# Generic Programming: Template

C++ Object Oriented Programming
Pei-yih Ting
NTOU CS

polymorphism: { static / dynamic / **parametric**

**templates** are code generators
default polymorphism for weak-typed languages



# Contents





# Generic Functions

- Suppose all you want to do is copy an array regardless of type

```
void copy(int arrayTo[], int arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
```

The same function won't work on an array of doubles.

- A traditional solution

```
typedef int genericType;
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
```

This only works one type at a time.

- A C++ solution: overload the function

```
void copy(int arrayTo[], int arrayFrom[], int n);
void copy(double arrayTo[], double arrayFrom[], int n);
```

You still have to write separate functions for each type.
You have to know in advance what types you need.



# Using Template to Achieve Generality

- Template for toplevel functions

C++98/03

keyword only (class) — parameter (genericType)

```
template<class genericType>
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
```

or equivalently,

```
template <typename genericType>
```

- Usage

```
void main() {
   int firstArray[] = {1, 2, 3};
   int secondArray[3];
   copy(secondArray, firstArray, 3);
}
```

or explicitly

```
copy<int>(secondArray, firstArray, 3)
```

- What happens: the compiler instantiates the function with *int* as argument. Compiler derives the type from your statement. If you call the same function with arrays of doubles, the compiler will instantiate a second overloaded function with *double* as argument.

# Multiple Template Parameters

- A template parameter cannot represent more than one type

```
template<class genericType>
void copy(genericType arrayTo[], genericType arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
void main() {
   int firstArray[] = {1, 2, 3};
   double secondArray[3];
   copy(secondArray, firstArray, 3);
}
```

```
error C2782: 'void __cdecl copy(genericType [],genericType [],int)' :
template parameter 'genericType' is ambiguous could be 'int' or 'double'
```

- **The solution**

```
copy<double,int>(secondArray, firstArray, 3)
```

```
template<class typeA, class typeB>
void copy(typeA arrayTo[], typeB arrayFrom[], int n) {
   for (int i=0; i<n; i++)
      arrayTo[i] = arrayFrom[i];
}
```



# Template Errors

- You may not violate syntax rules in a template

```
int array1[] = {1, 2, 3};
char *array2[3];
copy(array2, array1, 3);
```

```
error C2440: '=' : cannot convert from 'int' to 'char *'
```

- You may not violate semantics

```
template<class type>
type add(type x, type y) {
   return x+y;
}
void main() {
   int int1=5, int2=6;
   double double1=7.2, double2=4.3;
   int array1[] = {1,2,3}, array2[] = {4,5,6};
   cout << add(int1, int2) << endl;  // OK
   cout << add(double1, double2) << endl;  // OK
   cout << add(int1, double2) << endl;  // bad syntax
   cout << add(array1, array2) << endl;  //error C2110: cannot add two pointers
}
```

```
error C2782: 'type __cdecl add(type,type)' :
        template parameter 'type' is ambiguous
        could be 'double' or 'int'
```



# Improving the Semantics

```
template<class type>
type add(type x, type y) {
   return x+y;
}
class Array {
public:
   Array();
   void insert(int slot, double element);
   double get(int slot) const;
   void display() const;
   Array operator+(const Array &rhs) const;
private:
   double m_array[cArraySize];
};
void main() {
   Array array1, array2, array3;
   array1.insert(0, 2.2); array2.insert(0, 4.5);
   array3 = add(array1, array2);
   array3.display();
}
```

```
Array Array::operator+(const Array &rhs) const {
   Array tmp;
   for (int i=0; i<cArraySize; i++)
      tmp.m_array[i] = m_array[i] + rhs.m_array[i];
   return tmp;
}
```

```
Array::Array() {
   for (int i=0; i<cArraySize; i++)
      m_array[i] = 0;
}
```

Output
6.7 0 0



# Templates and Overloading

- You can overload a template function with another template function

```
template<class type>
type add(type x, type y) {
   return x+y;
}
template<class type>
type add(type x, type y, type z)
   {
   return x+y+z;
}
void main() {
   int x = 5;
   int y = 4;
   int z = 1;
   cout << add(x, y) << endl;
   cout << add(x, y, z) << endl;
}
```

Output
9
10

- Overloading is more commonly used to *avoid* a template, see next page

# Template and Overloading (cont'd)

- The template below will work fine with integers, doubles and chars

```
template <class type>
bool greaterThan(type x, type y) {
   return x > y;
}
```

But this template will fail with C char arrays.

- The solution is to provide an overloaded non-template function in addition to the template function

```
template <class type>
bool greaterThan(type x, type y)
   {
   return x > y;
}
```

```
bool greaterThan(char *str1, char *str2) {
   return strcmp(str1, str2) > 0;
}
```

- Rule for "signature matching" with templates: **non-template functions have precedence over template functions in matching function calls**



# Program Linkage Notes

- In a multi-file C++ project, we
  - put function prototypes in *.h file and put the definitions of each function in *.cpp files
  - put class declarations in *.h file and put the member function definitions in *.cpp files

  Which files should we put the **template function** into?

  *.cpp ? **No**. we should put template definitions into ***.h** file.

  - Remember that the compiler needs to instantiate the real function body according to the **template function call statement**. Therefore, the compiler need to know the complete template definitions before it can instantiate a template function after seeing the function call statement.
  - Previously, the compiler only need to know the declaration of each class or function. The actual function codes are only required at linkage step.



# Template Classes

- A template array example

```
template <class type>
class Array {
public:
   Array(int arraySize);
   ~Array();
   void insert(int slot, type element);
   type get(int slot) const;
private:
   int m_arraySize;
   type *m_data;
};
```

```
template<class type>
Array<type>::~Array() {
   delete[] m_data;
}
```

```
template<class type>
Array<type>::Array(int arraySize): m_arraySize(arraySize) {
   m_data = new type[arraySize];
}
```



# Template Classes (cont'd)

```
template<class type>
void Array<type>::insert(int slot, type element) {
   if (slot<m_arraySize && slot>=0)
      m_data[slot] = element;
   else
      cout << "Warning, out of range!\n";
}
template<class type>
type Array<type>::get(int slot) const {
 if (slot<m_arraySize && slot>=0)
      return m_data[slot];
   else
      cout << "Warning, out of range!\n";
   return 0; // return something
}
void main() {
   Array<int> array(20);
   array.insert(0, 10);
   cout << array.get(0);
}
```

Now you have a configurable array class that can hold chars, ints, doubles, strings, and other classes

# Templates with Constant Parameters

- Templates can have constant expressions (non-type parameter)

```
template<class type, int arraySize>
class Array {
public:
   void insert(int slot, type element);
   type get(int slot) const;
private:
   type m_data[arraySize];
};
```

also can be used to replace a **functor** in case a function requires compile time pre-configuration

- Usage

```
void main() {
   Array<int, 100> array;
   array.insert(99, 123);
   cout << array.get(99) << endl;
}
```

- Sample member function

```
template<class type, int arraySize>
void Array<type, arraySize>::insert(int slot, type element) {
   if (slot<arraySize && slot>=0)
      m_data[slot] = element;
   else
      cout << "Warning, out of range!\n";
}
```



# Templates and Static Data Member

- When a template class contains a static data member, each instantiation type has its own static data member
- Consider the following modification of the previous array template

```
template <class type>
class Array {
public:
   Array(int arraySize);
   ~Array();
   void insert(int slot, type element);
   type get(int slot) const;
private:
   int m_arraySize;
   type *m_data;
   static type sDefault;
};
```

**Or on a type by type basis (in .cpp file)**

```
int Array<int>::sDefault = 0;
char Array<char>::sDefault = '#';
```

- Every static data member must be *defined* outside the class
In the case of templates, we can do this generically (in .h file)

```
template<class type>
type Array<type>::sDefault = 0;
```



# Static Data Members (cont'd)

- The get() function returns the static data member

```
template<class type>
type Array<type>::get(int slot) const {
   if (slot<m_arraySize && slot>=0)
      return m_data[slot];
   else {
      cout << "Warning, out of range!\n";
      return sDefault; // return something
   }
}
```

- Usage

```
void main() {
   Array<char> array1(100);
   Array<int> array2(100);
   array1.insert(0, 'A');
   cout << array1.get(0) << endl;
   cout << array1.get(100) << endl; // out of range
   array2.insert(0, 5);
   cout << array2.get(0) << endl;
   cout << array2.get(100) << endl; // out of range
}
```

**Output**
**A**
**Warning, out of range!**
**#**
**5**
**Warning, out of range!**
**0**

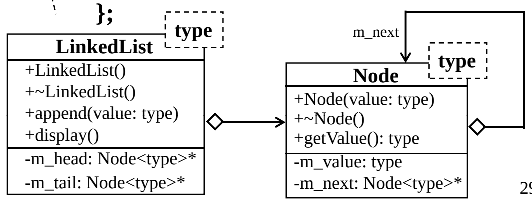


# Template Classes Working Together

- If a member object within a template class contains the parameterized variable, it must also be a template
- Example (linked list)

```
template <class type>
class LinkedList {
public:
   LinkedList();
   ~LinkedList();
   void append(type value);
   void display();
private:
   Node<type> *m_head;
   Node<type> *m_tail;
};
```

```
template <class type>
class Node {
   friend class LinkedList<type>;
private:
   Node(type value): m_value(value),
                     m_next(0) {}
   ~Node() { delete m_next; }
private:
   type m_value;
   Node<type> *m_next;
};
```



# Templates and Friends

```
template <class type>
LinkedList<type>::LinkedList(): m_head(0), m_tail(0) {
}
```

- Sample member function of the linked list

```
template<class type>
void LinkedList<type>::append(type value) {
   Node<type> *tmp = new Node<type>(value);
   if (m_head == 0)
      m_head = tmp;
   else
      m_tail->m_next = tmp;
   m_tail = tmp;
}
```

- Usage

```
void main() {
   LinkedList<char> myLinkedList;
   myLinkedList.append('A');
}
```



# Template Member Function

```
//------ MyClass.h -----
class MyClass {
public:
   MyClass(void);
   template <class T> void func(T x);
};
```

```
template <class T>
void MyClass::func(T x) {
   std::cout << x << std::endl;
}
```

```
#include <iostream>
template <class T>
void MyClass::func(T x) {
   std::cout << x << std::endl;
}
//------ end of MyClass.h -----
```



# Design Considerations

- Usage for templates: primarily for container classes, i.e. arrays, stacks, linked lists, map, etc
  Commonly used in class libraries
- Example of template library
  STL (Standard Template Library) … The Standard C++ Library
- How to write a good template?
  Avoid including elements to the template that will defeat its generality.
- Examples:
  - a function that adds together two components
  - you can't use the template on any class that doesn't overload+
- Document the template thoroughly.
  State which types will not work with the template.
  State which functions you expect to be available, e.g., +



# More Templates

- **Managed Pointers** **(CPP18-TheBigThree_splitted.pdf, pp.13-14, ppt)**
- **Inherit a template class** **(CPP25-AdvancedInheritance_splitted.pdf, pp.18-19, pp**
- **Java Generics** **(Java-Generics-Finalized.pdf, ppt)**
- **C++ Template Meta-programming (TMP, compile time programming, programming with types, see** https://en.wikibooks.org/wiki/C%2B%2B_Programming/Templates/Template_Meta-Programming**)** **For example**: factorial n! = n*(n-1)!

```
template <unsigned n> struct factorial {
   enum { value = n * factorial<n-1>::value };
};
template <> struct factorial<0> { enum { value = 1 }; };
int array[ factorial<7>::value ];  // compile-time constant
```

**7 * factorial<6>::value**
**7 * 6 * factorial<5>::value**
**7 * 6 * 5 * factorial<4>::value**
**…**
**7 * 6 * 5 * 4 * 3 * 2 * 1 * 1**

# References


- M. H. Austern, Generic Programming and the STL
  - 泛型程式設計與 STL, 侯捷 / 榮耀 / 姜宏 譯
- S. Meyers, Effective STL
  - Effective STL, 龚敏敏 譯
- 侯捷, STL 源碼解析, The Annotated STL Source – Using SGI STL
- D. Vandevoorde and N. M. Josuttis, C++ Templates – The Complete Guide
  - The C++ Templates, 侯捷 / 榮耀 / 姜宏 譯
- A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied
  - C++ 設計新思維,侯捷 / 於春景 譯
- D. Abrahams and A. Gurtovoy, C++ Template Metaprogramming
  - C++ 模板元編程 - 來自（並超越）Boost 的概念、工具、和技術, 侯捷 / 榮耀 / 姜宏 譯