Templates

- Templates for Algorithm Abstraction
- Templates for Data Abstraction

`swap_values` function for `int`

- Here is the function `swap_values` to swap two integers:

```
void swap_values (int& v1, int& v2)
{
     int temp = v1;
     v1 = v2;
     v2 = temp;
}
```

- Can we use this function to swap two characters? two doubles? two strings? …

A General `swap_values`

- A generalized version of `swap_values` is shown here

```
void swap_values(type_of_var& v1,
                 type_of_var& v2)
{
    type_of_var temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}
```

  This function, if `type_of_var` could accept any type, could be used to swap values of any type

Templates for Functions

- A C++ function template will allow `swap_values` to swap values of two variables of the same type

Type parameter

Template prefix

```
template<class T>
void swap_values(T& v1, T& v2)
{
      T temp;
      temp = v1;
      v1 = v2;
      v2 = temp;
}
```

Template Details

- `template<class T>` is the template prefix
  - Tells compiler that the declaration or definition that follows is a template
  - Tells compiler that `T` is a type parameter
    - `class` means type in this context
      (`typename` could replace `class` but `class` is usually used)
    - `T` can be replaced by any type argument – see next slide (whether the type is a class or not)

- A template overloads the function name by replacing `T` with the type used in a function call

The Type Parameter `T`

- `T` is the traditional name for the type parameter
  - Any valid, non-keyword, identifier can be used
  - If you don't hate capital T, just use it ☺

Calling a Template Function

- Calling a function defined with a template is identical to calling a normal function
  - Example: To call the template version of **`swap_values`**

```
char ch1, ch2;
int n1, n2;
…
swap_values(ch1, ch2);
swap_values(n1, n2);
```

- The compiler checks the argument types and generates an appropriate version of `swap_values`

Templates with Multiple Parameters

- Function templates may use more than one parameter
  - Example:

```
template<class T1, class T2>
```

    - All parameters must be used in the template function

Algorithm Abstraction

- Using a template function we can express more general algorithms in C++
- Algorithm abstraction means expressing algorithms in a very general way so we can ignore incidental detail
  - This allows us to concentrate on the substantive part of the algorithm

Example: A Generic Sorting Function

- The sort function below uses an algorithm that does not depend on the base type of the array

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used -1;index++)
    {
        index_of_next_smallest =
                index_of_smallest(a, index, number_used);

        swap_values(a[index],a[index_of_next_smallest]);
    }
}
```

The same algorithm could be used to sort an array of any type

Generic Sorting

- sort uses two helper functions
  - `index_of_smallest` also uses a general algorithm and could be defined with a template
  - `swap_values` has already been adapted as a template

- All three functions, defined with templates, are demonstrated in display 17.2 and 17.3, also on the following slides

DISPLAY 17.2 A Generic Sorting Function

```
// This is file sortfunc.cpp

template<class T>
void swap_values(T& variable1, T& variable2)
            <The rest of the definition of swap_values is given in Display 17.1.>

template<class BaseType>
int index_of_smallest(const BaseType a[], int start_index, int number_used)
{
    BaseType min = a[start_index];
    int index_of_min = start_index;

    for (int index = start_index + 1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through a[index]
        }

    return index_of_min;
}

template<class BaseType>
void sort(BaseType a[], int number_used)
{
int index_of_next_smallest;
for(int index = 0; index < number_used - 1; index++)
   {//Place the correct value in a[index]:
          index_of_next_smallest =
               index_of_smallest(a, index, number_used);
          swap_values(a[index], a[index_of_next_smallest]);
       //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
       //elements. The rest of the elements are in the remaining positions.
   }
}
```

```
//Demonstrates a generic sorting function.
#include <iostream>
using namespace std;

//The file sortfunc.cpp defines the following function:
//template<class BaseType>
//void sort(BaseType a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

#include "sortfunc.cpp"

int main( )
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
    sort(a, 10);
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << "Unsorted doubles:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
    sort(b, 5);
    cout << "In sorted order the doubles are:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
```

Many compilers will allow this function declaration to appear as a function declaration and not merely as a comment. However, including the function declaration is not needed, since the definition of the function is in the file `sortfunc.cpp`, and so the definition effectively appears before `main`.

```
    char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
    cout << "Unsorted characters:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;
    sort(c, 7);
    cout << "In sorted order the characters are:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;

    return 0;
}
```

Output

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

Templates and Operators

- The function `index_of_smallest` compares items in an array using the `<` operator
  - If a template function uses an operator, such as <, that operator must be defined for the types being compared
  - If a class type has the < operator overloaded for the class, then an array of objects of the class could be sorted with function template sort

Defining Templates

- When defining a template it is a good idea…
  - To start with an ordinary function that accomplishes the task with one type
    - It is often easier to deal with a concrete case rather than the general case
  - Then debug the ordinary function
  - Next convert the function to a template by replacing type names with a type parameter

Inappropriate Types for Templates

- Templates can be used for any type for which the code in the function makes sense
  - `swap_values` swaps individual objects of a type
  - This code would not work, because the assignment operator used in `swap_values` does not work with (non-dynamic) arrays:

```
int a[10], b[10];
<code to fill the arrays>
swap_values(a, b);
```

Templates for Data Abstraction

- Class definitions can also be made more general with templates
  - The syntax for class templates is basically the same as for function templates
    - `template<class T>` comes before the template definition
    - Type parameter `T` is used in the class definition just like any other type
    - Type parameter `T` can represent any type

A Class Template

- The following is a class template
  - An object of this class contains a pair of values of type T

```
template <class T>
class Pair
{
public:
    Pair( );
    Pair( T first_value, T second_value);

    …
    // continued on next slide
```

Template Class `Pair` (cont.)

```
void set_element(int position, T value);
//Precondition: position is 1 or 2
//Postcondition:  position indicated is set to
  value

T get_element(int position) const;
// Precondition:  position is 1 or 2
// Returns value in position indicated

private:
    T first;
    T second;
};
```

Declaring Template Class Objects

- Once the class template is defined, objects may be declared
  - Declarations must indicate what type is to be used for `T`
  - Example: To declare an object so it can hold a pair of integers:

```
Pair<int> score;
```

or for a pair of characters:

```
Pair<char> seats;
```

Using the Objects

- After declaration, objects based on a template class are used just like any other objects
  - Continuing the previous example:

```
score.set_element(1,3);
score.set_element(2,0);
seats.set_element(1, 'A');
```

Defining the Member Functions

- Member functions of a template class are defined the same way as member functions of ordinary classes
  - The only difference is that the member function definitions are themselves templates

Defining a `Pair` Constructor

- This is a definition of the constructor for class `Pair` that takes two arguments

```
template<class T>
Pair<T>::Pair(T first_value, T second_value)
            : first(first_value),
              second(second_value)
{}
```

  - The class name includes `<T>`

Defining `set_element`

- Here is a definition for `set_element` in the template class `Pair`

```
template<class T>
void Pair<T>::set_element(int position, T value)
{
      if (position == 1)
            first = value;
      else if (position == 2)
            second = value;
      // could check whether position is valid …
}
```

Template Class Names as Parameters

- The name of a template class may be used as the type of a function parameter
  - Example: To create a parameter of type `Pair<int>`:

```
int add_up(const Pair<int>& the_pair);
//Returns the sum of two integers in the_pair
```

Exercise: Implement the entire `Pair` class defined on slide #19 and #20. Then write some testing code to use this class.

Template Functions with Template Class Parameters

- Function `add_up` from a previous example can be made more general as a template function:

```
template<class T>
T add_up(const Pair<T>& the_pair)
//Precondition: operator + is defined for T
//Returns sum of the two values in the_pair
```

Program Example: An Array Class

- The example in the following displays is a class template whose objects are lists
  - The lists can be lists of any type

- See textbook and source code --

  The interface can be found in Display 17.4
  The program is in Display 17.5
  The implementation is in Display 17.6