# Object-Oriented Programming and Data Structures

# COMP2012: Generic Programming With Class and Function Templates

Prof. Brian Mak
Prof. C. K. Tang

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

# How Many my_max( ) Functions Do You Need?

```
int my_max(int a, int b) { return (a > b) ? a : b; }

char my_max(char a, char b) { return (a > b) ? a : b; }

double my_max(double a, double b) { return (a > b) ? a : b; }

#include <string>
string my_max(string a, string b) { return (a > b) ? a : b; }

#include "teacher.h"
Teacher my_max(const Teacher& a, const Teacher& b)
    { return (a > b) ? a : b; }
```

# How Many Stack Classes Do You Need?

```
class int_stack {
  private: int data[100]; int top_index;
  public: int_stack( );
    int top( ) const; void push(int); void pop( );
    bool empty( ) const; bool full( ) const; int size( ) const;
};

class char_stack {
  private: char data[100]; int top_index;
  public: char_stack( );
    char top( ) const; void push(char); void pop( );
    bool empty( ) const; bool full( ) const; int size( ) const;
};

#include "student.h"
class student_stack {
  private: Student data[100]; int top_index;
  public: student_stack( );
    Student top( ) const; void push(Student); void pop( );
    bool empty( ) const; bool full( ) const; int size( ) const;
};
```

# Generic Programming using Templates

- A lot of times, we find functions and data structures that look alike: they differ only in the types of objects they manipulate.
- Since C++ allows function overloading, one may define many **my_max( )** functions, one for each type of values/objects **T**, but they all have the following general form:

  T my_max(`const` T& a, `const` T& b) { ... }
- For stacks of different types of objects, one has to make up different class names for them (int_stack, char_stack, etc.).
- Again, we don't like the solution of creating the various **my_max( )** or stacks by "copy-and-paste-and-modify".
- The solution is generic programming using function templates and class templates.
- They are similar to function definitions and class definitions but the types of objects they manipulate are parameterized with type variables.
- Generic programming allows programmers to write just one version of code that works for different types of objects.

- It starts with the keyword template.

```
template <typename T>                /* File: max-template.cpp */
T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

- The typename keyword may be replaced by class.

```
template <class T>                   /* File: max-template2.cpp */
T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

- This is just a function template definition; it itself is not a real function and cannot be called directly.

# Example: Use of the my_max( ) Function Template

- Now we can use **my_max( )** for any types, as long as the function body codes make sense for the types they represent.
- In the case of **my_max( )**, it is required that the types can be compared by the operator " >".

```
#include <iostream>                                   /* File: max-calls.cpp */
using namespace std;
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }

int main( )
{
    int x = 4, y = 8;
    cout << my_max(x, y) << " is a bigger number!" << endl;

    string a("cheetah"), b("gorilla");
    cout << my_max(a, b) << " is stronger!" << endl;
    return 0;
}
```

# Function Template Instantiation

- Based on the function template definition, the compiler will create the real functions when they are called.
- This is called template instantiation. The parameter **T** in the template definition is called the formal parameter or formal argument.
- In our case, the compiler creates two **my_max( )** functions using the function template.

```
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }
```

- The template is instantiated with the actual arguments **int** and **string**, respectively.

```
#include <iostream>                              /* File: max-match-arg.cpp */
using namespace std;
template <typename T>
T my_max(const T& a, const T& b) { return (a > b) ? a : b; }

int main( ) {
    cout << my_max(3, 5) << endl;                               // T is int;
    cout << my_max(4.3, 5.6) << endl;                           // T is double
}
```

- When the compiler instantiates a template, it determines the actual type of the template parameter by looking at the types of the actual arguments.
- However, there is no automatic type conversion for template arguments.

```
cout << my_max(4, 5.5);     // Error
```

- You can help by explicitly instantiating the function template.

```
cout << my_max<double>(4, 5.5);
```

# Function Template w/ More Than One Formal Argument

```
#include <iostream>                          /* File: fcn-template-2arg.cpp */
using namespace std;

template <typename T1, typename T2>
T1 max(const T1& a, const T2& b) { return (a > b) ? a : b; }

int main( )
{
    cout << max(4, 5.5) << endl;             // T1 is int, T2 is double
    cout << max(5.5, 4) << endl;             // T1 is double, T2 is int
}
```

- A template may take more than one type arguments, each using a different typename.
- However, there is a subtle problem in this case: the return type of this **my_max** is the type of the first argument.
- So what will the above code print?

- The following template definition does not suffer from the problem but it doesn't return a value.

```
#include <iostream>                /* File: fcn-template-2arg-ok.cpp */
using namespace std;

template <typename T1, typename T2>
void print_max(const T1& a, const T2& b)
{
    if (a > b)
        cout << a << endl;
    else
        cout << b << endl;
}

int main( ) { print_max(4, 5.5); print_max(5.5, 4); }
```

# Template Arguments: Too Many Combinations

```
/* File: many-combination.cpp */
short s = 1; char c = 'A';
int i = 1023; double d = 3.1415;

print_max(s, s); print_max(s, c);
print_max(c, s); print_max(s, i);
// ... And all other combinations; 16 in total.
```

- With the above code, the compiler will instantiate a **print_max( )** for each of the 16 different combinations of arguments.
- With the current compiler technology, this means that we get 16 (almost identical) fragments of code in the executable program. There is no sharing of code.
- So a simple program may have a surprisingly large binary size, if we are not careful.

# Function Template: Common Errors

```
#include <iostream>                    /* File: fcn-template-error.cpp */
using namespace std;
template <class T> T* create( ) { return new T; };
template <class T> void f( ) { T a; cout << a << endl; }
int main( ) { create( ); f( ); }
```

```
fcn-template-error.cpp:5:15: error: no matching function for call to 'create'
int main( ) { create( ); f( ); }
              ^~~~~~
fcn-template-error.cpp:3:23: note: candidate template ignored: couldn't infer
      template argument 'T'
template <class T> T* create( ) { return new T; };
                      ^
fcn-template-error.cpp:5:26: error: no matching function for call to 'f'
int main( ) { create( ); f( ); }
                         ^
fcn-template-error.cpp:4:25: note: candidate template ignored: couldn't infer
      template argument 'T'
template <class T> void f( ) { T a; cout << a << endl; }
                        ^
```

The compiler can't deduce the actual object types from such calls.

# Class Template for Nodes of a List

- The template mechanism works for classes as well. This is particularly useful for defining container classes — classes that contains objects of the same kind such as arrays, lists, and sets.

```
template <typename T>                          /* File: listnode.h */
class List_Node
{
  public:
    List_Node(const T& data)
        : _data(data), _next(0), _prev(0) { }

    List_Node* _next;
    List_Node* _prev;
    T _data;
};
```

# Class Template for a List

```
#include "listnode.h"                                   /* File: list.h */
template <typename T> class List                        // A doubly linked list
{
  public:
    List( ) : _head(0), _tail(0) { }
    void append(const T& item) {
        List_Node<T>* new_node = new List_Node<T>(item);
        if(!_tail)
            _head = _tail = new_node;
        else
        {                                               /* incomplete */ }
    }
    void print( ) const {
        for (const List_Node<T>* p = _head; p; p = p->_next)
            cout << p->_data << endl;
    }
    // ... Other member functions
  private:
    List_Node<T>* _head;
    List_Node<T>* _tail;
};
```

# Class Template: List Example

- Now we can use the parameterized list class template to create lists to store any type of element that we want, without having to resort to "code re-use by copying".

```
#include <iostream>                    /* File: list-example.cpp */
using namespace std;
#include "list.h"
#include "student.h"

int main( )
{
    List<char> letters; letters.append('a');
    cout << "*** print char list *** \n"; letters.print( );

    List<int> primes; primes.append(2);
    cout << "### print int list ###\n"; primes.print( );

    List<Student> students;
    students.append(Student("James", CSE, 4.0));
    students.append(Student("Billy", ECE, 3.5));
}
```

# Nontype Parameters for Templates

- Template may also have nontype parameters, which are not type variables.

```
#include "listnode.h"                                 /* File: nontype-list.h */
template <typename T, int max_num_items>               // A doubly linked list
class List {
  public:
    List( ) : num_items(0), _head(0), _tail(0) { }
    bool append(const T& item)
    {
        if (num_items == max_num_items) {
            std::cerr << "List is full\n"; return false;
        }
        else {                          /* incomplete */ return true; }
    }
    // ... Other member functions

  private:
    int num_items;
    List_Node<T>* _head, _tail;
};
```

# Difference Between Class and Function Templates

- For function templates, the compiler can deduce the template arguments.

```
int i = my_max(4, 5);   // Rely on compilers to deduce my_max<int>
int j = my_max<int>(7, 2);                 // Explicitly instantiation
```

- For class templates, you always have to specify the actual template arguments; the compiler does not deduce the template arguments.

```
List primes;          // Error: how can compilers deduce the type?
primes.append(2);     // Error: too late; compilers can't lookahead!
```

# Separate Compilation For Templates??

- For regular non-template functions, we usually put their declarations in a header file, and their definitions in a corresponding .cpp file.
- Should we do the same for templates?

```
/* File: max.h */
template <typename T> T my_max(const T& a, const T& b);

/* File: max.cpp */
template <typename T> T my_max(const T& a, const T& b)
{
    return (a > b) ? a : b;
}
```

- But a function/class is instantiated only when it is used.
- No, we usually put the function definitions in the header file as well and include the template header file in every files which use the template.