# C++11 User-defined Literals

Michael Rüegg on behalf of Prof. Peter Sommerlad

ACCU 2013 / Bristol, UK / April 10, 2013

## Who are we?

Figure : University of Applied Sciences Rapperswil, Switzerland

## Built-in literals

## Basic type literals

- C++ has literals for its basic data types:
  - integer literals
  - floating-point literals
  - character literals
  - string literals
  - boolean literals
- Determine value and basic type:

```
decltype(42)      => int
decltype('i')     => char
decltype("accu")  => const char[4+1]
decltype(nullptr) => nullptr_t
```

## Prefixes and suffixes

```
decltype(42UL)       => unsigned long
decltype(L'i')       => wchar_t
decltype(U"UTF-32") => const char32_t[]
```

- They allow us to pick a particular **function overload**:

```
void foo(int);
void foo(unsigned);
foo(42);  // => foo(int)
foo(42U); // => foo(unsigned)
```

- And they also influence **interpretation**:

```
auto i = 42;   // decimal value 42
auto j = 0x42; // decimal value 66
auto s = "(C:\\foo.txt)"; // results in (C:\foo.txt)
auto t = R"(C:\foo.txt)"; // results in C:\foo.txt
```

## Motivation

## Why do we need UDLs?

- What is the meaning of this?

```
const int distanceToTarget = 42;
```

- Constants without dimension are confusing!
  - e.g., miles vs. kilometers
  - speed in mph, km/h, mach or c units
- Wouldnt it be nice to write out **values and units together**:

```
assert(1_kg == 2.2_lb);
```

- String literals in C++ aren't `std::strings`:

```
auto s = "hello"; // => const char[]
```

## Why do we need UDLs? ...continued

- To spell out complex numbers:

```
auto val = 3.14_i; // complex<long double>(0, 3.14)
```

- Or even to have binary literals:

```
int answer = 101010_B; // answer = 42
```

- And to **prevent silly bugs**:

```
Probability p1 = 0.5_P    // OK
Probability p2 = 1.2_P;   // NOK: 0 <= P <= 1
Probability p3 = -0.5_P;  // NOK: 0 <= P <= 1
css::Margin::Size top1 = 42_pt; // OK
css::Margin::Size top2 = 42;    // NOK
```

## User-defined literals 101

## Basic form

```
namespace distance_in_meters {
  constexpr double operator"" _m(long double x) {
    return x;
  }
  constexpr double operator"" _m(unsigned long long x) {
    return x;
  }
}
```

- Overload a **special non-member operator** function type `operator"" _suffix(arg)`
- Blank following `""` and _ prefix are required
- It is *only* possible to define **suffix operators** (no prefixes!)

## Best practices

- If possible, define UDL operator `constexpr` (literals are often used to initialize compile-time constants)
- Suffixes will be short and might easily clash => Put those suffix operator functions in **separate namespaces**, e.g. `namespace suffix { }`
- To use it: `using namespace suffix;`
- Fully qualified suffix qualifiers are *not* possible (e.g., `distance_in_meters::_m`)
- No ADL possible because UDLs only work with arguments of built-in types

## Extended example

```
namespace distance_in_meters { // norm things to meters
  constexpr long double operator"" _feet(long double d) {
    return d * 0.3048;
  }
  constexpr long double operator"" _feet(
      unsigned long long d) {
    return d * 1.0_feet;
  }
  constexpr long double operator"" _yard(long double d) {
    return d * 3_feet;
}} // and so on...

void testYardsAndFeet() {
  using namespace distance_in_meters;
  ASSERT_EQUAL_DELTA(0.9144_m, 1_yard, 0.00001);
  ASSERT_EQUAL_DELTA(0.9144_m, 1e0_yard, 0.00001);
  ASSERT_EQUAL(1_yard, 3_feet);
}
```

## How does it work?

- What happens if a C++11 compiler encounters `0.9144_m`?
  1. Recoginition of the floating point literal
  2. Lookup of user-defined literal operators
  3. Match suffix if possible
  4. Interpret `0.9144` to the longest floating point type
- Computation of `0.9144` to `long double` is called **cooking**

## String literal operator

## Define real string literals

```
namespace mystring {
  std::string operator"" _s(char const *s,
                              std::size_t len) {
    return std::string{s, len};
  }
}
```

- Example allows to use auto with string literals and get a std::string variable:

  ```
  using namespace mystring;
  auto s = "hello"_s;
  s += " world\n";
  std::cout << s;
  ```

## Raw literal operators

## Motivation

- There are cases when we are forced to analyze literals character by character:
  - Interpretation of literals with validity checks
  - Prevent loss of number precision while type coercion
- Syntax to get an **"uncooked"** string:

```
namespace mystring {
std::string operator"" _s(char const*s) { //no size_t
  return std::string{s};
}}
```

- Raw UDL operators work on numbers and string literals
- **Avoid them if possible**, because they cannot be evaluated at compile-time in general

## Example parsing binary literals

```
unsigned operator"" _b(const char* str) {
  unsigned ans = 0;
  for (size_t i = 0;  str[i] != '\0';  ++i) {
    char digit = str[i];
    if (digit != '1' && digit != '0')
      throw std::runtime_error("only 1s and 0s allowed");
    ans = ans * 2 + (digit - '0');
  }
  return ans;
}
// Credit: Andrzej Krzemieński
```

- Cannot be used to define compile-time constants:

  ```
  constexpr unsigned i = 110_b; // does not compile
  ```

- => **Variadic templates** to the rescue!

## Literal operators with variadic templates (VT)

## VT 101

- Example Boost Tuple library:

```
struct null_type {};
template<
class T0=null_type, class T1=null_type,
class T2=null_type, class T3=null_type,
class T4=null_type, class T5=null_type,
class T6=null_type, class T7=null_type,
class T8=null_type, class T9=null_type >
class tuple { /* ... */ };
```

- With variadic templates:

```
template <typename... Elements> // parameter pack
class tuple;
typedef tuple <int, int> coordinate;
```

## VT 101 continued

```
template <typename... Args>  // parameter pack
struct ArgCounter;

template <> // basic case
struct ArgCounter <> {
  static const int result = 0;
};

template <typename T, typename... Args> // recursive case
struct ArgCounter <T, Args...> {
  // Args... => pack expansion
  static const int result =
                 1 + ArgCounter <Args...>::result;
};
// ArgCounter <int, float, double, char>::result == 4

// Credit: Douglas Gregor et al., "Variadic Templates"
```

## static_assert 101

- Syntax: `static_assert(b, msg)`?
- It is a **compile-time assert**
- Allows for nicer compile errors and for checking your codes assumptions about its environment
- Message is **not optional** and must be a **string literal**! (already an issue for the next C++ std)
- Useful for compile-time checks, such as:

```
static_assert(sizeof(int) == 8, "int is not 64 bit");
```

## VT applied

```
template <unsigned VAL> // basic case
constexpr unsigned build_bin_literal() { return VAL; }

template <unsigned VAL, char DIGIT, char... REST>
constexpr unsigned build_bin_literal() { // recursion
  static_assert(is_binary(DIGIT), "only 0s and 1s");
  return build_bin_literal<(2*VAL+DIGIT-'0'), REST...>();
}
template <char... STR>
constexpr unsigned operator"" _b() { // literal operator
  static_assert(sizeof...(STR) <= NumberOfBits<unsigned>()
                , "too long");
  return build_bin_literal<0, STR...>();
} // Credit: Andrzej Krzemieński
```

- Now, the compiler **yields an error** for faulty binary literals:

  ```
  unsigned i = 102_b; // Does not compile => Hurray!
  ```

## Wrap-up

## Trade-offs

- Usage of TMP for implementing UDLs can **increase compile-times** significantly
- TMP with UDLs needs **unit tests** too! (tests that are evaluated at compile-time)
- Some consider UDLs an unnecessary complication of the language. Alternatives are:
  - Usage of <chrono> makes code also very readable:

    ```
    using namespace std::chrono;
    auto duration = hours{7} + minutes{30};
    ```

  - `Boost.Units` library with SI units and a very readable physics notation:

    ```
    using namespace boost::units;
    using namespace boost::units::si;

    quantity<nanosecond> t = 1.0 * seconds; // 1e9 ns
    quantity<length> L     = 2.0 * meters;
    ```

## Benefits and Outlook

- User-defined literals can help
  - to get dimensions of constants right
  - to make code more readable
  - as a short-hand for type conversions
  - implement compile-time constants of literal types
  - but also to obfuscate program code :-) if applied mercilessly
- Future standard will provide suffixes for `std::string`, `std::complex`, `std::chrono::duration`
- For more information consult **N3531**

## Questions?

INSTITUTE
FOR
SOFTWARE

Figure : http://ifs.hsr.ch

- *Credits*:
  - Slides from Prof. Peter Sommerlad
  - Code examples from Andrzej Krzemieńskis C++ Blog

IFS INSTITUTE FOR SOFTWARE

Free Trial

**Contact Info:** Prof. Peter Sommerlad
**phone:** +41 55 222 4984
**email:** ifs@hsr.ch

# Linticator

**Linticator gives you immediate feedback on programming style and common programmer mistakes by integrating Gimpel Software's popular PC-lint and FlexeLint static analysis tools into Eclipse CDT.**
PC-lint and FlexeLint are powerful tools, but they are not very well integrated into a modern developer's workflow. **Linticator** brings the power of Lint to the Eclipse C/C++ Development Tools by fully integrating them into the IDE. With Linticator, you get continuous feedback on the code you are working on, allowing you to write better code.

- **Automatic Lint Configuration**
  Lint's configuration, like include paths and symbols, is automatically updated from your Eclipse CDT settings, freeing you from keeping them in sync manually.
- **Suppress Messages Easily**
  False positives or unwanted Lint messages can be suppressed directly from Eclipse, without having to learn Lint's inhibition syntax–either locally, per file or per symbol.
- **Interactive "*Linting*" and Information Display**
  Lint is run after each build or on demand, and its findings are integrated into the editor by annotating the source view with interactive markers, by populating Eclipse's problem view with Lint's issues and by linking these issues with our *Lint Documentation View*.
- **Quick-Fix Coding Problems**
  Linticator provides automatic fixes for a growing number of Lint messages, e.g, making a reference-parameter const can be done with two keystrokes or a mouse-click.

Register at http://linticator.com if you want to try it for 30 days or order via email. Linticator is available for Eclipse CDT 3.4 (Ganymede) up to 4.2 (Juno). It is compatible with Freescale CodeWarrior and other Embedded C/C++ IDEs based on Eclipse CDT.

More information:
http://linticator.com

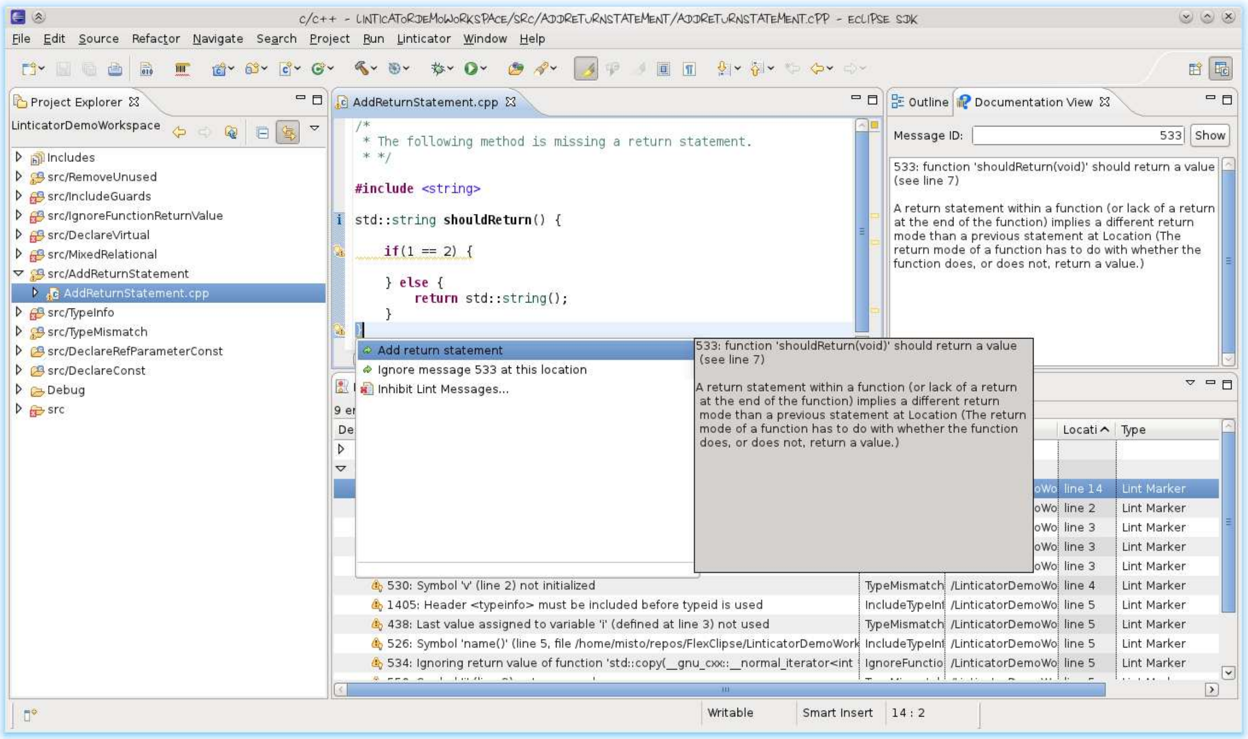
Pricing for Linticator is **CHF 500.-** per user (non-floating license). A maintenance contract that is required for updates costs 20% of license fee per year. The compulsory first maintenance fee includes 6 month of free updates.
**Orders, enquiries for multiple, corporate or site licenses are welcome** at ifs@hsr.ch.

Linticator requires a corresponding PC-Lint (Windows) or FlexeLint license per user.

## IFS Institute for Software

IFS is an institute of HSR Rapperswil, member of FHO University of Applied Sciences Eastern Switzerland.
In January 2007 IFS became an associate member of the Eclipse Foundation.

The institute manages research and technology transfer projects of four professors and hosts a dozen assistants and employees. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**
http://ifs.hsr.ch



```
/*
 * The following method is missing a return statement.
 * */

#include <string>

std::string shouldReturn() {

    if(1 == 2) {

    } else {
        return std::string();
    }
}
```

533: function 'shouldReturn(void)' should return a value (see line 7)

A return statement within a function (or lack of a return at the end of the function) implies a different return mode than a previous statement at Location (The return mode of a function has to do with whether the function does, or does not, return a value.)

```
                                                                                          Locati   Type
                                                                                    oWo   line 14  Lint Marker
                                                                                    oWo   line 2   Lint Marker
                                                                                    oWo   line 3   Lint Marker
                                                                                    oWo   line 3   Lint Marker
                                                                                    oWo   line 3   Lint Marker
530: Symbol 'v' (line 2) not initialized                           TypeMismatch  /LinticatorDemoWo  line 4   Lint Marker
1405: Header <typeinfo> must be included before typeid is used     IncludeTypeInf /LinticatorDemoWo  line 5   Lint Marker
438: Last value assigned to variable 'i' (defined at line 3) not used  TypeMismatch /LinticatorDemoWo line 5 Lint Marker
526: Symbol 'name()' (line 5, file /home/misto/repos/FlexClipse/LinticatorDemoWork IncludeTypeInf /LinticatorDemoWo line 5 Lint Marker
534: Ignoring return value of function 'std::copy(__gnu_cxx::__normal_iterator<int IgnoreFunctio /LinticatorDemoWo line 5 Lint Marker
```

IFS INSTITUTE FOR SOFTWARE

Contact Info: Prof. Peter Sommerlad
**phone:** +41 55 222 4984
**email:** ifs@hsr.ch

Free Trial

# Includator

**#include Structure Analysis and Optimization for C++ for Eclipse CDT**

The **Includator** plug-in analyzes the dependencies of C++ source file structures generated by #include directives, suggests how the *#include structure* of a C++ project can be optimized and performs this optimization on behalf of the developer. The aim of these optimizations is to improve code readability and quality, reduce coupling and thus reduce duration of builds and development time of C++ software.

**Includator Features**

- **Find unused includes**
  Scans a single source file or a whole project for superfluous #include directives and proposes them to be removed. This also covers the removal of #include directives providing declarations that are (transitively) reachable through others.
- **Directly include referenced files**
  Ignores transitively included declarations and proposes to #include used declarations directly, if they are not already included. This provides an “include-what-you-use” code structure.
- **Organize includes**
  Similar to Eclipse Java Development Tool's *Organize imports* feature for Java. Adds missing #include directives and removes superfluous ones.
- **Find unused files**
  Locates single or even entangled header files that are never included in the project’s source files.
- **Replace includes with forward declarations (EXPERIMENTAL)**
  Locates #include directives for definitions that can be omitted, when replacing them with corresponding forward declarations instead. This one is useful for minimizing #includes and reducing dependencies required in header files.
- **Static code coverage (EXPERIMENTAL)**
  Marks C++ source code as either *used*, *implicitly used* or *unused* by transitively following C++ elements’ definitions and usages. This helps to trim declarations and definitions not used from your source code. In contrast to dynamic code coverage, such as provided by our CUTE plug-in (http://cute-test.com) it allows to determine required and useless C++ declarations and definitions instead of executed or not-executed statements.

#

More information:
http://includator.com

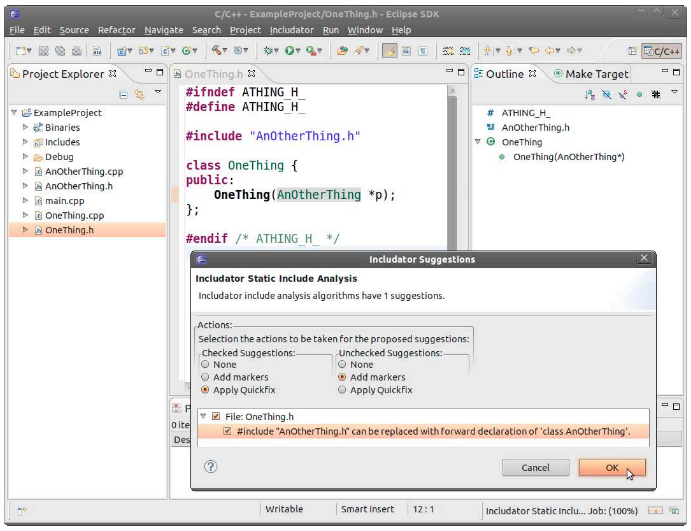
Includator is **CHF 830.-** per user (non-floating license). A maintenance contract is required for updates at 20% of the license fee per year. The compulsory first maintenance fee includes 6 month of free updates. **Orders, enquiries for multiple, corporate or site licenses are welcome** at ifs@hsr.ch.

**Enquiries for corporate or site licenses are welcome** at ifs@hsr.ch. **Significant volume discounts** apply.

Get Swiss Quality
CUTE

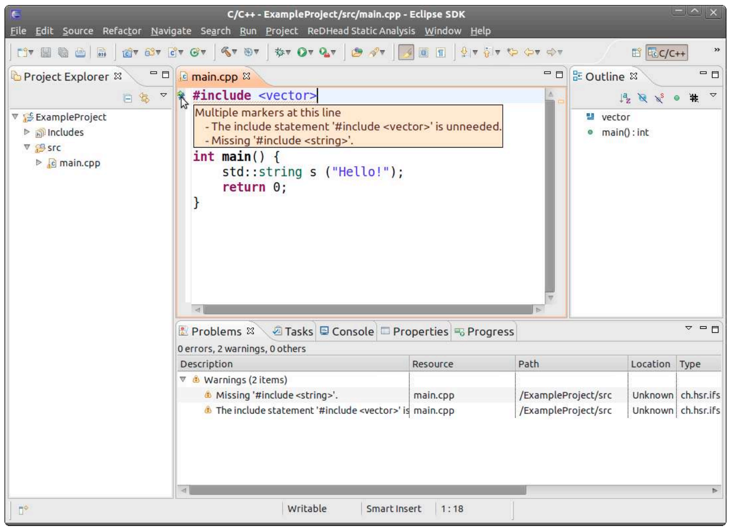
**User Feedback and Participation**

Includator is free to try and test. Register at http://includator.com for a 30-day trial license.



```
#ifndef ATHING_H_
#define ATHING_H_

#include "AnOtherThing.h"

class OneThing {
public:
    OneThing(AnOtherThing *p);
};

#endif /* ATHING_H_ */
```

Includator Suggestions
Includator Static Include Analysis
Includator include analysis algorithms have 1 suggestions.
Actions:
Selection the actions to be taken for the proposed suggestions:
Checked Suggestions: ( ) None ( ) Add markers [✓ Apply Quickfix]
Unchecked Suggestions: ( ) None [✓ Add markers] ( ) Apply Quickfix
☑ File: OneThing.h
☑ #include "AnOtherThing.h" can be replaced with forward declaration of 'class AnOtherThing'.



```
#include <vector>
int main() {
    std::string s ("Hello!");
    return 0;
}
```

Multiple markers at this line
- The include statement '#include <vector>' is unneeded.
- Missing '#include <string>'.

0 errors, 2 warnings, 0 others

| Description | Resource | Path | Location | Type |
|---|---|---|---|---|
| Warnings (2 items) | | | | |
| Missing '#include <string>'. | main.cpp | /ExampleProject/src | Unknown | ch.hsr.ifs |
| The include statement '#include <vector>' is | main.cpp | /ExampleProject/src | Unknown | ch.hsr.ifs |

IFS INSTITUTE FOR SOFTWARE

Free

**Contact Info:** Prof. Peter Sommerlad
**phone:** +41 55 222 4984
**email:** ifs@hsr.ch

# Green Bar for C++ with CUTE

Get Swiss Quality CUTE+

ECLIPSE FOUNDATION MEMBER

**Eclipse plug-in for C++ unit testing with CUTE**

Automated unit testing improves program code quality, even under inevitable change and refactoring. As a side effect, unit tested code has a better structure. Java developers are used to this because of JUnit and its tight integration into IDEs like Eclipse. We provide the modern C++ Unit Testing Framework CUTE (C++ Unit Testing Easier) and a corresponding Eclipse plug-in. The **free CUTE plug-in** features:

- wizard creating test projects (including required framework sources)
- test function generator with automatic registration
- detection of unregistered tests with quick-fix for registration
- test navigator with green/red bar (see screen shots to the right)
- diff-viewer for failing tests (see screen shot down to the right)
- code coverage view with gcov (see screen shot below)

**Support for Test-driven Development (TDD) and automatic Code Generation**

The CUTE plug-in incorporates support for Test-Driven Development (TDD) in C++ and preview of Refactoring features developed by IFS and its students.

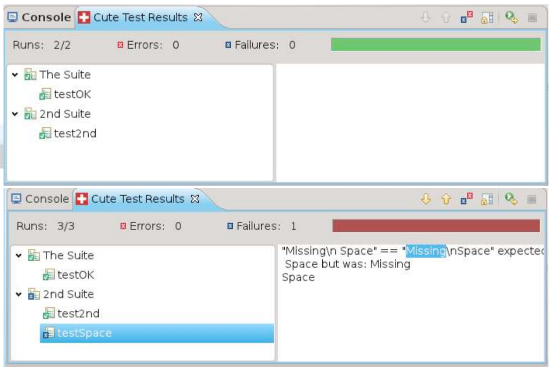
- create unknown function, member function, variable, or type from its use in a test case as a quick-fix (see screen shots below)
- move function or type from test implementation to new header file, after completion TDD cycle.
- toggle function definitions between header file and implementation file, for easier change of function signature, including member functions (part of CDT itself)
- extract template parameter for dependency injection, aka instrumenting code under test with a test stub through a template argument (instead of Java-like super-class extraction)
- check out Mockator flyer page for further code refactoring and generation features.

More information:
http://cute-test.com

**IFS Institute for Software**

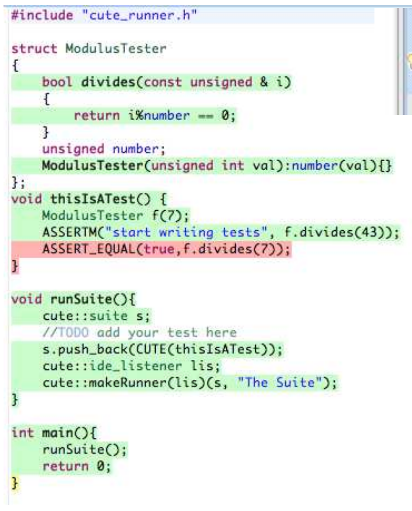
IFS is an Institute of HSR Rapperswil, member of FHO University of Applied Sciences Eastern Switzerland.
In January 2007 IFS became an associate member of the Eclipse Foundation.
The institute manages research and technology transfer projects of four professors and hosts a dozen assistants and employees. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**
http://ifs.hsr.ch

**Eclipse update site for installing the free CUTE plug-in:**
http://cute-test.com/updatesite

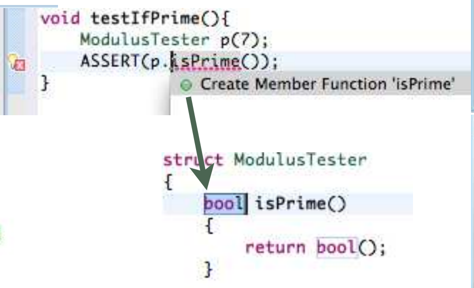


```
Console | Cute Test Results
Runs:  2/2      Errors:  0      Failures:  0
▾ The Suite
    testOK
▾ 2nd Suite
    test2nd
```

```
Console | Cute Test Results
Runs:  3/3      Errors:  0      Failures:  1
▾ The Suite                 "Missing\n Space" == "Missing\nSpace" expecte
    testOK                   Space but was: Missing
▾ 2nd Suite                 Space
    test2nd
    testSpace
```

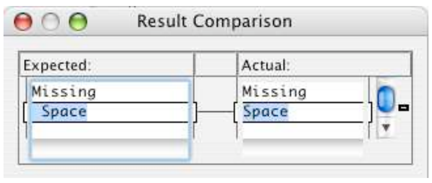


```
#include "cute_runner.h"

struct ModulusTester
{
    bool divides(const unsigned & i)
    {
        return i%number == 0;
    }
    unsigned number;
    ModulusTester(unsigned int val):number(val){}
};
void thisIsATest() {
    ModulusTester f(7);
    ASSERTM("start writing tests", f.divides(43));
    ASSERT_EQUAL(true,f.divides(7));
}

void runSuite(){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(thisIsATest));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main(){
    runSuite();
    return 0;
}
```



```
void testIfPrime(){
    ModulusTester p(7);
    ASSERT(p.isPrime());
}
```

```
struct ModulusTester
{
    bool isPrime()
    {
        return bool();
    }
```



| Expected: | Actual: |
|---|---|
| Missing | Missing |
| Space | Space |

INSTITUTE FOR SOFTWARE

Free

**Contact Info:** Prof. Peter Sommerlad

**phone:** +41 55 222 4984

**email:** ifs@hsr.ch

# Mockator - Eclipse CDT Plug-in for C++ Seams and Mock Objects

## Refactoring Towards Seams

Breaking dependencies is an important task in refactoring legacy code and putting this code under tests. Michael Feathers' seam concept helps in this task because it enables to inject dependencies from outside to make code testable. But it is hard and cumbersome to apply seams without automated refactorings and tool chain configuration assistance. Mockator provides support for seams and creates the boilerplate code and necessary infrastructure for the following four seam types:

- ***Object seam***: Based on inheritance to inject a subclass with an alternative implementation. Mockator helps in extracting an interface class and in creating the missing test double class including all used virtual member functions.
- ***Compile seam***: Inject dependencies at compile-time through template parameters. Apply the “Extract Template Parameter” refactoring and Mockator creates the missing test double template argument class including all used member functions.
- ***Preprocessor seam***: With the help of the C++ preprocessor, Mockator redefines function names to use an alternative implementation without changing original code.
- ***Link seam***: Mockator supports three kinds of link seams that also allow replacing dependencies without changing existing code:
  - Shadow functions through linking order (override functions in libraries with new definitions in object files)
  - Wrap functions with GNU's linker option -wrap (GNU Linux only)
  - Run-time function interception with the preload functionality of the dynamic linker for shared libraries (GNU Linux and MacOS X only)

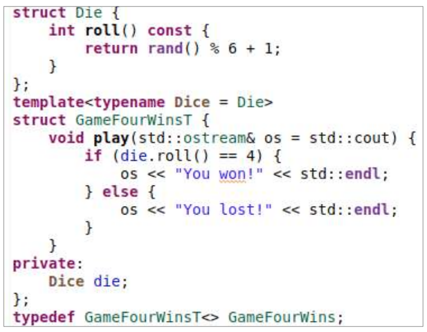
## Creating Test Doubles Refactorings

Mockator offers a header-only mock object library and an Eclipse CDT plug-in to create test doubles for existing code in a simple yet powerful way. It leverages new C++11 language facilities while still being compatible with C++03. Features include:

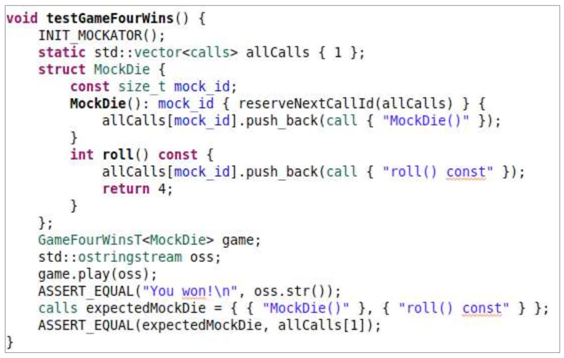
- Mock classes and free functions with sophisticated IDE support
- Easy conversion from fake to mock objects that collect call traces
- Convenient specification of expected calls with C++11 initializer lists or with Boost assign for C++03. Support for regular expressions to match calls.

Integrates easily with CUTE!

Get Swiss Quality CUTE+



```
struct Die {
    int roll() const {
        return rand() % 6 + 1;
    }
};
template<typename Dice = Die>
struct GameFourWinsT {
    void play(std::ostream& os = std::cout) {
        if (die.roll() == 4) {
            os << "You won!" << std::endl;
        } else {
            os << "You lost!" << std::endl;
        }
    }
private:
    Dice die;
};
typedef GameFourWinsT<> GameFourWins;
```



```
void testGameFourWins() {
    INIT_MOCKATOR();
    static std::vector<calls> allCalls { 1 };
    struct MockDie {
        const size_t mock_id;
        MockDie(): mock_id { reserveNextCallId(allCalls) } {
            allCalls[mock_id].push_back(call { "MockDie()" });
        }
        int roll() const {
            allCalls[mock_id].push_back(call { "roll() const" });
            return 4;
        }
    };
    GameFourWinsT<MockDie> game;
    std::ostringstream oss;
    game.play(oss);
    ASSERT_EQUAL("You won!\n", oss.str());
    calls expectedMockDie = { { "MockDie()" }, { "roll() const" } };
    ASSERT_EQUAL(expectedMockDie, allCalls[1]);
}
```

More information:
http://Mockator.com

## IFS Institute for Software

IFS is an Institute of HSR Rapperswil, member of FHO University of Applied Sciences Eastern Switzerland.
In January 2007 IFS became an associate member of the Eclipse Foundation.
The institute manages research and technology transfer projects of four professors and hosts a dozen assistants and employees. Contact us if you look for **Code Reviews**, **UI**, **Design** and **Architecture Assessments.**
http://ifs.hsr.ch
Also check out IFS' other plug-ins
http://cute-test.com
http://sconsolidator.com
http://linticator.com
http://includator.com

**Eclipse update site for installing Mockator for free:**
http://mockator.com/update/indigo
http://mockator.com/update/juno

IFS INSTITUTE FOR SOFTWARE

**Contact Info:** Prof. Peter Sommerlad
**phone:** +41 55 222 4984
**email:** ifs@hsr.ch

Free

# SConsolidator

## Eclipse CDT plug-in for SCons

*SCons* (http://www.SCons.org/) is an open source software build tool which tries to fix the numerous weaknesses of *make* and its derivatives. For example, the missing automatic dependency extraction, make's complex syntax to describe build properties and cross-platform issues when using shell commands and scripts. *SCons* is a self-contained tool which is independent of existing platform utilities. Because it is based on Python a SCons user has the full power of a programming language to deal with all build related issues.

However, maintaining a SCons-based C/C++ project with Eclipse CDT meant, that all the intelligence SCons puts into your project dependencies had to be re-entered into Eclipse CDT's project settings, so that CDT's indexer and parser would know your code's compile settings and enable many of CDT's features. In addition, SCons' intelligence comes at the price of relatively long build startup times, when SCons (re-) analyzes the project dependencies which can become annoying when you just fix a simple syntax error.

SConsolidator addresses these issues and provides tool integration for SCons in Eclipse for a convenient C/C++ development experience. The **free** plug-in features:

- conversion of existing CDT managed build projects to SCons projects
- **import of existing SCons projects into Eclipse with wizard support**
- SCons projects can be managed either through CDT or SCons
- interactive mode to quickly build single source files speeding up round trip times
- a special view for a convenient build target management of all workspace projects
- graph visualization of build dependencies with numerous layout algorithms and search and filter functionality that enables debugging SCons scripts.
- quick navigation from build errors to source code locations

More information:
http://SConsolidator.com

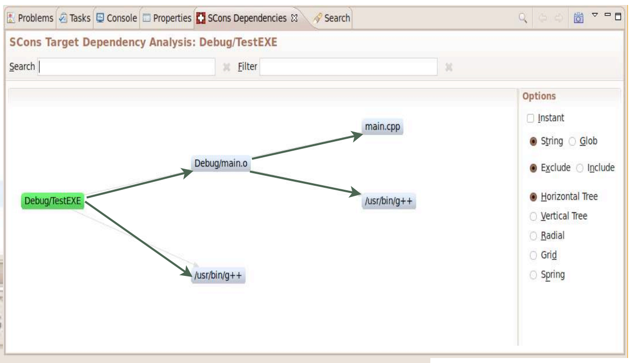
Install the **free SConsolidator plug-in** from the following Eclipse update site:

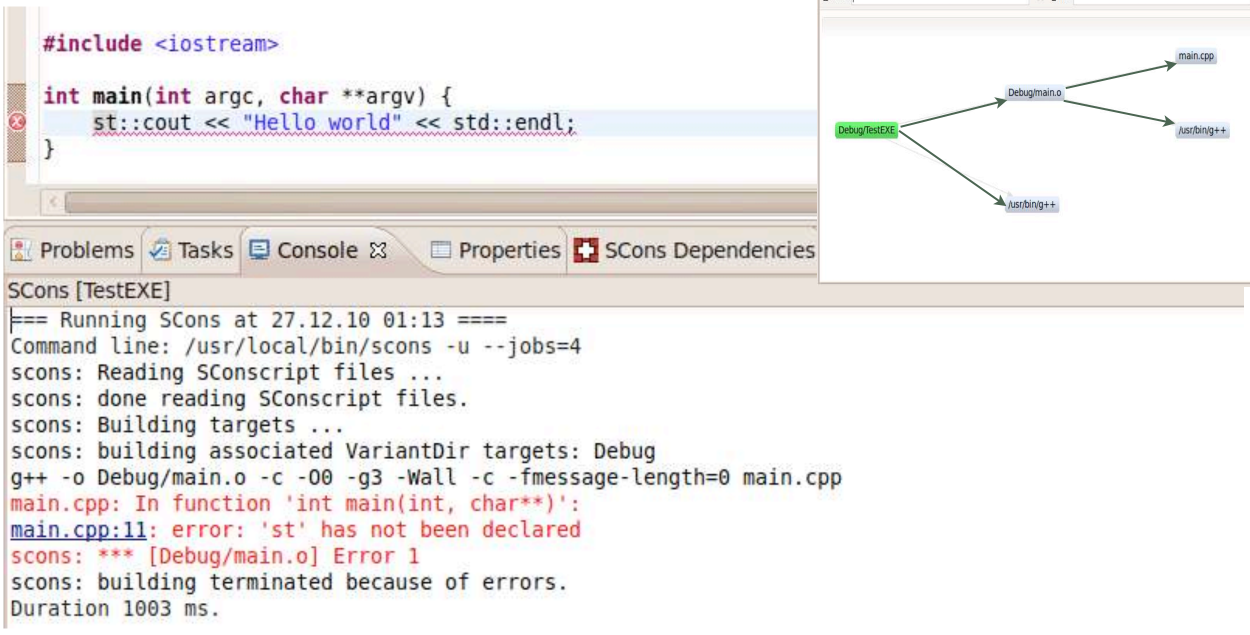
http://SConsolidator.com/update

Also check out IFS' other plug-ins at:
http://cute-test.com
http://mockator.com
http://linticator.com
http://includator.com

SConsolidator has been successfully used to import the following SCons-based projects into Eclipse CDT:

- MongoDB
- Blender
- FreeNOS
- Doom 3
- COAST (http://coast-project.org)





```
#include <iostream>

int main(int argc, char **argv) {
    st::cout << "Hello world" << std::endl;
}
```

```
=== Running SCons at 27.12.10 01:13 ====
Command line: /usr/local/bin/scons -u --jobs=4
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: Debug
g++ -o Debug/main.o -c -O0 -g3 -Wall -c -fmessage-length=0 main.cpp
main.cpp: In function 'int main(int, char**)':
main.cpp:11: error: 'st' has not been declared
scons: *** [Debug/main.o] Error 1
scons: building terminated because of errors.
Duration 1003 ms.
```