from intStack\n";

   while ( intStack.pop( intValue ) )
      cout << intValue << ' ';

   cout << "\nStack is empty. Cannot pop\n";

   return 0;
```

Invoke function **pop** of class-template specialization **Stack< double >**.

Note similarity of code for **Stack< int >** to code for **Stack< double >**.

Outline

fig11_03.cpp
(3 of 3)

fig11_03.cpp
output (1 of 1)

```
} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Outline

fig11_04.cpp
(1 of 2)

```
// Fig. 11.4: fig11_04.cpp
// Stack class template test program. Function main uses a
// function template to manipulate objects of type Stack< T >.
#include <iostream>

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

#include "tstack1.h"  // Stack class template definition

// function template to manipulate Stack< T >
template< class T >
void testStack(
   Stack< T > &theStack,   // reference to Stack< T >
   T value,                // initial value to push
   T increment,            // increment for subsequent values
   const char *stackName ) // name of the Stack < T > object
{
   cout << "\nPushing elements onto " << stackName << '\n';

   while ( theStack.push( value ) ) {
      cout << value << ' ';
      value += increment;

   } // end while
```

Function template to manipulate **`Stack< T >`** eliminates similar code from previous file for **`Stack< double >`** and **`Stack< int >`**.

Outline

fig11_04.cpp
(2 of 2)

```

   cout << "\nStack is full. Cannot push " << value
        << "\n\nPopping elements from " << stackName << '\n';

   while ( theStack.pop( value ) )
      cout << value << ' ';

   cout << "\nStack is empty. Cannot pop\n";

} // end function testStack

int main()
{
   Stack< double > doubleStack( 5 );
   Stack< int > intStack;

   testStack( doubleStack, 1.1, 1.1, "doubleStack" );
   testStack( intStack, 1, 1, "intStack" );

   return 0;

} // end main
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop
```

Note output identical to that of **`fig11_03.cpp`**.

Outline

fig11_04.cpp
output (1 of 1)

Class Templates and Nontype Parameters

- Class templates
  - Nontype parameters
    - Default arguments
    - Treated as **`consts`**
      `template< class T, int elements >`
      `Stack< double, 100 > mostRecentSalesFigures;`
      - Declares object of type **`Stack< double, 100>`**
  - Type parameter
    - Default type example: **`template< class T = string >`**
- Overriding class templates
  - Class for specific type
    - Does not match common class template
  - Example:

```
template<>
Class Array< Martian > {
        // body of class definition
};
```

Templates and Inheritance

- Several ways of relating templates and inheritance
  - Class template derived from class-template specialization
  - Class template derived from non-template class
  - Class-template specialization derived from class-template specialization
  - Non-template class derived from class-template specialization
- Friendships between class template and
  - Global function
  - Member function of another class
  - Entire class

Templates and Friends

- **`friend`** functions
  - Inside definition of **`template< class T > class X`**
    - **`friend void f1();`**
      - **`f1()`** **`friend`** of all class-template specializations
    - **`friend void f2( X< T > & );`**
      - **`f2( X< float > & )`** **`friend`** of **`X< float >`** only,
        `f2( X< double > & )` **`friend`** of **`X< double >`** only,
        `f2( X< int > & )` **`friend`** of **`X< int >`** only,
        …
    - **`friend void A::f4();`**
      - Member function **`f4`** of class **`A`** **`friend`** of all class-template specializations

Templates and Friends

- **`friend`** functions
  - Inside definition of **`template< class T > class X`**
    - **`friend void C< T >::f5( X< T > & );`**
      - Member function **`C<float>::f5( X< float> & )`** **`friend`** of **`class X<float>`** only
- **`friend`** classes
  - Inside definition of **`template< class T > class X`**
    - **`friend class Y;`**
      - Every member function of **`Y`** friend of every class-template specialization
    - **`friend class Z<T>;`**
      - **`class Z<float>`** **`friend`** of class-template specialization **`X<float>,`** etc.

Templates and static Members

- Non-template class
  - **`static`** data members shared between all objects
- Class-template specialization
  - Each has own copy of **`static`** data members
  - **`static`** variables initialized at file scope
  - Each has own copy of **`static`** member functions

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

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

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`**

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`**

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`**

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;
```

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

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`**

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`

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

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.

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`**

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

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`**

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`

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)

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

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

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***

Outline

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";

```

typedefs help clarify program. This declares an integer multiset that stores values in ascending order.

Outline

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**.

Outline

fig21_19.cpp
(3 of 3)

```
   // 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**.

```
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)

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

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

Outline

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.

Outline

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`**.

Outline

fig21_21.cpp
output (1 of 1)

```
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
```

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 > >;`**

Outline

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 definition

// 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.

Outline

fig21_22.cpp
(2 of 2)

```
    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.

Outline

fig21_22.cpp
output (1 of 1)

```
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
```

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)

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>`**

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

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

Outline

fig21_42.cpp
(1 of 4)

```
// 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**.

Outline

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()**.

Outline

fig21_42.cpp
(3 of 4)

```
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;

```

`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.

Outline

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
```