CONCEPTS

The primary motivation of concepts is to improve the quality of compiler error messages. If a programmer attempts to use a type that does not provide the interface a template requires, the compiler will generate an error. However, such errors are often difficult to understand. There are two main reasons for this:

1. Error messages are often displayed with template parameters spelled out in full, this leads to extremely large error messages.

2.They often do not immediately refer to the actual location of the error.

The main goal is to produce short, meaningful error messages for errors in the use of templates.

To understand “concepts” , we plan to

1. Statically analyse any class and determine the operations supported by it.

2. Statically analyse any template function and determine the operations that the template parameter type should support.

3. Whenever a template function is called, report errors if the argument type does not support operations required by the template parameter type.

Highlights of this project

Case 1:

```
class InputClass
{
   private:

   public:
        InputClass();
        ~InputClass();
        InputClass(const InputClass& ic);
        InputClass& operator=(const InputClass& ic);
};
```

After static analysis of the class, a class table is generated, which is as follows:

InputClass	0_arg_ctor	dtor	copy_ctor	Operator=

```
template<typename T>
T add(T a,T b)
{
        T c;
        c = a + b;
        return c;
}
```

After static analysis of the template function, a function table is generated, which is as follows:
add:

T	0_arg_ctor	dtor	copy_ctor	operator=	operator+

When add function is called with two args of type Input class, function table and class table are compared, if any requirements of the function are not satisfied by the class, an error is generated.

Case 2: Inheritance

```
class A
{
   private:
          A& operator*(const A& a);
   public:
         A();
         ~A();
         A(const A& a);
         A& operator=(const A& a);
          A operator+(const A& a);
};

class B : public A
{
        public:
                B();
                ~B();
                B(const B& b);
                B& operator=(const B& b);
};
```

Class table of base class is as follows:

A	ctor	dtor	copy_Ctor	operator=	operator+

The derived class B, inherits all the public member functions of class A. Hence the class table is as follows:

B	ctor	dtor	copy_Ctor	operator=	operator+ (of base class A)

Case 3: Friend function

```
class InputClass
{
        public:
                InputClass();
                ~InputClass();
                InputClass(const InputClass& ic);
                InputClass& operator=(const InputClass& ic);
                friend InputClass operator*(const InputClass&,const
InputClass&);
};
```

Class table for InputClass is as follows:

InputClass	0_arg_ ctor	dtor	copy_ctor	operator=	friend_operator*

```
template<typename T>
T mul_by_2(T a)
{
        T c;
        c = 2 * a;
        return c;
}
```

Function table for mul_by_2 is as follows:

mul_by_2	0_arg_ ctor	dtor	copy_ctor	operator=	friend_operator*