More C++

The Standard Template Library

The Standard Template Library (STL) includes generic libraries for such data structures as stacks, queues, etc. Here we give a brief introduction through some examples. The STL classes are often referred to as *container* classes. Although the STL is not part of the core C++ language, it is part of the C++ standard, and so any implementation of C++ that conforms to the standard includes the STL. As a practical matter, you can consider the STL to be part of the C++ language.

The STL uses **iterators** to access items in a container class. An iterator is not a pointer, but you will not go far wrong if you think of it and use it as if it were a pointer. Like a pointer variable, an iterator variable is located at (points to) one data entry in the container. You manipulate iterators using the following overloaded operators that apply to iterator objects:

- Prefix and postfix increment operators (++) for advancing the iterator to the next data item.
- Prefix and postfix decrement operators (--) for moving the iterator to the previous data item.
- Equal and unequal operators (== and !=) to test whether two iterators point to the same data location.
- A dereferencing operator (*) so that if p is an iterator variable, then *p gives access to the data located at (pointed to by) p. This access may be read only, write only, or allow both reading and changing of the data, depending on the particular container class.

A container class has member functions that get the iterator process started.

- c.begin( ) returns an iterator for the container c that points to the “first” data item in the container c.
- c.end( ) returns something that can be used to test when an iterator has passed beyond the last data item in a container c. The iterator c.end( ) is completely analogous to NULL .The iterator c.end( ) is thus an iterator that is not located at a data item but that is a kind of end marker or sentinel.

Here is a program to demonstrate the vector class, which is like an arraylist in Java.

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

int main( )
{
    vector<int> container;

    for (int i = 1; i <= 4; i++)
        container.push_back(i);
```

```
    cout << "Here is what is in the container:\n";
    vector<int>::iterator p;
    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    cout << "Here is what is in the container:\n";
    for (int i = 0; i < container.size(); i++)
        cout << container[i] << " ";
    cout << endl;

    cout << "Setting entries to 0:\n";
    for (p = container.begin( ); p != container.end( ); p++)
         *p = 0;
    cout << "Container now contains:\n";

    for (p = container.begin( ); p != container.end( ); p++)
        cout << *p << " ";
    cout << endl;

    return 0;
}
```

There is also a list STL class, but the vector does essentially the same things as the list.

There is also a Stack class:

```
//Program to demonstrate use of the stack template class from the STL.
#include <iostream>
#include <stack>
using std::cin;
using std::cout;
using std::endl;
using std::stack;

int main( )
{
    stack<char> s;

    cout << "Enter a line of text:\n";
    char next;
    cin.get(next);
```

```
    while (next != '\n')
    {
        s.push(next);
        cin.get(next);
    }

    cout << "Written backward that is:\n";
    while ( ! s.empty( ) )
    {
        cout << s.top( );
        s.pop( );
    }
    cout << endl;

    return 0;
}
```

The STL includes classes for sets and maps. The map class is like an associate array where we can associate items of any data type to items of any other data type. Here is an example where we map from string→string:

```
//Program to demonstrate use of the map template class.
#include <iostream>
#include <map>
#include <string>
using std::cout;
using std::endl;
using std::map;
using std::string;

int main( )
{
    map<string, string> planets;

    planets["Mercury"] = "Hot planet";
    planets["Venus"] = "Atmosphere of sulfuric acid";
    planets["Earth"] = "Home";
    planets["Mars"] = "The Red Planet";
    planets["Jupiter"] = "Largest planet in our solar system";
    planets["Saturn"] = "Has rings";
    planets["Uranus"] = "Tilts on its side";
    planets["Neptune"] = "1500 mile per hour winds";
```

```
    planets["Pluto"] = "Dwarf planet";

    cout << "Entry for Mercury - " << planets["Mercury"]
         << endl << endl;

    if (planets.find("Mercury") != planets.end())
      cout << "Mercury is in the map." << endl;
    if (planets.find("Ceres") == planets.end())
      cout << "Ceres is not in the map." << endl << endl;

    // Iterator outputs planets in order sorted by key
    cout << "Iterating through all planets: " << endl;
    map<string, string>::const_iterator iter;
    for (iter = planets.begin(); iter != planets.end(); iter++)
    {
      cout << iter->first << " - " << iter->second << endl;
    }

    return 0;
}
```

PITFALL: Underlying Containers

If you specify an underlying container, be warned that you should not place two > symbols in the type expression without a space in between them, or the compiler can be confused.

Use `map<int, vector<int> >`, with a space between the last two >'s.

Do not use `map<int, vector<int>>`.

Overloading the Assignment Operator

Here is an example illustrating a shallow copy when we use assignment on objects without an assignment operator.

```
class Overload
{
  public:
    Overload();
    Overload(int num1, int num2);
    ~Overload();
    void print();
    int num1, *num2;
};
```

```
#include <iostream>
#include "overload.h"

using namespace std;

Overload::Overload()
{
   num1 = 0;
   num2 = new int;
   *num2 = 0;
}

Overload::Overload(int num1, int num2)
{
   this->num1 = num1;
   this->num2 = new int;
   *(this->num2) = num2;
}

Overload::~Overload()
{
   delete num2;
}

void Overload::print()
{
        cout << num1 << " " << *num2 << endl;
}

int main()
{
        Overload o1, o2(3,4);
        o1.print();
        o2.print();

        o1 = o2;
        *o1.num2 = 100;
        o1.print();
        o2.print();

        return 0;
}
```

This implementation has a couple of problems – it does a shallow copy of the pointer (o2.print outputs 100) and we end up deleting num2 twice in the destructor.

One way to avoid this problem is to perform a deep copy in the assignment operator. We can overload the assignment operator as follows:

Add to the .h file:

```
Overload& operator=(const Overload &rtSide);
```

Add to the .cpp file:

```
Overload& Overload::operator=(const Overload &rtSide)
{
    if (this == &rtSide)
    //if the right side is the same as the left side
    {
        return *this;
    }
    else
    {
        // Copy values, following the pointer
        num1 = rtSide.num1;
        *num2 = *rtSide.num2;
        return *this;
        /* Alternate version would deallocate/allocate
         * if we had a linked data structure
        num1 = rtSide.num1;
        delete num2;
        num2 = new int;
        *num2 = *rtSide.num2;
        return *this;
        */
    }
}
```

This new version makes a proper copy of the object when assigned and avoids the double delete problem.

Finally, here is an example program that uses Posix threads. This is a C library that allows us to run code in separate threads, possibly in parallel.

```
/* Need to compile and link with -pthread */
#include <iostream>
#include <pthread.h>

using namespace std;

// This structure passes data to and from the thread
typedef struct argdata
{
  int num1, num2;
  int return_val;
};

// Code to run in a thread
void *TaskCode(void *argument)
{
   argdata *p;

   p = (argdata *) argument;
   int n1, n2;
   n1 = p->num1;
   n2 = p->num2;
   p->return_val = (n1 + n2);  // Time intensive computation

   return NULL;
}

int main ()
{
   pthread_t thread1,thread2;
   argdata arg1, arg2;

   /* create two threads */
   arg1.num1 = 1;
   arg1.num2 = 2;
   arg2.num1 = 3;
   arg2.num2 = 4;
   pthread_create(&thread1, NULL, TaskCode, (void *) &arg1);
   pthread_create(&thread2, NULL, TaskCode, (void *) &arg2);
   /* wait for all threads to complete */
   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
```

```
   cout << "Done! Return values: " << arg1.return_val << " "
                        << arg2.return_val << endl;
   return 0;
}
```

Compiling on bigmazzy: g++ pthread.cpp –pthread