Overloaded Function (P.237)

- Recall "function overloading" on P.237
- We have three functions with the same name:
  - int max(int x[], int len);
  - long max(long x[], int len);
  - double max(double x[], int len);
- The code is essentially the same for each function, but with different parameter types.

Ex6_07.cpp

```
// Maximum of ints
int max(const int x[],const size_t len)
{
  int maximum = x[0] ;
  for (size_t i=1; i < len; i++)
      if(maximum < x[i])
         maximum = x[i];
   return maximum;
}

// Maximum of longs
long max(const long x[], const size_t len)
{
  long maximum = x[0] ;
  for (size_t i=1; i < len; i++)
      if(maximum < x[i])
         maximum = x[i];
   return maximum;
}
```

```
// Maximum of doubles
double max(const double x[], const size_t len)
{
  double maximum = x[0] ;
  for (size_t i=1; i < len; i++)
      if(maximum < x[i])
         maximum = x[i];
   return maximum;
}
```

Function Templates (P.241)

- C++ compiler can automatically generate functions with various parameter types.
- The functions generated by a function template all have the same basic code, but customized by the type arguments that you supply.

Function Templates (P.241)

type parameter

```
template <typename T>
T max (const T x[], int len)
{
    T maximum = x[0] ;
    for (size_t i = 1; i < len; i++)
        if (maximum < x[i])
            maximum = x[i];
    return maximum;
}
```

Instantiation

T is replaced by a specific type argument, such as “long”

```
long max (const long x[], int len)
{
    long maximum =x[0];
    for (size_t i = 1; i < len; i++)
        if (maximum < x[i])
            maximum = x[i];
    return maximum;
}
```

Ex6_08.cpp (P.243)

```
#include <iostream>
using std::cout;
using std::endl;

// Template for function to compute the maximum element of an array
template<typename T> T max(T x[], int len)
{
   T maximum(x[0]);
   for(int i = 1; i < len; i++)
      if(maximum < x[i])
         maximum = x[i];
   return maximum;
}
```

Ex6_08.cpp (cont.)

```
int main(void)
{
   int small[] = { 1, 24, 34, 22};
   long medium[] = { 23, 245, 123, 1, 234, 2345};
   double large[] = { 23.0, 1.4, 2.456, 345.5, 12.0, 21.0};

   int lensmall(sizeof small/sizeof small[0]);
   int lenmedium(sizeof medium/sizeof medium[0]);
   int lenlarge(sizeof large/sizeof large[0]);

   cout << endl << max(small, lensmall);
   cout << endl << max(medium, lenmedium);
   cout << endl << max(large, lenlarge);

   cout << endl;
   return 0;
}
```

Class Templates (P.362)

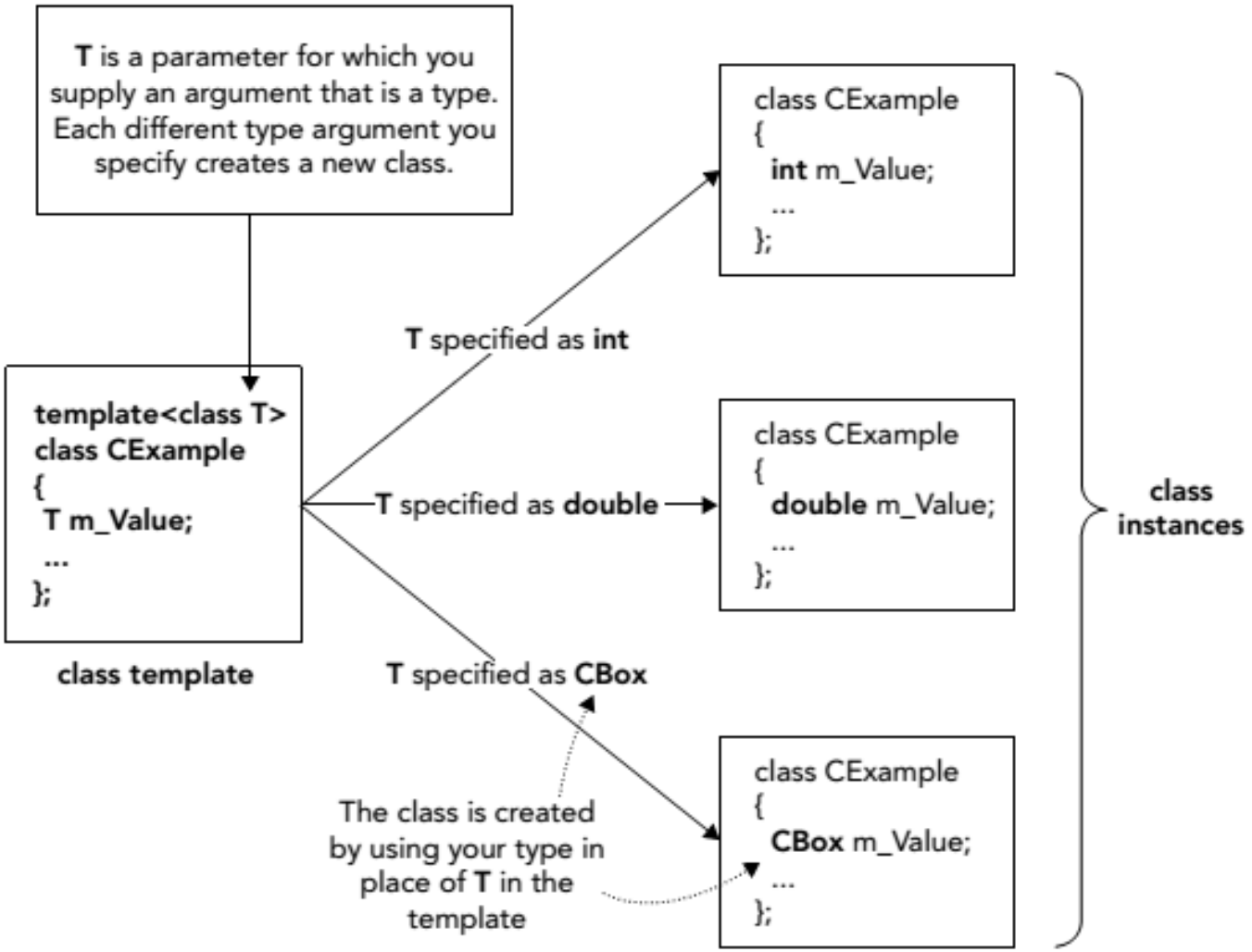


FIGURE 8-4

Chapter 10

Standard Template Library (STL)

The Standard Template Library (P.479)

- The STL is a large collection of class templates and function templates provided with your native C++ compiler.
- STL provides class templates that relieves you of the trouble of memory management.
- In the class of Data Structure, you will learn how to implement these by yourselves:
  - vector, list, hash, set, queue, stack

Some useful STL class templates

- vector
  - an array that can increase in capacity automatically when required.
  - You can only add new elements efficiently to the end of a vector.
- deque
  - Double-ended queue
  - Equivalent to a vector but you can add elements to the beginning efficiently.
- list
  - a doubly-linked list

Figure 10-2 (P.491)

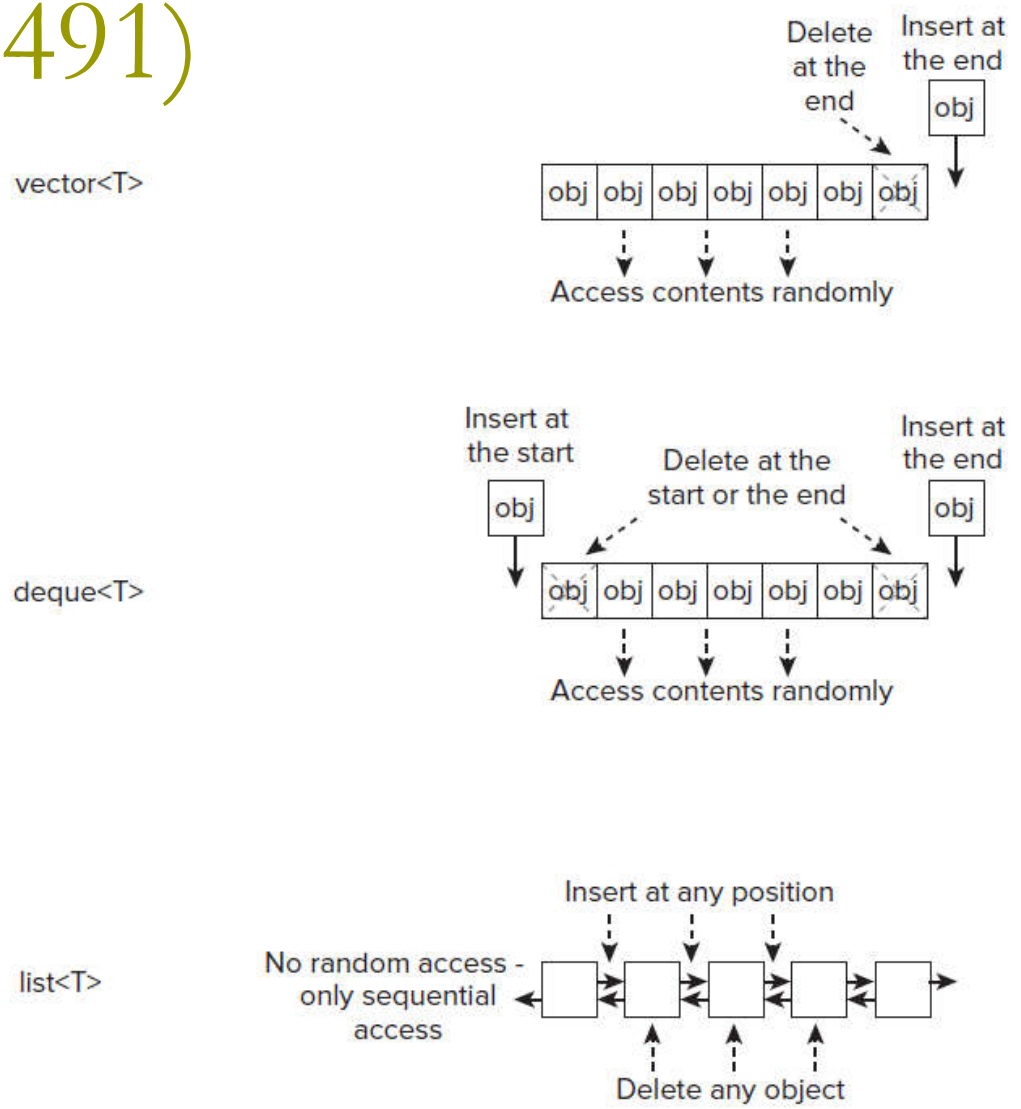


FIGURE 10-2

Creating Vector Containers

- Containers (P.480) are objects that you use to store and organize other objects.
  - an int vector – an array of integers
  - a string vector – an array of strings
- Instead of defining `CInt_Vector`, `CFloat_Vector`, `CString_Vector`, you have a class template vector<T>.
- Creating `vector<T>` containers:
  - `vector<string> strings;`
  - `vector<double> data;`
  - `vector<int> mydata;`

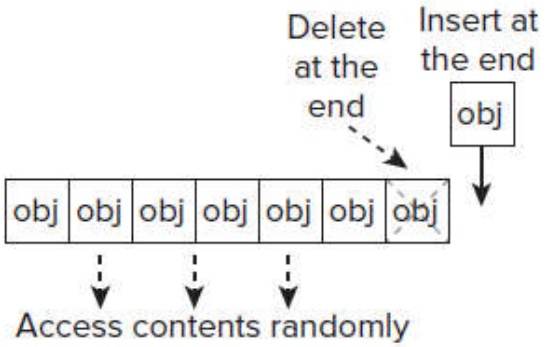
A Simple Example

```
#include <iostream>
#include <vector>

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

int main()
{
    vector<int> mydata;
    mydata.push_back(98);
    mydata.push_back(99);
    mydata.push_back(100);

    for (size_t i=0; i<mydata.size(); i++)
        cout << mydata[i] << ' ';
    cout << endl;
    return 0;
}
```



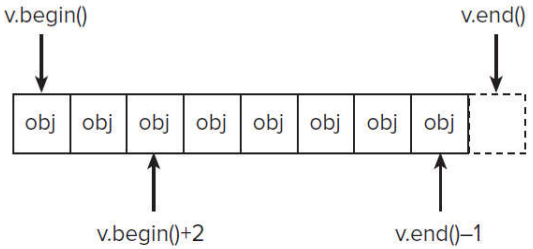
Visual C++ implements this as a 4-byte unsigned integer
On lilina, g++ implements this as an 8-byte unsigned integer

Create a Vector

- vector<int> mydata;
  mydata.push_back(99);
  - If you add new elements to this vector, the memory allocated for the vector will be increased automatically.
- vector<int> mydata(100);
  - Create a vector that contains 100 elements that are all initialized to zero.
- vector<int> mydata(100, -1);
  - Create a vector that contains 100 elements that are all initialized to -1.

Create a Vector (cont.)

- Create a vector with initial values for elements from an external array.
  - ```
    int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> mydata(data, data+8);
    // mydata will contain 1 2 3 4 5 6 7 8
    ```
  - You may also specify
    ```
    vector<int> hisdata(mydata.begin()+1, mydata.end()-1);
    // hisdata will contain 2 3 4 5 6 7
    ```
    - A sequence of elements is specified in the STL by two iterators (imagine iterators as something like pointers, P.483):
      - begin() pointing to the first element in the sequence
      - end() pointing to one past the last element in the sequence



Iterators for a vector v

FIGURE 10-3
    ```

Capacity and Size of a Vector Container

- capacity() – the maximum number of objects a container can currently accommodate.
- size() – the number of objects stored in the container.
- empty() – Boolean function which tests whether size() is 0.
- Both values are returned as type vector<T>::size_type
  - In Visual C++, this is implemented as size_t, a 4-byte unsigned integer.

Resize a Vector

```
vector<int> values(5, 66); // 66 66 66 66 66
values.resize(7,88);       // 66 66 66 66 66 88 88
values.resize(10);         // 66 66 66 66 66 88 88 0 0 0
    // If you don't specify a value for new elements,
    // the default value of the object will be produced.
values.resize(4);          // 66 66 66 66
```

Accessing Elements in a Vector

- Use the subscript operator, just like an array
  - cout << numbers[i];
- Use the at() function where the argument is the index position.
  - cout << numbers.at(i);

Inserting and Deleting Elements in a Vector (P.500)

- push_back();
- pop_back();
- clear();
  - the size will be 0; the capacity will be left unchanged.
- insert();
- erase();
  - Both the erase() and insert() operations are slow.

size() and push_back()

```
#include <iostream>
#include <vector>
using std::vector;

void Print(vector<int>& V) {
    for (size_t i=0; i<V.size(); ++i)
        std::cout << V.at(i) << ' ';
    std::cout << std::endl;
}

int main()
{
    vector<int> A;
    for (int i=0; i<6; ++i)
        A.push_back(i*10);
    Print(A);
    return 0;
}
```

1. pass-by-reference
2. runs faster if the vector is very large

insert() and erase()

```
int main()
{
    vector<int> A;
    for (int i=0; i<6; ++i)
        A.push_back(i*10);
    A.insert(A.begin() + 3, 33);
    A.erase( A.begin() + 1 );

    Print(A);
    return 0;
}
```

Sorting Vector Elements

- sort(data, data+count);
  - Note the pointer marking the end of the sequence of elements must be one past the last element.
- sort(data, data+count, greater<int>());
  - Sorting in descending order.
  - Remember to "using std::greater"

Sorting a Vector of CRational

```
#include <iostream>
#include <vector>
#include <algorithm>
using std::sort;
using std::vector;

class CRational {
public:
    int numerator;
    int denominator;

    CRational(int n = 0, int d = 1): numerator(n), denominator(d) {}
};

void Print(const vector<CRational>& A) {
    for (size_t i=0; i<A.size(); ++i)
        std::cout << A[i].numerator << '/' << A[i].denominator << std::endl;
}
```

initializer list

Define Your Comparison Function

```
bool less(const CRational& a, const CRational& b) {
    return a.numerator < b.numerator;
}

int main()
{
    vector<CRational> A;
    A.push_back( CRational(1, 2) );
    A.push_back( CRational(3, 4) );
    A.push_back( CRational(5, 6) );
    A.push_back( CRational(7, 1) );
    A.push_back( CRational(2, 3) );
    A.push_back( CRational(4, 5) );
    A.push_back( CRational(6, 7) );
    Print(A);
    std::cout << std::endl;

    sort(A.begin(), A.end(), less);
    Print(A);
}
```

Or Use the Member Function `operator<()`

```
using std::less;

bool operator<(const CRational& b) const {
        return numerator*b.denominator <
                b.numerator*denominator;

sort(A.begin(), A.end(), less<CRational>());
```

How to count the number of occurrence of each object?

(Suppose you don't have a list of objects in advance.)

- apple
- apple
- coconut
- coconut
- apple

- apple
- apple
- coconut
- coconut
- apple

- apple
- apple
- coconut
- coconut
- apple

name	count
apple	1

- apple
- apple
- coconut
- coconut
- apple

name	count
apple	2

- apple
- apple
- coconut
- coconut
- apple

name	count
apple	2
coconut	1

- apple
- apple
- coconut
- coconut
- apple

name	count
apple	2
coconut	2

- apple
- apple
- coconut
- coconut
- apple

name	count
apple	3
coconut	2

Associate Containers (P.543)

- map<K, T>
  - You don't need to search the whole list to retrieve a particular object.
  - The location of an object of type T within an associative container is determined from a key of type K.
  - The mechanism is also known as dictionary or hash.
- The objects you store in a map are always a key/object pair
  - The template type pair<K,T> is automatically included by the map header.

A Simple Example

```
#include <iostream>
#include <map>
#include <string>

using std::cout;
using std::endl;
using std::string;
using std::map;

int main()
{
    map<string, int> score;
    score["Alice"] = 100;
    score["Bob"] = 70;
    score["Charlie"] = 60;

    cout << score["Bob"] << endl;
    cout << score.size() << endl;
    return 0;
}
```

fruit_count

```
#include <iostream>
#include <string>
#include <map>
using std::cin;
using std::cout;
using std::endl;
using std::string;
using std::map;

int main()
{
    string fruit;
    map<string, int> fruit_count;
    for (int i=0; i<5; i++)
    {
        cin >> fruit;
        if (fruit_count.find(fruit) == fruit_count.end() )
            fruit_count[fruit] = 1;
        else
            fruit_count[fruit]++;
    }

    for (map<string, int>::iterator i = fruit_count.begin();
        i != fruit_count.end(); i++)
    {
        cout << i->first << '\t' << i->second << endl;
    }
    return 0;
}
```

key

value

```
Apple
Banana
Apple
Apple
Coconut
```

```
Apple   3
Banana  1
Coconut 1
```

Iterate Through a Map

```
#include <iostream>
#include <string>
#include <map>
using std::string;
using std::map;
using std::cout;
using std::endl;

int main()
{
    map<string, int> m;
    m["Alice"] = 10;
    m["Bob"] = 20;
    m["Carol"] = 30;
    for (map<string,int>::iterator it = m.begin(); it != m.end(); ++it)
        cout << it->first << "\t: " << it->second << endl;
    for (auto it = m.begin(); it != m.end(); ++it)
        cout << it->first << "\t: " << ++it->second << endl;
    for (auto item : m)
        cout << item.first << "\t: " << ++item.second << endl;
    for (auto& [key, value] : m)
        cout << key << "\t: " << value << endl;

    return 0;
}
```

Accessing Objects

- Use the subscript operator to retrieve the object from a map corresponding to a key
  - string number = phonebook[Person("Jack", "Jones")];
- However, if the key is not in the map, the subscript operator will insert a pair entry for this key, with an object of default value.
  - You could use the `find()` function to check whether there is an entry for a given key.

find()

```
string number;
Person key = Person("Jack", "Jones");
auto iter = phonebook.find(key);

if (iter != phonebook.end())
{
    number = iter->second;
    cout << "The number is " << number << endl;
}
else
{
    cout << "No number for the key ";
    key.showPerson();
}
```