Unit testing C++ template metaprograms

Ábel Sinkovics

ELTE, Budapest
16th July 2010.

C++ template metaprograms

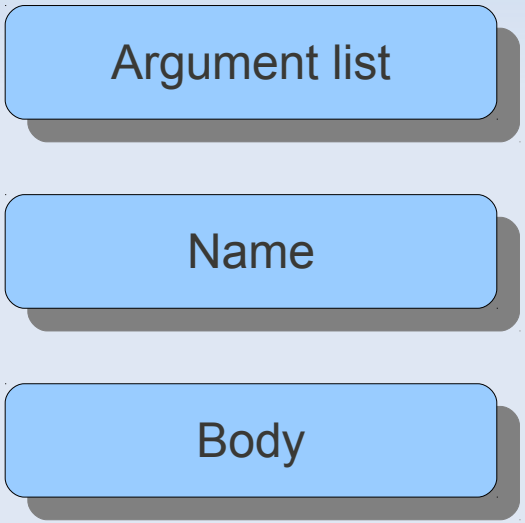
- Code evaluated at compilation time
- Erwin Unruh, 1994.
- Useful in many areas: extra verifications, DSL, etc.
- Strong connection with functional programming
- Library support: Boost metaprogramming library
- Syntax is complex, complexity increases rapidly

Unit testing frameworks for C++

- CppUnit
  - C++ port of JUnit
  - Manual test case registration
- Google C++ testing framework
  - Based on the xUnit architecture
  - Automatic test case registration
- Boost::test
  - Automatic and manual test registration
  - Supports output and template functions

C++ template metaprograms

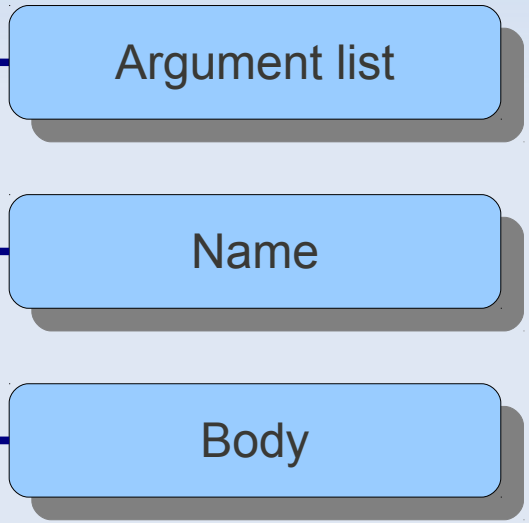
```
template <class T>
struct makeConst
{
  typedef const T type;
};
```




```
makeConst<int>::type
```

C++ template metaprograms

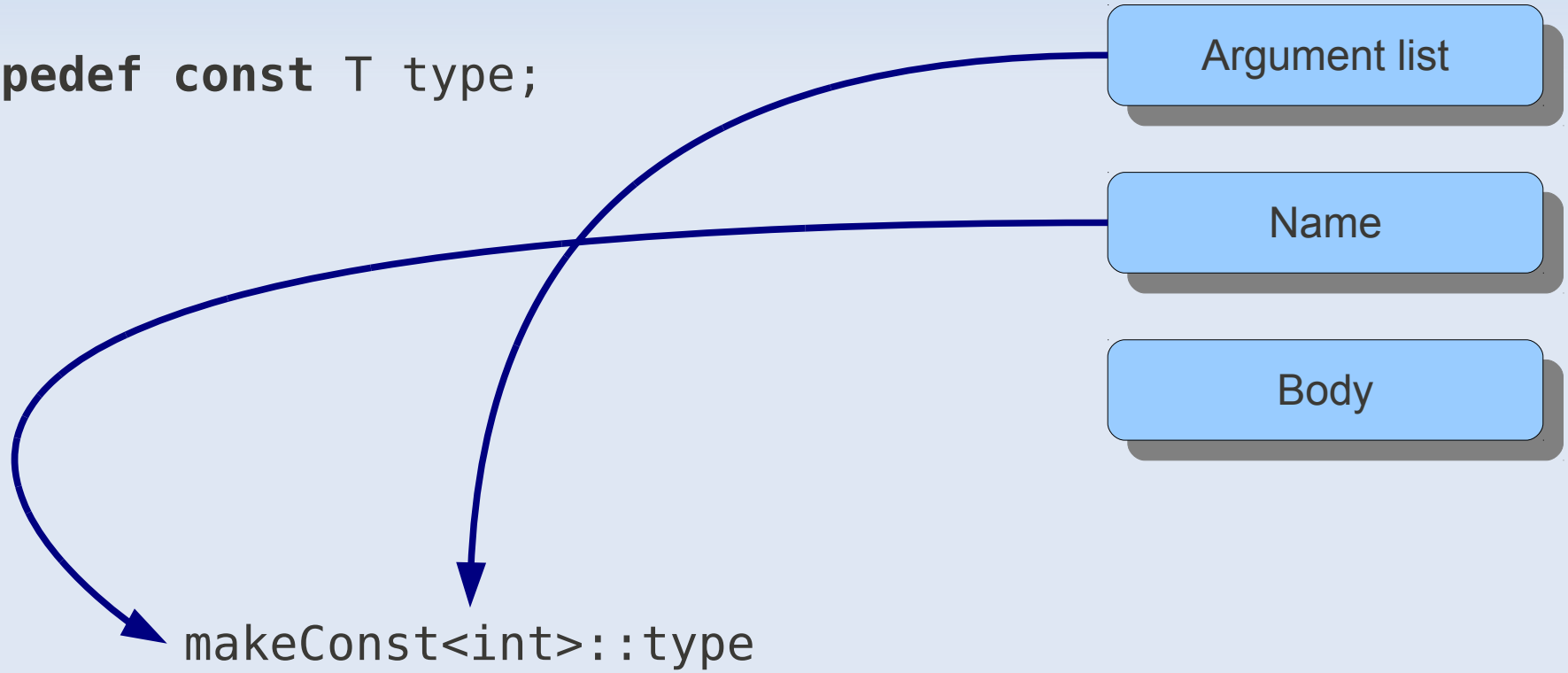
```
template <class T>
struct makeConst
{
  typedef const T type;
};
```




```
makeConst<int>::type
```

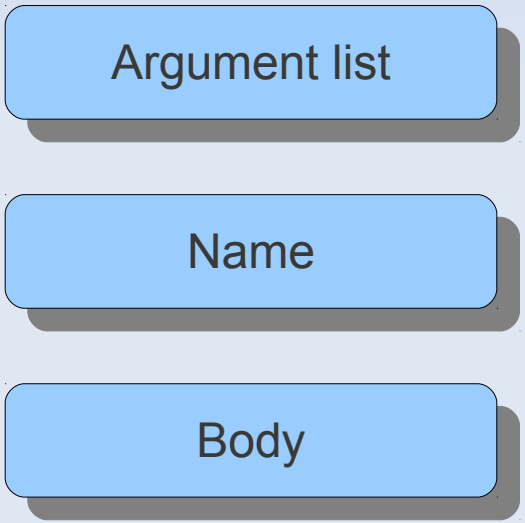
C++ template metaprograms

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```



Template metafunction class

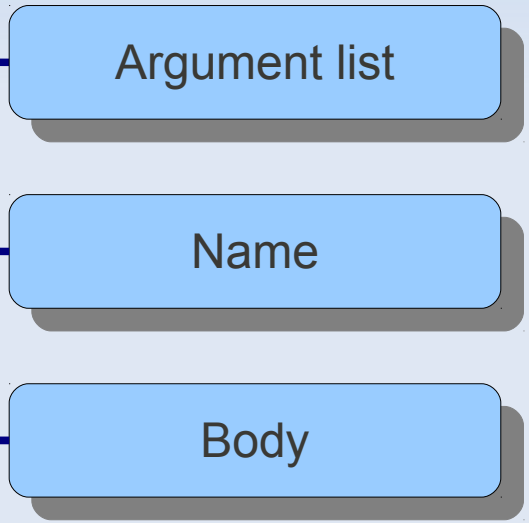
```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```




```
makeConst::apply<int>::type
```

Template metafunction class

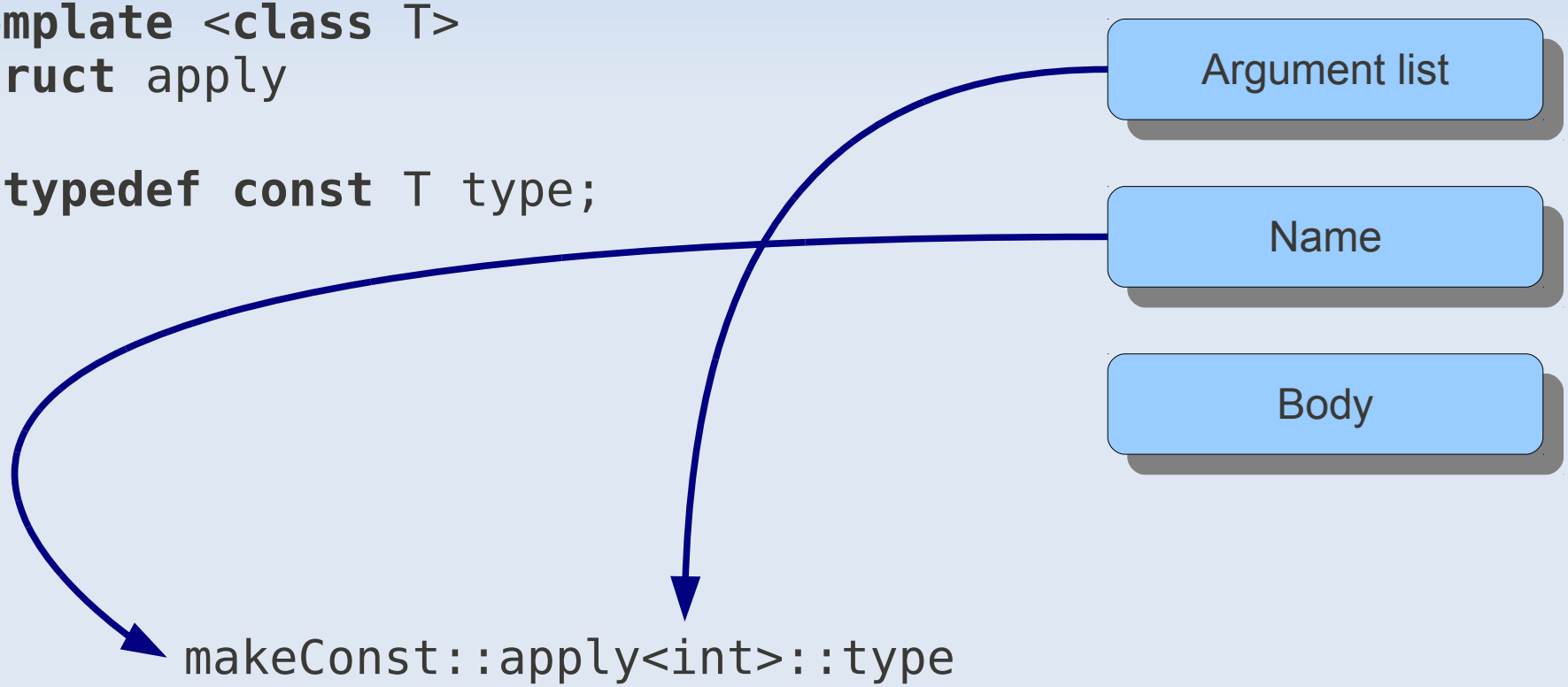
```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```




```
makeConst::apply<int>::type
```

Template metafunction class

```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```



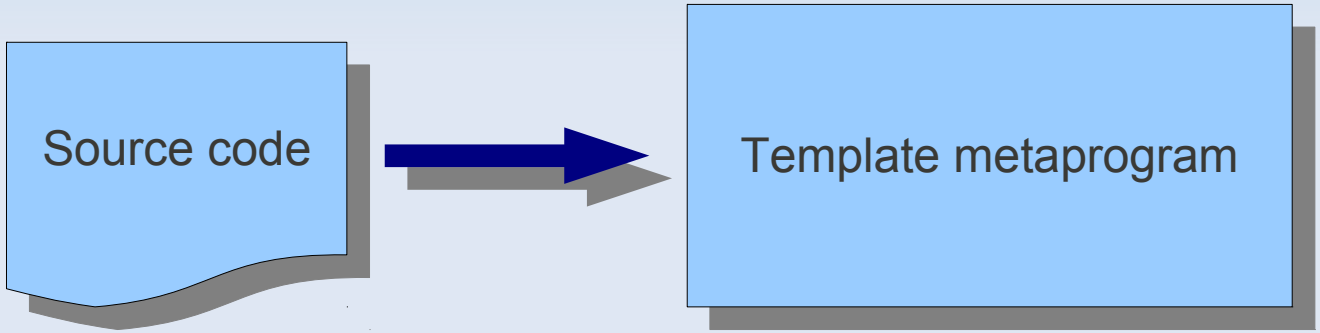
What is a test case?

- Test cases are template metafunctions
- They evaluate to a boolean value

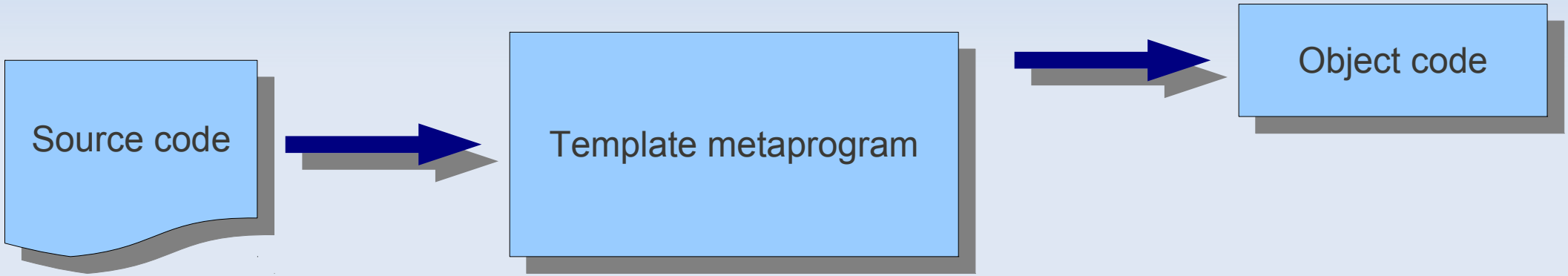
```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

Input and output of template metaprograms

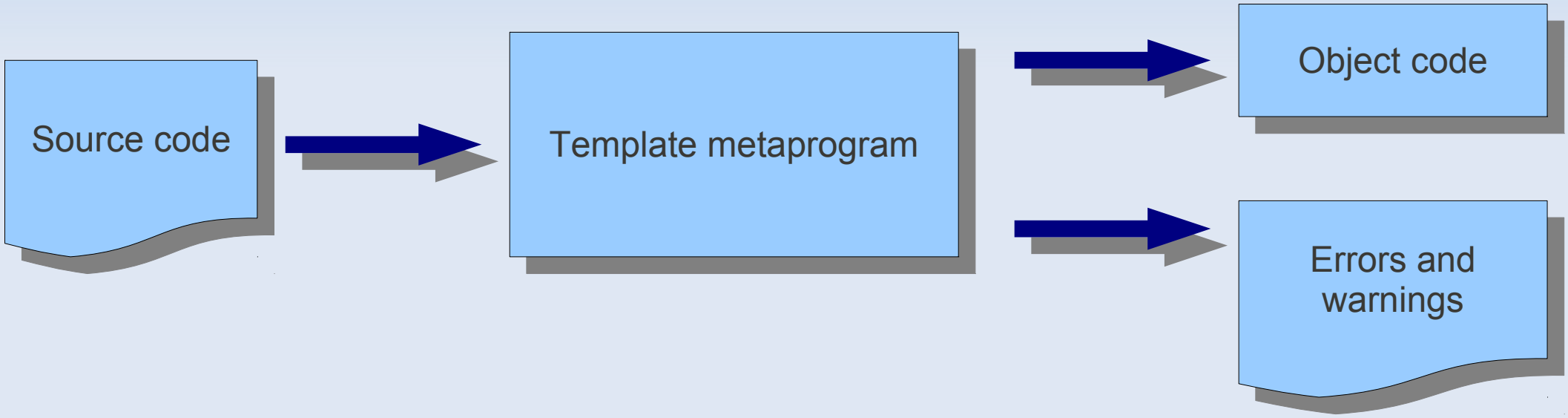
Input and output of template metaprograms



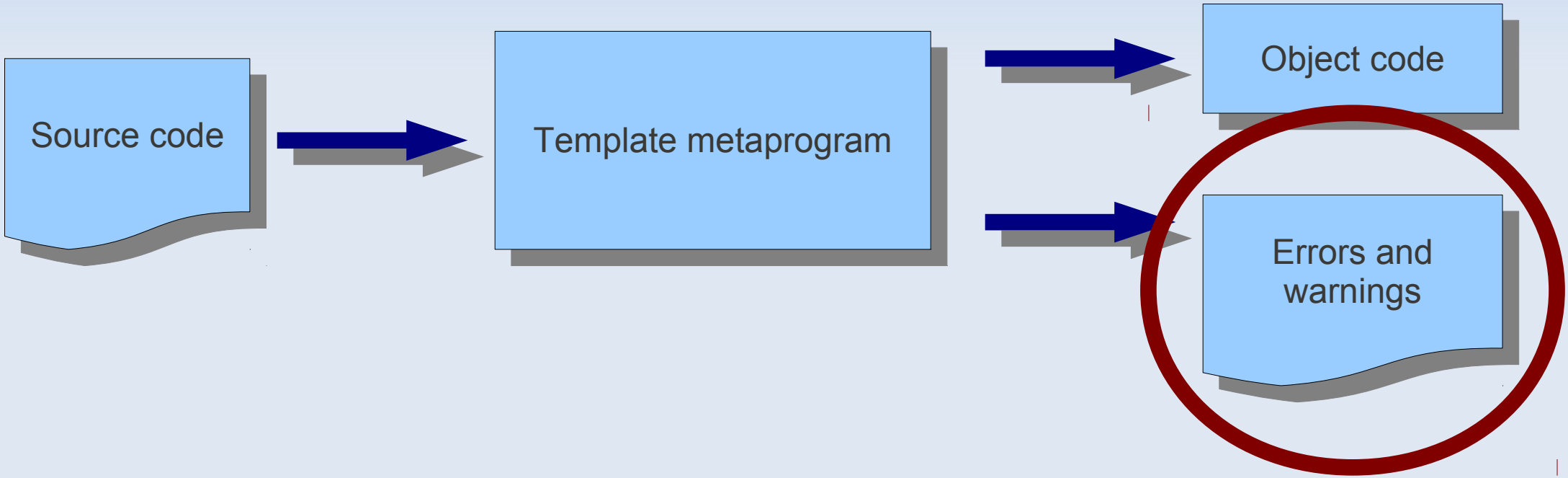
Input and output of template metaprograms



Input and output of template metaprograms



Input and output of template metaprograms

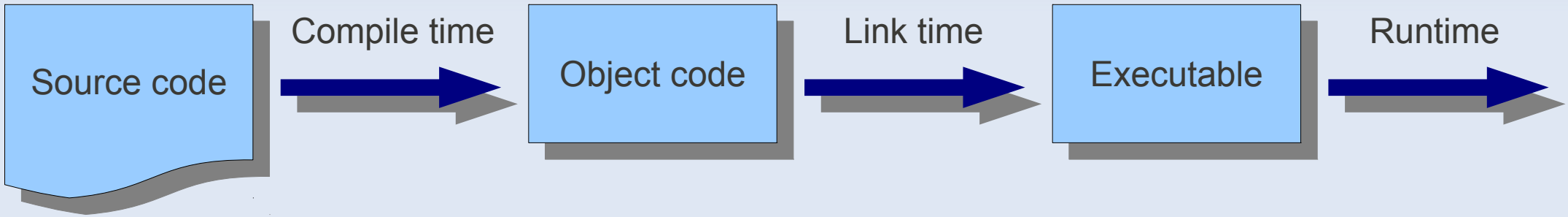


Using asserts

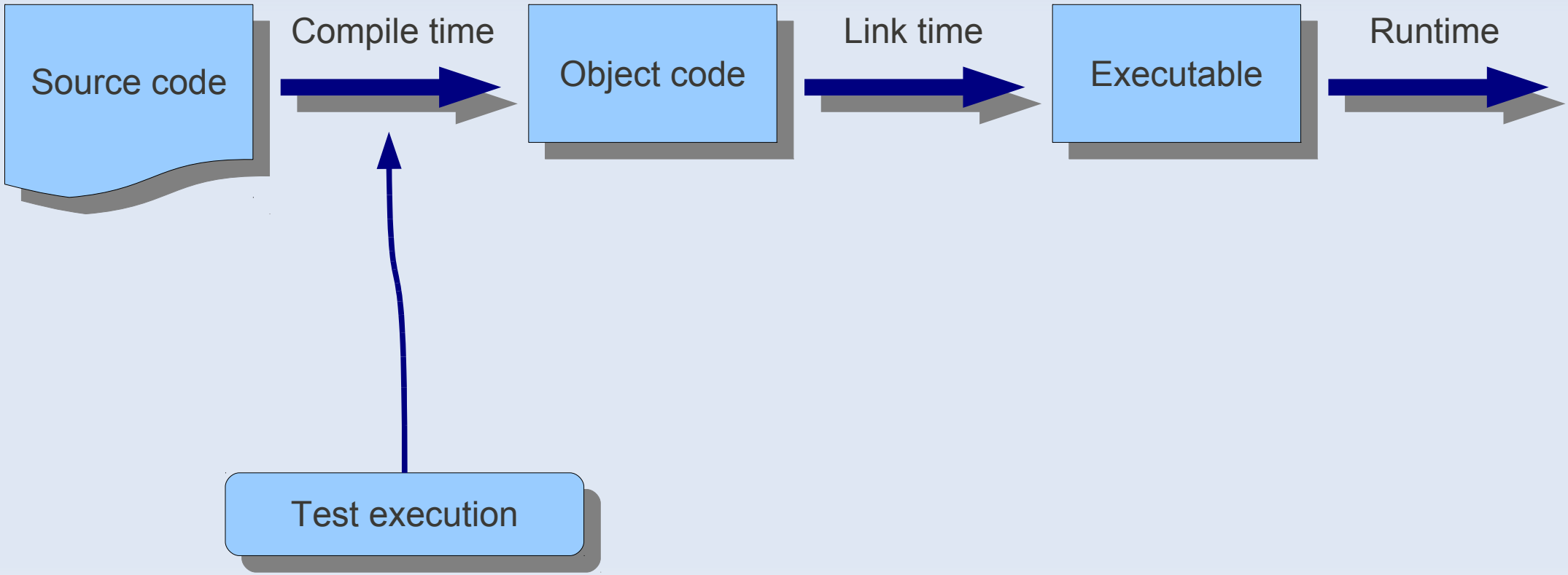
- Boost provides compile-time assertion
- It breaks the compilation when the assertion fails

```
BOOST_STATIC_ASSERT(my_test::type::value);
```

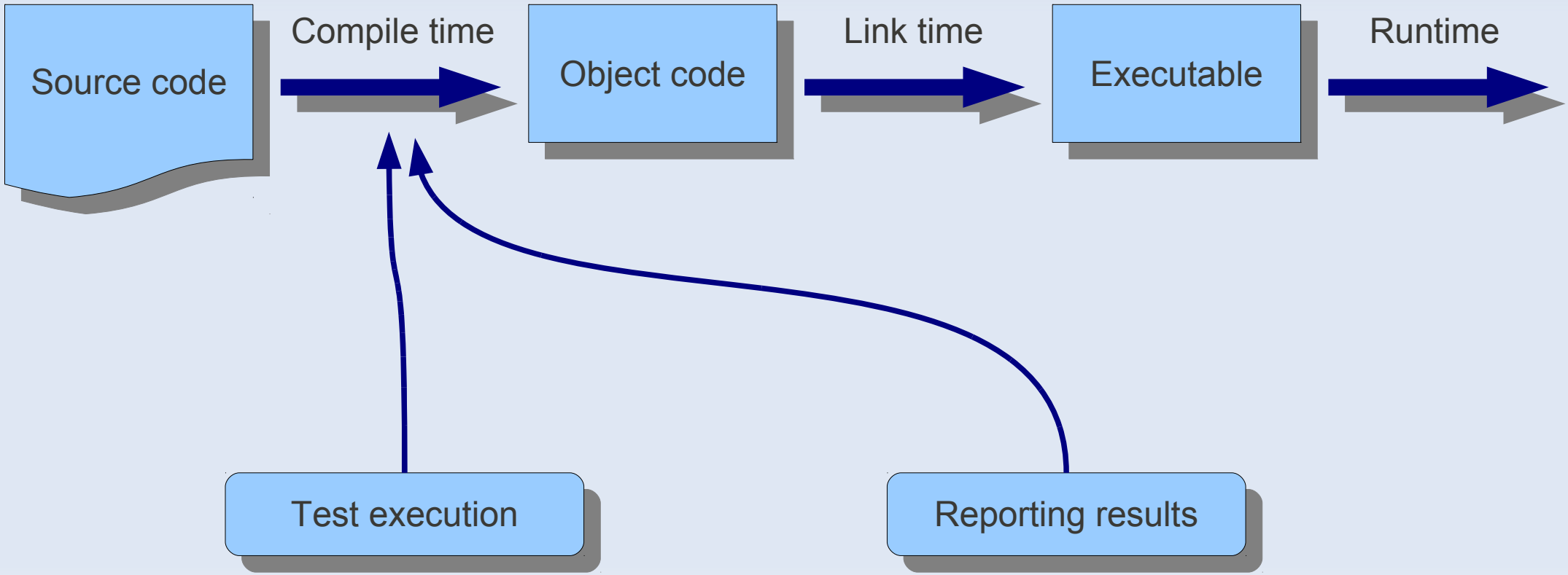
Testing workflow



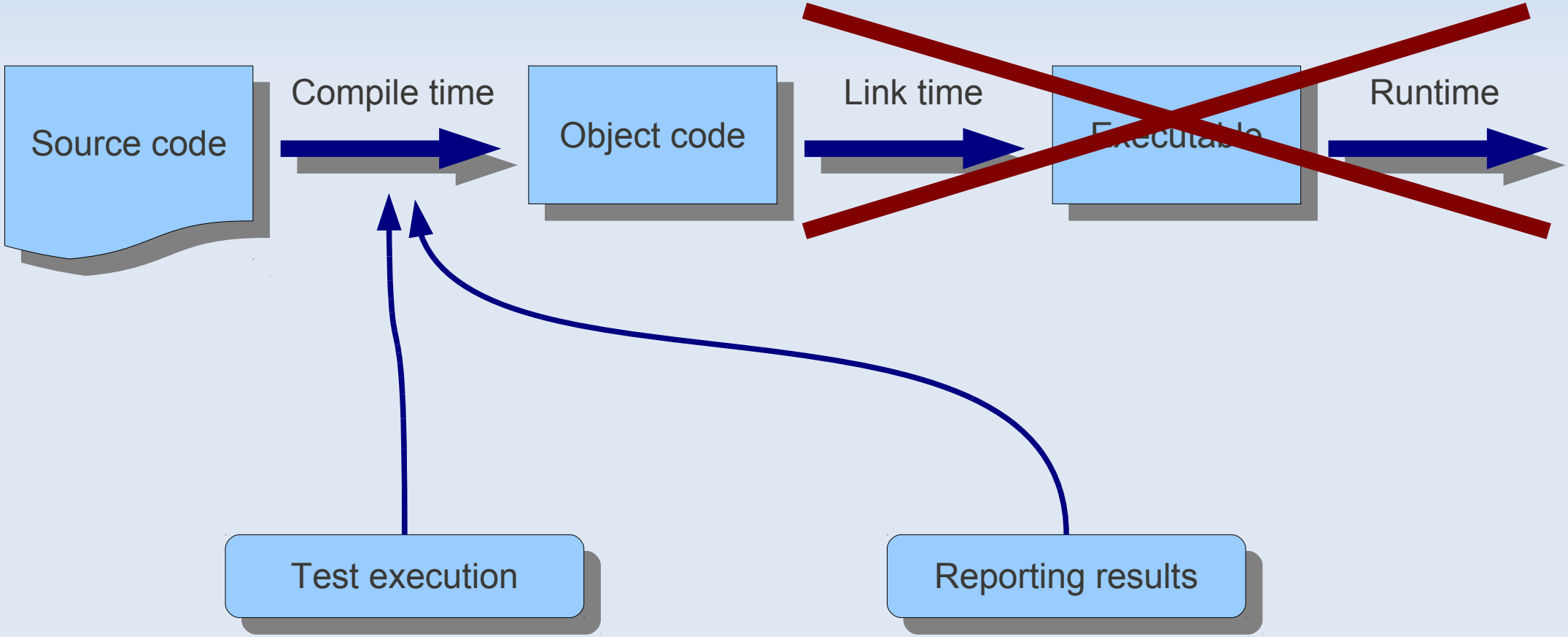
Testing workflow



Testing workflow



Testing workflow



Errors and warnings

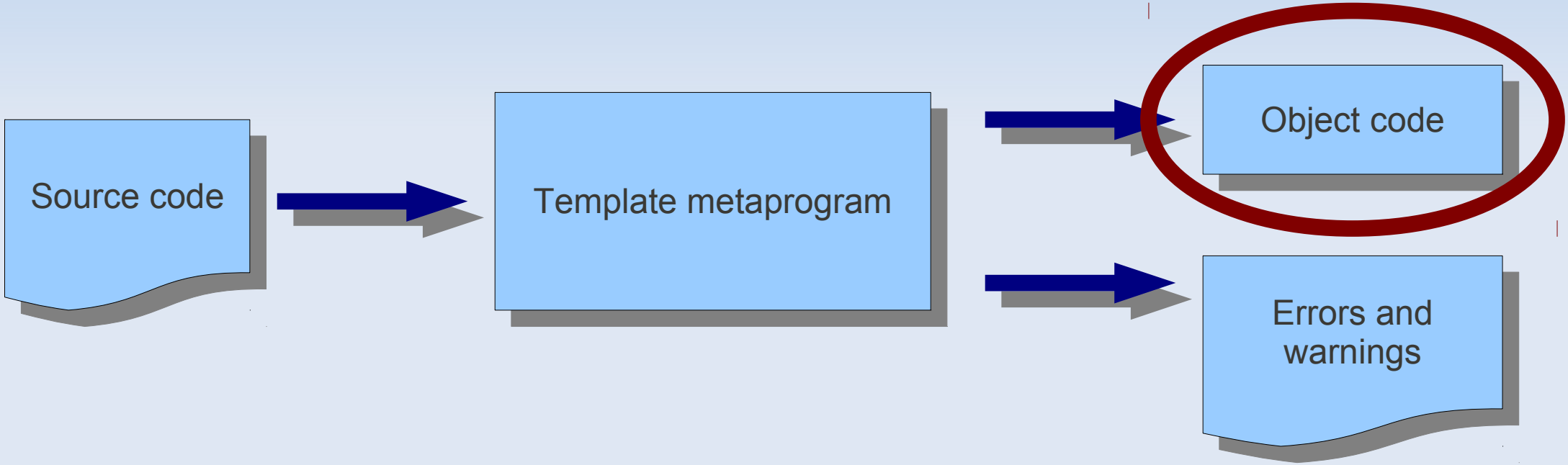
```
struct test1
{
  // ...
};

struct test2
{
  // ...
};

struct test3
{
  // ...
};
```

Errors and warnings

```
struct test1
{
  // ...
};

struct test2
{
  // ...
};

struct test3
{
  // ...
};
```

```
In file included from /usr/include/boost/mpl/aux_/include_pr
eprocessed.hpp:37,
                 from /usr/include/boost/mpl/aux_/arithmetic
                 _op.hpp:34,
                 from /usr/include/boost/mpl/divides.hpp:19,
                 from t1.cpp:2:
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp: In
 instantiation of 'boost::mpl::divides_impl<mpl_::integral_c
 _tag, mpl_::integral_c_tag>::apply<mpl_::int_<7>, mpl_::int
 _<0> >':
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp:70:
   instantiated from 'boost::mpl::divides<mpl_::int_<7>, mpl
   _::int_<0>, mpl_::na, mpl_::na, mpl_::na>'
t1.cpp:16:   instantiated from 'divide<mpl_::int_<7>, mpl_::
int_<0> >'
t1.cpp:20:   instantiated from here
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp:142
: error: '(7 / 0)' is not a valid template argument for type
 'int' because it is a non-constant expression
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp: In
 instantiation of 'boost::mpl::divides<mpl_::int_<7>, mpl_::
 int_<0>, mpl_::na, mpl_::na, mpl_::na>':
t1.cpp:16:   instantiated from 'divide<mpl_::int_<7>, mpl_::
int_<0> >'
t1.cpp:20:   instantiated from here
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp:70:
 error: no type named 'type' in 'struct boost::mpl::divides_
 impl<mpl_::integral_c_tag, mpl_::integral_c_tag>::apply<mpl
 _::int_<7>, mpl_::int_<0> >'
t1.cpp: In instantiation of 'divide<mpl_::int_<7>, mpl_::int
_<0> >':
t1.cpp:20:   instantiated from here
t1.cpp:16: error: no type named 'type' in 'struct boost::mpl
::divides<mpl_::int_<7>, mpl_::int_<0>, mpl_::na, mpl_::na,
mpl_::na>'
t1.cpp: In function 'int main()':
t1.cpp:20: error: 'type' is not a member of 'divide<mpl_::in
t_<7>, mpl_::int_<0> >'
t1.cpp:20: error: expected ';' before 'x'
```

Input and output of template metaprograms



Object code

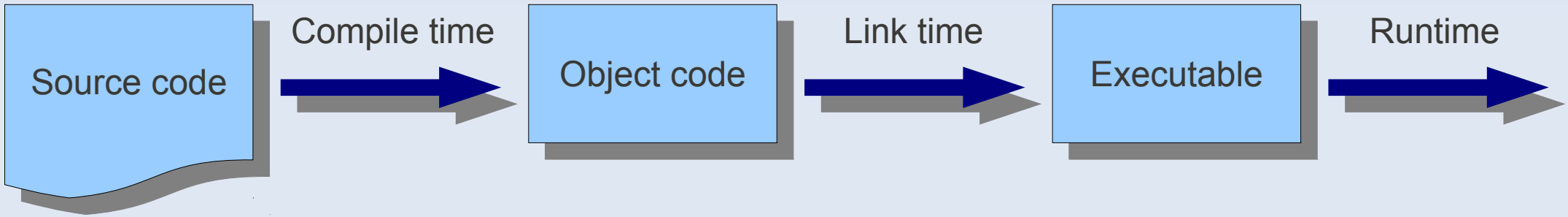
```
struct test1
{
  // ...
};

struct test2
{
  // ...
};

struct test3
{
  // ...
};
```

Object code

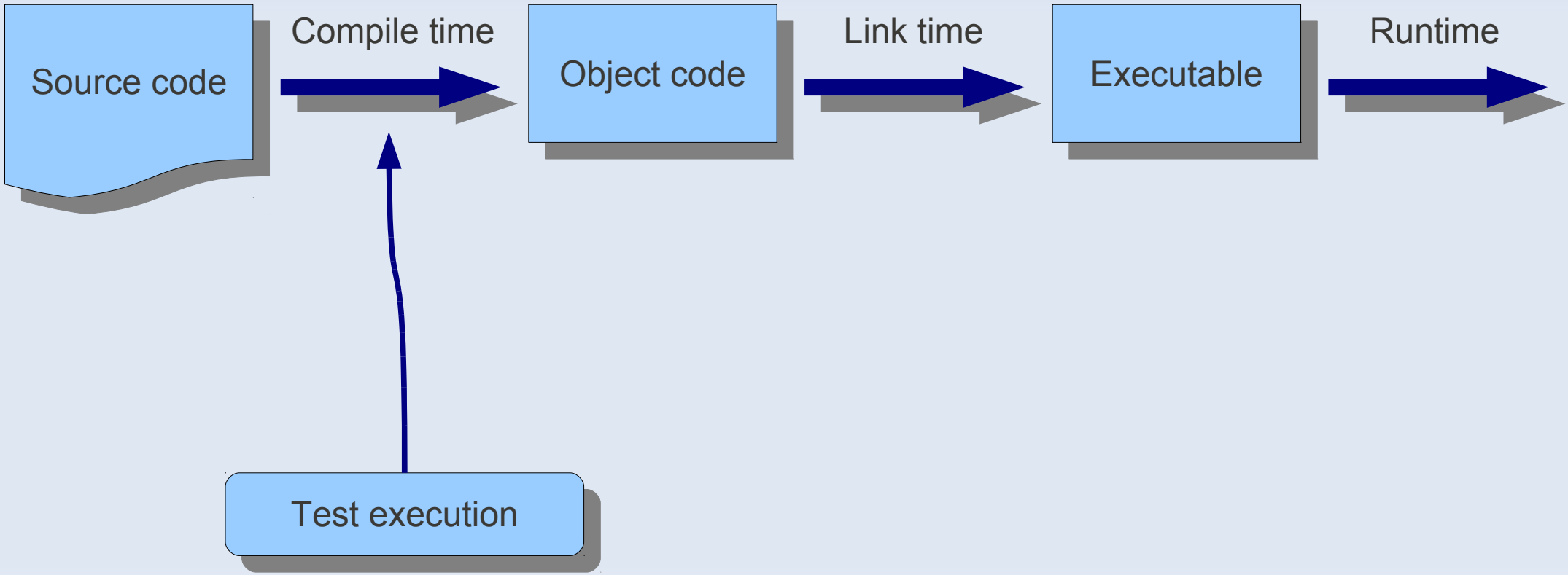
```
struct test1
{
  // ...
};

struct test2
{
  // ...
};

struct test3
{
  // ...
};
```

```
std::cout
  << ”test1 PASSED”
  << std::endl;


std::cout
  << ”test2 FAILED”
  << std::endl;


std::cout
  << ”test3 PASSED”
  << std::endl;
```

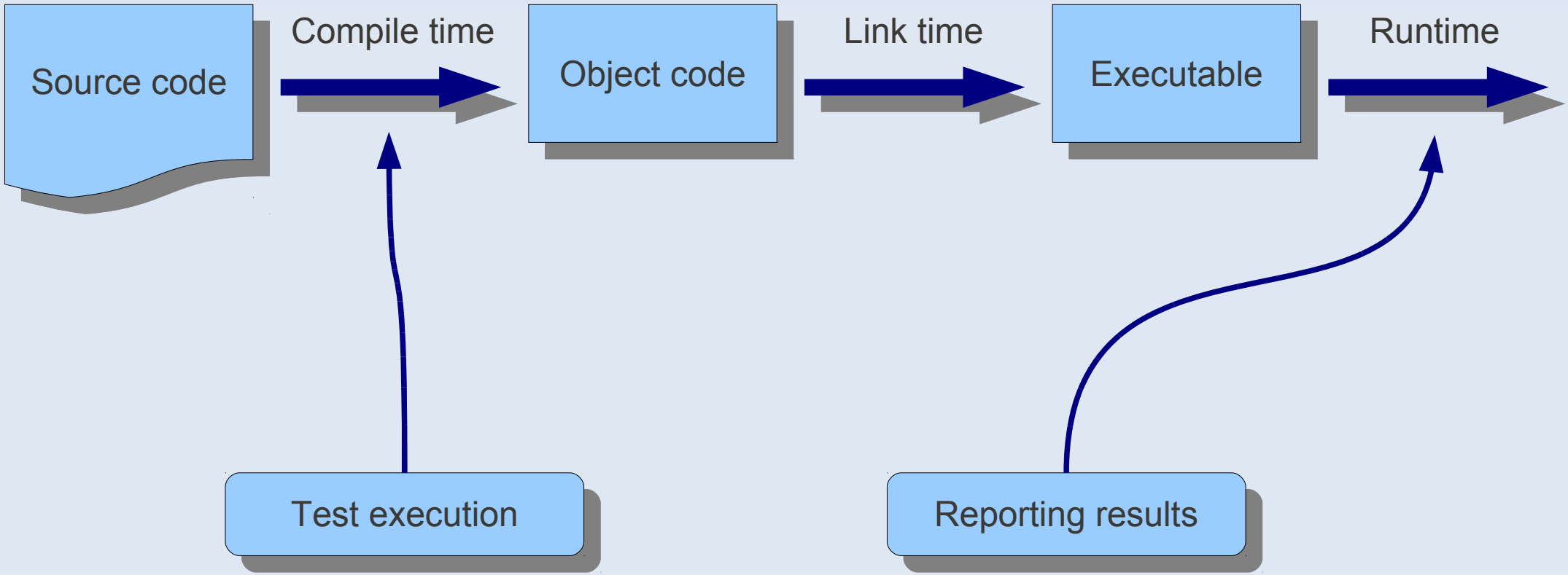
Testing workflow



Testing workflow



Testing workflow



Object code

```
struct test1
{
  // ...
};

struct test2
{
  // ...
};

struct test3
{
  // ...
};
```

```
test1 PASSED
test2 FAILED
test3 PASSED
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

```
my_test::type
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

```
my_test::type::value
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

```
if (my_test::type::value) std::cout << "PASSED";
  else std::cout << "FAILED";
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;



  run_my_test()
  {

    if (my_test::type::value) std::cout << "PASSED";
      else std::cout << "FAILED";
  }
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;

struct run_my_test
{
  run_my_test()
  {

    if (my_test::type::value) std::cout << "PASSED";
      else std::cout << "FAILED";
  }


};
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;

struct run_my_test
{
  run_my_test()
  {

    if (my_test::type::value) std::cout << "PASSED";
      else std::cout << "FAILED";
  }

  static run_my_test instance;
};

run_my_test run_my_test::instance;
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;

struct run_my_test
{
  run_my_test()
  {
    std::cout << "my suite::my_test:";
    if (my_test::type::value) std::cout << "PASSED";
      else std::cout << "FAILED";
  }

  static run_my_test instance;
};

run_my_test run_my_test::instance;
```

How it works

```
typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;
```

```
TestSuite suite("my suite");

MPLLIBS_ADD_TEST(suite, my_test)
```

Evaluation

- 8 criteria in the documentation of boost::test's unit testing framework
- They are for testing run time, not compile time code

Evaluation

- ”Writing a unit test module should be simple and obvious for new users.”

```
TestSuite suite("my suite");

typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;


MPLLIBS_ADD_TEST(suite, my_test)
```

Evaluation

- ”The framework should allow advanced users to perform nontrivial tests.”

```
TestSuite suite("my suite");

typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;


MPLLIBS_ADD_TEST(suite, my_test)
```

Evaluation

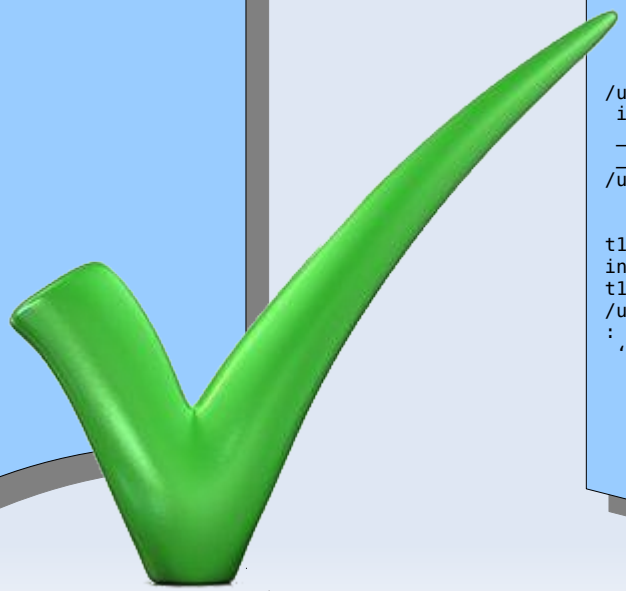
- ”Test module should be able to have many small test cases and developer should be able to group them into test suites.”

```
TestSuite suite("my suite");

typedef
  equal_to< int_<13>, my_complex_function<> >
  my_test;


MPLLIBS_ADD_TEST(suite, my_test)
```

Evaluation

- ”At the beginning of the development users want to see verbose and descriptive error message, whereas during the regression testing they just want to know if any tests failed.”

```
test1 PASSED
test2 FAILED
test3 PASSED
```



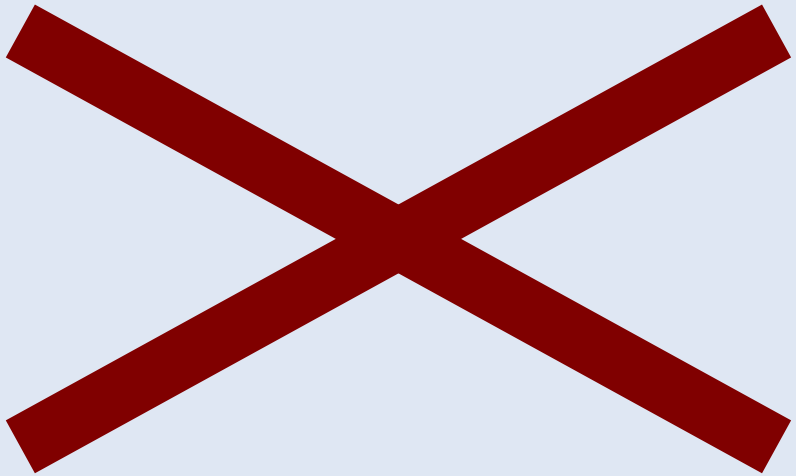
```
In file included from /usr/include/boost/mpl/aux_/include_pr
eprocessed.hpp:37,
                 from /usr/include/boost/mpl/aux_/arithmetic
                 _op.hpp:34,
                 from /usr/include/boost/mpl/divides.hpp:19,
                 from t1.cpp:2:
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp: In
 instantiation of ‘boost::mpl::divides_impl<mpl_::integral_c
 _tag, mpl_::integral_c_tag>::apply<mpl_::int_<7>, mpl_::int
 _<0> >’:
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp:70:
   instantiated from ‘boost::mpl::divides<mpl_::int_<7>, mpl
   _::int_<0>, mpl_::na, mpl_::na, mpl_::na>’
t1.cpp:16:   instantiated from ‘divide<mpl_::int_<7>, mpl_::
int_<0> >’
t1.cpp:20:   instantiated from here
/usr/include/boost/mpl/aux_/preprocessed/gcc/divides.hpp:142
: error: ‘(7 / 0)’ is not a valid template argument for type
 ‘int’ because it is a non-constant expression
```

Evaluation

- ”For a small test modules run time should prevail over compilation time: user don't want to wait a minute to compile a test that takes a second to run.”

Evaluation

- ”For long and complex tests users want to be able to see the test progress.”

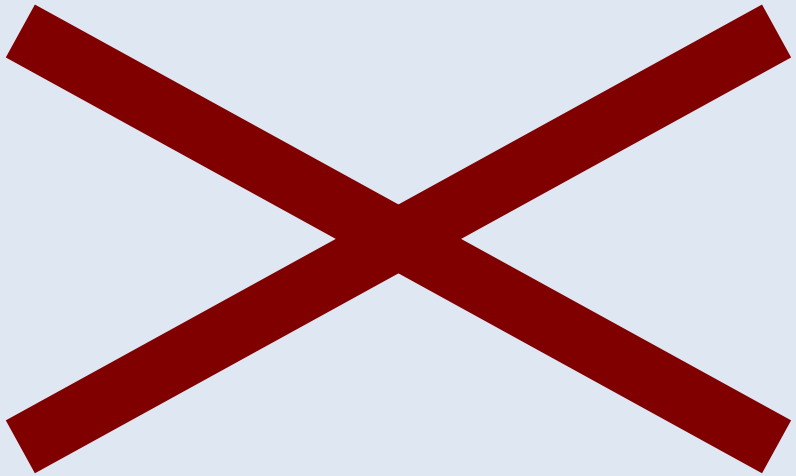


Evaluation

- ”Simplest tests shouldn't require an external library.”

Evaluation

- ”For long term usage users of a unit test framework should be able to build it as a standalone library.”



Summary

- C++ template metaprograms are complex
- Automatic testing is essential
- We provide a framework for doing this
- We provide human readable summary
- We meet most of the criteria

Unit testing C++ template metaprograms

Ábel Sinkovics
abel@elte.hu

http://abel.web.elte.hu/mpllibs/