Object-oriented Programming
Generic Programming using Templates

YoungWoon Cha

Computer Science and Engineering

KU KONKUK UNIVERSITY

Review

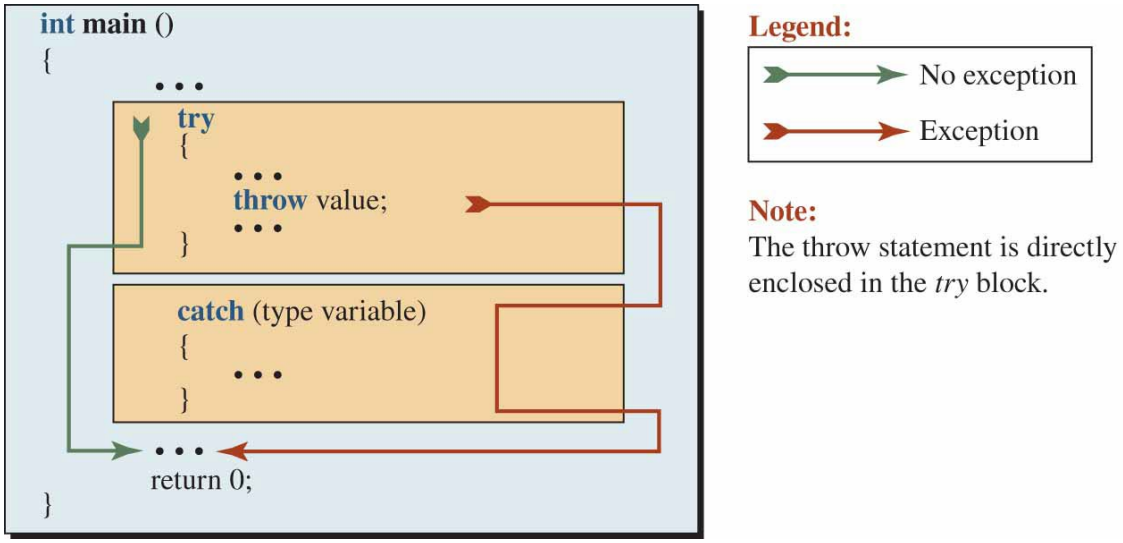
Exception Handling Using Try-Catch Block

The *try* clause includes the code that may cause the program to abort.

The run-time environment tries to execute the code. If the code can be executed, the program flow continues.

If the code cannot be executed, the system *throws* an *exception* using a throw expression. The rest of the try clause after the throw statement is ignored and control moves to the *catch* clause.

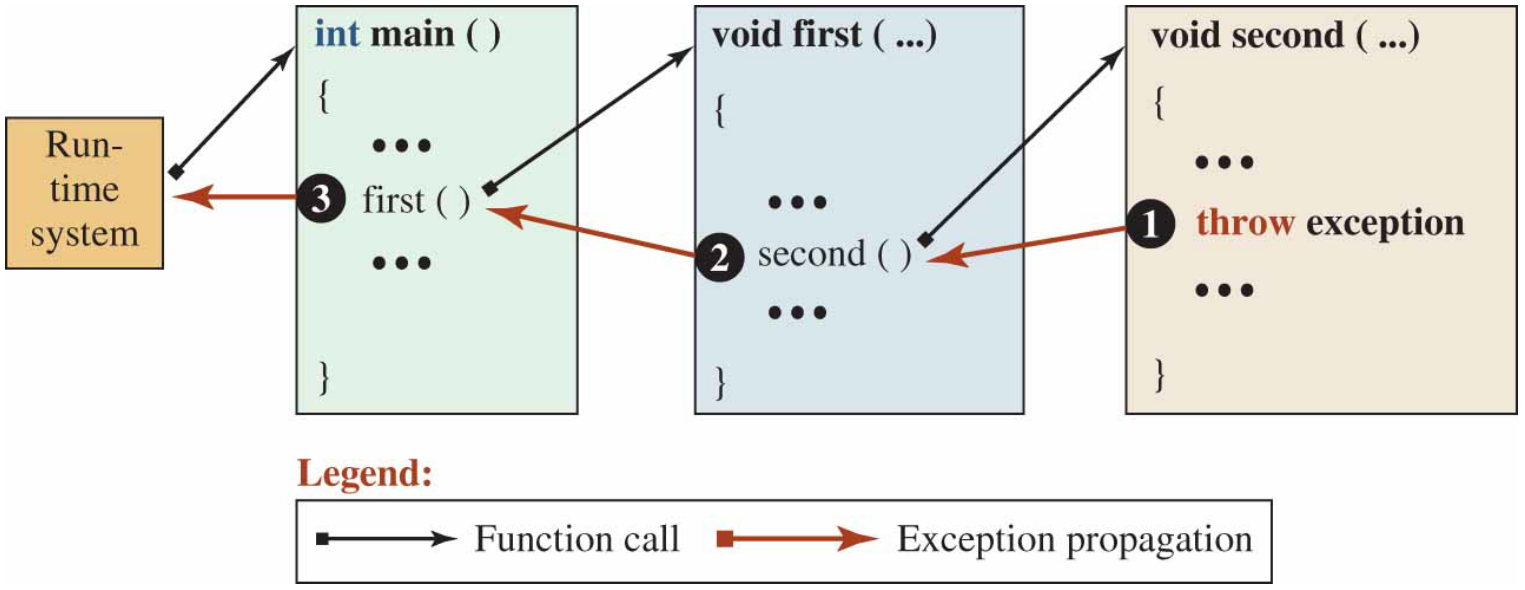
The *catch clause* lets the program handle the exception and continue with the rest of program if possible.



Exception Propagation

A function may throw an exception and it not caught and handled where it has been thrown, it will be automatically propagated to the previous function in the hierarchy of function calls until one of the functions in the path captures and handles it.

If it is not caught and handled by the *main* function (the last function) it will be propagated to the run-time system, where it is caught and causes the *main* function to be aborted, which aborts the whole program.

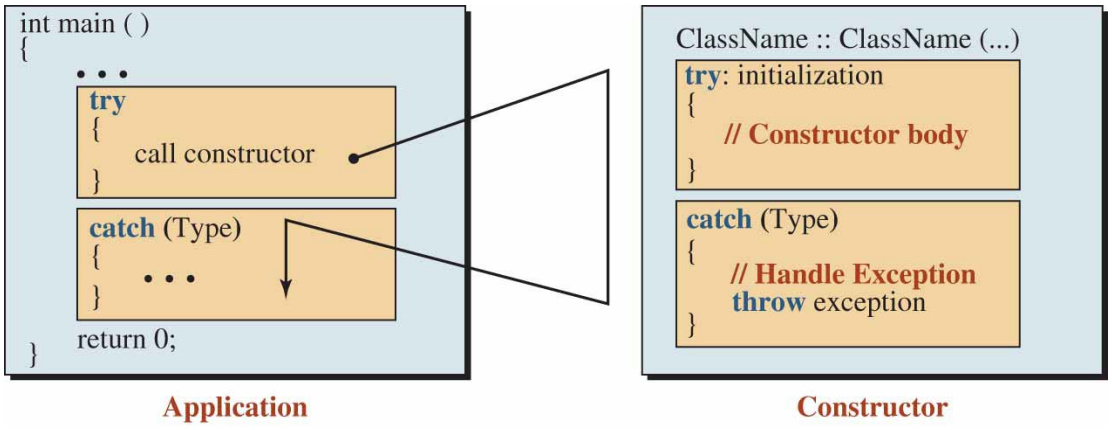


Exceptions in Classes

Although handling exceptions in member functions other than constructors and destructors is the same as the exceptions in a stand-alone function, we need to be careful about exceptions in constructors and destructors.

Constructors

To be able to throw an exception if it happens in the initializer list, we need to use what is called a *function-try* block, which combines a *try-catch* block with an initialization list.



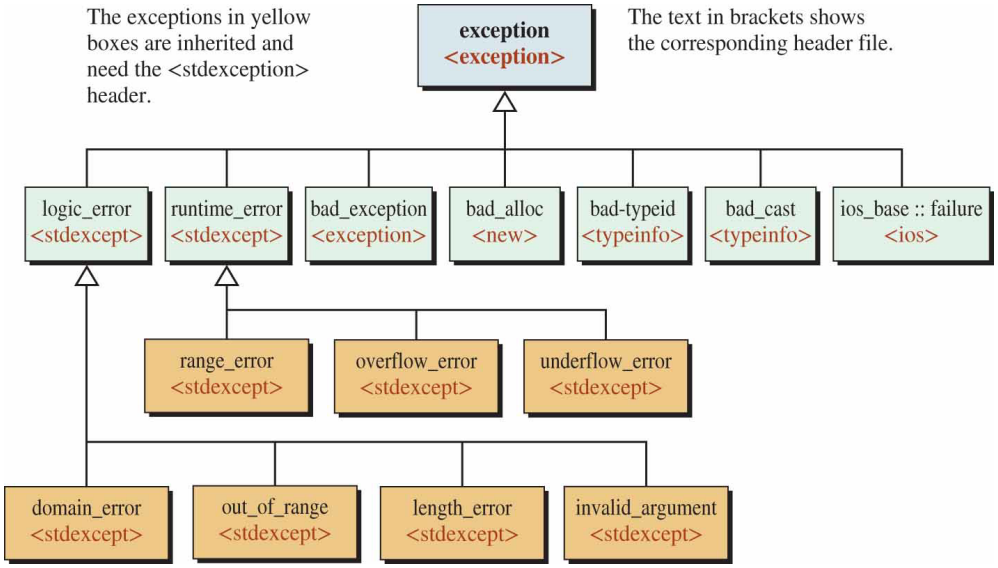
Destructors

If any exception is thrown in the destructor, C++ calls a global function named *terminator* that terminates the whole program.

Exception throwing in a destructor must be avoided.

STANDARD EXCEPTION CLASSES

C++ defines a set of standard exception classes that are used in its library.



At the top of the hierarchy is the *exception* class defined in the <exception> header.

The Table shows the public interface for the *exception* class.

```cpp
#include <iostream>
#include <string>
#include <exception>
using namespace std;

int main()
{
   string str = "Test";
   for (unsigned int i = 0; i < str.length(); i++)
   {
      cout << str.at(i) << endl;
   }

   try
   {
      cout << str.at(str.length()); // exception here
   }
   catch (exception & e) // catch-by-reference
   {
      cout << e.what() << endl;
   }

   return 0;
}
```

```cpp
exception() throw() // constructor
exception(const exception&) throw()  // copy constructor
exception& operator =(const exception&) throw() // Assignment operator
virtual ~exception() throw() // destructor
virtual const char* what() const throw() // member function
```

KU KONKUK UNIVERSITY

Function Templates

FUNCTION TEMPLATE

When programming in any language, we sometimes need to apply the same code on different data types.

For example, we may have a function to find the smaller of two integer data types, another function to find the smaller of two floating-point data types, and so on.

We need to define a family of functions, each with one or more different data types.

We can write a program to solve a problem with a generic data type.

We can then apply the program to the specific data type we need.

This is known to as generic programming or template programming.

Using function templates, actions can be defined when we write a program and the data types can be defined when we compile the program.

Using A Family of Functions

If we do not use template and generic programming, we need to define a family of functions. (using overloaded functions)

```
#include <iostream>
using namespace std;

// Function to find the smaller between two characters
char smaller (char first, char second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
// Function to find the smaller between two integers
int smaller (int first, int second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
// Function to find the smaller between two doubles
double smaller (double first, double second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
```

```
int main ()
{
      cout << "Smaller of 'a' and 'B': " << smaller ('a', 'B')
      << endl;
      cout << "Smaller of 12 and 15: " << smaller (12, 15)
      << endl;
      cout << "Smaller of 44.2 and 33.1: " << smaller (44.2,
      33.1) << endl;
      return 0;
}
```

```
Run:
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

Note that in ASCII uppercase letters come before lowercase letters, which means 'B' is smaller that 'a'.

Using Function Template

Syntax

To create a template function, we can use a placeholder for each generic type.

Below shows the general syntax for a generic function, in which T, U, ..., and Z are replaced by actual types when the function is called.

```
template <typename T, typename U, ..., typename Z>
T functionName(U first, ... Z last)
{
    ...
}
```

The template header contains the keyword *template* followed by a list of symbolic types inside two angle brackets.

The template function header follows the rules for function headers except that the types are symbolic as declared in the template header.

We use the keyword *typename* because it is the current standard and is used in the C++ library.

Program Using One Function Template

```
/*****************************************************************
 * A program that uses a template function to find the smaller *
 * of two values of different types                              *
 *****************************************************************/
#include <iostream>
using namespace std;

// Definition of a template function
template <typename T>
T smaller (T first, T second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}

int main ( )
{
      cout << "Smaller of a and B: " << smaller ('a', 'B') << endl;
      cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
      cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
      return 0;
}
```

```
Run:
Smaller of a and B: B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

Program That Swap Two Values

```
/*****************************************************************
 * A program that uses a template function to swap two values. *
 *****************************************************************/
#include <iostream>
using namespace std;

// Definition of template function
template <typename T>
void exchange (T& first, T& second)
{
      T temp = first;
      first = second;
      second = temp;
}

int main ()
{
      // Swapping two int types
      int integer1 = 5;
      int integer2 = 70;
      exchange (integer1, integer2);
      cout << "After swapping 5 and 70: ";
      cout << integer1 << " " << integer2 << endl;
      // Swapping two double types
      double double1 = 101.5;
      double double2 = 402.7;
      exchange (double1, double2);
      cout << "After swapping 101.5 and 402.7: ";
      cout << double1 << " " << double2 << endl;
      return 0;
}
```

```
Run:
After swapping 5 and 70: 70 5
After swapping 101.5 and 402.7: 402.7 101.5
```

Function Template Instantiation

Using template functions postpones creating non-template function definitions until compilation time.

This means that when a program involving a function template is compiled, the compiler actually creates as many versions of the function as needed by function calls.

This process is referred to as *template instantiation*, but it should not be confused with instantiation of an object from a type.

```
template <typename T>
T smaller (T& first, T& second)
{
    ...
}
```

Before compilation

```
char smaller (char first, char second)
{
int smaller (int first, int second)
{
double smaller (double first, double second)
{
    ...
}
```

After compilation

Non-type Template Parameter

Sometimes, we need to define a value instead of a type in our function template.

In other words, the type of a parameter may be the same for all template functions that we want to use.

In this case, we can define a template, but the type can be specifically defined.

```
#include <iostream>
using namespace std;

// Definition of the print template function
template <typename T, int N>
void print (T (&array) [N])
{
      for (int i = 0; i < N ; i++)
      {
           cout << array [i] << " ";
      }
      cout << endl;
}

int main ()
{
      // Creation of two arrays
      int arr1 [4] = {7, 3, 5, 1};
      double arr2 [3] = {7.5, 6.1, 4.6};
      // Calling template function
      print (arr1);
      print (arr2);
      return 0;
}
```

Assume we want to have a function that prints the elements of any array regardless of the type of the element and the size of the array.

We know that the type of each element can vary from one array to another, but the size of an array is always an integer (or unsigned integer).

We have two template parameters, *T* and *N*.

The parameter *T* can be any type; the parameter *N* is a non-type (the type is predefined as integer).

```
Run:
7 3 5 1
7.5 6.1 4.6
```

Explicit Type Determination Part 1

If we try to call the function template to find the smaller between an integer and a floating-point value such as the following, we get an error message.

```
cout << smaller(23, 67.2); // Errors! Two types for the same T
```

In other words, we are giving the compiler an integral value of 23 for the first argument of type T and a floating-point value of 67.2 for the second argument of type T.

The type T is one template type and the values for it must always be the same type.

The error can be avoided if we define the explicit type conversion during the call.

This is done by defining the type inside the angle brackets as shown below.

```
cout << smaller<double>(23, 67.2);  // 23 will be changed to 23.0
```

We are telling the compiler that we want to use the version of the *smaller* programs in which the value of T is of type *double*.

The compiler then creates that version and convert 23 to 23.0 before finding the *smaller*.

Explicit Type Determination Part 2

Predefined Operation

The reason that we can compare two integers, two double, or two characters using the smaller function is because the less than operator (<) is defined for these types.

This means that we can use a function template for any type for which the overloaded operator less-than (<) is defined.

For example, we know that the library *string* class has overloaded this operator.

We can explicitly tell the compiler to replace T by a *string* and then call the function.

This means that the following statement is definitely valid and the result is the string "Bi".

```
cout << smaller("Hello" , "Bi");  //  The result is "Bi"
```

Explicit Type Determination Part 3

Specialization

The operator less than (<) is not defined for a C-Style string.

This means that we cannot use the template function to find the smaller of two C-Style strings. (We will get a compilation error if we do so.)

```
const char* s1 = "Hello";
const char* s2 = "Bi";
cout << smaller(s1, s2);  //  Compile error for const char*
```

However, there is a solution, which is called template specialization.

We can define another function with a specific type (instead of the template type).

Template Specialization

a. We need to use the template <> before the header to show that this is the specialization of a template function previously defined.

b. We need to be careful about the replacement of the T with a specialized type. A C-Style string is of type *const char**, which means every time we have T, we need to replace it with *const char**.

```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;

// Template Function
template <typename T>
T smaller (const T& first, const T& second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
// Specialization of template function
template <>
const char* smaller(const (const char*)& first, const (const char*) & second)
{
      if (strcmp (first, second ) < 0)
      {
           return first;
      }
      return second;
}
```

```
int main ( )
{
      // Calling template with two string objects
      string str1 = "Hello";
      string str2 = "Hi";
      cout << "Smaller (Hello , Hi): " << smaller (str1, str2) << endl;
      //Calling template function with two C-string objects
      const char* s1 = "Bye";
      const char* s2 = "Bye Bye";
      cout << "Smaller (Bye, Bye Bye)" << smaller (s1, s2) << endl;
      return 0;
}
```

```
Run:
Smaller (Hello , Hi): Hello
Smaller (Bye, Bye Bye): Bye
```

Template Overloading

The overloading concept can be applied to function templates.

We can overload a function template to have several functions with the same name but different signatures.

Normally, the template type is the same, but the number of parameters is different.

We overload the *smallest* template function to accept two or three parameters.

Note that we have defined the second function in terms of the first one.

```
#include <iostream>
using namespace std;

// Template function with two parameters
template <typename T>
T smallest (const T& first, const T& second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
// Template function with three parameters
template <typename T>
T smallest (const T& first, const T& second, const T& third)
{
      return smallest (smallest (first, second), third);
}
```

```
int main ( )
{
      // Calling the overloaded version with three integers
      cout << "Smallest of 17, 12, and 27 is ";
      cout << smallest (17, 12, 27) << endl;
      // Calling the overloaded version with three doubles
      cout << "Smallest of 8.5, 4.1, and 19.75 is ";
      cout << smallest (8.5, 4.1, 19.75) << endl;
      return 0;
}
```

```
Run:
Smallest of 17, 12, and 27 is 12
Smallest of 8.5, 4.1, and 19.75 is 4.1
```

Interface and Application Files

```
/*****************************************************************
 * The interface file for the function template named smaller  *
 *****************************************************************/
#ifndef SMALLER_H
#define SMALLER_H
#include <iostream>
using namespace std;

template <typename T>
T smaller (const T& first, const T& second)
{
      if (first < second)
      {
           return first;
      }
      return second;
}
#endif
```

```
/*****************************************************************
 * The application file to test a function template            *
 *****************************************************************/
#include "smaller.h"

int main ( )
{
      cout << "Smaller of 'a' and 'B': " << smaller ('a', 'B')
      << endl;
      cout << "Smaller of 12 and 15: " << smaller (12, 15) <<
      endl;
      cout << "Smaller of 44.2 and 33.1: " << smaller (44.2,
      33.1) << endl;
      return 0;
}
```

The definition of a template function can be put in an interface file and the header can be included in an application file.

This means that we can write one definition for a template function and then use it in different programs.

The interface file in this case needs to include the definition of the function, not only the declaration.

```
Run:
Smaller of a and B: B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
```

KU KONKUK UNIVERSITY

Class Templates

CLASS TEMPLATE

The concept of templates makes the C++ language very powerful.

We may also have a class with data types and another class, with the same functionality, but with different data types.

In these cases, we can use a class template.

Templates are used in C++ libraries such as string and stream classes.

They are also used in the Standard Template Classes, STL. To create a class template, we need to make both the data members and the member functions generic.

Interface and Implementation

As we know a class has an interface and an implementation. When we need to create a class template, we need to have generic parameters in both the interface and the implementation.

Interface

The interface of a class needs to define the *typename* for both data members and member functions that use the parameterized type.

Implementation

In the implementation, we need to mention the *typename* for each member function that uses the generic type.

```
template <typename T>
class Name
{
      private:
            T data;
      public:
            Name(); // default constructor
            T get() const; // accessor
            void set(T data);     // mutator function

};
```

```
// Implementation of the get function

template <typename T>
T name <T> :: get() const
{
      return data;
}
// Implementation of the set function

template <typename T>
void name <T> :: set(T  d)
{
      data = d;
}
```

Interface and Implementation for Class Fun

Interface File

```
#ifndef FUN_H
#define FUN_H
#include <iostream>
using namespace std;

template <typename T>
class Fun
{
      private:
            T data;
      public:
            Fun (T data);
            ~Fun ();
            T get () const;
            void set (T data);
};
#endif
```

Implementation File

```
#ifndef FUN_CPP
#define FUN_CPP
#include "fun.h"

// Constructor
template <typename T>
Fun <T> :: Fun (T d)
: data (d)
{
}
// Destructor
template <typename T>
Fun <T> :: ~Fun ()
{
}
// Accessor Function
template <typename T>
T Fun <T> :: get () const
{
      return data;
}
// Mutator Function
template <typename T>
void Fun <T> :: set (T d)
{
      data = d;
}
#endif
```

Application File Using Class Fun

The compiler needs to see the parameterized version of the template function when it compiles the application file.

In other words, the application file needs to have the implementation file as a header file, which means we need to add the macros *ifndef*, *define*, and *endif* to the implementation file.

```
#include "fun.cpp"

int main ( )
{
      // Instantiation of four classes each with different data type
      Fun <int> Fun1 (23);
      Fun <double> Fun2 (12.7);
      Fun <char> Fun3 ('A');
      Fun <string> Fun4 ("Hello");
      // Displaying the data values for each class
      cout << "Fun1: " << Fun1.get() << endl;
      cout << "Fun2: " << Fun2.get() << endl;
      cout << "Fun3: " << Fun3.get() << endl;
      cout << "Fun4: " << Fun4.get() << endl;
      // Setting the data values in two classes
      Fun1.set(47);
      Fun3.set ('B');
      // Displaying values for newly set data
      cout << "Fun1 after set: " << Fun1.get() << endl;
      cout << "Fun3 after set: " << Fun3.get() << endl;
      return 0;
}
```

Note that we have included *fun.cpp* as a header file to help the compiler create different versions of the class.

Also note that to instantiate template classes, we need to define the actual type that replace the *typename* T.

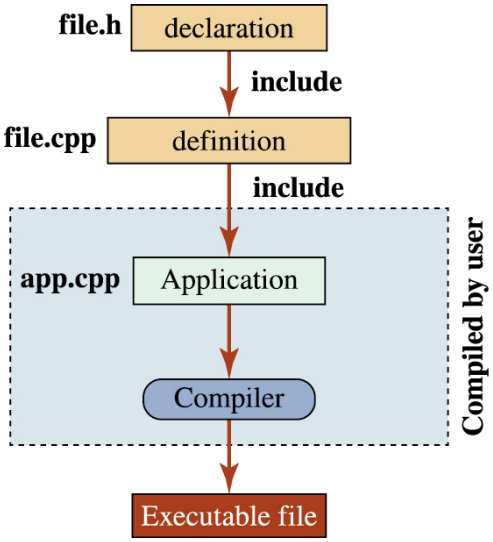
```
Run:
Fun1: 23
Fun2: 12.7
Fun3: A
Fun4: Hello
Fun1 after set: 47
Fun3 after set: B
```

Template Compilation Part 1

One issue is how we can compile and link different files related to a program when there are some function templates or class templates in our project.

Inclusion Approach

The first approach is called compilation using inclusion.
In this approach we put the declaration and definition in header files and then include the header files in the our application program.



We can put the declaration in a (.h) file and the definition in a (.cpp) file.

We do not compile the definition file separately, we just include it in the application file and compile only the application file.

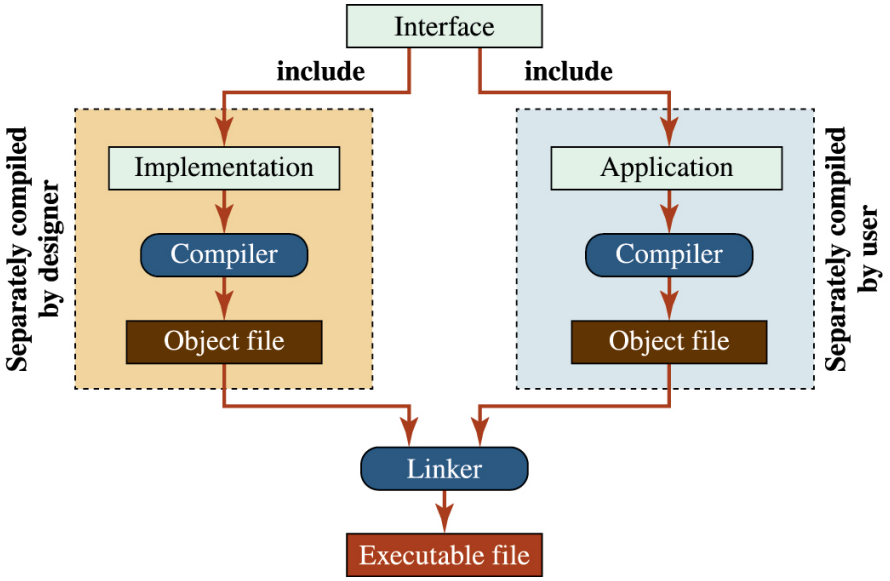
Note that the declaration file must be included in the definition file and the definition file must be included in the application file.

We only compile the application file.

Template Compilation Part 2

Separate Compilation

In this model we have three different files: interface files, implementation files, and one application file.



Although this approach has been used with non-template functions and classes, many compilers does not support this separate compilations with template functions or template classes.

The problem is that the implementation file cannot be compiled independently because the compiler needs to see the application file to figure out which instances of each template function or template class needs to be used in the compilation.

Recommendation for Template Compilation Part 1

fun.hpp

```
#pragma once

template <typename T>
class Fun
{
private:
  T data;
public:
  Fun(T data);
  ~Fun();
  T get() const;
  void set(T data);
};

#include "fun.tpp"
```

fun.tpp

```
#pragma once
#include "fun.hpp"

template <typename T>
Fun <T> ::Fun(T d)
  : data(d)
{
}

template <typename T>
Fun <T> :: ~Fun()
{
}

template <typename T>
T Fun <T> ::get() const
{
  return data;
}

template <typename T>
void Fun <T> ::set(T d)
{
  data = d;
}
```

app.cpp

```
#include "fun.hpp"
#include <iostream>
using namespace std;

int main()
{
  Fun <int> Fun1(23);
  Fun <double> Fun2(12.7);
  Fun <char> Fun3('A');
  Fun <string> Fun4("Hello");

  cout << "Fun1: " << Fun1.get() << endl;
  cout << "Fun2: " << Fun2.get() << endl;
  cout << "Fun3: " << Fun3.get() << endl;
  cout << "Fun4: " << Fun4.get() << endl;

  Fun1.set(47);
  Fun3.set('B');

  cout << "Fun1 after set: " << Fun1.get() << endl;
  cout << "Fun3 after set: " << Fun3.get() << endl;
  return 0;
}
```

Recommendation for Template Compilation Part 2

fun.hpp

```
#pragma once

template <typename T>
class Fun
{
private:
  T data;
public:
  Fun(T data);
  ~Fun();
  T get() const;
  void set(T data);
};

template <typename T>
Fun <T> ::Fun(T d)
  : data(d)
{
}

template <typename T>
Fun <T> :: ~Fun()
{
}

template <typename T>
T Fun <T> ::get() const
{
  return data;
}

template <typename T>
void Fun <T> ::set(T d)
{
  data = d;
}
```

app.cpp

```
#include "fun.hpp"
#include <iostream>
using namespace std;

int main()
{
  Fun <int> Fun1(23);
  Fun <double> Fun2(12.7);
  Fun <char> Fun3('A');
  Fun <string> Fun4("Hello");

  cout << "Fun1: " << Fun1.get() << endl;
  cout << "Fun2: " << Fun2.get() << endl;
  cout << "Fun3: " << Fun3.get() << endl;
  cout << "Fun4: " << Fun4.get() << endl;

  Fun1.set(47);
  Fun3.set('B');

  cout << "Fun1 after set: " << Fun1.get() << endl;
  cout << "Fun3 after set: " << Fun3.get() << endl;
  return 0;
}
```

Recommendation for Template Compilation Part 3

fun.hpp

```
#pragma once

template <typename T>
class Fun
{
private:
  T data;
public:
  Fun(T d)
    :data(d)
  {
  }
  ~Fun() {}
  T get() const
  {
    return data;
  }
  void set(T d)
  {
    data = d;
  }
};
```

app.cpp

```
#include "fun.hpp"
#include <iostream>
using namespace std;

int main()
{
  Fun <int> Fun1(23);
  Fun <double> Fun2(12.7);
  Fun <char> Fun3('A');
  Fun <string> Fun4("Hello");

  cout << "Fun1: " << Fun1.get() << endl;
  cout << "Fun2: " << Fun2.get() << endl;
  cout << "Fun3: " << Fun3.get() << endl;
  cout << "Fun4: " << Fun4.get() << endl;

  Fun1.set(47);
  Fun3.set('B');

  cout << "Fun1 after set: " << Fun1.get() << endl;
  cout << "Fun3 after set: " << Fun3.get() << endl;
  return 0;
}
```

KU KONKUK UNIVERSITY

Issues on Templates

Other Issues

Friends

The declaration of a template class can have friends. This can be done in three different ways.

A template class can have a non-template function as a friend. A template class can have a template function as a friend.

A template class can also have a specialized template function as a friend.

Aliases

It is sometimes easier to define aliases for template classes using the key term *typedef*.

This allows us to use the alias as the full definition of the class in the program.

```
typedef stack <int> iStack;
typedef stack <double> dStack;
typedef stack <string, int> siStack;
```

Then we can use the type definitions in our program as shown below:

```
iStack s1;
dStack s2;
siStack s3;
```

Inheritance

We can derive a template class from another template class or from a non-template class.

For example, assume we have defined the class First as a template class:

```
template < typename T >
class First
{
      ...
}
```

We can then define another class Second which is publicly derived from the First class as shown below:

```
class Second : public First <T>
{
      ...
}
```

In Retrospect

We have discussed some classes in the past that they were actually template classes. In this section we quickly look at these classes as generic classes.

String Classes

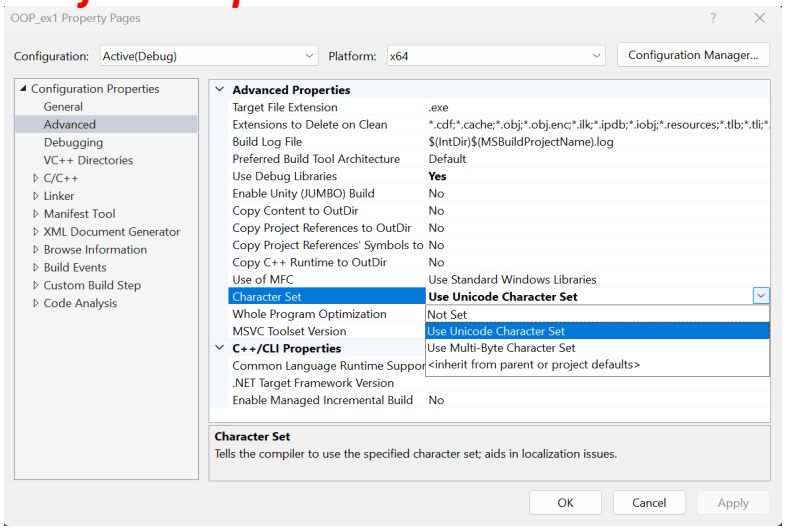
The string class is a specialization of a template class named *basic_string*. The following shows the general layout of this class.

```
template < typename charT >
class basic_string
{
      ...
};
```

```
typedef basic_string <char> string;
typedef basic_string <wchar_t> wstring;
```

The library class defines two specialization from this class, one for a *char* type and another for a *wchar_t* type.

Project Properties on Visual Studio



MSVC/include/xstring

```
#endif // ^^^ !_HAS_CXX20 ^^^

using string  = basic_string<char, char_traits<char>, allocator<char>>;
using wstring = basic_string<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>;
#ifdef __cpp_lib_char8_t
using u8string = basic_string<char8_t, char_traits<char8_t>, allocator<char8_t>>;
#endif // __cpp_lib_char8_t
using u16string = basic_string<char16_t, char_traits<char16_t>, allocator<char16_t>>;
using u32string = basic_string<char32_t, char_traits<char32_t>, allocator<char32_t>>;
```

In Retrospect

Input/Output Classes

Another set of classes that are actually generic classes are the input/output classes.

All of the input classes we discussed before are actually specialization of the template classes.

For example, the *istream* class is a specialization of the *basic_istream* class as shown below:

```
typedef basic_istream <char> istream;
typedef basic_istream <wchar_t> wistream;
```

MSVC/include/iosfwd

```
using ios           = basic_ios<char, char_traits<char>>;
using streambuf     = basic_streambuf<char, char_traits<char>>;
using istream       = basic_istream<char, char_traits<char>>;
using ostream       = basic_ostream<char, char_traits<char>>;
using iostream      = basic_iostream<char, char_traits<char>>;
using stringbuf     = basic_stringbuf<char, char_traits<char>, allocator<char>>;
using istringstream = basic_istringstream<char, char_traits<char>, allocator<char>>;
using ostringstream = basic_ostringstream<char, char_traits<char>, allocator<char>>;
using stringstream  = basic_stringstream<char, char_traits<char>, allocator<char>>;
using filebuf       = basic_filebuf<char, char_traits<char>>;
using ifstream      = basic_ifstream<char, char_traits<char>>;
using ofstream      = basic_ofstream<char, char_traits<char>>;
using fstream       = basic_fstream<char, char_traits<char>>;
#if _HAS_CXX20
using syncbuf     = basic_syncbuf<char>;
using osyncstream = basic_osyncstream<char>;
#endif // _HAS_CXX20

using wios           = basic_ios<wchar_t, char_traits<wchar_t>>;
using wstreambuf     = basic_streambuf<wchar_t, char_traits<wchar_t>>;
using wistream       = basic_istream<wchar_t, char_traits<wchar_t>>;
using wostream       = basic_ostream<wchar_t, char_traits<wchar_t>>;
using wiostream      = basic_iostream<wchar_t, char_traits<wchar_t>>;
using wstringbuf     = basic_stringbuf<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>;
using wistringstream = basic_istringstream<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>;
using wostringstream = basic_ostringstream<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>;
using wstringstream  = basic_stringstream<wchar_t, char_traits<wchar_t>, allocator<wchar_t>>;
using wfilebuf       = basic_filebuf<wchar_t, char_traits<wchar_t>>;
using wifstream      = basic_ifstream<wchar_t, char_traits<wchar_t>>;
using wofstream      = basic_ofstream<wchar_t, char_traits<wchar_t>>;
using wfstream       = basic_fstream<wchar_t, char_traits<wchar_t>>;
#if _HAS_CXX20
```

Summary

Highlights

- Function Templates
- Class Templates
- Issues on Templates

What's Next? (Reading Assignment)

- Read Chap. 19. Standard Template Library (STL)
- Read Chap. 16. Input/Output Streams

End of Class

Thank you

E-mail: youngcha@konkuk.ac.kr