

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*&)
```

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

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 ostream
```

```
{
```

```
private:
```

```
    // private copy operations
```

```
    ostream(const ostream&) = delete;
```

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

```
public:
```

```
    ostream() { };
```

```
};
```

```
...
```

```
swap(cout, cerr); /* compiler will try generate code for  
swap(ostream&,ostream&), 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?

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.
4. **Within** same quality, prefer non-template over template.

Example 1

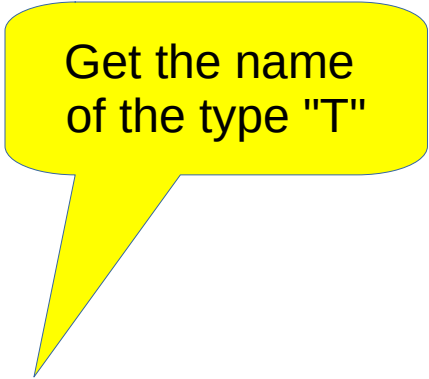
```
#include <iostream>
#include <typeinfo>

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;
}
```



Get the name
of the type "T"

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 screenshot of 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.

```
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 screenshot of a terminal window with a black background and white text. It shows the output of the C++ program: 'foo(double)' on the first line, 'foo(double)' on the second line, 'foo<char>(T*)' on the third line, and 'Press any key to continue . . . _' on the fourth 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 terminal window showing the output of the program. The first two lines are "foo<int>" and "foo<int>". The third line is "foo<char><T*>". The fourth line is "Press any key to continue . . . _".

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


Example 4

```
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');
}
```



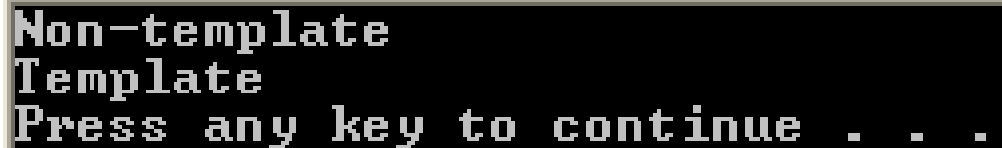
```
Non-template
Template
Non-template
Press any key to continue . . . _
```

Example 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( 2 , 1.2);
    f( 2.2 , 1.2);
}
```

A screenshot of a terminal window with a black background and white text. It shows the output of the program: "Non-template" on the first line, "Template" on the second line, and "Press any key to continue . . ." on the third line.

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

Example 6

```
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() {
```

```
    cout << circular_area(5); // 75
```

```
    cout << circular_area(5.0); // 78...
```

```
}
```