Containers

The Standard C++ Library provides C++ programmers with
a collection of container types

a library of common data structures
Linked lists, vectors, deques, sets, maps

a set of fundamental algorithms that operate on them.

Container Library Components

- Principal components of Container Library
  - **Containers** – data structures that manage a collection of objects.
  - **Iterators** - mechanisms for traversing and examining the elements in a container.
  - **Algorithms** - procedures that operate on the contents of a container.

- Other components of Container Library
  - **Function objects** - provide a more efficient mechanism than pointers to functions.
  - **Adaptors** - provide a new interface to a container or to an iterator.
  - **Allocators** - encapsulate the memory model for different machine architectures.

Types of Containers

- **Sequence** containers - containers that organise objects, all of the same type, into a strictly linear arrangement.
  – **Vector** - an array which can grow at one end
  – **List** - linked list
  – **Deque** - an array which can grow at either end

- **Associative** containers - provide for fast retrieval of objects via keys. The elements are sorted, so fast binary search is possible.
  – **Set** - supports unique keys (at most one of each key value) and provides fast retrieval of keys themselves.
  – **Multiset** - a set where equal keys (possibly multiple copies of the one key) are allowed.
  – **Map** - supports unique keys for fast retrieval of another type based on the key.
  – **Multimap** - a map where equal keys are allowed.

`vector, deque` Containers

- These containers use the basic one-dimensional array.

- The container may be subscripted for storing or retrieving items in random order - but use of an out-of-range subscript leads to unspecified results.

- Items can be inserted at the beginning, end or within the container. If insertions are at
  – *the end* - if there is space, the item is added after the last. If there is insufficient space, twice the existing space is allocated, the existing items are copied, and the new item is added.
  – *the middle* - items are moved towards the end to make room and the item is added. If necessary, the space is reallocated.
  – *the beginning* -
    - *vector* - as for the middle.
    - *deque* - unused space is divided between both ends, so insertion at the beginning is as efficient as at the end.

- Deletions are the inverse of insertions. If necessary, other items are moved to occupy the space of the deleted item.

`list` Containers

- The underlying structure is a doubly-linked list.
- Fast random access to items is not supported.
- Insertions are more efficient than with `vector` or `deque`.
- Memory is allocated and released for individual items as required

`set, multiset, map, multimap`

- The underlying data structure is the binary tree.
- Fast access is provided to individual items - based on content, not on position within the container.
- `operator<` must be overloaded for the class of item being stored.

Container Declaration

Elements held in a container can be

- Primitive language types, e.g. `int`, `double`…
- Pointer types.
- User defined types (classes).

  In this case, a default constructor and a copy constructor must be available for the class

Sequential Container Declaration

- Vector

```
vector<Person> people1;
```

  – A Vector of default size with no elements.

```
vector<Person> people2(5);
```

  – A Vector of size 5 with no elements.

```
vector<Person> people3(5, aPerson);
```

  – A Vector, size 5, containing 5 copies of aPerson.

- Deque

```
deque<Person> people1;
```

  – Etc as for Vector

- List

```
list<Person> people1;
```

  – A List with no elements.

```
list <Person> people2(5);
```

  – A List with 5 elements, each initialised by default constructor.

```
list <Person> people3(5, aPerson);
```

  – A Vector, size 5, each element containing copy of aPerson.

`set` or `multiset` Declaration

- `Set`

  set<Employee, less<Employee> > empSet;
  set<Employee, greater<Employee> > empSet;

  A Set of Employees, whose ordering is defined by the less-than comparison operator (`operator<`)giving ascending order.

  – Alternatively the greater-than comparison operator (`operator>`) can be used to give descending order.

  – The function object requires the overloading of the appropriate comparison operator for the class being stored.

- If `multiset` is specified rather than `set`, objects that are equal to each other may be stored

`map` or `multimap` Declaration

- Pair data structure – used extensively by `map` and `multimap` (and in other contexts).
  – A `pair` is a `struct` with two components – `first` and `second`.
  (It is a template structure – *to be explained later* – so that the two components can be defined to be of any types .)

```
map<int, Employee* , less<int> > empMap
```

- A `map` containing `pairs` – `first` an `int` and `second` a pointer to an Employee. The `int` is the key and the ordering is defined by the `operator<` defined over `int`s.

- `multimap` – As for a map except that duplicates are allowed

Iterators

- Iterators are a generalisation of pointers that allow a programmer to work with different data structures in a uniform manner.
- e.g. compare printing the contents of a vector

```
#include <vector>
#include <iostream>
using namespace std;
main(){
  vector <int> v(3);
  v[0] = 5; v[1] = 2; v[2] = 7;
  vector<int>::iterator
              vFirst = v.begin();
  vector<int>::iterator
              vLast = v.end();
      while(vFirst != vLast)
        cout << *vFirst++ << " “;
  system(“PAUSE”):
  return EXIT_SUCCESS:
}
```

with printing the contents of a list

```
#include <list>
#include <iostream>
using namespace std;
main(){
  list <int> l;
  l.push_back(5); l.push_back(2);
  l.push_back(7);
  list<int>::iterator
            lFirst = l.begin();
  list<int>::iterator
            lLast = l.end();
  while(lFirst != lLast)
     cout << *lFirst++ << " ";
  system(“PAUSE”)
  return EXIT_SUCCESS;}
```

In each case the iterator moves from one element to the next - only the iterator cares whether via address arithmetic or a pointer.

Types of iterators

- *Input* and *output* iterators
  - Used with *istream* and *ostream* for I/O
  - Valid operations
    - `*` (dereference)
    - ++ (pre- or post-increment)
- *Forward* iterators
  - Usable with all containers
  - Valid operations as for *input* and *output*
- *Bidirectional* iterators
  - Usable with all containers
  - Valid operations as for *forward* plus
    - -- (pre- or post-decrement)
- *Random access* iterators
  - Only usable with vector or deque
  - Valid operations as for *bidirectional* plus
    `+(int), -(int)`

Operations on `vector`

- The following are the most important operations that may be applied to a Container. (Below `value_type` refers to the class or type of objects stored in the container.)

- `vector`

  `iterator begin()` Returns an iterator referring to the **first** element.
  `iterator end()` Returns an invalid iterator referring **beyond the last** element.
  `value_type front()` Returns the **first** element
  `value_type back()` Returns the **last** element
  `iterator insert(iterator, value_type)`
  Inserts the value_type **before** the specified iterator.
  `void erase(iterator)` Removes the element referred to by the iterator.
  `void push_back(value_type)` Adds the element to the **end** of the vector.
  `void pop_back()` Removes the **last** element.
  `int size()` Returns the number of elements currently in the vector.
  `bool empty()` Is the vector Empty?

Operations on `deque, list`

- `deque`
  - ***As for*** `vector` ***plus:***

  `void push_front(value_type)`
  Adds the element to the **front** of the `vector`.
  `void pop_front()`
  Removes the **first** element.

- `list`
  - ***As for*** `deque` ***plus:***

  `void sort()`
  Sorts list into ascending order.

Operations on `set, multiset`

- Set or Multiset

`iterator begin()` Returns an iterator referring to the **first** element.

`iterator end()` Returns an invalid iterator referring **beyond the last** element.

`int count(value_type)` The number of elements equal to `value_type`.

`void erase(iterator)` Remove the element pointed to by the iterator.

`int erase(value_type)`
Remove the element (or elements for Multiset) equal to `value_type`. Return the number of elements removed.

`iterator find(value_type)`
Find the element equal to `value_type` and return an iterator pointing to it – or `end()` if not found.

`pair<iterator, bool> insert(value_type)`
Insert an element equal to `value_type` and return an iterator to it and a `bool` that is `true` if the element was inserted, `false` otherwise. (In a `set` element not inserted if it matches one already in the set.)

Operations on `map, multimap`

- `map` or `multimap`

`iterator begin()` Returns an iterator referring to the **first** element.

`iterator end()` Returns an invalid iterator referring **beyond the last** element.

`int count(key_type)` The number of elements having the key `key_type` .

`void erase(iterator)` Remove the element pointed to by the iterator.

`int erase(key_type)` Remove the element (or elements for Multimap) having key `key_type`. Return the number of elements removed.

`iterator find(key_type)`
- Find the element having key `key_type` and return an iterator pointing to it – or `end()` if not found.

`pair<iterator, bool> insert(value_type)`
- Insert an element equal to `value_type` and return an iterator to it and a `bool` that is `true` if the element was inserted, `false` otherwise. (In a `map` element not inserted if it matches one already in the set.)
- For a `map` or `multimap value_type` is a `key/value` pair.

`value& operator[](key_type)`
- Use subscript-like syntax, providing a key as the subscript to return a reference to the associated value.

Container usage

- To illustrate the usage of some containers in the Container Library, the objects stored will be of the following class:

```
class Employee{
    friend ostream& operator<<(ostream& out, Employee& emp){
        out << emp.empNo << " " << emp.details << endl;
        return out;
    }
  public:
    Employee(int no = 0, char* dets = 0):empNo(no){
      if(dets != 0){
        details = new char[strlen(dets)+1];
        strcpy(details, dets);
      }
      else
        details = 0;
    }
    int operator<(const Employee& other)const{
           return empNo < other.empNo;}
     int GetEmpNo(){return empNo;};
  protected:
    int    empNo;
    char*  details;
};
```


Container usage (cont'd)

- These objects will be used in the following illustrations :

```
Employee empArr[10];
empArr[0] = Employee(123, "John");
empArr[1] = Employee(234, "Mary");
empArr[2] = Employee(100, "Elizabeth");
empArr[3] = Employee(765, "Natasha");
empArr[4] = Employee(222, "Hahai");
empArr[5] = Employee(12, "Sudesh");
empArr[6] = Employee(444, "Sujan");
empArr[7] = Employee(135, "Miyako");
empArr[8] = Employee(531, "Ulianov");
empArr[9] = Employee(333, "Do");
```

Using a `vector`

```
#include <vector.h>
#include "Employee.h"
main(){
   Employee empArr[10];
        // . . . .fill array . . . .
   vector<Employee> empVec(3);        // initial size 3
   empVec[0] = empArr[0];
   empVec[1] = empArr[1];
   empVec[2] = empArr[2];
   for(int i = 3; i < 10; i++)        // append to array –
        empVec.push_back(empArr[i]);  //   grows if necessary
   typedef vector<Employee>::iterator vIter;
                // the safe way to process container contents
   for(vIter vit = empVec.begin(); vit != empVec.end(); vit++)
        cout << *vit;
   for(int i = 0; i < 10; i++)        // if I goes outside limit,
        cout << empVec[i];            //   results undefined
}
```


Using a `vector` (output)

- Output from running previous program

```
123 John
234 Mary
100 Elizabeth
765 Natasha
222 Hahai
12 Sudesh
444 Sujan
135 Miyako
531 Ulianov
333 Do
```

Using a Set

```
#include <set.h>
#include "Employee.h“
main(){
   Employee empArr[10];
   // . . . .fill array . . . .
   set<Employee, less<Employee> > empSet;// less is "function object"
   for(int j = 0; j < 10; j++){      // items inserted in set in order
   if(!empSet.insert(empArr[j]).second)  //     specified by less
      cout << "Employee " << empArr[j] << "is a duplicate" << endl;
   }

   typedef set<Employee, less<Employee> > iterator sIter;
   for(sIter sit = empSet.begin(); sit != empSet.end(); sit++)
      cout << *sit;         // output all elements of set
   cout << endl;

   sIter sFind;               // find elements from set
   int nums[] = {123, 765, 888, 12};
   for(int k = 0; k < 4; k++){
      sFind = empSet.find(Employee(nums[k]));
      if(sFind != empSet.end())   cout << *sFind;
      else      cout << "No employee has number“ << nums[k] << endl;
   }
}
```


Using a `set` (output)

- Output from running previous program (Employee number for Sujan changed to 234 to test duplicate insertion)

```
Employee 234 Sujan
Is a duplicate
12 Sudesh
100 Elizabeth
123 John
135 Miyako
222 Hahai
234 Mary
333 Do
531 Ulianov
765 Natasha

123 John
765 Natasha
No Employee has number 888
12 Sudesh
```

Using a map

```
#include <map.h>
#include "Employee.h"
main(){
   Employee empArr[10];
   // . . . .fill array . . . .
   map<int, Employee*, less<int> > empMap;
   for(int j = 0; j < 10; j++)   // items inserted in set in order
      empMap[empArr[j].GetEmpNo()] = &empArr[j]; //spec'd by less

   typedef map<int, Employee*, less<int> > :: iterator mIter;
   for(mIter mit = empMap.begin(); mit != empMap.end(); mit++)
      cout << *((*mit).second);     // output all elements of set
   cout << endl;

   mIter mFind;
   int nums[] = {123, 765, 888, 12};
   for(int k = 0; k < 4; k++){
        mFind = empMap.find(nums[k]) // find elements from set
        if(mFind != empMap.end())     cout << *((*mFind).second);
        else  cout << "No employee has number" << nums[k] << endl;
   }
}
```


Using a map (output)

- Output from running previous program

```
12 Sudesh
100 Elizabeth
123 John
135 Miyako
222 Hahai
234 Mary
333 Do
444 Sujan
531 Ulianov
765 Natasha

123 John
765 Natasha
No Employee has number 888
12 Sudesh
```

Exception Handling

A mechanism that allows
two separately developed program components to communicate
when a program anomaly
(an error condition - an *exception*)
is encountered during execution of a program.
.

Error Handling

- When a function, for some reason, is unable to perform its designated function it can
  - Ignore the problem, possibly causing garbage results, or an Operating System initiated abort (e.g. GPF).
    - This is what happens when an out-of-range subscript is used with a C-style array.
  - Return an error code to the calling program, which can then be tested.
    - Operator new returns the invalid value 0 if there is insufficient memory. Similarly, most Windows Application Program Interface calls return 0 if the operation requested can not be performed.
  - Abort with an error message specifying the source code line that detected the error
    - *The assert() macro provides this facility.*
  - ANSI Standard C++ Exception Handling provides a more flexible solution to this problem.
    - *This will be described in some detail.*

Assertions

- Assertions provide a fairly simple means of locating error conditions

```
//#define NDEBUG
#include <assert>
. . .
assert(int-expr);
. . .
```

The expression in the assert statement is evaluated.

If it returns a non-zero value, nothing happens.

If it returns zero, the program aborts with the message

`Assertion failed:int-expr, file filename, line linenum`

Where, *`filename`* and *`linenum`* identify the position of the failing `assert()` statement in the source code.

Uncommenting the *`#define NDEBUG`* statement has the same effect as commenting out all `assert()` statements in the program file.

Assertions example

- A common example of the use of assertions:

```
#include <assert>
main(){
   . . .
   Person* pPtr = new Person(...);
   assert(pPtr != 0);
   . . .
}
```

If there is insufficient memory to allocate a new object, there is no point in continuing execution.

This error is almost certainly due to programming errors resulting in “memory leakage” – failure to delete unused objects.

Limitations of Assertions

- Assertions are principally of use during debugging of programs.
- Assertions have limitations:
  - The first assertion to fail will abort the job - it is not possible to perform any processing after this failure.
  - Apart from indicating the source code position of the failing assertion, no other information may be passed to the user to indicate values of variables that caused the failure.
  - There is a significant run time penalty associated with evaluating the condition being asserted.
- For these reasons, assertions are not suitable for informing the rest of the application that an unusual, but recoverable, condition has occurred.

Exceptions

- Exceptions provide the most flexible means for a function to inform the calling program that an unusual condition has occurred.
- As much information as is appropriate may be passed to the calling program.
- On receipt of the exception, a special piece of code in the calling program may be executed to analyse the information passed back.

  If appropriate, the calling program may take corrective action and continue execution.

  Alternatively, the calling program may elect to abort after providing any information to the user.

Using Exceptions

- The basic use of exceptions may be illustrated as follows:

```
class MyExcept{
   public:
      MyExcept(. . . );
   . . .
}
void SomeFunct(. . .)throw(MyExcept){
   if(unusualCondition)
      throw(MyExcept(. . . ));       //throw the exception
   . . .
}
main(){
   . . .
   try{                       //a try block
      SomeFunct(. . . )
   }
        //a catch block - must immediately follow a try block
   catch(MyExcept exc){
      get information from exc
   }
   . . .
}
```


Throwing an Exception

Notes on previous code outline

- It is desirable (but not required) that a function declare the types of exception objects that it may throw.
  `retType FuncName(Params)throw(type1, type2, ..)`
  A warning diagnostic occurs if any other exception is thrown.

- When a function detects an abnormal situation, it creates an object of the appropriate type, passing any appropriate parameters to the constructor.
  `throw ClassName(params);`
  A `throw` operation with this object as operand is then executed. This terminates the function in the same manner as a `return`.

Responding to an Exception

- To process the exception, the statements which call the function (either directly or indirectly), are placed in a `try` block.

  ```
  try{ . . . . . }
  ```

  If the function call results in an exception, the remaining statements in the `try` block are by-passed.

- Immediately following the try block are one or more catch blocks.

  ```
  catch(type name){......}
  ```

  If an exception occurs within the try block, the catch block with the matching type is executed.

Exceptions – an example

```
#include <iostream>
class TooBig{
   public:
      TooBig(int v) : val(v) {};
      int val;
};
class TooSmall{
   public:
      TooSmall(int v) : val(v) {};
      int val;
};
void UseInt(int xx)throw(TooSmall,
   TooBig){
   if(xx > 10)
        throw(TooBig(xx));
   if(xx < 5)
        throw(TooSmall(xx));
}
```

```
main(){
   for(int i = 1; i < 15; i++){
      try{
         UseInt(i);
         cout << “i = “ << i
               << endl;
       }
       catch(TooBig& exc){
          cout << “Too big: “
             << exc.val << endl;
       }
       catch(TooSmall& exc){
          cout <<“Too small: “
             << exc.val << endl;
       }
   }
}
```