# Topic 13 Template

Generic Programming

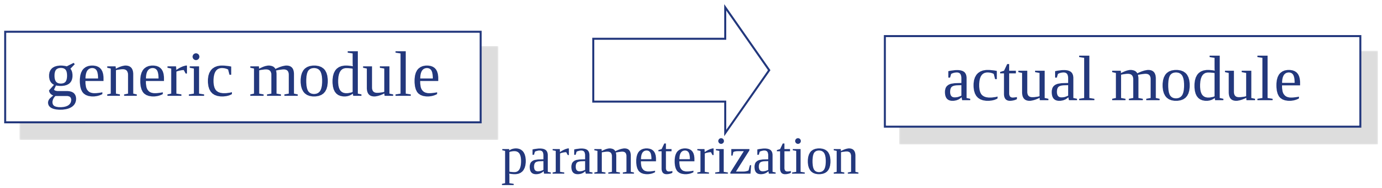
Macros vs. Template

Template Functions and Template Classes

Template Arguments

# Genericity

- Genericity
  - the ability to define parameterized modules
  - typically, the parameters (called *formal generic parameters*) stand for types
  - 범용성, 일반성
- Generic Module
  - parameterized module
  - not directly usable; rather a module pattern.
  - Actual modules, called instances of the generic module, are obtained by providing actual *types(actual generic parameters*) for each of the formal generic parameters.



# Generic Behavior of Function

- Mathematical operations, data sorting, searching, or swapping algorithms can be applied to different data types

```
int min(int x, int y) {
  if ( x > y ) return y ;
  else return x ;
}
```

```
float min(float x, float y) {
  if ( x > y ) return y ;
  else return x ;
}
```

```
Complex min(Complex  x, Complex  y) {
  if ( x > y ) return y ;
  else return x ;
}
```

# Generic Function Using Macros

## Macro Function

- macro functions are not real functions: the preprocessor handles them
- the parameters of macro functions has no type

```
#define MIN(x, y)       (x > y)? y: x
...
cout << MIN(2, 3) << endl;
cout << MIN(3.0, 2.5) << endl;
...
```

# Defects of Macros

- Operator precedence
  - 잘 못 사용하면 이해할 수 없는 오류 메시지/결과 유발

```
#define MIN(x, y)       (x > y)? y: x
...
cout << MIN(2, 3) * MIN(2, 3) << endl;
// '<<' : illegal, right operand has type 'class ostream &(__cdecl *)(class ostream &)'
cout << MIN(2, 3) ; // the result is 0, not 2. why ?
```

- Parameter occurrences
  - 증감연산자와 함께 사용할 경우 문제 발생 가능

```
#define MIN(x, y)       ( ((x) > (y))? (y): (x) )
...
cout << MIN(i++, j++);
...
```

# Macros in C++

- Rule of Thumb
  - Don't use macros unless you have to Read the document about your preprocessor before you use macros
  - Don't be too clever
  - Use capital letters in macro names
- Limitations of Macros
  - Macro names cannot be overloaded
  - Macro functions cannot be recursive

# Generic Function Using Template

- Templates can replace macro functions

```
# include <iostream>
template <class T>
const T& min(const T& data1, const T& data2) {
        return ( data1 < data2 ) ? data1 : data2 ;
}

int main() {
  int i1 = 3, i2 = 5 ;
  float f1 = 9.8F, f2 = 10.5F ;
  std::cout << min(i1, i2) << std::endl ; // compiler generates min for int
  std::cout <<   min(f1, f2) << std::endl ; // compiler generates min for float
}
```

# Templates and Operators

- Template functions may assume some operators be defined for their arguments types

```
class Complex {
  private:
    float r, i ;
  public:
    Complex(float _r=0.F, float _i=0.F) {
      r  = _r ; i = _i ;
    }
} ;
```

```
void f() {
  Complex c1, c2 ;
  min(c1, c2) ; // compiler generates min(Complex,Complex)
}
```

```
Complex &min(
    const Complex &data1,
    const Complex &data2) {
     return ( data1 < data2 ) ? data1 : data2 ;
}
```

no match for operator
< (class const Complex&,
class Complex&)

# Templates and Operators

- We should define the operators assumed by templates

```
class Complex {
  private:
    float r, i ;
  public:
    Complex(float _r=0.F, float _i=0.F) {
      r  = _r ; i = _i ;
    }
    bool operator < (const Complex& c) const {
      return size() < c.size() ;
    }
    float size() const {
      return sqrt(r*r + i*i) ;
    }
} ;
```

```
void f() {
  Complex c1(3, 4), c2(4, 5)
  Complex min_c = min(c1, c2) ;
}
```

# Exercise

- Implement class "Complex" and template function "find" for the following code.

```
int main() {
  int a[10] = { 0, 1,  2, 3, 4, 5, 6, 7, 8, 9} ;
  bool bFound = find(a, 10, 100) ;

  char b[5] = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’} ;
  bFound = find(b, 5, ‘e’) ;

  Complex c[5] = { Complex(0, 0), Complex(0, 1),
                   Complex(1, 1), Complex(1, 2), Complex(2, 2)} ;
  bFound = find(c, 5, Complex(2, 2)) ;
}
```

```
# include <cmath>

template <class T>
bool find(const T array[], int size, const T& value) {
   for ( int i = 0 ; i < size ; i ++ ) {
      if ( array[i] == value ) return true ; // assignment operator required !
   }
   return false ;
}

class Complex {
      float r, i ;
   public:
      Complex(float _r = 0.F, float _i = 0.F) { r  = _r ; i = _i ; }
      bool operator < (const Complex& c) const {
         return size() < c.size() ;
      }
      float size() const { return sqrt(r*r + i*i) ; }
      bool operator == (const Complex& c) const {
         return ( r == c.r && i == c.i ) ;
      }
} ;
```

# Template Class

- Template Class
  - A class can also have type parameters
- Generic Container Class
  - One of the most useful kinds of classes is the container class, that is, a class that holds objects of some type.
  - Examples: lists, arrays, sets, queues, stacks, etc.
  - Types of objects contained is of little interest to container class

# Template Stack

```
const int SIZE = 200 ;
template <class T>
class Stack {
      T elems[SIZE] ;
      int top ;
   public:
      Stack() { top = 0 ; }
      void Push(const T& elem) { elems[top++] = elem ; }
      T Pop() { return elems[--top] ; }
      bool Empty() const { return top == 0 ; }
} ;
```

```cpp
int main() {
    Stack<int> is;
    is.Push(100) ;
    cout << is.Pop() << endl ;

    Stack<char> cs;
    cs.Push('a'); cs.Push('b') ;
    cout << cs.Pop() << endl ;
```

```cpp
    Stack<Complex> cos ;
    cos.Push(Complex(10, 10)) ;
    cos.Push(Complex(20, 20)) ;
    cos.Pop() ;
    cos.Push(Complex(30,30)) ;
    while ( ! cos.Empty() )
            cout << cos.Pop() << endl ;
}
```

```cpp
class Complex {
   public:
      friend ostream& operator << (ostream& os, const Complex& c) ;
} ;
ostream& operator << (ostream& os, const Complex& c) {
   os << '(' << c.r << ',' << c.i << ')' ;
   return os ;
}
```

# Template Class in STL: vector<>

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

template <class T>
T* find(vector<T> data, T v) {
    for (int i = 0; i < data.size(); i++)
        if (data[i] == v)
            return &data[i];
    return 0;
}
int main() {
    vector<int> v;
    // insert some elements
    int *found = find(v, 7);
    if (found) ...
}
```

You can use more safe function data.at(i)

# Template Class in STL: limits

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

int main()
{
   cout << numeric_limits<char>::digits
        << " char₩n";
   cout << numeric_limits<unsigned char>::digits
        << " u  char₩n";
   cout << numeric_limits<wchar_t>::digits
        << " wchar_t₩n";
   cout << numeric_limits<int>::max()
        << " max int₩n";
   cout << numeric_limits<double>::max()
        << " max double " << endl;
}
```

**실행 결과:**
**7 char**
**8 u  char**
**16 wchar_t**
**2147483647 max int**
**1.79769e+308 max double**

# Another Template Stack

```
template <class T>
class Stack {
    T* pElems, *pTop;
    int size ;
    void copyFrom(const Stack&) ;
  public:
    Stack(int s = 50) { pElems = pTop = new T[size=s] ; }
    Stack(const Stack& s) { copyFrom(s) ; }
    Stack& operator = (const Stack& s) {
       delete [] pElems ;
       copyFrom(s) ;
       return *this ;
    }
    ~Stack() { delete [] pElems ; }
    T Pop() { return *--pTop ; }
    void Push(const T& elem) { *pTop ++ = elem ; }
    bool Empty() const { return pElems == pTop ; }
} ;
```

# Another Template Stack

```
template <class T>
void Stack<T>::copyFrom(const Stack<T>& s) {
  pElems = pTop = new T[size=s.size] ;
  for ( T* p = s.pElems  ; p != s.pTop ; p ++ )
    *pTop++ = *p ;
}
```

# Template List

```
template <class T>
class List {
      T * pElems;
      const  int size;
   public:
      List(int len = 100) : size(len) { pElems = new T[size]; }
      T& operator [] (int i) { return pElems[i] ; }
      int getSize() const { return size; }
      // Destructor, Copy constructor, and Assignment operator required!
} ;
template <class T>
ostream& operator << (ostream& os, List<T>& l) {
   os << '[';
   for (int i = 0 ; i < l.getSize(); i ++ ) os << l[i] << ' ' ;
   os << ']';
   return os;
}
```

# Template List

```
int main() {
   List<int> il;
   for ( int i = 0 ; i < il.getSize() ; i ++ )
      il[i] = i ;
   cout << il << endl ;

   List<char> cl(10);
   for ( int i = 0 ; i < cl.getSize() ; i ++ )
      cl[i] = 'A' + i ;
   cout << cl << endl ;

   List<Complex> col(20) ;
   for ( int i = 0 ; i < col.getSize() ; i ++ )
      col[i] = Complex(il[i], il[i]) ;
   cout << col << endl ;
}
```

# Genetic Sorting

- We want to sort a List of arbitrary type

```
template <class T> void sort(List<T>& v) {
  // bubble sorting
  unsigned int n = v.size() ;
  for ( int i = 0 ; i < n –1 ; i ++ )
    for ( int j = n-1 ; i<j ; j -- ) {
      if ( v[j] < v[j-1] ) {
        T temp(v[j]) ; // copy constructor
        v[j] = v[j-1] ;
        v[j-1] = temp ;
      }
    }
}
```

```
void f() {
  List<String> strList(3) ;
  strList[0] = String("DB");
  strList[1] = String("PL");
  strList[2] = String("C++");
  sort(strList) ;
}
```

# Genetic Sorting for a List of int

- Sorting an integer list

```
#include <cstdlib>
#include <ctime>

int main() {
  const int SIZE = 5;
  List<int> intList(SIZE);
  srand(static_cast<unsigned>(time(static_cast<time_t*>(0))));
  for (int i = 0; i < SIZE; ++i)
    intList[i] = rand() % 100;

  cout << "Before: " << intList << endl;
  sort(intList);
  cout << "After:  " << intList << endl;
}
```

```
time_t time( time_t *timer );
void srand( unsigned int seed );
int rand( void );
```

# class String

```
# include <cstring>
class String {
    char* str ;
  public:
    String() : str(0) {}
    String(char *s) { str = new char[strlen(s)]; strcpy(str, s); }
    ~String() { delete [] s ; }
    bool operator < (const String& s) { return (strcmp(str, s.getStr()) < 0); }
    String& operator = (const String& s) {
      if (str) delete [] str ;
      str = new char[s.getSize()+1] ;
      strcpy(str, s.getStr()) ;
    }
    int getSize() const { return strlen(str) ; }
    char* getStr() const { return str ; }
} ;
```

```
ostream& operator << (ostream& o, String s)
{
  return (o << s.getStr());
}
```

# Sorting of Strings

- Sorting a list of String: not working (Why?)

```
int main() {
  List<String> strList(3);
  strList[0] = String("DB");
  strList[1] = String("PL");
  strList[2] = String("C++");
  cout << "Before: " << strList << endl;
  sort(strList) ;
  cout << "After:  " << strList << endl;
}
```

# Sorting of Strings

- Shallow copy problem: the copy constructor should be supplied !

```
class String {
    …
    String(const String& s) {
      str = new char[s.getSize()+1] ;
      strcpy(str, s.getStr()) ;
    }
    …
} ;
```

```
template <class T> void sort(List<T>& v) {
  …
     if ( v[j] < v[j-1] ) {
       T temp(v[j])
       v[j] = v[j-1] ;
       v[j-1] = temp ;
       // temp freed here; v[j].str also freed
     }
  …
}
```

# Specialization

- A specialized version of template instantiation may be provided
  - a valuable degree of flexibility
  - important tool for performance tuning

```
template <> void sort(List<char *>& v) {
  // bubble sorting
  unsigned int n = v.size() ;
  for ( int i = 0 ; i < n ; i++ )
    for ( int j = n-1 ; i < j ; j-- ) {
      if ( strcmp(v[j], v[j-1]) < 0 ) {
        char *temp = v[j] ;
        v[j] = v[j-1] ;
        v[j-1] = temp ;
      }
    }
}
```

# Order of Specializations

- The order of specialization is important
  - more general version should appear before a special version

```
template <class T> void sort(List<T>& v)
{
  …
}
template <> void sort(List<char *>& v)
{
  …
}
```

# Template Arguments

- A template argument need not be a type name; constant expressions can be used
- In particular, integers can be useful as template arguments

```
template <class T, int sz>
class Stack {
    T items[sz] ;
    int nTop ;
  public:
  Stack() {…}
  T Pop() { return items[--nTop] ; }
  void Push(T v) { items[nTop++] = v ; }
} ;
```

```
void f() {
  Stack<int, 100> intStack ;
  Stack<float, 200> floatStack ;
}
```

# Template Arguments

- Two template class names refer to the same class if
  - their template names are identical and
  - their arguments have identical values

```
void f() {
    Stack<int, 20> s1 ;
    Stack<float, 20> s2 ;
    Stack<int, 20> s3 ;
    Stack<float, 100> s4 ;

    s1 = s2 ;        // error: type mismatch
    s1 = s3 ;        // ok
    s2 = s4 ;        // error: type mismatch
}
```

# typename

- You can use the keyword "typename" instead of "class"

```
template <typename T>
class List {
  …
} ;
```

- Keyword "typename" is also used for indicating that a qualified name is a type name.

```
template <class C> void h(C& v) {
  typename C::iterator i = v.begin();
  typename T1::y * z ;  // 포인터 선언
  // typename이 없으면 곱하기 연산자인지 아닌지 모호함

  …
} ;
```

# Exercise

- Implement a generic list using templates

```
int main() {
  List<Complex, 100> cList ;
  List<MyString, 200> sList ;

  int i1 = cList.add(Complex(0, 0)) ;
  cList.add(Complex(1, 1)) ;
  int i2 = sList.add("abc") ;
  sList.add("def") ;
  cList.find(Complex(1, 0)) ;
  sList.find("def") ;
  cList.remove(i1) ;
  sList.remove("abc") ;

  List<String, 200> s2List(sList) ;
  List<String, 200> s3List ;
  s3List.add("123") ;
  s3List = s2List ;
  s3List.remove("def") ;
}
```