

Function Templates

Version 1: Dr. Ofir Pele

Version 2: Dr. Erel Segal-Halevi

Motivation

A useful routine to have is

```
void swap( int& a, int &b )  
{  
    int tmp = a;  
    a = b;  
    b = tmp;  
}
```

Example

What happens if we want to swap a double ? or a string?

For each one, we need different function:

```
void swap( double& a, double &b )  
{  
    double tmp = a; a = b; b = tmp;  
}
```

```
void swap( string& a, string &b )  
{  
    string tmp = a; a = b; b = tmp;  
}
```

Generics Using void*

C approach:

```
void swap( void *a, void *b, size_t size )
{
    for (size_t i=0; i<size; i++) {
        char t = *(char *)(a+i);
        *(char *)(a+i) = *(char*)(b+i);
        *(char*)(b+i) = t;
    }
    // or can be done using malloc and memcpy
}
```

Function Templates

The `template` keyword defines “templates”

Piece of code that will be regenerated with different arguments each time

```
template<typename T> // T is a
                      //"type argument"
void swap( T& a, T& b )
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Template Instantiation

Explicit

```
double d1 = 4.5, d2 = 6.7;  
swap<double>(d1, d2);
```

The compiler generates and compiles:

```
swap<double>(double&,double&)
```

Implicit

```
double d1 = 4.5, d2 = 6.7;  
swap(d1, d2);
```

The compiler generates and compiles:

```
swap<double>(double&,double&)
```

Template Instantiation

Different instantiations of a template can be generated by the same program

```
int a = 2, b = 3;  
swap( a, b ); // Compiler generates swap(int&,int&)  
swap( b, a ); // No need to generate a new function
```

```
double x = 0.1, y = 1.0;  
swap( x, y ); // Compiler generates swap(double&,double&)
```

```
char* s = "sss", t = "ttt";  
swap( s, t ); // Compiler generates swap(char*&,char*&)
```

```
Member s, t;  
swap( s, t ); // Compiler generates swap(Member&,Member&)
```

Technical comment

These two definitions are exactly the same, if we have both, we will have a compilation error:

```
template <typename T>
void swap(T &a, T &b)
{
    T c = a;
    a = b;
    b = c;
}
```

```
template <class P>
void swap(P &a, P &b)
{
    P c = a;
    a = b;
    b = c;
}
```


Template Instantiation – godbolt.org example

```
template <typename T> void swap(T& a, T& b) {  
    T tmp = a;  a = b; b = tmp;  
}
```

```
void swap (double& a, double& b) {  
    a = a + b;  b = a - b;  a = a - b;  
}
```

```
int main() {  
    int a=4,b=5;  
    swap(a,b);  
    double c=4.1,d=5.1;  
    swap(c,d);  
    swap<double>(c,d);  
}
```

Templates & Compilation

- A template is a **declaration**
- The compiler performs functions/operators calls checks only when a template is instantiated with specific arguments - then the generated code is compiled.



Not just the
declaration

Implications:

1. Template **code** has to be visible by the code that uses it (i.e., appear in header *.h/.hpp* file)
2. Compilation errors can occur only in a specific instance

Template assumptions (folder 1)

// example of a uncopyable class

```
class Cookie
```

```
{
```

```
private:
```

```
    // private copy operations
```

```
    Cookie(const Cookie&);
```

```
    Cookie & operator=(const Cookie&);
```

```
public:
```

```
    Cookie() { };
```

```
};
```

```
...
```

```
    Cookie vanilla, chocolate;
```

```
    swap(vanilla, chocolate); /* compiler will try generate  
code for swap(Cookie&,Cookie&), but will fail, claiming an  
error somewhere at the declaration of swap*/
```



?Why

Another Example - max

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

What are the
template
assumptions

Another Example - sort

```
// Inefficient generic sort
template< typename T >
void sort( T* begin, T* end )
{
    for( ; begin != end; begin++ )
        for( T* q = begin+1; q != end; q++ )
        {
            if( *q < *begin )
                swap( *q, *begin );
        }
}
```

What are the
template
assumptions

Another Example

```
// Inefficient generic sort
template< typename T >
void sort( T* begin, T* end )
{
    for( ; begin != end; begin++ )
        for( T* q = begin+1; q != end; q++ )
        {
            if( *q < *begin )
                swap( *q, *begin );
        }
}
```

What are the
template
assumptions

Usage

Suppose we want to avoid writing operator != for new classes

```
template <typename T>
bool operator!=(T const& lhs, T const& rhs)
{
    return !(lhs == rhs);
}
```

When is this template used?

Usage

```
class MyClass
{
public:
    bool operator==
        (MyClass const & rhs) const;
};

int a, b;
if( a != b ) // uses built in
               // operator!=(int,int)

...
MyClass x,y;
if( x != y ) // uses template with
               // T= MyClass
```


Puzzle

Can you define all 6 operators

($<$ $>$ $==$ $<=$ $>=$ $!=$)

using only 1 of them?

Which one?

When Templates are Used? – overloads, again

When the compiler encounters

...

f(a, b)

...

1. Look for all functions named f.
2. Creates instances of all templates named f according to parameters.
3. Order them by matching quality (1..4).
4. **Within** same quality, prefer non-template over template.

Example 1

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

void foo(double x) {
    std::cout << "foo(double)\n";
}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}

int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

Example 1

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

void foo(double x) {
    std::cout << "foo(double)\n";
}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}

int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

A terminal window showing the output of the program. The output consists of four lines: 'foo(int)', 'foo(double)', 'foo(char)<T*>', and 'Press any key to continue . . . _'. The text is displayed in a monospaced font on a black background with a blue border.

```
foo(int)
foo(double)
foo(char)<T*>
Press any key to continue . . . _
```

Example 2

```
#include <iostream>
#include <typeinfo>

//void foo(int x) {
//    std::cout << "foo(int)\n";
//}
void foo(double x) {
    std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

Example 2

```
#include <iostream>
#include <typeinfo>

//void foo(int x) {
//    std::cout << "foo(int)\n";
//}
void foo(double x) {
    std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

A terminal window showing the output of the C++ program. The output consists of three lines: 'foo(double)', 'foo(double)', and 'foo<char>(T*)'. Below these lines is a prompt 'Press any key to continue' followed by a series of dots and a cursor line.

```
foo(double)
foo(double)
foo<char>(T*)
Press any key to continue . . . _
```

Example 3

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

//void foo(double x) {
//    std::cout << "foo(double)\n";
//}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}

int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

Example 3

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

//void foo(double x) {
//    std::cout << "foo(double)\n";
//}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}

int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

A screenshot of a terminal window showing the output of the program. The output consists of three lines: 'foo<int>', 'foo<int>', and 'foo<char>(T*)'. Below these lines is a prompt 'Press any key to continue' followed by a cursor and three dots.

```
foo<int>
foo<int>
foo<char>(T*)
Press any key to continue . . . _
```


Example 3

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

//void foo(double x) {
//    std::cout << "foo(double)\n";
//}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}

int main() {
    foo(42);
    foo(42.2);
    foo("abcdef");
    return 0;
}
```

Won't compile with the flag:
-Wconversion (forbids lossy
conversions)

Example 4

```
#include <iostream>
#include <typeinfo>

//void foo(int x) {
//    std::cout << "foo(int)\n";
//}
void foo(double x) {
    std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
int main() {
    foo(42);
    foo(42.0);
    foo("abcdef");
    return 0;
}
```

Will compile with the flag:
-Wconversion (allows lossless
conversion)

Example 5

```
template<typename T>
void f(T x, T y)
{
    cout << "Template" << endl;
}

void f(int w, int z)
{
    cout << "Non-template" << endl;
}

int main(){
    f( 1 , 2 );
    f('a', 'b');
    f( 1 , 'b');
}
```



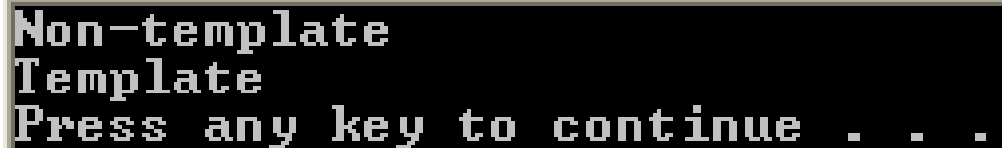
```
Non-template
Template
Non-template
Press any key to continue . . . _
```

Example 6

```
template<typename T>
void f(T x, T y)
{
    cout << "Template" << endl;
}

void f(int w, int z)
{
    cout << "Non-template" << endl;
}

int main(){
    f( 2 , 1.2);
    f( 2.2 , 1.2);
}
```



```
Non-template
Template
Press any key to continue . . .
```

Example 7

```
template <typename T> void f(T) { cout << "Less specialized"; }  
template <typename T> void f(T*) { cout << "More specialized"; }  
int main() {  
    int i = 0;  
    int *pi = &i;  
    f(i); // Calls less specialized function.  
    f(pi); // Calls more specialized function.  
}
```

1

2

Is there a type that fits 1 and will not fit 2? If so, 2 is more specialized and should be preferred.

Variable Templates

Template variables (folder 1)

```
template<typename T> const T pi =  
    T(3.1415926535897932385L); // variable template
```

```
template<typename T> T circular_area(T r) {  
    return pi<T> * r * r;  
}
```

```
int main() {  
    return circular_area(5); // 75?  
}
```

Class Templates

String Stack

```
class StrStk {
public:
    StrStk():m_first(nullptr) { }
    void push (string const& s ) {m_first=new Node(s,m_first);}
    bool isEmpty() const {return m_first==nullptr;}
    const string& top () const {return m_first->m_value;}
    void pop ()
    {Node *n=m_first; m_first=m_first->m_next; delete n;}
    ~StrStk() { while (!isEmpty()) pop(); }
private:
    StrStk(StrStk const& rhs);      StrStk& operator=(StrStk const& rhs);

    struct Node {
        string m_value;
        Node* m_next;
        Node(string const& v ,Node* n):m_value(v),m_next(n) { }
    };
    Node* m_first;
};
```

Generic Classes

- The actual code for maintaining the stack has nothing to do with the particulars of the string type.
- Can we have a generic implementation of stack?

Generic Stack (folder 2)

```
template <typename T> class Stk {
public:
    Stk():m_first(nullptr) { }
    ~Stk() { while (!isEmpty()) pop(); }
    void push (const T& s) {m_first=new Node(s,m_first);}
    bool isEmpty() const {return m_first==nullptr;}
    const T& top () const {return m_first->m_value;}
    void pop ()
    {Node *n=m_first; m_first=m_first->m_next; delete n;}
private:
    Stk(const Stk& rhs); Stk& operator=(const Stk& rhs);
    struct Node {
        T m_value;
        Node* m_next;
        Node(const T& v ,Node* n):m_value(v),m_next(n) { }
    };
    Node* m_first;
};
```

Class Templates

```
template<typename T>  
class Stk  
{  
    ...  
};
```

```
Stk<int> intList; // T = int
```

```
Stk<string> stringList; // T = string
```

Class Templates

The code is similar to non-template code, but:

- Add `template<...>` statement before the class definition
- Use template argument as type in class definition
- To implement methods outside the class definition (but still in header: .h.hpp, not in a cpp file!):

```
template <typename T>
bool Stk<T>::isEmpty() const
{
    return m_first==nullptr;
}
```

Example of generic programming - Iterators

Constructing a List

- We want to initialize a stack from a primitive array.
- `int arr[6];`
- We can use a pointer to initial position and **one to the position after the last:**
- `Stk<int> myStack(
 arr, arr+sizeof(arr)/sizeof(*arr));`

Constructing a List

// Fancy copy from array

```
template< typename T >
```

```
Stk<T>::Stk<T>(const T* begin, const T* end) {
```

```
    for(; begin!=end; ++begin) {
```

```
        push(*begin);
```

```
    }
```

```
}
```


Pointer Paradigm

Code like:

```
const T* begin=theList;  
const T* end=  
list+sizeof(theList)/sizeof(*theList);  
for(; begin!=end; ++begin)  
{  
    // Do something with *begin  
}
```

- Applies to all elements in [begin,end-1]
- Common in C/C++ programs
- Can we extend it to other containers?

Iterator

- **Object that behaves “almost” like a pointer**
- Allows to iterate over elements of a container

Example:

```
Stk<int> L;
```

```
...
```

```
Stk<int>::iterator i;
```

```
for( i = L.begin(); i != L.end(); ++i)
```

```
    cout << " " << *i << "\n";
```

Iterators

To emulate pointers, we need:

1. copy constructor
2. `operator=` (copy)
3. `operator==`, `!=` (compare)
4. `operator*` (access value)
5. `operator++` (increment)

And maybe:

6. `operator[]` (random access)
7. `operator+=` / `-=` (random jump)
8. ...

Stk<T> iterator (folder 2)

Create an inner class, keep a pointer to a node.

```
class iterator
{
private:
    Node *m_pointer;
};
```

Provides encapsulation, since through such an iterator we cannot change the structure of the list

Initializing a Stk

We now want to initialize a stack from using parts of another stack. Something like:

```
Stk(iterator begin, iterator end) {  
    for(; begin!=end; ++begin) {  
        push(*begin);  
    }  
}
```

Initializing a Stk

Compare:

```
Stk<T>::Stk<T>(iterator begin, iterator end) {  
    for(; begin!=end; ++begin) {  
        push(*begin);  
    }  
}
```

To:

```
Stk<T>::Stk<T>(const T* begin, const T* end) {  
    for(; begin!=end; ++begin) {  
        push(*begin);  
    }  
}
```

Generic Constructor

The code for copying using

- T^*
- `Stk<T>::iterator`

are essentially identical on purpose --- iterators mimic pointers!

Can we write the code once?

Yes: template inside template (folder 2)

Class exercise

Write a function for summing all elements of a container.

The function should work with all kinds of containers:

- Native array;
- Linked list;
- vector;
- user defined...