2 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

2.1 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);

                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;
};
```

2.2 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;
```

2.2.1 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');
```

2.3 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

2.3.1 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)
{
        //No body needed due to initialization above
}
```

  - **The class name includes <T>**

2.3.2 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;
        else
                :
                :
}
```

2.4 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
```

2.4.1 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
```

2.5 typedef and Templates

- You specialize a class template by giving a type argument to the class name such as Pair<int>

  - The specialized name, Pair<int>, is used just like any class name

- You can define a new class type name with the same meaning as the specialized name:

```
typedef Class_Name<Type_Arg> New_Type_Name;
```

- For example:

```
typedef Pair<int> PairOfInt;
PairOfInt pair1, pair2;
```

2.6 Template non-type parameters

- C++ allows values to be used as template parameters.

```cpp
#include <iostream>

template<int init>
class value
{
public:
    double x;
    value();
};

template<int init>
value<dim>::value():x(init)
{
}

int main(int argc, const char * argv[])
{
    value<3> a;
    std::cout << a.x << std::endl;
    return 0;
}
```