Generics Everywhere

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM

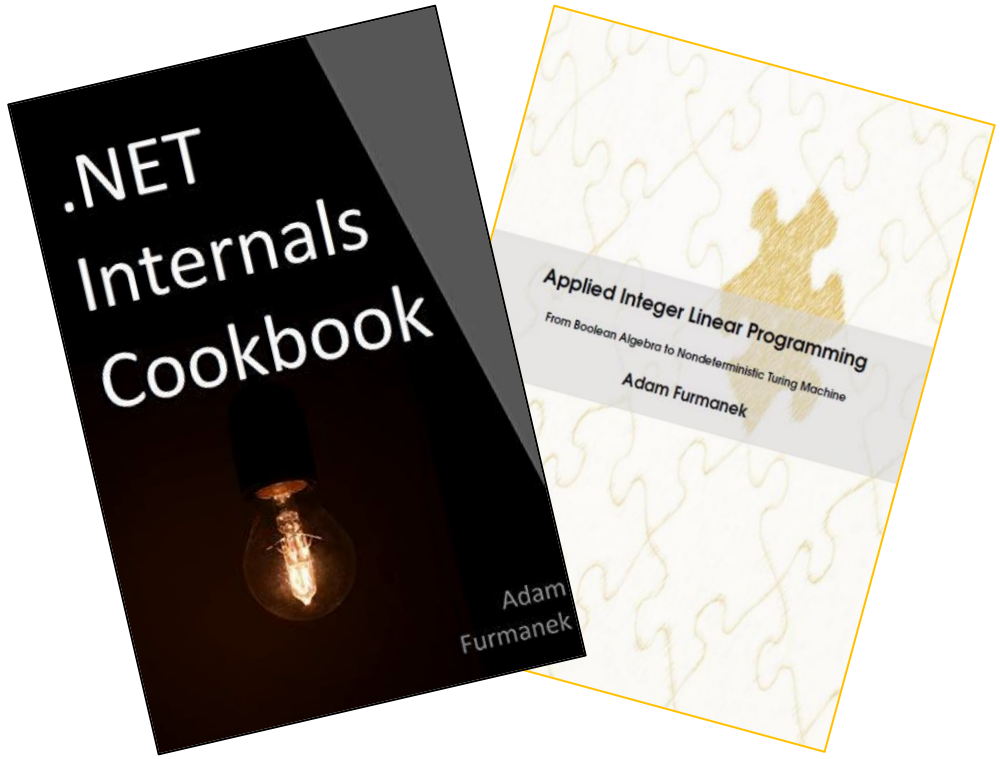
About me

Software Engineer, Blogger, Book Writer, Public Speaker.
Author of ***Applied Integer Linear Programming*** and ***.NET Internals Cookbook***.

http://blog.adamfurmanek.pl

contact@adamfurmanek.pl

furmanekadam



Random IT Utensils

IT, operating systems, maths, and more.

Agenda

Why do we need reusability.

Templates in C++.

Erasure in Java.

Reification in .NET.

Reusability

Use of existing assets in some form within the software product development process.

```
int maximum(int left, int right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}


int main()
{
    int a = 3;
    int b = 2;
    maximum(a, b);
}
```

Why do we need reusability

```
int main()
{
    std::string a = "3";
    std::string b = "2";
    maximum(a, b);
}
```

std::string a

no suitable conversion function from "std::string" to "int" exists

```
std::string maximum(std::string left, std::string right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main()
{
    std::string a = "3";
    std::string b = "2";
    maximum(a, b);
}
```

Why do we need reusability

```
int maximum(int left, int right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}
```

```
std::string maximum(std::string left, std::string right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}
```

Generic programming

Style of computer programming in which algorithms are written in terms of types to-be-specified-later that are then instantiated when needed for specific types provided as parameters.

Generic programming

Not new – started in ML in 1973.

Most popular on function and class level.

Multiple flavours – templates, generics, type classes, polytypic functions.

Used for code reuse, static-time reflection, metaprogramming.

Templates in C++

```
template<typename T>
T maximum(T left, T right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main()
{
    maximum(2, 3);
    maximum("2", "3");
}
```

```cpp
template<typename T>
T maximum(T left, T right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main()
{
    maximum(2, 3);
    maximum("2", "3");
}
```

```cpp
template<>
int maximum(int left, int right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

template<>
std::string maximum(std::string left, std::string right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main() {
    maximum(2, 3);
    maximum("2", "3");
}
```

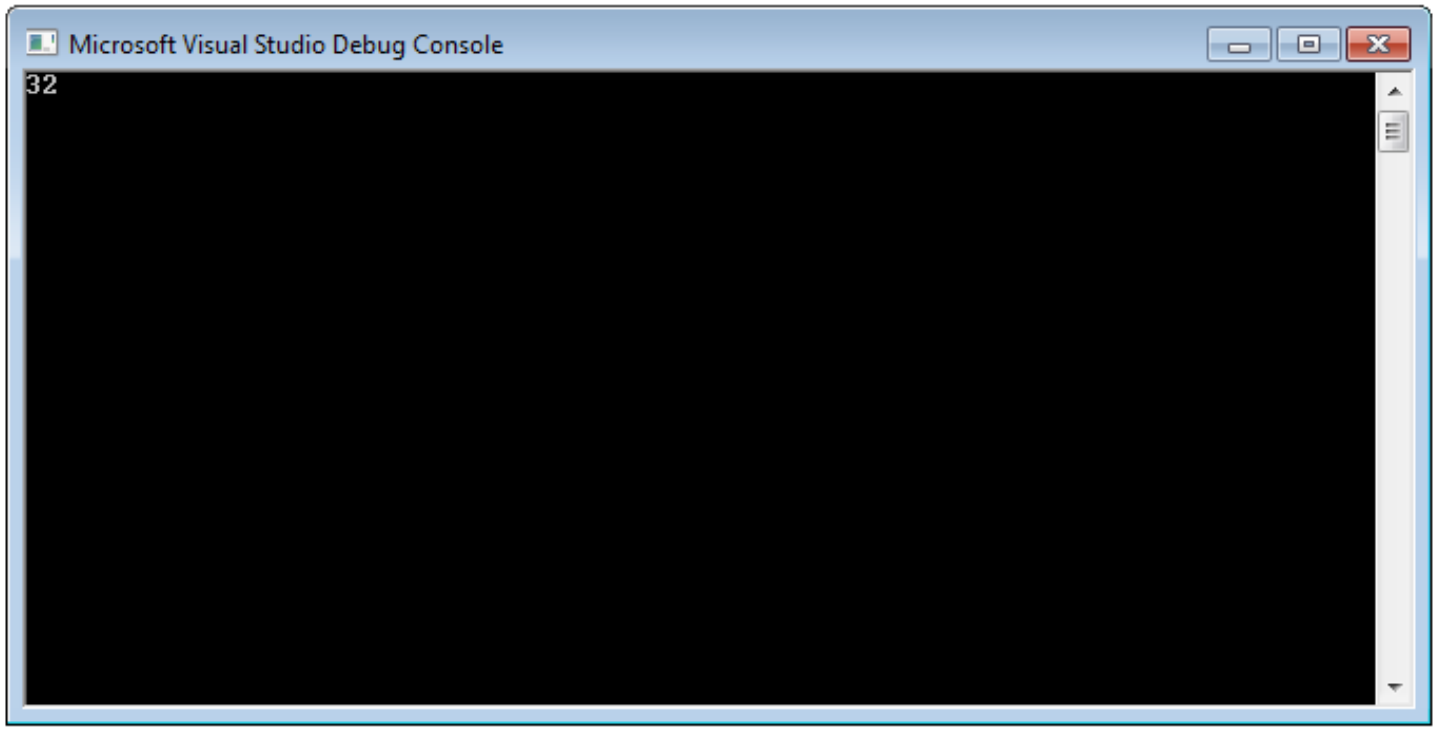
C++ templates

C++ „copies” each template and replaces type placeholder with actual type.

Then it tries to compile it.

If it works – great!

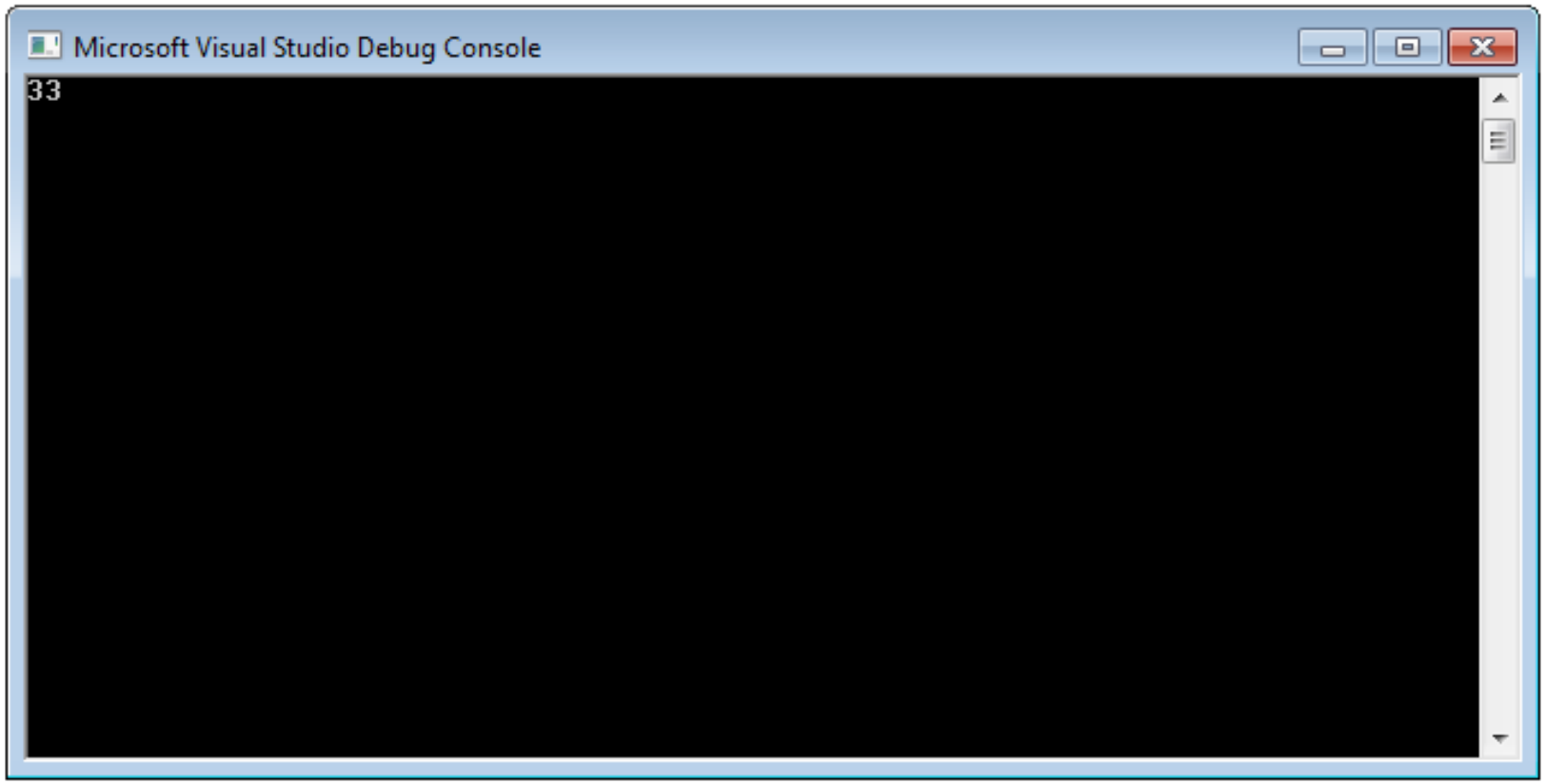
```cpp
template<typename T>
T maximum(T left, T right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main()
{
    std::cout<<maximum(2, 3);
    std::cout<<maximum("2", "3");
}
```



```
std::cout<<maximum("2", "3");
```

```
const char *maximum<const char *>(const char *left, const char *right)
```

```cpp
template<typename T>
T maximum(T left, T right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

int main()
{
    std::cout<<maximum<int>(2, 3);
    std::cout<<maximum<std::string>("2", "3");
}
```



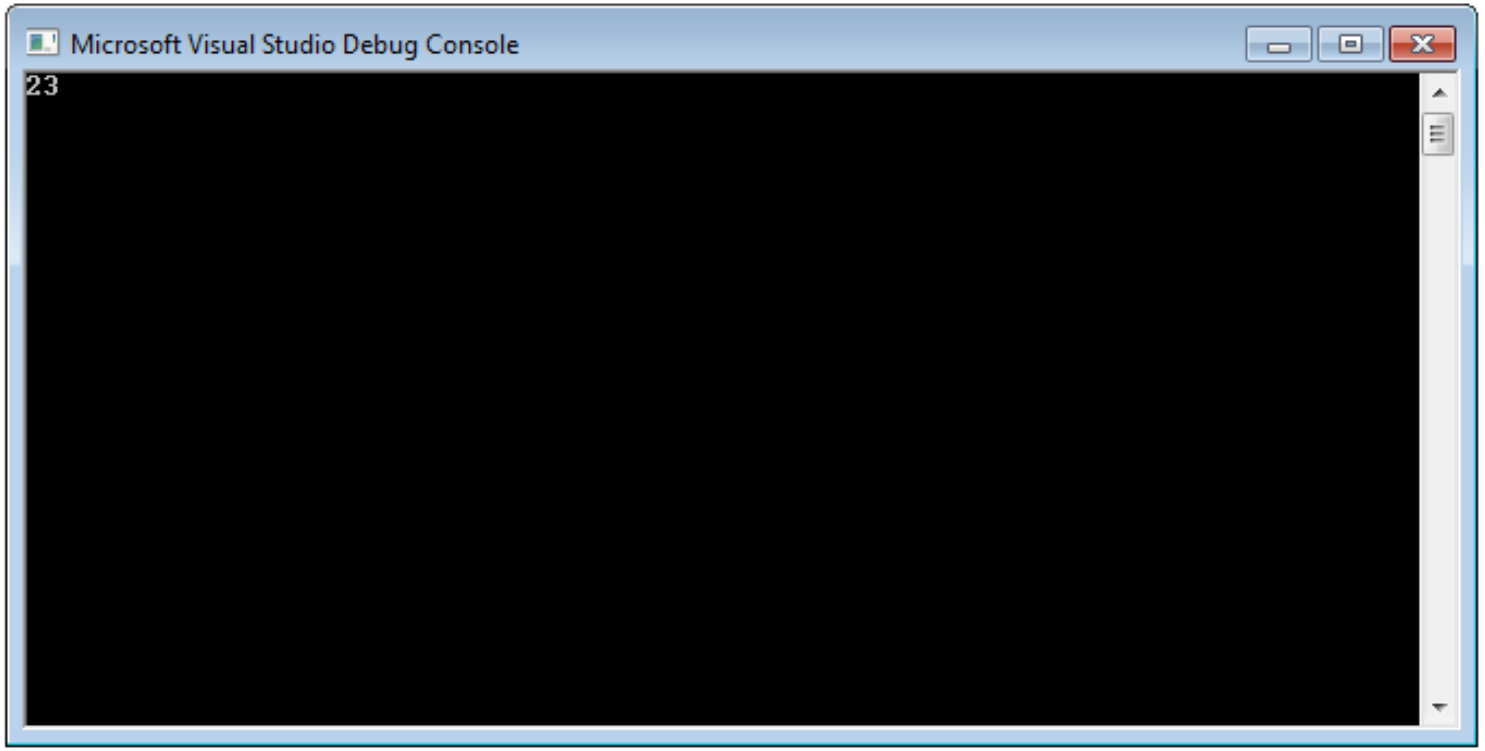
How to see the code?

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

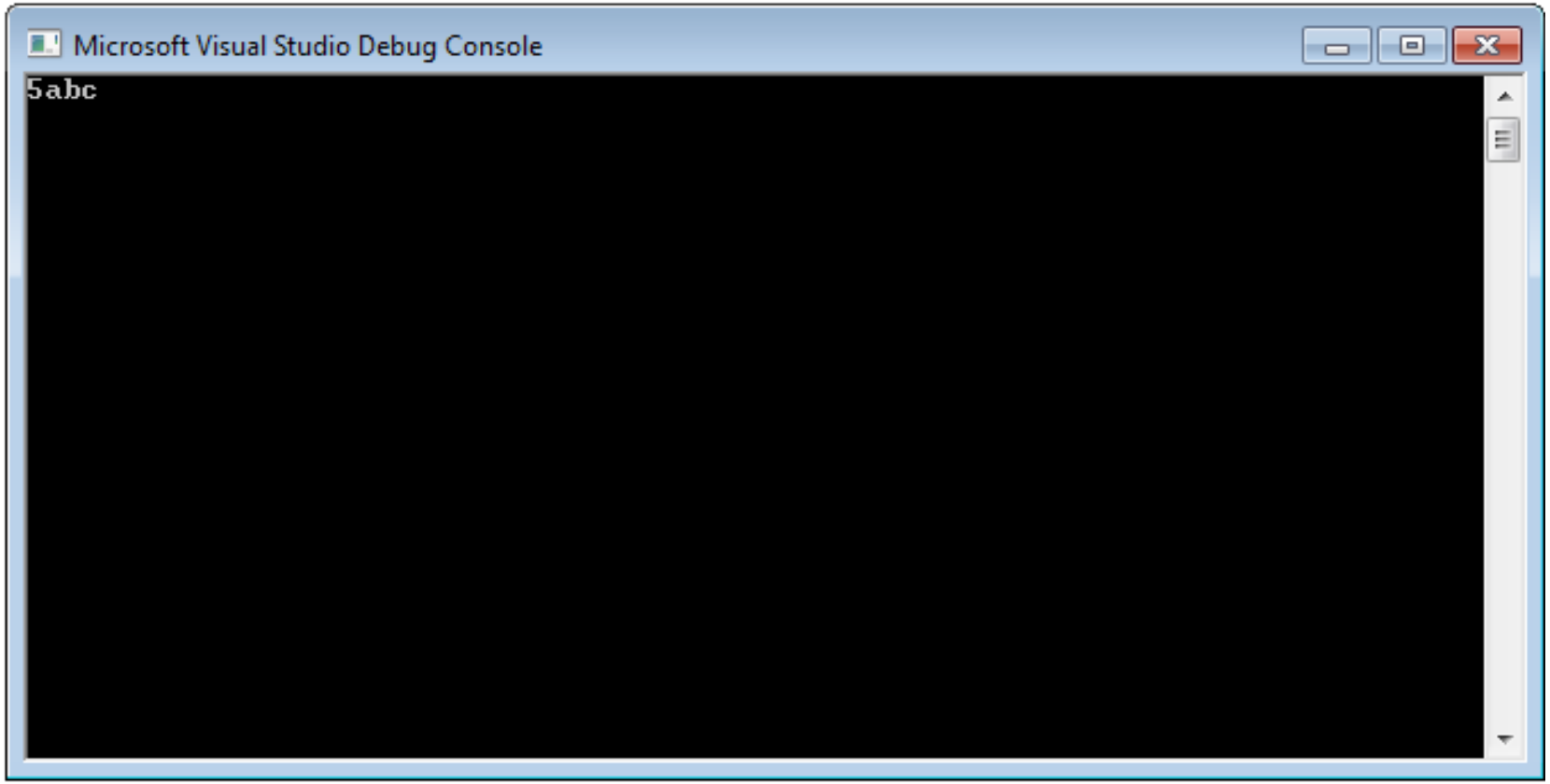
```
clang++ -Xclang-ast-print -fsyntax-only  test.cpp
```

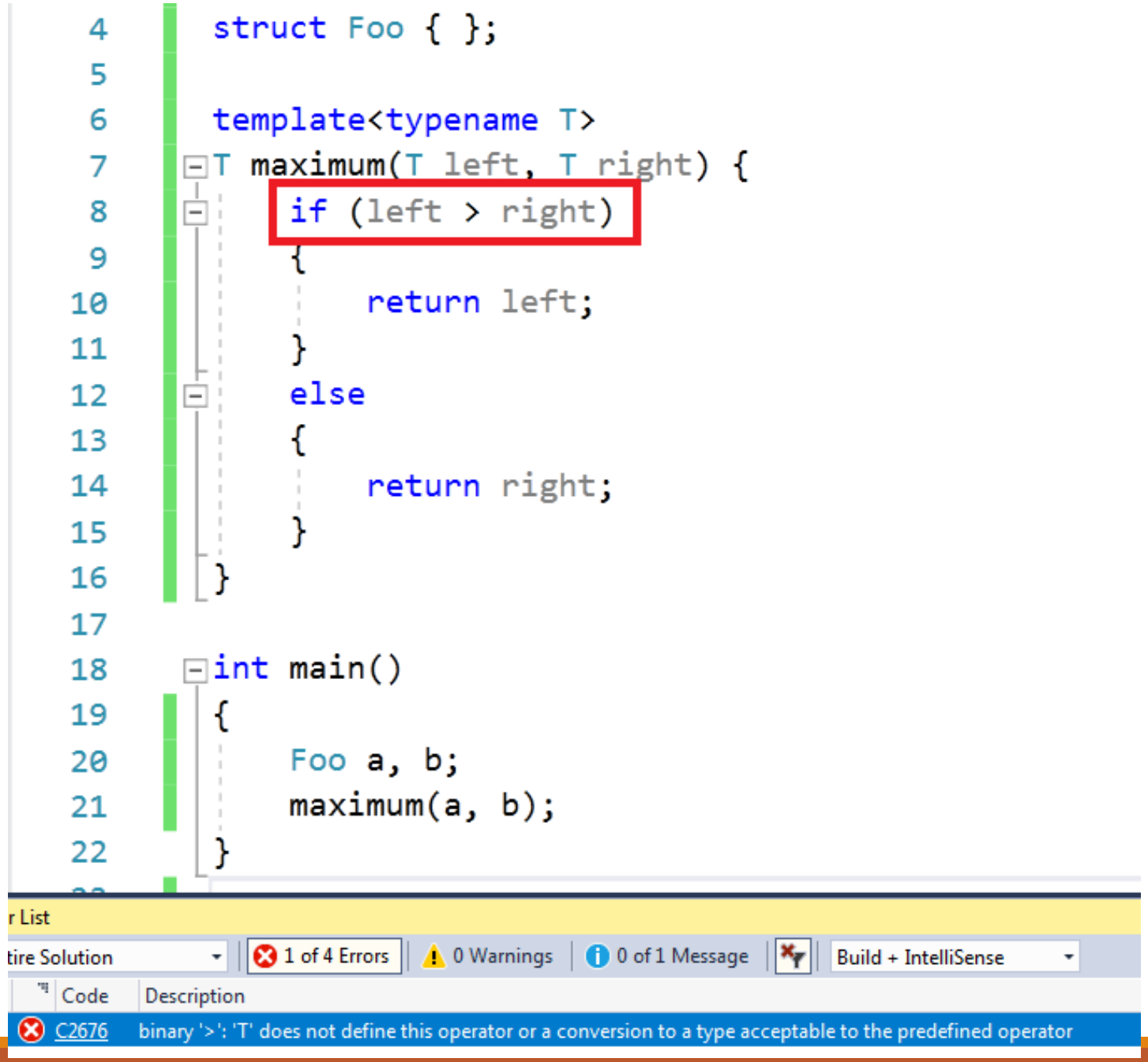
```
template<> int add<int>(int a, int b) {
    return a + b;
}

int main() {
    add<int>(10, 2);
}
```

```cpp
template<typename T>
T maximum(T left, T right) {
    if (left > right)
    {
        return left;
    }
    else
    {
        return right;
    }
}

template<>
int maximum(int left, int right) {
    return left;
}

int main()
{
    std::cout<<maximum<int>(2, 3);
    std::cout<<maximum<std::string>("2", "3");
}
```



```cpp
template<typename T>
class Wrapper {
    T field;
public:
    void set(T value) {
        field = value;
    }

    T get() {
        return field;
    }
};

int main()
{
    Wrapper<int> integer;
    integer.set(5);
    std::cout << integer.get();

    Wrapper<std::string> literal;
    literal.set("abc");
    std::cout << literal.get();
}
```




4 struct Foo { };
5
6 template<typename T>
7 T maximum(T left, T right) {
8 if (left > right)
9 {
10 return left;
11 }
12 else
13 {
14 return right;
15 }
16 }
17
18 int main()
19 {
20 Foo a, b;
21 maximum(a, b);
22 }
r List
tire Solution
1 of 4 Errors
0 Warnings
0 of 1 Message
Build + IntelliSense
Code
Description
C2676
binary '>': 'T' does not define this operator or a conversion to a type acceptable to the predefined operator

```cpp
#include <vector>
#include <algorithm>
int main()
{
    int a;
    std::vector<std::vector<int>> v;
    std::vector<std::vector<int>>::const_iterator it = std::find(v.begin(), v.end(), a);
}
```

C2678 binary '==': no operator found which takes a left-hand operand of type 'std::vector<int,std::allocator<_Ty>>' (or there is no acceptable conversion)

```cpp
3515    template<class _InIt,  <T>
3516        class _Ty> inline
3517    _InIt _Find_unchecked1(_InIt _First, const _InIt _Last, const _Ty& _Val, false_type)
3518    {   // find first matching _Val
3519    for (; _First != _Last; ++_First)
3520        if (*_First == _Val)
3521            break;
3522    return (_First);
3523    }
```

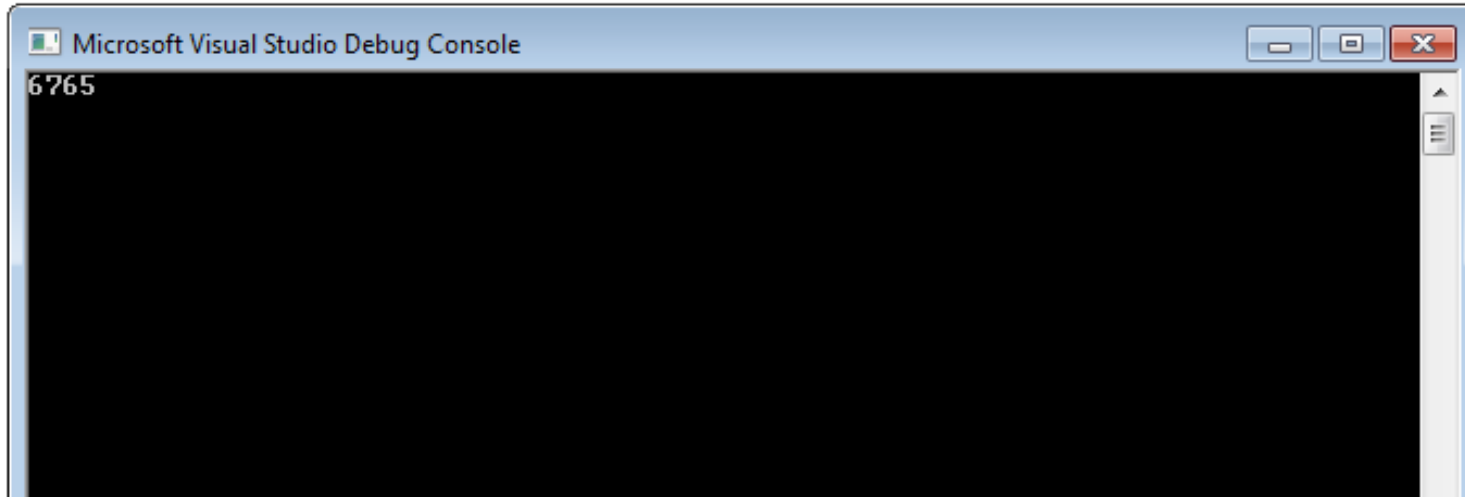
```
In file included from /usr/include/c++/4.6/algorithm:63:0,
                 from error_code.cpp:2:
/usr/include/c++/4.6/bits/stl_algo.h: In function '_RandomAccessIterator std::__find(_RandomAccessIterator, _RandomAccessIterator, const _Tp&, std::random_access_iterator_tag) [with _Rando
/usr/include/c++/4.6/bits/stl_algo.h:4403:45:   instantiated from '_IIter std::find(_IIter, _IIter, const _Tp&) [with _IIter = __gnu_cxx::__normal_iterator*, std::vector > >, _Tp = int]'
error_code.cpp:8:89:   instantiated from here
/usr/include/c++/4.6/bits/stl_algo.h:162:4: error: no match for 'operator==' in '__first.__gnu_cxx::__normal_iterator::operator* [with _Iterator = std::vector*, _Container = std::vector >,
/usr/include/c++/4.6/bits/stl_algo.h:162:4: note: candidates are:
/usr/include/c++/4.6/bits/stl_pair.h:201:5: note: template bool std::operator==(const std::pair&, const std::pair&)
/usr/include/c++/4.6/bits/stl_iterator.h:285:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:335:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/allocator.h:122:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/allocator.h:127:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/stl_vector.h:1273:5: note: template bool std::operator==(const std::vector&, const std::vector&)
/usr/include/c++/4.6/ext/new_allocator.h:123:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::new_allocator&, const __gnu_cxx::new_allocator&)
/usr/include/c++/4.6/bits/stl_iterator.h:805:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:799:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
/usr/include/c++/4.6/bits/stl_algo.h:4403:45:   instantiated from '_IIter std::find(_IIter, _IIter, const _Tp&) [with _IIter = __gnu_cxx::__normal_iterator*, std::vector > >, _Tp = int]'
error_code.cpp:8:89:   instantiated from here
/usr/include/c++/4.6/bits/stl_algo.h:166:4: error: no match for 'operator==' in '__first.__gnu_cxx::__normal_iterator::operator* [with _Iterator = std::vector*, _Container = std::vector >,
/usr/include/c++/4.6/bits/stl_algo.h:166:4: note: candidates are:
/usr/include/c++/4.6/bits/stl_pair.h:201:5: note: template bool std::operator==(const std::pair&, const std::pair&)
/usr/include/c++/4.6/bits/stl_iterator.h:285:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:335:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/allocator.h:122:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/allocator.h:127:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/stl_vector.h:1273:5: note: template bool std::operator==(const std::vector&, const std::vector&)
/usr/include/c++/4.6/ext/new_allocator.h:123:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::new_allocator&, const __gnu_cxx::new_allocator&)
/usr/include/c++/4.6/bits/stl_iterator.h:805:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:799:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
/usr/include/c++/4.6/bits/stl_algo.h:170:4: error: no match for 'operator==' in '__first.__gnu_cxx::__normal_iterator::operator* [with _Iterator = std::vector*, _Container = std::vector >,
/usr/include/c++/4.6/bits/stl_algo.h:170:4: note: candidates are:
/usr/include/c++/4.6/bits/stl_pair.h:201:5: note: template bool std::operator==(const std::pair&, const std::pair&)
/usr/include/c++/4.6/bits/stl_iterator.h:285:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:335:5: note: template bool std::operator==(const std::reverse_iterator&, const std::reverse_iterator&)
/usr/include/c++/4.6/bits/allocator.h:122:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/allocator.h:127:5: note: template bool std::operator==(const std::allocator&, const std::allocator&)
/usr/include/c++/4.6/bits/stl_vector.h:1273:5: note: template bool std::operator==(const std::vector&, const std::vector&)
/usr/include/c++/4.6/ext/new_allocator.h:123:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::new_allocator&, const __gnu_cxx::new_allocator&)
/usr/include/c++/4.6/bits/stl_iterator.h:805:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
/usr/include/c++/4.6/bits/stl_iterator.h:799:5: note: template bool __gnu_cxx::operator==(const __gnu_cxx::__normal_iterator&, const __gnu_cxx::__normal_iterator&)
```

What is T?

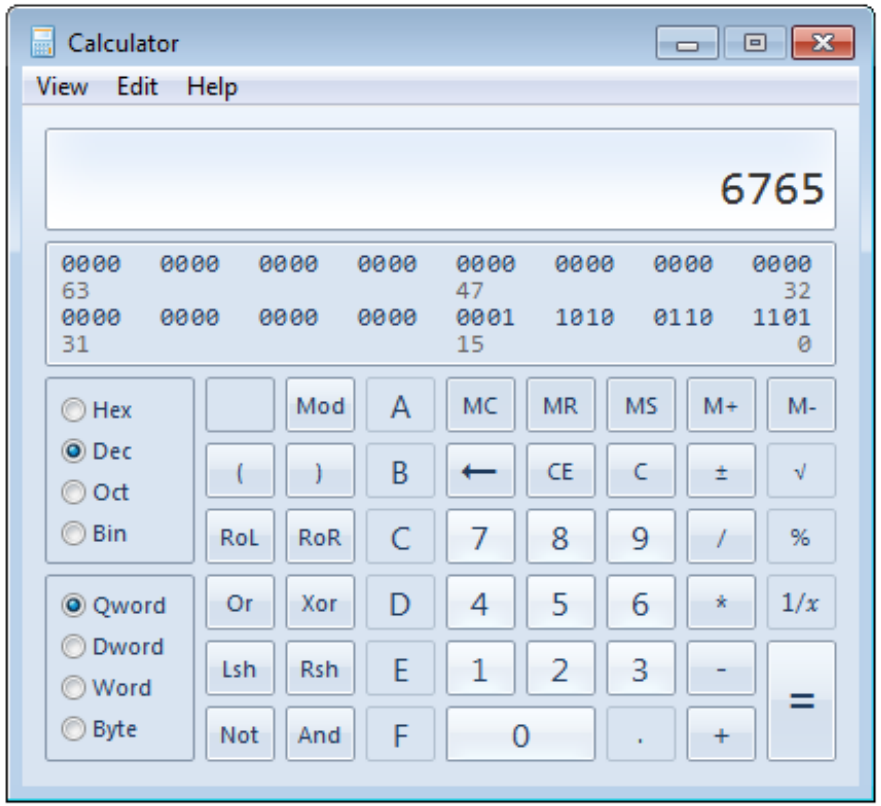
```
template<typename T>

template<typename T = void>

template<typename... Ts>

template<int I>

template<typename K, typename V, template<typename>
typename C = my_array>
```

```cpp
template <int a>
struct isodd {
    static constexpr bool value = a % 2;
};

int main() {
    std::cout<<isodd<5>::value; // 1
}
```

```
template <int n> struct Fib
{
    enum { val = Fib<n - 1>::val + Fib<n - 2>::val };
};

template<> struct Fib<0> { enum { val = 0 }; };
template<> struct Fib<1> { enum { val = 1 }; };

int main()
{
    std::cout << Fib<20>::val << std::endl;
}
```



```
00EE1F30  push        ebp
00EE1F31  mov         ebp,esp
00EE1F33  sub         esp,0C0h
00EE1F39  push        ebx
00EE1F3A  push        esi
00EE1F3B  push        edi
00EE1F3C  lea         edi,[ebp-0C0h]
00EE1F42  mov         ecx,30h
00EE1F47  mov         eax,0CCCCCCCCh
00EE1F4C  rep stos    dword ptr es:[edi]
00EE1F4E  mov         ecx,offset _7791189E_maximum@cpp (0EF5008h)
00EE1F53  call        @__CheckForDebuggerJustMyCode@4 (0EE13BBh)
    std::cout << Fib<20>::val << std::endl;
00EE1F58  mov         esi,esp
00EE1F5A  push        offset std::endl<char,std::char_traits<char> > (0EE16CCh)
00EE1F5F  mov         edi,esp
00EE1F61  push        1A6Dh
00EE1F66  mov         ecx,dword ptr [_imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A (0EF30D8h)]
00EE1F6C  call        dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (0EF309Ch)]
00EE1F72  cmp         edi,esp
00EE1F74  call        __RTC_CheckEsp (0EE13D9h)
00EE1F79  mov         ecx,eax
00EE1F7B  call        dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (0EF30E4h)]
00EE1F81  cmp         esi,esp
00EE1F83  call        __RTC_CheckEsp (0EE13D9h)
}
00EE1F88  xor         eax,eax
00EE1F8A  pop         edi
00EE1F8B  pop         esi
00EE1F8C  pop         ebx
00EE1F8D  add         esp,0C0h
00EE1F93  cmp         ebp,esp
```

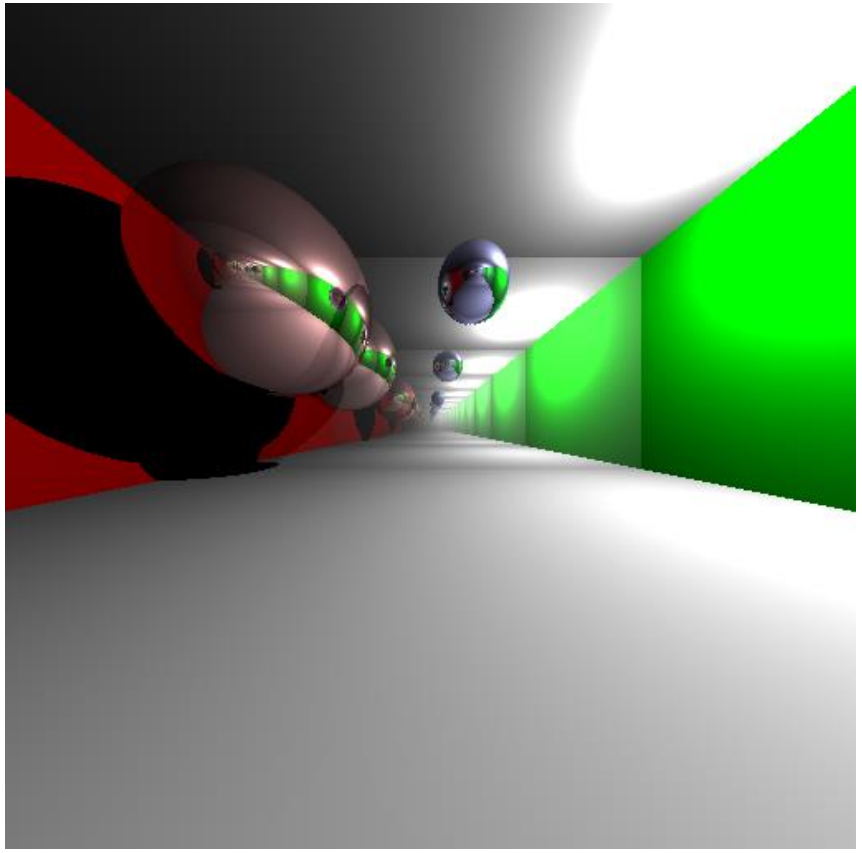


```cpp
template <class T>
inline void hash_combine(std::size_t& seed, T const& v)
{
    seed ^= hash<T>()(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
}

template <class Tuple, size_t Index = std::tuple_size<Tuple>::value - 1>
struct HashValueImpl
{
    static void apply(size_t& seed, Tuple const& tuple)
    {
        HashValueImpl<Tuple, Index - 1>::apply(seed, tuple);
        hash_combine(seed, get<Index>(tuple));
    }
};

template <class Tuple>
struct HashValueImpl<Tuple, 0>
{
    static void apply(size_t& seed, Tuple const& tuple)
    {
        hash_combine(seed, get<0>(tuple));
    }
};
```

```cpp
template <typename ... TT>
struct hash<std::tuple<TT...>>
{
    size_t
        operator()(std::tuple<TT...> const& tt) const
    {
        size_t seed = 0;
        HashValueImpl<std::tuple<TT...> >::apply(seed, tt);
        return seed;
    }

};
```

```
template<class F, class... Args>
auto cached(F&& f, const Args&... x) {
    static unordered_map<tuple<Args...>, typename result_of<F(Args...)>::type> cache;

    auto key = tuple<Args...>(x...);
    auto y = cache.find(key);
    if (y != cache.end()) return y->second;
    return cache[key] = f(x...);
}

int fib(int n) {
    if (n < 2) return n;
    return cached(fib, n - 2) + cached(fib, n - 1);
}
```

Raytracing



https://github.com/phresnel/metatrace

C++ templates

PROS

Relatively easy to understand.

Doesn’t require sophisticated runtime support.

Works without any notion of „supertype” or „base type” for all types.

Doesn’t require extensive constraints for passed types.

Can be easily specialized for specific types.

Works nice with const expressions.

Turing complete – perfect for compile time calculations.

CONS

„Something” needs to instantiate the type for given parameter.

A lot of machine code duplication.

We cannot instantiate the code in runtime.

Compiler messages are often less than ideal.

We cannot constrain input type declaratively – we can only use metaprogramming tricks.

JVM Erasure

Java type system

OBJECTS

java.lang.Object is the root of the class hierarchy. **Every class** has *Object* as a superclass. All objects, including arrays, implement the methods of this class.

Provides method for equality checking (equals), calculating hashcode (hashCode), checking runtime type (getClass), serializing to string literal (toString).

String is an Object.

PRIMITIVE VALUES

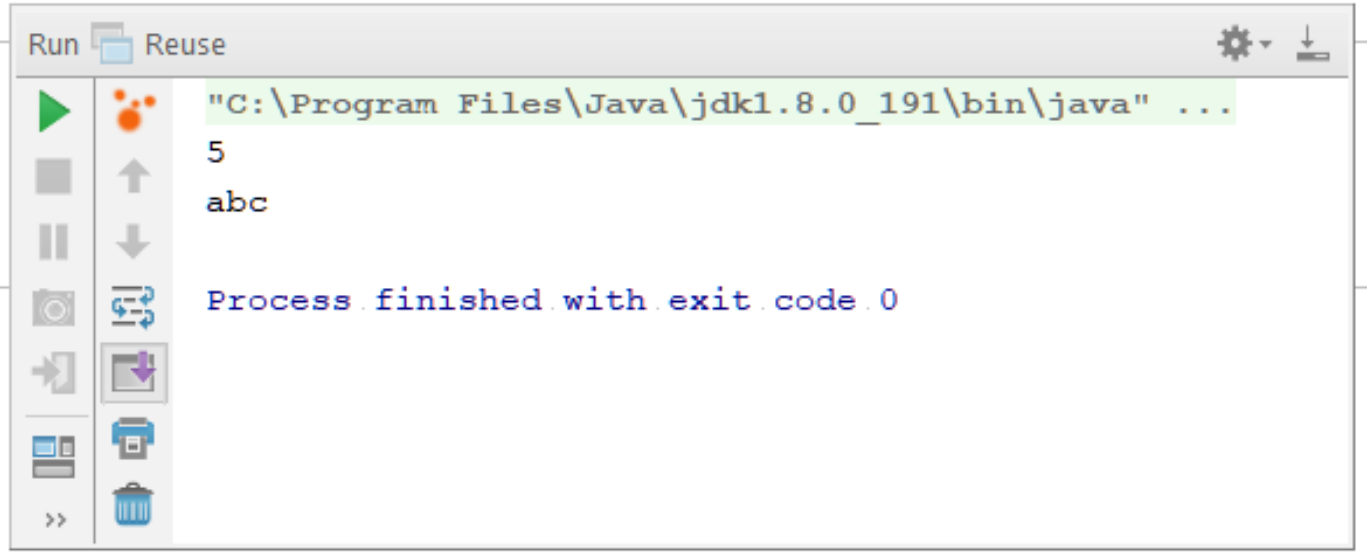
Primitive types correspond to primitive values.

These include int, short, long, byte, char, float, double, boolean.

We can implicitly convert between them (most of the times).

There is a reference type for each primitive type respectively (Integer, Short, Long, Byte, Character, Float, Double, Boolean).

```java
class Wrapper{
    Object field;
    public void set(Object value){
        field = value;
    }

    public Object get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper integer = new Wrapper();
        integer.set(5);
        System.out.println(integer.get());

        Wrapper literal = new Wrapper();
        literal.set("abc");
        System.out.println(literal.get());
    }
}
```



```
class Wrapper{
    Object field;
    public void set(Object value){
        field = value;
    }

    public Object get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper integer = new Wrapper();
        integer.set(5);
        int i = integer.get();

        Wrapper literal = new Wrapper();
        literal.set("abc");
        String l = literal.get();
    }
}
```

Incompatible types.

Required: **java.lang.String**

Found: **java.lang.Object**

```java
class Wrapper{
    Object field;
    public void set(Object value){
        field = value;
    }

    public Object get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper integer = new Wrapper();
        integer.set(5);
        int i = integer.get();

        Wrapper literal = new Wrapper();
        literal.set("abc");
        String l = literal.get();
    }
}
```

Incompatible types.
Required: **java.lang.String**
Found: **java.lang.Object**

```java
class Wrapper{
    Object field;
    public void set(Object value){
        field = value;
    }

    public Object get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper integer = new Wrapper();
        integer.set(5);
        int i = (int)integer.get();

        Wrapper literal = new Wrapper();
        literal.set("abc");
        String l = (String)literal.get();
    }
}
```

```java
class Wrapper{
    Object field;
    public void set(Object value){
        field = value;
    }

    public Object get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper integer = new Wrapper();
        integer.set(5);
        int i = (int)integer.get();

        Wrapper literal = new Wrapper();
        literal.set("abc");
        String l = (String)literal.get();
    }
}
```

```java
class Wrapper<T> {
    T field;
    public void set(T value){
        field = value;
    }

    public T get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper<Integer> integer = new Wrapper<>();
        integer.set(5);
        int i = integer.get();

        Wrapper<String> literal = new Wrapper<>();
        literal.set("abc");
        String l = literal.get();
    }
}
```

```
class Wrapper {

  // compiled from: Reuse.java

  // access flags 0x0
  // signature TT;
  // declaration: T
  Ljava/lang/Object; field
```

```
public static main([Ljava/lang/String;)V
  L0
    LINENUMBER 16 L0
    NEW Wrapper
    DUP
    INVOKESPECIAL Wrapper.<init> ()V
    ASTORE 1
  L1
    LINENUMBER 17 L1
    ALOAD 1
    ICONST_5
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/Integer;
    INVOKEVIRTUAL Wrapper.set (Ljava/lang/Object;)V
  L2
    LINENUMBER 18 L2
    ALOAD 1
    INVOKEVIRTUAL Wrapper.get ()Ljava/lang/Object;
    CHECKCAST java/lang/Integer
    INVOKEVIRTUAL java/lang/Integer.intValue ()I
    ISTORE 2
```

JVM erasure

Java replaces generic placeholder with java.lang.Object. In other words, it **gets rid of** the generic type.

There is one version of the code – no copying of generic for different types.

It then compiles the code with the java.lang.Object as a generic type.

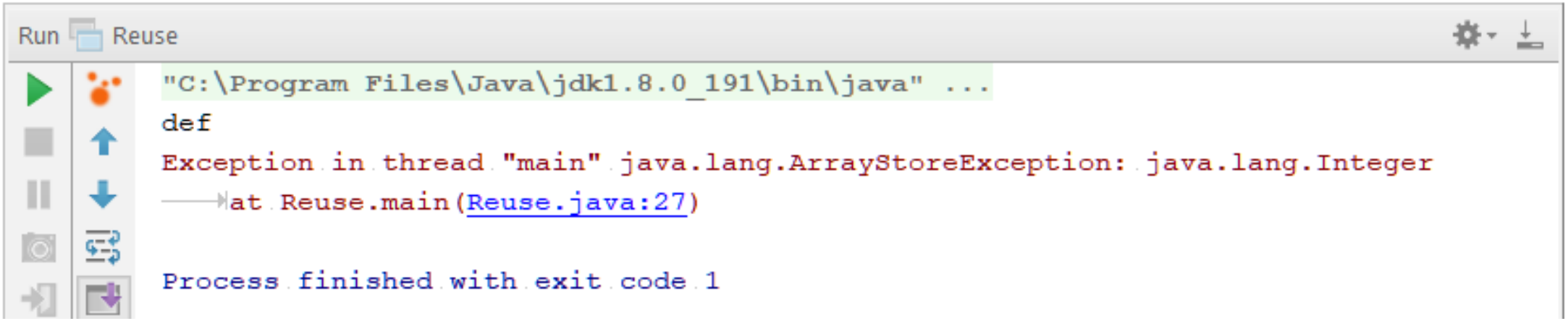
```java
class Wrapper<T> {
    T field;
    public void set(T value){
        field = value;
    }

    public T get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper<Integer> integer = new Wrapper();
        integer.set(5);
        int i = integer.get();

        Wrapper literal = new Wrapper<String>();
        literal.set("abc");
        String l = literal.get();
    }
}
```

Unchecked assignment: 'Wrapper' to 'Wrapper<java.lang.Integer>'

```java
class Wrapper<T extends Number> {
    T field;

    public void set(T value){
        field = value;
    }

    public T get(){
        return field;
    }
}

public class Reuse {
    public static void main(String[] args){
        Wrapper<Integer> integer = new Wrapper<Integer>();
        integer.set(5);
        int i = integer.get();

        Wrapper<String> literal = new Wrapper<String>();
        literal.set("abc");
        String l = literal.get();
    }
}
```

```
class Wrapper {

  // compiled from: Reuse.java

  // access flags 0x0
  // signature TT;
  // declaration: T
  Ljava/lang/Number; field
```

```java
public class Reuse {
    public static void main(String[] args){
        String[] strings = new String[1];
        strings[0] = "abc";

        Object[] objects = strings;
        objects[0] = "def";
        System.out.println(objects[0]);

        objects[0] = 123;
        System.out.println("We will never get here");
    }
}
```



```java
public class Reuse {
    public static void main(String[] args){
        List<String> strings = new ArrayList<String>( 1);
        strings.set(0, "abc");

        List<Object> objects = strings;
    }
}
```

Incompatible types.

Required: List <**java.lang.Object**>

Found: List <**java.lang.String**>

```
class Util {
    static void foo(List<Object> objects){
    }
}
```

```
public class Reuse {
    public static void main(String[] args){
        List<Object> objects = new ArrayList<Object>();
        Util.foo(objects);

        List<String> strings = new ArrayList<String>();
        Util.foo(strings);
    }
}
```

```
class Util {
    static void foo(List<?> objects){
    }
}
```

```
public class Reuse {
    public static void main(String[] args){
        List<Object> objects = new ArrayList<Object>();
        Util.foo(objects);

        List<String> strings = new ArrayList<String>();
        Util.foo(strings);
    }
}
```

```java
class Util {
    static void foo(List<? extends String> objects){
    }
}
```

```java
public class Reuse {
    public static void main(String[] args){
        List<Object> objects = new ArrayList<Object>();
        Util.foo(objects);

        List<String> strings = new ArrayList<String>();
        Util.foo(strings);
    }
}
```

```java
class Util {
    static void foo(List<? super String> objects){
    }
}
```

```java
public class Reuse {
    public static void main(String[] args){
        List<Object> objects = new ArrayList<Object>();
        Util.foo(objects);

        List<String> strings = new ArrayList<String>();
        Util.foo(strings);
    }
}
```

```
public class Reuse {
    public static void main(String[] args){
        List<String> strings = new ArrayList<String>( 1);
        strings.set(0, "abc");

        List<?> objects = strings;
        objects.set(0, "def");
    }
}
```

Wrong 2nd argument type. Found: 'java.lang.String', required: '?' more...

```
public class Reuse {
    public static void main(String[] args){
        List<String> strings = new ArrayList<String>( i: 1);
        strings.set(0, "abc");

        List<? super String> objects = strings;
        objects.set(0, "def");

        strings.set(0, 123);
    }
}
```

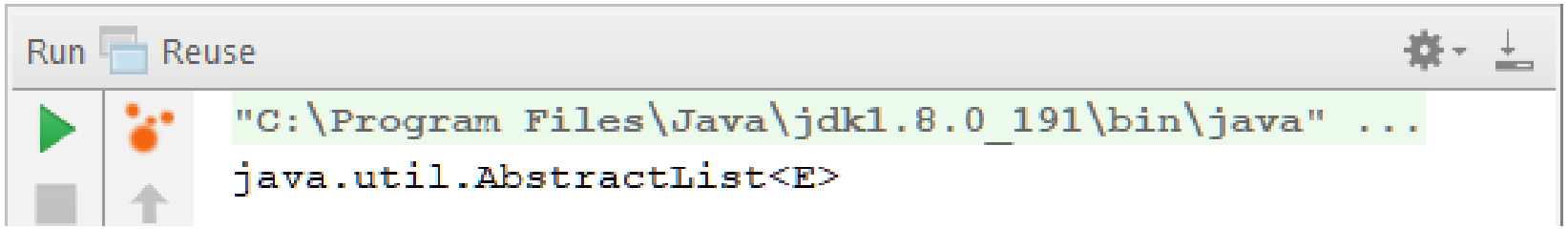
Wrong 2nd argument type. Found: 'int', required: 'java.lang.String' more...

```
class Util {
    static <T> void foo(){
        T t = new T();
    }
}
```

```
class Util {
    static <T> void foo(Class<T> klass) throws Exception {
        T t = klass.newInstance();
    }
}
```

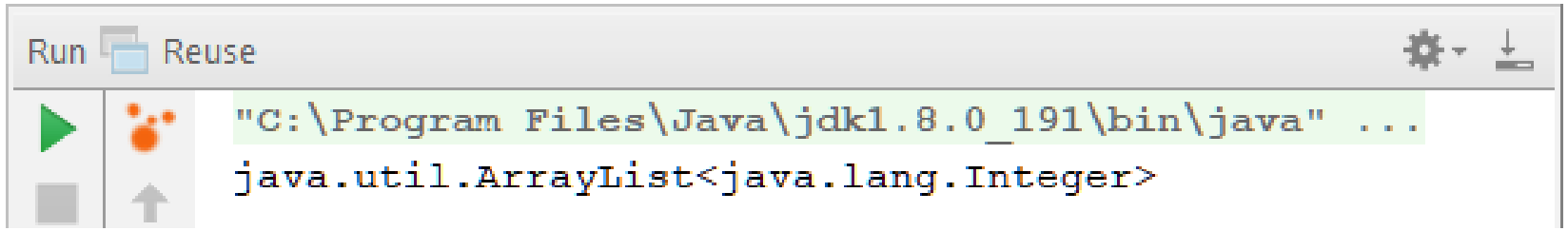
```
class Util {
    static <T> void foo(T value) {
        System.out.println(value.getClass().getGenericSuperclass());
    }
}
```

```
public class Reuse {
    public static void main(String[] args){
        List<Integer> integers = new ArrayList<Integer>();
        Util.foo(integers);
    }
}
```



```
class Util {
    static <T> void foo(T value) {
        System.out.println(value.getClass().getGenericSuperclass());
    }
}
```

```
public class Reuse {
    public static void main(String[] args){
        List<Integer> integers = new ArrayList<Integer>(){};
        Util.foo(integers);
    }
}
```



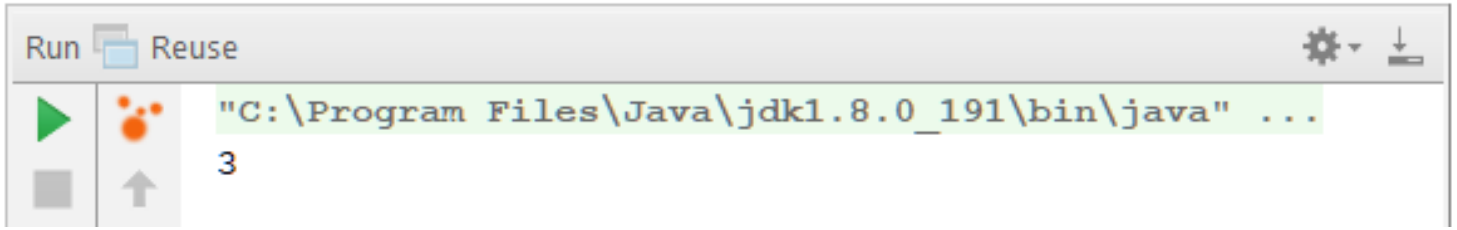
```
class Max{
    static <T> T maximum(T left, T right){
        if(left > right){
            return left;
        }else{
            return right;
        }
    }
}
```

Operator '>' cannot be applied to 'T', 'T'

```
public class Reuse {
    public static void main(String[] args){
        Max.maximum( left: 2, right: 3);
    }
}
```

```java
class Max {
    public static <T extends Comparable<? super T>> T maximum(T left, T right){
        if(left.compareTo(right) > 0){
            return left;
        }else{
            return right;
        }
    }
}
```

```java
public class Reuse {
    public static void main(String[] args){
        System.out.println(Max.maximum( left: 2,  right: 3));
    }
}
```



```
class Util {
    static void foo(List<Object> value) {
    }

    static void foo(Wrapper<Object> value){
    }
}
```

```
class Util {
    static <T> void foo(T<Object> value) {
    }
}
```

Type 'T' does not have type parameters

```scala
trait Expr[+A]

trait FixExpr {
  val in: Expr[FixExpr]
}

trait Fix[F[_]] {
  val in: F[Fix[F]]
}
```

JVM erasure

PROS

Maintains backwards comaptibility.

Stores one code for all types.

Works with higher kinded types (partially).

We can constrain generic parameter (bounds).

Relatively easy to implement.

Works with covariance and contravariance better.

CONS

We lose the type information.

We have a performance hit for primitive types.

We don’t have full compiler support and type safety.

It is harder to perform optimizations for particular types.

We need to encode „trivial” things with type system.

We cannot provide specialization for given type.

Reification in .NET

.NET Type system

Mostly based on Java type system.

System.Object is the root of the hierarchy.

Primitive types are called *value types*. Users can create their own value types (structures).

Few more things: delegates, events, expression trees, dynamic.

```
class Wrapper<T>
{
    T field;
    void set(T value)
    {
        this.field = value;
    }

    T get()
    {
        return field;
    }
}
```

```
.class private auto ansi beforefieldinit Reusability.Wrapper`1<T>
    extends [mscorlib]System.Object
{
    // Fields
    // Token: 0x04000001 RID: 1
    .field private !T 'field'
```

```
var myObject = new Wrapper<object>();

EEClass:          00007fff8e862510
Module:           00007fff8e752fc8
Name:             Wrapper`1[[System.Object, mscorlib]]
mdToken:          0000000002000003
MethodDesc Table
Entry               MethodDesc        JIT     Name
00007fff8e870210 00007fff8e754280     JIT Wrapper`1[[System.__Canon, mscorlib]]..ctor()
00007fff8e75c098 00007fff8e754278    NONE Wrapper`1[[System.__Canon, mscorlib]].get()

var myString = new Wrapper<string>();

EEClass:          00007fff8e862510
Module:           00007fff8e752fc8
Name:             Wrapper`1[[System.String, mscorlib]]
mdToken:          0000000002000003
MethodDesc Table
Entry       MethodDesc    JIT Name
00007fff8e870210 00007fff8e754280     JIT Wrapper`1[[System.__Canon, mscorlib]]..ctor()
00007fff8e75c098 00007fff8e754278    NONE Wrapper`1[[System.__Canon, mscorlib]].get()
```

```
Class Name:      Wrapper`1[[System.__Canon, mscorlib]]
mdToken:         0000000002000003
MT                Field          Offset  Type            VT Attr       Value Name
00007fffecf05c80  4000002        8       System.__Canon  0 instance    field
```

```
var myInt = new Wrapper<int>();
```

```
EEClass:          00007fff8e862628
Module:           00007fff8e752fc8
Name:             Wrapper`1[[System.Int32, mscorlib]]
mdToken:          0000000002000003
MethodDesc Table
Entry       MethodDesc    JIT Name
00007fff8e870260 00007fff8e7544b8    JIT Wrapper`1[[System.Int32, mscorlib]]..ctor()
00007fff8e75c0c0 00007fff8e7544b0   NONE Wrapper`1[[System.Int32, mscorlib]].get()
```

```
var myObject = new Wrapper<object>();
```

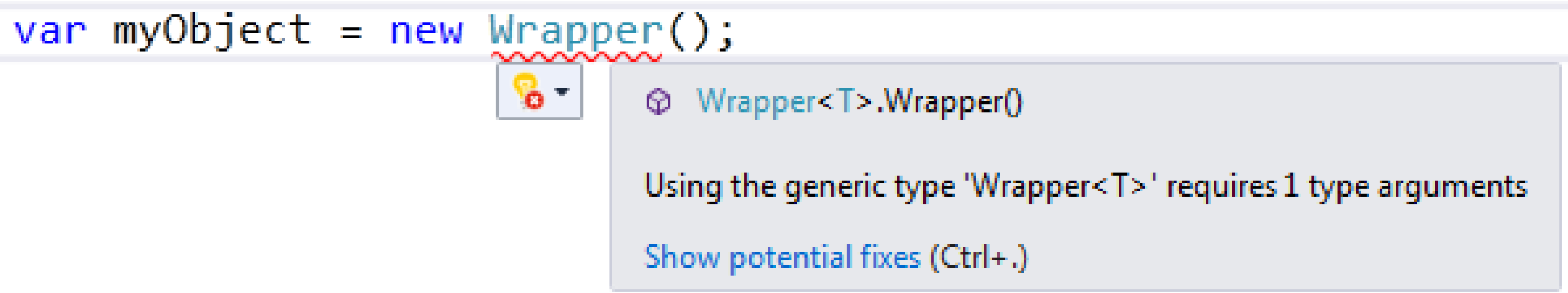
```
EEClass:          00007fff8e862510
Module:           00007fff8e752fc8
Name:             Wrapper`1[[System.Object, mscorlib]]
mdToken:          0000000002000003
MethodDesc Table
Entry               MethodDesc        JIT     Name
00007fff8e870210 00007fff8e754280     JIT Wrapper`1[[System.__Canon, mscorlib]]..ctor()
00007fff8e75c098 00007fff8e754278    NONE Wrapper`1[[System.__Canon, mscorlib]].get()
```

Reification in .NET

.NET replaces type placeholder with System.__Canon for reference types and actual type for value types.

It has one code for all reference types and different code for each value type.

It doesn’t lose the generic type. Generics are **reified** and **remember their type**.


var myObject = new Wrapper();
Wrapper<T>.Wrapper()
Using the generic type 'Wrapper<T>' requires 1 type arguments
Show potential fixes (Ctrl+.)

```
class Util
{
    static void Foo<T>()
    {
        var t = new T();
    }
}
```

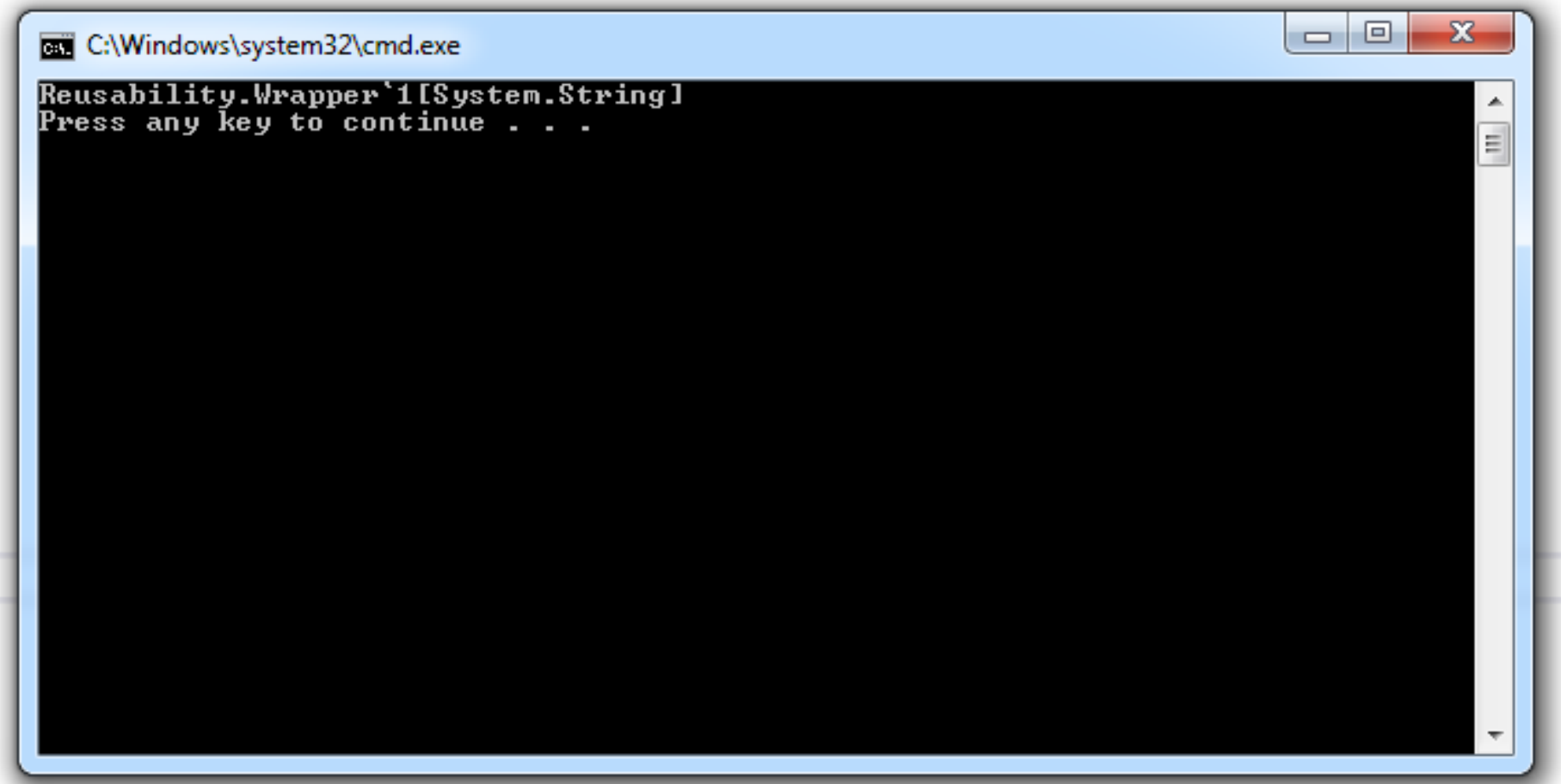
```
class Util
{
    static void Foo<T>() where T: new()
    {
        var t = new T();
    }
}
```

```
class Util
{
    static void Foo<T>() where T: IDisposable
    {

    }
}
```

```
class Util
{
    static void Foo<T>() where T: struct
    {

    }
}
```

```
class Util
{
    static void Foo<T>() where T: class
    {

    }
}
```

```
class Wrapper<T>
{
    T field;
    void set(T value)
    {
        this.field = value;
    }

    T get()
    {
        return field;
    }
}

class Program
{
    static void Main(string[] args)
    {
        var myString = new Wrapper<string>();
        Console.WriteLine(myString.GetType());
    }
}
```

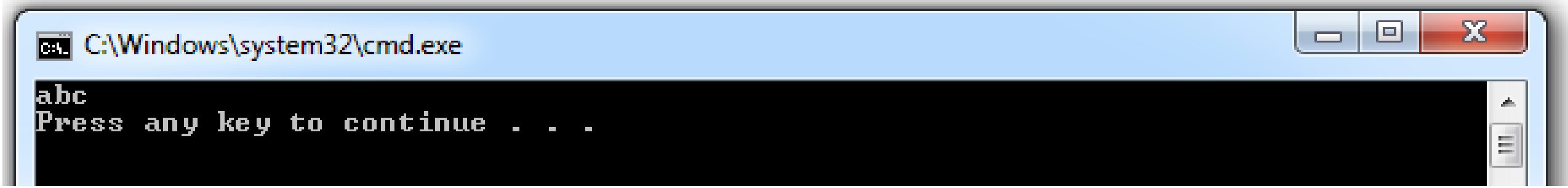


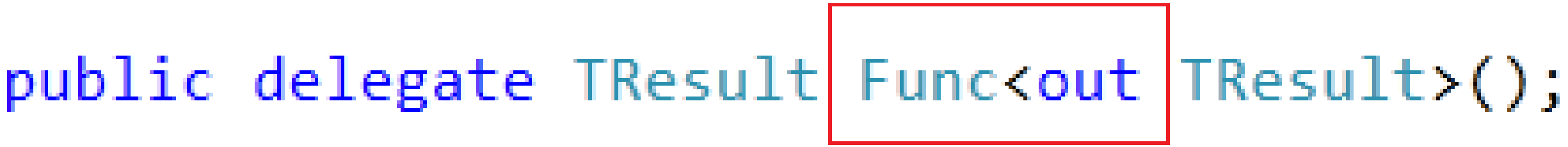
```
List<string> myStrings = new List<string>(1);
myStrings[0] = "abc";

List<object> myObjects = myStrings;
```

(local variable) List<string> myStrings

Cannot implicitly convert type 'System.Collections.Generic.List<string>' to 'System.Collections.Generic.List<object>'

```
Func<string> myStringGenerator = () => "abc";
Func<object> myObjectGenerator = myStringGenerator;
Console.WriteLine(myObjectGenerator());
```




public delegate TResult Func<out TResult>();

```
IEnumerable<string> strings = new List<string>();
IEnumerable<object> stringObjects = strings;

IEnumerable<int> integers = new List<int>();
IEnumerable<object> integerObjects = integers;
```

(local variable) IEnumerable<int> integers

Cannot implicitly convert type 'System.Collections.Generic.IEnumerable<int>' to 'System.Collections.Generic.IEnumerable<object>'. An explicit conversion exists (are you missing a cast?)

```
object foo = Util.GetFoo();
typeof(Wrapper<>).MakeGenericType(foo.GetType());
```

```
interface IMarker
{
}

struct LeftStruct : IMarker { }
struct RightStruct : IMarker { }

static int ChooseStruct<T>(int left, int right)
{
    if (typeof(T) == typeof(LeftStruct))
    {
        return left;
    }
    else
    {
        return right;
    }
}
```

```
L0000: mov eax, ecx
L0002: ret
```

```
L0000: mov eax, edx
L0002: ret
```

```
public class Class
{
    public static void Main(string[] args)
    {
        Bar(new A());
        Bar(new B());
        Bar(new C());
    }

    public static void Bar<T>(T val) where T : IFoo
    {
        val.Foo();
    }
}
```

```
public interface IFoo
{
    void Foo();
}

public class A : IFoo
{
    public void Foo()
    {
        Console.WriteLine("A");
    }
}

public class B : IFoo
{
    public void Foo()
    {
        Console.WriteLine("B");
    }
}

public struct C : IFoo
{
    public void Foo()
    {
        Console.WriteLine("C");
    }
}
```

```
Class.Main(System.String[])
    L0000: sub rsp, 0x28
    L0004: mov rcx, 0x7ffe7390d580
    L000e: call 0x7ffec7e9c520
    L0013: mov rcx, rax
    L0016: mov r11, 0x7ffe73907020
    L0020: cmp [rcx], ecx
    L0022: call qword [rip+0x6b58]
    L0028: mov rcx, 0x7ffe7390d6f8
    L0032: call 0x7ffec7e9c520
    L0037: mov rcx, rax
    L003a: mov r11, 0x7ffe73907028
    L0044: cmp [rcx], ecx
    L0046: call qword [rip+0x6b3c]
    L004c: mov rcx, 0x211bffd7ef0
    L0056: mov rcx, [rcx]
    L0059: call System.Console.WriteLine(System.String)
    L005e: nop
    L005f: add rsp, 0x28
    L0063: ret
```

```
static void Foo<T>(T<object> value)
{

}
```

T in Util.Foo<T>

The type parameter 'T' cannot be used with type arguments

Reification in .NET

PROS

Maintains generic type – no need for passing type information in runtime.

Guarantees type safety.

Easier optimizations for value types.

No overhead for value type (no boxing).

CONS

No way to do higher kinded types without runtime support.

Harder to implement.

Duplicates code for value types.

Need to encode „trivial” things with type system.

No easy way to provide specialization for given type.

Summary

3 different approaches for reusability:

- C++ copying code, changing the type and compiling it.
- JVM removing generic type and pretending it is still there.
- .NET storing generic type and generating different code for value types.

Each approach has pros and cons

- C++ approach is Turing complete but requires a lot of code duplication.
- JVM approach is easy to implement but doesn't guarantee type safety.
- .NET approach is much stronger and guarantees type safety but doesn't support higher kinded types = less powerful type system.

Q&A

References

Jeffrey Richter — „CLR via C#”

Mario Hewardt — „Advanced .NET Debugging”

Adam Furmanek – „.NET Internals Cookbook”

Andrei Alexandrescu — „Modern C++ Design: Generic Programming and Design Patterns Applied: Applied Generic and Design Patterns (C++ in Depth)”

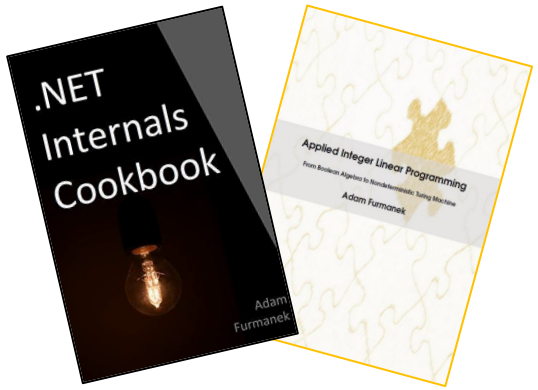
References

https://blog.adamfurmanek.pl/2019/08/31/net-inside-out-part-10-using-type-markers-for-low-level-optimizations/ — type markers in .NET

http://dreixel.net/research/pdf/gdmh.pdf — generics in Haskell

https://alexandrnikitin.github.io/blog/dotnet-generics-under-the-hood/ — .NET generics under the hood

http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html — Java generics under the hood



Random IT Utensils

IT, operating systems, maths, and more.

Thanks!

CONTACT@ADAMFURMANEK.PL

HTTP://BLOG.ADAMFURMANEK.PL

FURMANEKADAM