Module 9

Templates & STL
Objectives

• In this module
  – Learn about templates
  – Construct function templates and class templates
  – STL
Introduction

• **Templates**: enable you to write generic code for related functions and classes
• **Function templates**: used to simplify function overloading
Templates

• **Template**: a single code body for a set of related functions (**function template**) and related classes (**class template**)

• **Syntax:**

```
template <class Type>
declaration;
```

  – **Type** is the data type

  – Declaration is either a function declaration or a class declaration
• **class** in the heading refers to any user-defined type or built-in type
• **Type**: a formal parameter to the template
• Just as variables are parameters to functions, data types are parameters to templates
Function Templates

• Syntax of the function template:

```
template <class Type>
function definition;
```

• Type is a formal parameter of the template used to:
  – Specify type of parameters to the function
  – Specify return type of the function
  – Declare variables within the function
```cpp
#include <iostream>
using namespace std;

template <class T>
T getMax (T x, T y)
{
    if (x > y)
        return x;
    else
        return y;
}

template <class T>
void swap (T* x, T* y)
{
    T temp = *x;
    *x = *y;
    *y = temp;
}

int main ()
{
    string str1, str2;
    cout << "Enter str1 and str2: ";
    cin >> str1 >> str2;

    string max = getMax (str1, str2);
    // or     = getMax <string> (str1, str2);

    cout << "The maximum string is 
        << max
        << endl;

    int x = 88, y = 99;
    swap (&x, &y);
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    }
```
Class Templates

• **Class template**: a single code segment for a set of related classes
  – Called parameterized types

• **Syntax**:  
  
  template <class Type>  
  class declaration

• A template instantiation can be created with either a built-in or user-defined type

• The function members of a class template are considered to be function templates
// template class
#include <iostream>
using namespace std;

template <class T>
class Point
{
  // Compiler issues warning message  
  //      friend ostream& operator<< (ostream&, const Point <T>&);

  friend ostream& operator<< (ostream&, const Point <int>&);
  friend ostream& operator<< (ostream&, const Point <float>&);

  public:
  Point ();
  Point (T x, T y);

  void print () const;

  private:
  T x;
  T y;
};
ostream& operator<< (ostream& os, const Point <int>& p)
{
    os << "Point (";
    os << p.x << " , " << p.y << " )";
    return os;
}

ostream& operator<< (ostream& os, const Point <float>& p)
{
    os << "Point (";
    os << p.x << " , " << p.y << " )";
    return os;
}
template <class T>
Point <T>::Point ()
{
    // Better don't initialize as
    // T is not a specific type!!!
}

template <class T>
Point <T>::Point (T x, T y)
{
    this -> x = x;
    this -> y = y;
}

template <class T>
void Point <T>::print () const
{
    cout << "Point (" << x << ", " << y << ")";
}

int main ()
{
    Point <int> p1 (2, 3);
    p1.print ();
    cout << endl;
    Point <float> p2 (2.3, 3.4);
    p2.print ();
    cout << endl;
    cout << p1 << " and " << p2 << endl;
}
// A stack is implemented using a linear linked list
#include <iostream>
#include <string>
using namespace std;

template <class T>
class Stack {
  public:
    Stack();
    void push(T);
    T pop();
    bool isEmpty() const;

  private:
    struct Node;
    typedef Node* NodePtr;

    struct Node
    {
      T data;
      NodePtr next;
    };
    NodePtr head;
};
template <class T>  
Stack<T>::Stack()
{
    head = NULL;
}

template <class T>  
void Stack<T>::push(T item)
{
    NodePtr temp = new Node;
    temp -> data = item;
    temp -> next = head;
    head = temp;
}

template <class T>  
bool Stack<T>::isEmpty () const
{
    return head == NULL;
}

T Stack<T>::pop()
{
    T item;
    if (head != NULL)
    {
        NodePtr temp = head;
        head = head -> next;
        item = temp -> data;
        delete temp;
    }
    return item;
}
int main ()
{
    Stack <int> sInt;
    Stack <string> sStr;

    for (int i = 1; i < 10; i++)
        sInt.push (i);

    while (!sInt.isEmpty ())
        cout << sInt.pop () << " ";
    cout << endl;

    sStr.push ("AAA");
    sStr.push ("BBB");
    sStr.push ("CCC");

    while (!sStr.isEmpty ())
        cout << sStr.pop () << " ";
    cout << endl;
}
#include <iostream>
using namespace std;

enum Gender {Male, Female};

template <class T, class S>
class Student {
  public:
    Student ();
    Student (T, S);
    void print () const;
  private:
    T name;
    S sex;
};

template <class T, class S>
Student <T, S>::Student ()
{
}

template <class T, class S>
Student <T, S>::Student (T name, S sex)
{
    this -> name = name;
    this -> sex = sex;
}

template <class T, class S>
void Student <T, S>::print () const
{
    cout << "Student (" << name << ", \\
        " << sex << ")" << endl;
}
int main ()
{
    string name = "Heng A K";
    Gender g = Male;
    Student <string, Gender> s1 (name, g);
    s1.print ();

    name = "Nancy Tan";
    char sex [] = "Female";
    Student <string, char *> s2 (name, sex);
    s2.print ();

    name = "Robert Lim";
    char type = 'M';
    Student <string, char> s3 (name, type);
    s3.print ();
}
• Passing a parameter to a function takes effect at run time
• Passing a parameter to a class template takes effect at compile time
• Cannot compile the implementation file independently of the client code
  – Can put class definition and definitions of the function templates directly in the client code
  – Can put class definition and the definitions of the function templates in the same header file
Header File and Implementation File of a Class Template (cont’d.)

• Another alternative: put class definition and function definitions in separate files
  – Include directive to implementation file at the end of header file

• In either case, function definitions and client code are compiled together
STL - Objectives

In this topic, you will:

• Learn about the Standard Template Library (STL)
• Explore the basic components of the STL: containers, iterators, and algorithms
• Use containers to manipulate data in a program
• Use iterators
• Learn about various generic algorithms
Introduction

• ANSI/ISO Standard C++ is equipped with a Standard Template Library (STL)
• STL includes class templates to process lists, stacks, and queues
• This topic:
  – Discusses many important features of the STL
  – Shows how to use its tools
Components of the STL

- Components of the STL:
  - Containers
  - Iterators
  - Algorithms
- Containers and iterators: class templates
- Iterators: used to step through the elements of a container
- Algorithms: used to manipulate data
Sequence Containers

• Sequence container: every object in the container has a specific position
• Two important predefined sequence containers:
  – vector
  – list
Iterators are a generation of pointers.

For example, if we define,

```cpp
iterator curr;
cout << “The item in the container is “ << *curr << endl;
++curr; // move the iterator to the next item in the container
-- curr; // move the iterator to the previous item in the container
```

Two useful functions:
- `iterator begin();` -- initializes an iterator to reference the 1st item in the container
- `iterator end();` -- returns a value that indicates whether you have reached the end of the container.
Sequence Container: `vector`

- Stores and manages its objects in a dynamic array
- **Must use:** `#include <vector>`
- **To define an object of type** `vector`, **specify the type of the object**
  - **Examples:**
    ```cpp
    vector<int> intList;
    vector<string> stringList;
    ```
- `vector` **contains several constructors**
### Sequence Container: `vector` (cont’d.)

**TABLE 21-1** Various Ways to Declare and Initialize a Vector Container

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;elemType&gt; vecList;</code></td>
<td>Creates the empty vector container <code>vecList</code>. (The default constructor is invoked.)</td>
</tr>
<tr>
<td><code>vector&lt;elemType&gt; vecList(otherVecList);</code></td>
<td>Creates the vector container <code>vecList</code> and initializes <code>vecList</code> to the elements of the vector <code>otherVecList</code>. <code>vecList</code> and <code>otherVecList</code> are of the same type.</td>
</tr>
<tr>
<td><code>vector&lt;elemType&gt; vecList(size);</code></td>
<td>Creates the vector container <code>vecList</code> of size <code>size</code>. <code>vecList</code> is initialized using the default constructor.</td>
</tr>
<tr>
<td><code>vector&lt;elemType&gt; vecList(n, elm);</code></td>
<td>Creates the vector container <code>vecList</code> of size <code>n</code>. <code>vecList</code> is initialized using <code>n</code> copies of the element <code>elm</code>.</td>
</tr>
<tr>
<td><code>vector&lt;elemType&gt; vecList(beg, end);</code></td>
<td>Creates the vector container <code>vecList</code>. <code>vecList</code> is initialized to the elements in the range <code>[beg, end]</code> that is, all the elements in the range <code>beg...end-1</code>. Both <code>beg</code> and <code>end</code> are pointers, called iterators in STL terminology. (Later in this chapter, we explain how iterators are used.)</td>
</tr>
</tbody>
</table>
Sequence Container: vector (cont’d.)

- Basic vector operations:
  - Item insertion
  - Item deletion
  - Stepping through the elements
- Vector elements can be processed just as they can in an array
## Sequence Container: vector (cont’d.)

**TABLE 21-2  Operations to Access the Elements of a Vector Container**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vecList.at(index)</code></td>
<td>Returns the element at the position specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>vecList[index]</code></td>
<td>Returns the element at the position specified by <code>index</code>.</td>
</tr>
<tr>
<td><code>vecList.front()</code></td>
<td>Returns the first element. (Does not check whether the container is empty.)</td>
</tr>
<tr>
<td><code>vecList.back()</code></td>
<td>Returns the last element. (Does not check whether the container is empty.)</td>
</tr>
</tbody>
</table>
# Sequence Container: `vector` (cont’d.)

## TABLE 21-3 Operations to Determine the Size of a Vector Container

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vecCont.capacity()</code></td>
<td>Returns the maximum number of elements that can be inserted into the container <code>vecCont</code> without reallocation.</td>
</tr>
<tr>
<td><code>vecCont.empty()</code></td>
<td>Returns <code>true</code> if the container <code>vecCont</code> is empty, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>vecCont.size()</code></td>
<td>Returns the number of elements currently in the container <code>vecCont</code>.</td>
</tr>
<tr>
<td><code>vecCont.max_size()</code></td>
<td>Returns the maximum number of elements that can be inserted into the container <code>vecCont</code>.</td>
</tr>
</tbody>
</table>
## Sequence Container: `vector` (cont’d.)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vecList.clear()</code></td>
<td>Deletes all of the elements from the container.</td>
</tr>
<tr>
<td><code>vecList.erase(position)</code></td>
<td>Deletes the element at the position specified by <code>position</code>.</td>
</tr>
<tr>
<td><code>vecList.erase(beg, end)</code></td>
<td>Deletes all of the elements starting at <code>beg</code> until <code>end-1</code>.</td>
</tr>
<tr>
<td><code>vecList.insert(position, elem)</code></td>
<td>A copy of <code>elem</code> is inserted at the position specified by <code>position</code>. The position of the new element is returned.</td>
</tr>
<tr>
<td><code>vecList.insert(position, n, elem)</code></td>
<td><code>n</code> copies of <code>elem</code> are inserted at the position specified by <code>position</code>.</td>
</tr>
<tr>
<td><code>vecList.insert(position, beg, end)</code></td>
<td>A copy of the elements, starting at <code>beg</code> until <code>end-1</code>, is inserted into <code>vecList</code> at the position specified by <code>position</code>.</td>
</tr>
<tr>
<td><code>vecList.push_back(elem)</code></td>
<td>A copy of <code>elem</code> is inserted into <code>vecList</code> at the end.</td>
</tr>
<tr>
<td><code>vecList.pop_back()</code></td>
<td>Deletes the last element.</td>
</tr>
<tr>
<td><code>vecList.resize(num)</code></td>
<td>Changes the number of elements to <code>num</code>. If <code>size()</code> increases, the default constructor creates the new elements.</td>
</tr>
<tr>
<td><code>vecList.resize(num, elem)</code></td>
<td>Changes the number of elements to <code>num</code>. If <code>size()</code> increases, the new elements are copies of <code>elem</code>.</td>
</tr>
</tbody>
</table>
Declaring an Iterator to a Vector Container

• A vector contains a typedef iterator
• Examples:
  
  ```cpp
  vector<int>::iterator intVecIter;
  ```
  
  – Declares `intVecIter` to be an iterator into a vector container of type `int`
  
  ```cpp
  ++intVecIter
  ```
  
  – Advances the iterator
  
  ```cpp
  *intVecIter
  ```
  
  – Returns element at current iterator position
Containers and the Functions begin and end

• Every container contains member functions:
  – `begin`: returns the position of the first element
  – `end`: returns the position of the last element

• Both functions have no parameters
### Member Functions Common to All Containers

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default constructor</td>
<td>Initializes the object to an empty state.</td>
</tr>
<tr>
<td>Constructor with parameters</td>
<td>In addition to the default constructor, every container has constructors with parameters. We will describe these constructors when we discuss a specific container.</td>
</tr>
<tr>
<td>Copy constructor</td>
<td>Executes when an object is passed as a parameter by value and when an object is declared and initialized using another object of the same type.</td>
</tr>
<tr>
<td>Destructor</td>
<td>Executes when the object goes out of scope.</td>
</tr>
<tr>
<td>ct.empty()</td>
<td>Returns <code>true</code> if container <code>ct</code> is empty, <code>false</code> otherwise.</td>
</tr>
<tr>
<td>ct.size()</td>
<td>Returns the number of elements currently in container <code>ct</code>.</td>
</tr>
</tbody>
</table>
### Member Functions Common to All Containers (cont’d.)

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ct.max_size()</code></td>
<td>Returns the maximum number of elements that can be inserted in container <code>ct</code>.</td>
</tr>
<tr>
<td><code>ctl.swap(ct2)</code></td>
<td>Swaps the elements of containers <code>ctl</code> and <code>ct2</code>.</td>
</tr>
<tr>
<td><code>ct.begin()</code></td>
<td>Returns an iterator to the first element into container <code>ct</code>.</td>
</tr>
<tr>
<td><code>ct.end()</code></td>
<td>Returns an iterator to the position after the last element into container <code>ct</code>.</td>
</tr>
<tr>
<td><code>ct.rbegin()</code></td>
<td>Reverse begin. Returns a pointer to the last element into container <code>ct</code>. This function is used to process the elements of <code>ct</code> in reverse.</td>
</tr>
<tr>
<td><code>ct.rend()</code></td>
<td>Reverse end. Returns a pointer to the position before the first element into container <code>ct</code>.</td>
</tr>
<tr>
<td><code>ct.insert(position,elem)</code></td>
<td>Inserts <code>elem</code> into container <code>ct</code> at the position specified by <code>position</code>. Note that here, <code>position</code> is an iterator.</td>
</tr>
<tr>
<td><code>ct.erase(beg, end)</code></td>
<td>Deletes all of the elements between <code>beg</code>...<code>end-1</code> from container <code>ct</code>. Both <code>beg</code> and <code>end</code> are iterators.</td>
</tr>
<tr>
<td><code>ct.clear()</code></td>
<td>Deletes all of the elements from the container. After a call to this function, container <code>ct</code> is empty.</td>
</tr>
</tbody>
</table>
## Member Functions Common to All Containers (cont’d.)

<table>
<thead>
<tr>
<th>Member function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operator functions</strong></td>
<td></td>
</tr>
<tr>
<td><code>ctl1 = ctl2;</code></td>
<td>Copies the elements of <code>ctl2</code> into <code>ctl1</code>. After this operation, the elements in both containers are the same.</td>
</tr>
<tr>
<td><code>ctl1 == ctl2</code></td>
<td>Returns <code>true</code> if containers <code>ctl1</code> and <code>ctl2</code> are equal, <code>false</code> otherwise.</td>
</tr>
<tr>
<td><code>ctl1 != ctl2</code></td>
<td>Returns <code>true</code> if containers <code>ctl1</code> and <code>ctl2</code> are not equal, <code>false</code> otherwise.</td>
</tr>
</tbody>
</table>
### Member Functions Common to All Sequence Containers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>seqCont.insert(position, elem)</code></td>
<td>A copy of <code>elem</code> is inserted at the position specified by the iterator <code>position</code>. The position of the new element is returned.</td>
</tr>
<tr>
<td><code>seqCont.insert(position, n, elem)</code></td>
<td><code>n</code> copies of <code>elem</code> are inserted at the position specified by the iterator <code>position</code>.</td>
</tr>
<tr>
<td><code>seqCont.insert(position, beg, end)</code></td>
<td>A copy of the elements, starting at <code>beg</code> until <code>end-1</code>, is inserted into <code>seqCont</code> at the position specified by the iterator <code>position</code>. Also, <code>beg</code> and <code>end</code> are iterators.</td>
</tr>
<tr>
<td><code>seqCont.push_back(elem)</code></td>
<td>A copy of <code>elem</code> is inserted into <code>seqCont</code> at the end.</td>
</tr>
<tr>
<td><code>seqCont.pop_back()</code></td>
<td>Deletes the last element.</td>
</tr>
</tbody>
</table>
### Member Functions Common to Sequence Containers (cont’d.)

**TABLE 21-6  Member Functions Common to All Sequence Containers (continued)**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>seqCont.erase(position)</code></td>
<td>Deletes the element at the position specified by the iterator <code>position</code>.</td>
</tr>
<tr>
<td><code>seqCont.erase(beg, end)</code></td>
<td>Deletes all of the elements starting at <code>beg</code> until <code>end-1</code>. Both <code>beg</code> and <code>end</code> are iterators.</td>
</tr>
<tr>
<td><code>seqCont.clear()</code></td>
<td>Deletes all of the elements from the container.</td>
</tr>
<tr>
<td><code>seqCont.resize(num)</code></td>
<td>Changes the number of elements to <code>num</code>. If <code>size()</code> grows, the new elements are created by their default constructor.</td>
</tr>
<tr>
<td><code>seqCont.resize(num, elem)</code></td>
<td>Changes the number of elements to <code>num</code>. If <code>size()</code> grows, the new elements are copies of <code>elem</code>.</td>
</tr>
</tbody>
</table>
```cpp
#include <iostream>
#include <vector>
using namespace std;

class Circle {
friend ostream& operator<< (ostream&, const Circle&);

public:
    Circle () ;
    Circle (int);

private:
    int radius;
};

ostream& operator<< (ostream& os, const Circle& c) {
    os << "C (" << c.radius << ")";
    return os;
}

Circle::Circle () {}

Circle::Circle (int r) {
    radius = r;
}

int main ()
{
    vector <Circle> myVector;

    // Use like an array
    for (int i = 0; i < 5; i++)
    {
        Circle c (i);
        myVector.push_back (c);
        cout << myVector [i] << "\t";
    }

    // Use of iterator
    vector <Circle>::iterator it;

    for (it = myVector.begin() ; it < myVector.end() ; it++ )
    {
        cout << *it << "\t";
    }
    cout << endl;

    return 0;
}
```
// A stack is implemented using
// a vector
#include <iostream>
#include <vector>
using namespace std;

template <class T>
class Stack {
    public:
        Stack();
        void push(T);
        T pop();
        bool isEmpty () const;
    private:
        
vector <T> v;
};

int main() {
    Stack<int> stack;
    stack.push(1);
    stack.push(2);
    stack.push(3);
    std::cout << stack.pop() << std::endl;
    std::cout << stack.pop() << std::endl;
    std::cout << stack.pop() << std::endl;
    return 0;
}

Interesting!!! I think better data structure
int main ()
{
    Stack <int> s;

    for (int i = 1; i <= 10; i++)
    {
        int element = i;
        s.push (element);
    }

    while (!s.isEmpty ())
    {
        cout << s.pop () << "\t";
    }
    cout << endl;
}
// Please refer to Carrano Page 224 -225

template <class T> class std::list {
    list ();
    list (size_type num, const T& val = T());
    list (const list <T> & anotherList);
    bool empty () const;
    size_type size () const;
    size_type max_size ();
    iterator insert (iterator i, const T& val = T());
    void remove (const T& val);
    iterator erase (iterator i);
    iterator begin ();
    iterator end ();
    void sort ();
};
#include <list>
#include <iostream>
#include <string>
using namespace std;

void printList (list <string> ll) {
    list <string>::iterator i = ll.begin ();
    while (i != ll.end ()) {
        cout << *i << "\n";
        ++i;
    }
    cout << "\n";
}

int main () {
    list <string> ll;
    list <string>::iterator i = ll.begin ();
    i = ll.insert (i, "bbb");
    i = ll.insert (i, "aaa");
    i = ll.insert (i, "ccc");
    i = ll.insert (i, "eee");
    i = ll.insert (i, "ddd");
    cout << "No of objects: " << ll.size() << endl;
    printList (ll);
    cout << "After sorting\n";
    ll.sort ();
    printList (ll);
}

No of objects: 5
ddd   eee   ccc   aaa   bbb

After sorting
aaa   bbb   ccc   ddd   eee
```cpp
#include <list>
#include <iostream>
using namespace std;

int main () {
    list < list <int> > listlist;
    list <int> alist;

    list < list <int> >::iterator ll = listlist.begin ();
    list <int> ::iterator al = alist.begin ();

    for (int n = 1; n <= 5; n++) {
        // construct a list
        alist.insert (al, n);
        // insert back
        listlist.insert (ll, alist);
        // add a list to another list
        alist.insert (al, 99);
        alist.insert (al, 100);
        listlist.insert (ll, alist);
        // add another list to another list
    }

    // to print out the list of list
    ll = listlist.begin ();
    while (ll != listlist.end ()) {
        list<int>::iterator i = (*ll).begin ();
        while (i != (*ll).end ()) {
            cout << *i << '	';
            ++i;
            // move to another object in the list
        }
        cout << endl;
        ll++;
        // move to another list
    }

    return 0;
}
```

One more STL map

• **What is a map?**
  A map is a super-cool and useful method of storing data. It's kind of a cross between a vector and an array.

• **How does a map work?**
  Basically, when we create a map, we give it two data types, as opposed to one (when we use a vector). The first is used as the *key*, which is basically used like when we use arrays. The second is the *value*, which is the actual value that this member will store.
```cpp
#include <iostream>
#include <map>
using namespace std;

enum UOWGrade {HD, D, C, P, F};

void display (map <string, UOWGrade> grade);

int main (){
    map <string, UOWGrade> grade;
    grade ["Heng AK"] = D;
    grade ["Tan AB"] = C;
    grade ["Robert"] = HD;
    grade ["Mohamed"] = F;
    grade ["Nancy"] = P;
    display (grade);
}

void display (map <string, UOWGrade> grade) {
    map <string, UOWGrade>::iterator it;
    it = grade.begin ();
    while (it != grade.end ())
    {
        cout << it -> first << "\t" << it -> second << endl;
        ++it;
    }
}
```

Output:
```
Heng AK 1
Mohamed 4
Nancy 3
Robert 0
Tan AB 2
```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <set>
using namespace std;

int main ()
{
    set <int> s;

    for (int i = 1; i <= 20; i++)
    {
        int n = rand () % 10;
        s.insert (n);
        cout << "Inserted " << n << endl;
    }

    cout << endl;
    cout << "\nThe size of set: " << s.size () << endl;

    set <int>::iterator it;
    for (it = s.begin(); it != s.end(); it++)
    {
        cout << *it << "\t";
    }
    cout << endl;
No duplication, set sorted

```
Inserted 1
Inserted 7
Inserted 4
Inserted 0
Inserted 9
Inserted 4
Inserted 8
Inserted 8
Inserted 2
Inserted 4
Inserted 5
Inserted 5
Inserted 1
Inserted 7
Inserted 1
Inserted 1
Inserted 5
Inserted 2
Inserted 7
Inserted 6
```

The size of set: 9
```cpp
// set of circles
#include <iostream>
#include <set>
#include <cstdlib>
#include <ctime>
using namespace std;

class Circle
{
    friend ostream& operator<< (ostream&, const Circle&);

public:
    Circle (int):
        // Compulsory function
        bool operator< (const Circle&) const:

    private:
        int radius;
};
```
Set of objects (continue)

```cpp
ostream& operator<<(ostream& os, const Circle& c)
{
    os << "C (" << c.radius << ")";
}

Circle::Circle(int radius)
{
    this->radius = radius;
}

bool Circle::operator<(const Circle& c) const
{
    return this->radius < c.radius;
}
```
Set of objects (continue)

```cpp
int main ()
{
    set <Circle> s:

    for (int i = 1; i <= 10; i++)
    {
        Circle c (rand () % 5 + 1);
        s.insert (c);
    }

    cout << "\nThe size of set: " << s.size () << endl;

    set <Circle>::iterator it:
    for ( it = s.begin(); it != s.end(); it++)
        cout << *it << "\t"
    cout << endl;
}
```

The size of set: 5
C (1)  C (2)  C (3)  C (4)  C (5)