Chapter 22 - C++ Templates

Outline
22.1 Introduction
22.2 Class Templates
22.3 Class Templates and Non-type Parameters
22.4 Templates and Inheritance
22.5 Templates and friends
22.6 Templates and static Members

Objectives
• In this chapter, you will learn:
  – To be able to use class templates to create a group of related types.
  – To be able to distinguish between class templates and template classes.
  – To understand how to overload template functions.
  – To understand the relationships among templates, friends, inheritance and static members.

22.1 Introduction
• Templates
  – Easily create a large range of related functions or classes
  – Function template - the blueprint of the related functions
  – Template function - a specific function made from a function template

22.2 Class Templates
• Class templates
  – Allow type-specific versions of generic classes
• Format:
  template <class T>
class ClassName{
definition }
  – Need not use "T", any identifier will work
  – To create an object of the class, type:
    ClassName< type > myObject;

22.2 Class Templates (II)
• Template class functions
  – Defined normally, but preceded by "template<class T>
  – Binary scope resolution operator used
  – Template class function definition:
    template<class T>
    MyClass< T >::MyClass(int size)
    {
      myArray = new T[size];
    }
  – Constructor definition - creates an array of type T
using std::endl;
using std::cin;
using std::cout;

// objects of type Stack<T>.
// Test driver for Stack template.
// Fig. 22.2: fig22_02.cpp

// return 1 if successful, 0 otherwise
// Push an element onto the stack
// returns true if successful, false otherwise
bool Stack<T>::push( const T &pushValue ) {
    if ( !isFull() ) {
        stackPtr[ ++top ] = pushValue; // place item in Stack
        return true;  // push successful
    }
    return false;     // push unsuccessful
}

// Constructor with default size 10
template< class T >
Stack< T >::Stack( int s )
    : size( s > 0 ? s : 10 ), stackPtr( new T[ size ] ), top( -1 ) {
}

// return 1 if successful, 0 otherwise
// Pop an element off the stack
// returns true if successful, false otherwise
bool Stack<T>::pop( T &popValue ) {
    if ( !isEmpty() ) {
        popValue = stackPtr[ top-- ];  // remove item from Stack
        return true;  // pop successful
    }
    return false;       // pop unsuccessful
}

// return 1 if successful, 0 otherwise
// Push an element onto the stack
// returns true if successful, false otherwise
bool Stack<T>::push( const T &pushValue ) {
    if ( !isFull() ) {
        stackPtr[ ++top ] = pushValue; // place item in Stack
        return true;  // push successful
    }
    return false;     // push unsuccessful
}

// return 1 if successful, 0 otherwise
// Pop an element off the stack
// returns true if successful, false otherwise
bool Stack<T>::pop( T &popValue ) {
    if ( !isEmpty() ) {
        popValue = stackPtr[ top-- ];  // remove item from Stack
        return true;  // pop successful
    }
    return false;       // pop unsuccessful
}

// return 1 if successful, 0 otherwise
// Push an element onto the stack
// returns true if successful, false otherwise
bool Stack<T>::push( const T &pushValue ) {
    if ( !isFull() ) {
        stackPtr[ ++top ] = pushValue; // place item in Stack
        return true;  // push successful
    }
    return false;     // push unsuccessful
}

// return 1 if successful, 0 otherwise
// Pop an element off the stack
// returns true if successful, false otherwise
bool Stack<T>::pop( T &popValue ) {
    if ( !isEmpty() ) {
        popValue = stackPtr[ top-- ];  // remove item from Stack
        return true;  // pop successful
    }
    return false;       // pop unsuccessful
}
22.3 Class Templates and Non-type Parameters

- Can use non-type parameters in templates
  - Default argument
  - Treated as const

- Example:
  ```cpp
template< class T, int elements >
Stack< double, 100 > mostRecentSalesFigures;
```
  - Defines object of type `Stack< double, 100>`
  - May appear in the class definition:
    ```cpp
t StackHolder[ elements ]; //array to hold stack
```
  - Creates array at compile time, rather than dynamic allocation at execution time

22.3 Class Templates and Non-type Parameters (II)

- Classes can be overridden
  - For template class `Array`, define a class named
    ```cpp
    Array<myCreatedType>
    ```
  - This new class overrides then class template for `myCreatedType`
  - The template remains for unoverridden types

22.4 Templates and Inheritance

- A class template can be derived from a template class
- A class template can be derived from a non-template class
- A template class can be derived from a class template
- A non-template class can be derived from a class template

22.5 Templates and friends

- Friendships allowed between a class template and
  - Global function
  - Member function of another class
  - Entire class
- friend functions
  - Inside definition of class template `X`:
    ```cpp
    friend void f1();
    friend void f2( X< T > & );
    friend void f3();
    ``
  - member function `f3` of class `A` is a friend of all template classes

- friend classes
  - friend class `Y`:
    - Every member function of `Y` is a friend with every template class made from `T`:
    ```cpp
    friend class Y;
    ```
    - Class `X< float >::friend of class X< int >` only
22.6 Templates and `static` Members

- **Non-template class**
  - `static` data members shared between all objects

- **Template classes**
  - Each class (e.g., `int`, `float`, etc.) has its own copy of `static` data members
  - `static` variables initialized at file scope
  - Each template class gets its own copy of `static` member functions