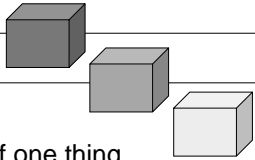


Templates



• Polymorphism

- Multiple variants of one thing
 - E.g., integer or floating "+"

• Can we do it too?

- User-defined operations (functions)
- User-defined types (classes)
 - Basis of Standard Template Library

Function Example

```
int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

```
float Max (float a, float b)
{
  return (a > b) ? a : b;
}
```

Similar, but different, functions

Simple Template Function

```
template <class T>
```

```
T Max (T a, T b)
{
  return (a > b) ? a : b;
}
```

Compiler "instantiates" template for each type, as needed

```
float x;
float y;
float z = Max (x,y);
```

Template Functions

- **Write basic function**
 - Test with common type (e.g., "int")
- **Replace type identifiers**
 - E.g., change all "int" to "T"
 - "T" not special, just a name
- **Add template prefix**
 - E.g., template<class T>
 - Or template<typename T>

Template Class

- **Similar to template function**
- **But generates an entire class**
 - Data members
 - Member functions
- **Example**
 - Generalize simple vector class

Vector Class Definition (1)

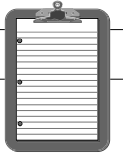
```

template<class T>
class Vec
{
public:
    Vec();
    explicit Vec(unsigned int sz,
                 const T& val = T());
    Vec(const Vec<T>& v);
    ~Vec();
    void push_back(const T& val);
    unsigned int size() const;
    bool empty() const;
    void clear();

```

Replace "int" with "T"

Default value



Vector Class Definition (2)

```

...
T& operator[](unsigned int indx);
const T& operator[]
(unsigned int indx) const;
Vec<T>& operator=
(const Vec<T>& right);
private:
unsigned int array_size;
unsigned int num_values;
T* array;
void CopyData (const Vec<T>& right);
};

```

Constructor Implementation

```

template<class T>
Vec<T>::Vec()
:array_size(0),num_values(0),array(0)
{ }

```

```

Vec<T>::Vec(unsigned int sz,const T& val)
:array_size(sz),num_values(sz),array(0)
{...
array = new T [array_size];
for (unsigned int indx = 0;
indx < num_values; indx++)
array[indx] = val;
...}

```

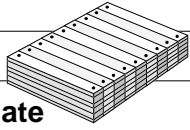
Template Member Function

```

template<class T>
void Vec<T>::push_back(const T& val)
{
if (array_size < (num_values + 1))
{
T* old_array = array;
if (array_size == 0) array_size = 1;
array_size *= 2;
array = new T [array_size];
for (unsigned int indx = 0;
indx < num_values; indx++)
array[indx] = old_array[indx];
delete [] old_array;
}
array[num_values++] = val;
}

```

Template Location



• **Compiler needs template**

- To generate instances
- At compile time

• **Template must be accessible**

- In header (".h") file
- Even function definitions!!
 - At least in current compilers

Template Summary



• **Write "generic" code**

- Functions or classes

• **Compiler specializes**

- Generates code for specific variant
- All template code in header file
- Duplicate code managed by compiler
 - But, template bloat ? . . .

Advanced Template Features

• **Non-type template parameters**

- See text, pages 717-718
 - `template <class T, int n>`



• **Template specializations**

- Defined implementations for specific variations (template parameters)
- Partial specializations (e.g., all pointers)

• **Too much to discuss now**

```
// vectmpl.h - Templated Vec container class.
//
// Mark Sebern, MSOE
// Version 1.0
// 23 April 1999

#ifndef VECTMPL_H
#define VECTMPL_H

#include <cassert>

template<class T>
class Vec
{
public:
    // No-argument constructor
    Vec();

    // Constructor
    // Arguments:
    //   sz - Number of elements to initialize in vector.
    //   val - Value to assign to initial elements.
    explicit Vec(unsigned int sz, const T& val = T());

    // Copy constructor
    Vec(const Vec<T>& v);

    // Destructor
    ~Vec();

    // push_back - add element to end of vector.
    // Arguments:
    //   val - value to add to end of vector.
    //   Note: passed by const reference since this
    //   value may not be a small built-in type.
    void push_back(const T& val);

    // size - return number of elements in vector.
    // Returns: current number of vector elements.
    unsigned int size() const;

    // empty - test if vector has no elements.
    // Returns: true if number of elements is zero; false otherwise.
    bool empty() const;

    // clear - delete all elements from vector
    void clear();

    // Non-const version of subscript operator.
    // Arguments:
    //   indx - subscript value.
    // Returns: reference to specified element.
    // Precondition: index must be between 0 and size-1.
    T& operator[](unsigned int indx);

    // Const version of subscript operator.
    // Arguments:
    //   indx - subscript value.
    // Returns: const reference to specified element.
    // Precondition: index must be between 0 and size-1.
    const T& operator[](unsigned int indx) const;

    // Assignment operator
    Vec<T>& operator= (const Vec<T>& right);
private:
    // Number of possible element positions currently in
    // array.
    unsigned int array_size;

    // Number of actual elements currently stored in array.
    unsigned int num_values;

    // Pointer to dynamic array of elements.
    T* array;
};
```

```
// CopyData - helper function to copy another array.
// Arguments:
//   right - array to copy values from.
// Precondition: "array" and "array_size" must be correct
//   and consistent.
//   If "array" is null, "array_size" must be zero.
//   If "array_size" is not zero, "array" must point to
//   a valid dynamic array of "array_size" elements.
void CopyData (const Vec<T>& right);
};

// Template class implementation

template<class T>
Vec<T>::Vec()
: array_size(0), num_values(0), array(0)
{
    // No further initialization required.
}

template<class T>
Vec<T>::Vec(unsigned int sz, const T& val)
: array_size(sz), num_values(sz), array(0)
{
    // The initializer list above sets the current
    // and maximum number of elements to the specified
    // size, and initializes the array pointer to null.

    // If a non-zero number of elements, try to allocate and
    // initialize the array.
    if (array_size > 0)
    {
        array = new T [array_size];
        assert (array != 0);

        for (unsigned int indx = 0; indx < num_values; indx++)
        {
            array[indx] = val;
        }
    }
}

template<class T>
Vec<T>::Vec(const Vec<T>& v)
: array_size(0), num_values(0), array(0)
{
    // Call the helper routine to copy the data.
    // Note that the data member initializations above
    // are required for CopyData to work properly.
    CopyData(v);
}

template<class T>
Vec<T>::~~Vec()
{
    // Destroy the dynamic array if there is one.
    delete [] array;
}

template<class T>
void Vec<T>::push_back(const T& val)
{
    // Check to see if the currently allocated dynamic
    // array is big enough to hold another element.
    if (array_size < (num_values + 1))
    {
        // If not, we need to reallocate the dynamic array.
        // First, set a pointer to keep track of the current
        // array.
        T* old_array = array;

        // Next, allocate a new array that is bigger.
        // To keep from doing this so often, we'll make the
        // new array twice as big as the old one.
        // If we didn't really have an array before (array_size == 0),
```

```
// then we'll pretend it was one element long.
if (array_size == 0) array_size = 1;
array_size *= 2;

array = new T [array_size];
assert (array != 0);

// Now we must copy over the previous data.
for (unsigned int indx = 0; indx < num_values; indx++)
{
    array[indx] = old_array[indx];
}

// Next, delete the previous array, if there was one.
delete [] old_array;
}

// One way or the other, now, the array should be big enough.
// Put the new value at the end of the array and increment
// the number of values.
assert (num_values < array_size);
array[num_values++] = val;
}

template<class T>
unsigned int Vec<T>::size() const
{
    // Return the number of actual elements.
    return num_values;
}

template<class T>
bool Vec<T>::empty() const
{
    // Return true if no elements, false otherwise.
    return (num_values == 0);
}

template<class T>
void Vec<T>::clear()
{
    // Just clear out the element count.
    // Don't disturb the dynamic array if there is one,
    // since we may be able to reuse it.
    num_values = 0;
}

template<class T>
T& Vec<T>::operator[](unsigned int indx)
{
    // Verify the precondition.
    assert (indx < num_values);

    // Do some extra checking (should not be necessary).
    assert (array != 0);
    assert (indx < array_size);

    // Return a reference to actual array element.
    return array[indx];
}

template<class T>
const T& Vec<T>::operator[](unsigned int indx) const
{
    // Verify the precondition.
    assert (indx < num_values);

    // Do some extra checking (should not be necessary).
    assert (array != 0);
    assert (indx < array_size);

    // Return a const reference to actual array element.
    return array[indx];
}
```

```
template<class T>
Vec<T>& Vec<T>::operator= (const Vec<T>& right)
{
    // Do assignment only if not assigning to self.
    // (Self-assignment would be relatively harmless here,
    // since CopyData would not reallocate an array that
    // was already the right size, but in general it is
    // dangerous to permit self-assignment. In any case,
    // it would be a waste of time and effort.)
    if (this != &right)
    {
        // Copy the data, reallocating the dynamic array if
        // necessary.
        CopyData(right);
    }
    // Return reference to self, permitting chained assignment.
    return *this;
}
```

```
template<class T>
void Vec<T>::CopyData (const Vec<T>& right)
{
    // If we don't have a dynamic array, or it isn't big
    // enough, make a new one.
    if (array_size < right.num_values)
    {
        // Delete the dynamic array if we have one.
        delete [] array;

        // Allocate a new dynamic array.
        array = new T [right.num_values];
        assert (array != 0);

        // Remember the new array size.
        array_size = right.num_values;
    }

    // Set the number of elements and copy over the data.
    num_values = right.num_values;
    for (unsigned int indx = 0; indx < num_values; indx++)
    {
        array[indx] = right.array[indx];
    }
}
```

```
#endif // VECTMPL_H
```

```
// testvect.cpp
// Test program for Vec templated container class.
//
// Mark Sebern, MSOE
// Version 1.0
// 23 April 1999

#include <iostream>
#include <fstream>
#include <string>
// #include <vector>
using namespace std;

#include "vectmpl.h"

// Local function prototypes:

// Note: the following functions are templated to handle vectors of
// different types, to demonstrate the use of templated functions.

// OutputVec
// Output a vector to a stream.
// Arguments:
//   os - output stream
//   vec - vector to output
//   delim - string to output after each element
template<class U>
void OutputVec (ostream& os, const Vec<U>& vec, const string& delim);

// OutputVecInfo
// Output vector information to a stream.
// Arguments:
//   os - output stream
//   vec - vector to describe
//   label - string used to identify output
template<class U>
void OutputVecInfo (ostream& os, const Vec<U>& vec, const string& label);

// ProcessVec
// Silly function to manipulate a vector passed by value.
// Arguments:
//   os - output stream for messages
//   vec - vector to manipulate
template<typename U> // Same as "template<class U>"
void ProcessVec (ostream& os, Vec<U> vec);

void main()
{
    // Create output data file.
    ofstream out("testvect.txt");

    // Define vectors of integers.
    Vec<int> vec_one(4,22);
    Vec<int> vec_two;
    Vec<string> vec_three;

    // Define some int data objects.
    int a = 2;
    int b = -5;

    // Put some elements into the vectors.
    vec_one.push_back(a);
    vec_one.push_back(9);
    vec_one.push_back(b);

    vec_two.push_back(vec_one[1]);
    vec_two.push_back(33);

    vec_three.push_back("Hello");
    vec_three.push_back("there");

    for(unsigned int indx = 0; indx < 12; indx++)
    {
        vec_two.push_back(indx*indx);
    }
}
```

```

out << "vec_one (A) = [";
OutputVec (out, vec_one, " ");
out << ']' << endl;

OutputVecInfo (out, vec_one, "vec_one");

out << "vec_two (A) = [";
OutputVec (out, vec_two, " ");
out << ']' << endl;

OutputVecInfo (out, vec_two, "vec_two");

out << "vec_three (A) = [";
OutputVec (out, vec_three, " ");
out << ']' << endl;

OutputVecInfo (out, vec_three, "vec_three");

// Test vector assignment.
vec_two = vec_one;

// Modify some vector elements.
vec_one[1] = vec_one[0] + 10;
ProcessVec(out, vec_two);
ProcessVec(out, vec_three);

// Write out vectors.
out << "vec_one (B) = [";
OutputVec (out, vec_one, " ");
out << ']' << endl;

OutputVecInfo (out, vec_one, "vec_one(B)");

out << "vec_two (B) = [";
OutputVec (out, vec_two, " ");
out << ']' << endl;

OutputVecInfo (out, vec_two, "vec_two(B)");

vec_one.clear();

out << "vec_three (B) = [";
OutputVec (out, vec_three, " ");
out << ']' << endl;

OutputVecInfo (out, vec_three, "vec_three");

OutputVecInfo (out, vec_one, "vec_one after clear");
}

template<class U>
void OutputVec (ostream& os, const Vec<U>& vec, const string& delim)
{
    for (unsigned int indx = 0; indx < vec.size(); indx++)
    {
        // Note that the 'const' form of the subscript operator
        // is used here, since we are accessing the vector by
        // const reference.
        os << vec[indx] << delim;
    }
}

template<class U>
void OutputVecInfo (ostream& os, const Vec<U>& vec, const string& label)
{
    os << "-----" << endl;
    os << "Vec " << label << ":" << endl;
    os << "Size is " << vec.size() << endl;
    os << "Vector is ";
    if (!vec.empty()) os << "NOT ";
    os << "empty." << endl;
    os << "-----" << endl;
}

```

```
template<typename U>
void ProcessVec (ostream& os, Vec<U> vec)
{
    unsigned int vec_size = vec.size();

    if (vec_size > 0)
    {
        U temp = vec[0];
        vec[0] = vec[vec_size-1];
        vec[vec_size-1] = temp;
    }

    os << "ProcessVec: vec = [ ";
    OutputVec(os, vec, " ");
    os << "]" << endl;

    OutputVecInfo(os,vec,"ProcessVec vec");
}
```

```
vec_one (A) = [22 22 22 22 2 9 -5 ]
-----
Vec vec_one:
Size is 7
Vector is NOT empty.
-----
vec_two (A) = [22 33 0 1 4 9 16 25 36 49 64 81 100 121 ]
-----
Vec vec_two:
Size is 14
Vector is NOT empty.
-----
vec_three (A) = [Hello there ]
-----
Vec vec_three:
Size is 2
Vector is NOT empty.
-----
ProcessVec: vec = [ -5 22 22 22 2 9 22 ]
-----
Vec ProcessVec vec:
Size is 7
Vector is NOT empty.
-----
ProcessVec: vec = [ there Hello ]
-----
Vec ProcessVec vec:
Size is 2
Vector is NOT empty.
-----
vec_one (B) = [22 32 22 22 2 9 -5 ]
-----
Vec vec_one(B):
Size is 7
Vector is NOT empty.
-----
vec_two (B) = [22 22 22 22 2 9 -5 ]
-----
Vec vec_two(B):
Size is 7
Vector is NOT empty.
-----
vec_three (B) = [Hello there ]
-----
Vec vec_three:
Size is 2
Vector is NOT empty.
-----
-----
Vec vec_one after clear:
Size is 0
Vector is empty.
-----
```