Generic Programming

- Certain kinds of functions or data structures are "generic" in the sense that they can be used for many data types.
- Minimum/Maximum: you can compute it as long as you can compare two items
- Sorting: you can sort as long as you can compare two items
- Vectors: you can have vectors of ints or vectors of strings.
- We do not want to rewrite similar (sometimes the same) code for different types.
- In C++, we use templates to write the code only once.

Class Template

- The vector is a class template: you can define the element type using <...>. e.g. vector<int>, vector<string>.
- The two vector classes are different classes, but they have the same member functions.
- The vector class template is only defined once. It is **instantiated** twice.

Class Template Syntax

The syntax for a class template is

```
template<typename T>
class Name {
  ...
};
```

Inside the class, `T` is used as a parameter and refers to a type name.

Example

Recall our `DArray` example:

```
template<typename T>
class DArray {
public:
  DArray(int size = 10);
    ...
private:
  int n;
  T *A;     // elements are type T
};
```

Then we can declare variables of type `DArray<int>`, `DArray<bool>`, etc.

It is as if we typed the same class interface and implementation twice: once with `T = int` and once with `T = bool`.

Example

```
template<typename T>
DArray<T>::DArray(int size)
  : n(size)
{
  A = new T[n];
  for (int i = 0; i < n; i++)
    A[i] = 0;   // only works if this makes sense:
                // It must be okay to assign 0 to type T
}
```

Implementation Issues

- Normally, the interface goes into `.h` file and the implementation goes into `.cc` file.
- For templates, put everything into `.h` file. (Alternatively, include the `.cc` file in `.h` file.)
- In order for the compiler to instantiate the templates, it needs to know the implementations as they are being used.
- Different compilers may handle this differently...
- Strange syntax for static members (see text) and friends (see example in class).
- You may get strange errors if the supplied type does not make sense.

Function Templates

- We can also write templates for functions.
- This is useful for certain functions that look the same for different types.
- e.g. minimum/maximum, sort, swap, etc.
- Write it once, and instantiated as many times as necessary.

Function Templates

```
template<typename T>
const T &mymin(const T &a, const T &b)
{
  if (a < b)      // operator< must be defined for T
    return a;
  else
    return b;
}
```

We use constant references because of potential inefficiencies if `T` is a complicated class.

Function Templates

To use:

```
int a, b;
double x, y;
string u, v;

cout << mymin(a, b) << endl;  // int version
cout << mymin(x, y) << endl;  // double version
cout << mymin(u, v) << endl;  // string version
cout << mymin(a, y) << endl;  // ambiguous!
cout << mymin<double>(a, y) << endl;  // double version
```

Complicated Example

```
template<typename T>
void transform(T A[], int n, T (*f)(const T &x))
{
  for (int i = 0; i < n; i++)
    A[i] = f(A[i]);
}

template<typename T>
T square(const T &x) { return x * x; }

int A[10];
transform(A, 10, square);
```

Some Standard Function Templates

- C++ provides a number of function templates for commonly used algorithms (#include <algorithm>)
- e.g. min(a,b) computes minimum of two elements
- e.g. sort(A, A+10) sorts an array of 10 elements from smallest to largest (pointer to beginning, pointer to one-past-the-end).
- Only works if element type can be compared by <.
- There are others: binary_search, find, etc.