Hands On With the C/C++ IDE

Severin Gehwolf, Jeff Johnston, Bernhard Merkle, Andrew Overholt
Hands On With The C/C++ IDE

In this tutorial, attendees will be led through focussed examples that illustrate how to effectively use the C/C++ IDE.

A set of C/C++ projects will show users how to take advantage of the CDT to develop, build, debug, test, and profile their code within Eclipse.
Virtual Images: VirtualBox/VM-Ware with Fedora14

Easy Tutorial Setup: Use Virtual Images:
- HIGHLY RECOMMENDED: ready to go
- 4GB Virtual Box Image File
- Fedora14 preinstalled with Eclipse CDT Linux Tools
- Available for
  - Oracle Virtual Box
  - VM-Ware: VM-Ware Workstation/Player

Setup CDT on Windows → Wascana (Doug Schaefer)

Setup on OS X ? (better use our Virtual Box Image)
Tutorial Setup 101

Copy **VirtualBox-Image** somewhere on HardDrive (4GB)

Install **VirtualBox-Installer** (for your OS)
We have Installers for Windows, MacOS, Linux, AMD/Intel

Startup VirtualBox

- Machine ➔ Add… (Ctrl-A)
- Select Fedora.vbox (Copied in Step1)
- Startup the “Fedora” Virtual Machine
- Should log in automatically, but if not:
  - User “eclipsecon2011”, Password “eclipsecon2011”

# Glossary and architecture

<table>
<thead>
<tr>
<th>Linux Tools Project</th>
<th>Other C/C++ plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Development Tooling (CDT)</td>
<td>Eclipse Platform</td>
</tr>
<tr>
<td>Native toolchain</td>
<td></td>
</tr>
</tbody>
</table>
Exercises

- Discovering and fixing source code errors
- Configuring the build
- Working with breakpoints and data available while debugging
- Finding memory usage problems
- Tracking down performance bottlenecks
- Performing refactorings
- Integration with UnitTests
- Finding bugs and errors with static analysis
Overview

Test Driven Development
- Eclipse plugins for TDD: CUTE
- Implementing an example

Static Analysis (SA)
- 3 rules of Scott Meyers “Effective C++ 2nd” (Item 3, 11, 14)
- Tools for SA:
  - Lint, gcc –weffc++
- Eclipse plugins for SA:
  - Codan
  - Linticator
  - Includator
Overview

Test Driven Development
- Eclipse plugins for TDD: CUTE
- Implementing an example

Static Analysis (SA)
- 3 rules of Scott Meyers “Effective C++ 2nd” (Item 3, 11, 14)
- Tools for SA:
  - Lint, gcc –weffc++
- Eclipse plugins for SA:
  - Codan
  - Linticator
  - Includator
CUTE

Project of IFS in Rapperswil, CH
- http://www.cute-test.com

Features
- “The JUnit for C/C++ Programmers”
- CUTE = C(++) Unit Testing Easy
  - Wizards to initialize and set up new tests
  - Test navigator with green/red bar
  - Diff-viewer for failing tests
Vicious Circle: Testing – Stress

Help:
- Write test FIRST!
- Automate tests
- Run them often
Unit-Testing 101

Test anything that **might** break
Test everything that **does** break

New code is guilty until proven innocent
Write at least as much test code as production code

Run local tests with each compile
Run all tests before check-in to repository
Structure of a typical Unit Testing Framework

- Setup
- Exercise
- Verify
- Teardown

Test Stub

SUT

Creation

Installation

Indirect Input

Return Values
Structure of a typical Unit Testing Framework

Test Assertion / Check statement
  – used in

Test (Member-)Function
  – defined in

TestCase Subclass bundling Tests
  – its objects contained in

Test Suite collecting test objects
  – executed by

Test Runner (often in a main() function)
  – delivers result

OK or Failure
```c
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

void thisIsATest() {
    ASSERTM("start writing tests", true);
}

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();
}
```
CUTE: Test Fixtures

```c
#include "cute.h"
#include "cute_equals.h"

#include "CircularBuffer.h" // if you have this class separate

struct ATest {
    CircularBuffer<int> buf; // SUT == System Under Test
    ATest():buf(4){}
    void testEmpty(){ ASSERT(buf.empty());}
    void testNotFull(){ ASSERT(!buf.full());}
    void testSizeZero(){ ASSERT_EQUAL(0,buf.size());}
};

#include "cute_testmember.h"

....
s.push_back(CUTE_SMEMFUN(ATest,testEmpty));
s.push_back(CUTE_SMEMFUN(ATest,testNotFull));
s.push_back(CUTE_SMEMFUN(ATest,testSizeZero));
...
Using CUTE: it IS EASY !!! 😊

#include "cute.h"

ASSERT(condition);
   – fails if condition is false

ASSERT_EQUAL(expected,actual);
   – fails if expected is not equal to actual

add a message by appending M
   – ASSERTM(msg,condition)
   – ASSERT_EQUALM(msg,exp,act)

FAIL(); FAILM(msg)
   – fails always, use to mark unwritten tests
   – or for checking exceptions
My first CUTE Test

Create new C++ CUTE project
- In Project Explorer
  - New Project
  - C++ Project
  - CUTE Project
  - give project name

Let the project compile

Run binary as a CUTE Test
- Observe Result in CUTE
- Results Tab and Console
- Navigate to the failing test

Fix the Test and observe
```cpp
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

void thisIsATest() {
    ASSERTM("start writing tests", true);
}

void run_suite() {
    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() {
    run_suite();
}
```
Collecting multiple Tests

CUTE collects test objects in `cute::test_suite`
  - this is just a `std::vector<cute::test>`

add your tests to your test suite
  - `s.push_back(CUTE(testfunction));`
  - `s.push_back(testfunctor());`

An overloaded operator `+=` could ease syntax:
  - `s += CUTE(testfunction);`
  - `s += testfunctor();`
- Test in C++

```cpp
void testAnException() {
    std::vector<int> v; // arrange
    try {
        v.at(0); // act
        FAILM("expected out_of_range exception"); // assert
    }
    catch(std::out_of_range &) {} // assert
}
```

- CUTE Version

```cpp
void testAnException() {
    std::vector<int> v;
    ASSERT_THROWS(v.at(0),std::out_of_range);
}
```
Member Functions as Tests in CUTE

CUTE_SMEMFUN(TestClass, memfun)
- instantiates a new object of TestClass and calls memfun on it
  ("simple" member function)

CUTE_MEMFUN(testobject, TestClass, memfun)
- uses pre-instantiated testobject as target for memfun
  - this is kept by reference, take care of its scoping/lifetime
  - allows reuse of testobject for several tests and thus of a fixture
    provided by it.
  - allows for classes with complex constructor parameters

CUTE_CONTEXT_MEMFUN(context, TestClass, memfun)
- keeps a copy of context object and passes it to TestClass'
  constructor before calling memfun on it
  - avoids scoping problems
  - allows single-parameter constructors
TDD Example

- Start with a TEST FIRST !!!

- See Requirements R1…R4 for more details

- Requirement Priorities
  - High (++):
    must be completed to reach minimum usable subset
  - Medium (+):
    useful and should have, but could in principle live without
  - Low :
    optional, nice to have but definitely not essential
RE1 (++): Creation and Output of Strings

Objective
- Allow to create a string with a initial or a default value
- Allow to print its value on the console
- Allow to print the length of the string value

Details:
- String s1();
- String s2("Hello world");
- s1.print() results in ""
- s2.print() results in "Hello world";
- s1.length() == 0;
- s2.length() == 11;
RE2 (+): Common String operations

Objective
- Allow common string manipulations, e.g. toUpper(), toLower(), trim()

Details
- String e("EclipseCon");
- e.toUpper() → ECLIPSECON
- e.toLower() → eclipsecon
- e.trim() → EclipseCon
**Objective**

- Extend with additional important convenience operations

**Details**

- String s1("one"), String s2("twenty");
- s1 = s2; // results in s1 == "twenty"

- String s3 = s2 + s1; // results in ☺ S3 == "twentyone"
RE4 (): Additional operations

Objective

- Support additional convenience operations

Details

- void clear()
- int compare(const MyString& other)
- support for operator \(<, \==, \>\) etc.
- boolean contains(const MyString& other)
- starts/endsWith(const MyString& other)
- char operator[int pos]/char at(int pos)
Overview

Test Driven Development
- Eclipse plugins for TDD: CUTE
- Implementing an example

Static Analysis (SA)
- 3 rules of Scott Meyers “Effective C++ 2nd” (Item 3, 11, 14)
- Tools for SA:
  • Lint, gcc –weffc++
- Eclipse plugins for SA:
  • Codan
  • Linticator
  • Includator
Possible levels of Static Analysis:

**Micro-Level**
- Code, MISRA-C
- e.g: =, ==, { },

**Macro-Level**
- Class-Design, Effective Rules for C++, Java, C#
- e.g: by reference, String concat, Exception-Handling

**Architecture-Level:**
- Layers, Graphs, Subsystems, Components, Interfaces
- e.g: Coupling, Dependency, etc…
Critical areas of C (C Standard)

...are described in Appendix F/ANSI or G/ISO
  – Unspecified behaviour
  – Undefined behaviour
  – Implementation-defined behaviour
  – Locale-specific behaviour

failures can be detected
  – at compilation stage / static
  – at run-time / dynamic
Unspecified behaviour

```cpp
for ( i = 0; i < 100; a[i++] = b[i] )
{
    ...;
}
```

```cpp
a * b + c;
...
(a * b) + c;
...
a * (b + c);

a * (f() + g());
a = i + b[++i];
a = 2 + b[3];  // valid compiler implementation
a = 3 + b[3];  // valid compiler implementation
```
Empirically determined misbehaviour

Errors of omission and addition

```c
int a, b;
...
if ( a = b )
{
    ...
}
...
```

- occurs every 3306 lines in commercial C code

```c
...
a == b;  
- occurs every 12325 lines in commercial C code
...
```

```c
...
if ( a == b );
{
    ...
}
```
Effective C++
Second Edition
50 Specific Ways to Improve Your Programs and Designs
Scott Meyers

Conforms to the new ISO/ANSI C++ standard!
Overview: EC++ 2nd Edition

- Shifting from C to C++ (Item 1 - 4)
- Memory Management (Item 5 - 10)
- Constructors, Destructors, Assignment Operators (Item 11 - 17)
- Classes and Functions: Design and Declaration (Item 18 - 28)
- Classes and Functions: Implementation (Item 29 - 34)
- Inheritance and Object-Oriented Design (Item 35 - 44)
- Miscellany (Item 45 - 50)
Support of Effective C++ in tools: e.g. g++ -WeffC++

-WeffC++ (C++ only)
Warn about violations of the following style guidelines from Scott Meyers’ Effective C++ book:

* Item 11: Define a copy constructor and an assignment operator for classes with dynamically allocated memory.
* Item 12: Prefer initialization to assignment in constructors.
* Item 14: Make destructors virtual in base classes.
* Item 15: Have "operator=" return a reference to *this.
* Item 23: Don’t try to return a reference when you must return an object.

Also warn about violations of the following style guidelines from Scott Meyers’ More Effective C++ book:

* Item 6: Distinguish between prefix and postfix forms of increment and decrement operators.
* Item 7: Never overload "&&", "!!", or ",".

When selecting this option, be aware that the standard library headers do not obey all of these guidelines; use grep -v to filter out those warnings.

-Wno-deprecated (C++ only)
Do not warn about usage of deprecated features.

-Wno-non-template-friend (C++ only)
Disable warnings when non-templatized friend functions are declared within a template. Since the advent of explicit template specification support in C++, if the name of the friend is an unqualified-id (i.e., friend foo<int>), the C++ language specification demands that the friend declare or define an ordinary, nontemplate func-
**Item 3:** Prefer `new/delete` to `malloc/free

Problem with malloc and free

- they don't know about constructors and destructors

```c
string *stringArray1 =
    static_cast<string*>(malloc(10 * sizeof(string)));
```

```c
string *stringArray2 = new string[10];
```

- `stringArray1` point to memory enough for 10 strings
- `stringArray2` point to memory with 10 fully constructed strings

Advantages of `new/delete`

- always calls default ctor / dtor
  - Can also be a disadvantage (then forbid default ctor)
- they are typesafe
Item 3: Prefer **new /delete** to **malloc/free**

Same for deallocation of memory

```cpp
free(stringArray1);
delete [] stringArray2;
```

- free only releases the memory, no dtor is called
- delete[] does what the programmer expects
- NOTE: delete and delete[] are discussed in Item5 (!!!)

Always use matching allocate / deallocate calls:

```cpp
new --> delete
new[] --> delete[]
malloc --> free
```

You are asking for trouble if you violate this rule
Constructor(s), Destructor, and Assignment Oper

Ctor, Dtor, (Cctor), operator=

every class you write will have
  – one or more constructors,
  – a destructor, and
  – an assignment operator

In fact, they already HAVE one if you don‘t define it (Item50)

these are your bread-and-butter functions
it's vital that you get them right
Item 11: cctor & operator= for classes with dny. memory

Example:

// a poorly designed String class
class String {
    public:
        String(const char *value);
        ~String();
        ...
    private:
        char *data;
};
Item 11: cctor & operator= for classes with dny. memory

```cpp
String::String(const char *value)
{
    if (value) {
        data = new char[strlen(value) + 1];
        strcpy(data, value);
    } else {
        data = new char[1];
        *data = '\0';
    }
}

inline String::~String() { delete [] data; }
```
Item 11: `cctor & operator=` for classes with dny. memory

```cpp
String a("Hello");
String b("World");
b = a; //...

- problems during assignment:
  - multiple pointers on the SAME data
  - multiple deletes are called on the SAME data
- there is no client-defined `operator=`
- default assignment operator performs memberwise assignment from the members (just a bitwise copy)

```cpp
void doNothing(String localString) {} 
```

```cpp
String s = "The Truth Is Out There";
doNothing(s); //...
```

- The case of the copy constructor differs a little from that of the assignment operator
Item 11: `cctor & op=` for classes with memory

solution to these kinds of pointer aliasing problems:

- write your own versions of
  - the copy constructor and
  - the assignment operator

if you have any pointers in your class

- Inside those functions, you can either
  - copy the pointed-to data structures, every object has its own copy
  - implement some kind of reference-counting scheme

if you want to **inhibit** assignment or copy of this class

- You *declare* the functions (`private`, as it turns out), but you don't define (i.e., implement) them at all (Item 27)
- Or use boost:non_copyable
**default and delete in C++0x**

```c++
struct NC { // NonCopyable „old style“
    NC() {...};
private:
    NC(const NC&); // no impl!
    NC& operator=(const NC&); // no impl!
};

struct NC { // NonCopyable in C++0x
    NC() = default;
    NC(const NC&) = delete;
    NC& operator=(const NC&) = delete;
};
```
Item 11: `cctor & operator=` for classes with dny. memory

Declare a copy constructor and an assignment operator for classes with dynamically allocated memory (ressources)

Example:

```cpp
// a poorly designed String class
class String {
public:
    String(const char *value);
    ~String();
    ...             // TODO !!! copy ctor AND operator=
private:
    char *data;
};
```
Item 14: have base classes have virtual dtors.

class Target {
public:
    Target() { ++numTargets; }
    Target(const Target&) { ++numTargets; }
    ~Target() { --numTargets; }

    static size_t numberOfTargets() { return numTargets; }
    virtual bool fire();
private:
    static size_t numTargets; // object counter
};

// Target.cpp init static member
size_t Target::numTargets = 0;

class EnemyTank: public Target {
public:
    EnemyTank() { ++numTanks; }
    EnemyTank(const EnemyTank& rhs): Target(rhs) { ++numTanks; }
    ~EnemyTank() { --numTanks; }

    static size_t numberOfTanks() { return numTanks; }
    virtual bool fire();
private:
    static size_t num Tanks; // object counter for tanks
};
**Item 14:** have base classes have virtual dtors.

```cpp
Target *targetPtr = new EnemyTank;
...
delete targetPtr;
```
Item 14: have base classes have virtual dtors.

Target *targetPtr = new EnemyTank;
...
delete targetPtr; //behaviour is undefined if no virtual dtor

- rule: declare a virtual destructor in a class if and only if that class contains at least one virtual function
- Efficiency in C++: declaring all destructors virtual is just as wrong as never declaring them virtual

- Finally, it can be convenient to declare pure virtual destructors in some classes
- one twist, however: you must provide a definition for the pure virtual destructor
Item 14: have base classes have virtual dtors.

- When you
  - try to delete a derived class object
  - through a base class pointer
  - and
  - the base class has a nonvirtual destructor
  - the results are undefined
- To avoid this problem you have only to make the destructor virtual
- If a class does not contain any virtual functions, that is often an indication that it is not meant to be used as a base class
Item 15: Have operator= return *this

- C++ and the creator strived to ensure that user-defined types would mimic the built-in types as closely as possible
- With built-in types, you can chain assignments together
  ```
  int w, x, y, z;
  w = x = y = z = 0;
  ```

- you should be able to chain together assignments for user-defined types, too
  ```
  String w, x, y, z;
  w = x = y = z = "hello";
  ```

  ```
  w = (x = (y = (z = "Hello")));
  ```

  ```
  w.operator=(x.operator=(y.operator=(z.operator=("Hello"))));
  ```
Item 15: Have `operator=` return `*this` 

```
operator=
  – return type of must be acceptable as an input to the function
  – define that return a reference to their left-hand argument, `*this`

String& String::operator=(const String& rhs)
{
  ...
  return *this;                     // return reference
                  // to left-hand object
}
```
Codan == CODe ANalysis

Tool Vendors
- create plugins containing end-user checkers and templates
- integrate command line static analysis tools into CDT

Software Architects, Process Enforcement
- create customized new checkers, based on templates (no programming involved)
- To create problem profiles

Developer, Tester, Code Inspector
- check for errors as you type and have a quick way to fix them
- find bugs, security violations, API violations, coding standard violations during code inspection and before code execution
Codan: Severity + Enablement on Workspace/Project
Codan: Launch Control

Run on demand from context menu

Run with Build

Run as you type
Codan: Problem Markers

```c
int main(void) {
    int a;
    int b;
    if (a == b) return 0;
    b += 1;
    puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
    return EXIT_SUCCESS;
}
```

**Description**

<table>
<thead>
<tr>
<th>Description</th>
<th>Resource</th>
<th>Path</th>
<th>Location</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ Problem</td>
<td>hello.c</td>
<td>/hello/src</td>
<td>line 19</td>
<td>Code</td>
</tr>
<tr>
<td>Code Analysis Problem</td>
<td>hello.c</td>
<td>/hello/src</td>
<td>line 14</td>
<td>Code</td>
</tr>
<tr>
<td>Possible assignment in condition</td>
<td>hello.c</td>
<td>/hello/src</td>
<td>line 19</td>
<td>Code</td>
</tr>
<tr>
<td>Bad function name &quot;Aara1&quot; (pattern /^[a-z]/)</td>
<td>hello.c</td>
<td>/hello/src</td>
<td>line 14</td>
<td>Code</td>
</tr>
<tr>
<td>Catch clause uses reference in declaration of exception</td>
<td>foo.cc</td>
<td>/hello/src</td>
<td>line 26</td>
<td>Code</td>
</tr>
<tr>
<td>Class 'a' has virtual method 'pre' but non-virtual destructor '~a'</td>
<td>foo.cc</td>
<td>/hello/src</td>
<td>line 16</td>
<td>Code</td>
</tr>
<tr>
<td>Statement has no effect</td>
<td>foo.cc</td>
<td>/hello/src</td>
<td>line 21</td>
<td>Code</td>
</tr>
<tr>
<td>Statement has no effect</td>
<td>foo.cc</td>
<td>/hello/src</td>
<td>line 22</td>
<td>Code</td>
</tr>
<tr>
<td>Statement has no effect</td>
<td>hello.c</td>
<td>/hello/src</td>
<td>line 20</td>
<td>Code</td>
</tr>
<tr>
<td>Suggested parenthesis around expression</td>
<td>foo.cc</td>
<td>/hello/src</td>
<td>line 27</td>
<td>Code</td>
</tr>
</tbody>
</table>
Codan: How the write own checkers

Internal Checker
- Problem scope is userdefine (you found e.g. a bug)
- Pick a model to find that problem e.g. AST, Index, ControlFlow-, DataFlow-, Call-Graph
- Extend abstract checker for that model + implement check
- Create Extension for finding
- Create Autofix Action?

External Checker
- Problem scope is defined by external tool
- Integrate output into eclipse concole/problems view (error parser)
- Offer Autofix Actions?
Linticator

Project of IFS in Rapperswil, CH
  – http://www.linticator.ch

Features
  – Autosetup + Project Configuration
  – Problems Overview
  – Message Explanation View
  – Quickfixes
  – Supressions
Linticator: Project Configuration

![Image of Eclipse Project Explorer with Linticator configuration options]

- **New**
- **Go Into**
- **Open in New Window**
- **Copy** (Ctrl+C)
- **Paste** (Ctrl+V)
- **Delete** (Delete)
- **Move...**
- **Rename...** (F2)
- **Build Configurations**
- **Make Targets**
- **Index**
- **Linticator** (Enable Linticator)
- **Properties** (Alt+Enter)
Linticator: Overview

Source Annotations

Message Explanations

Messages Overview
Linticator: Problems View + Message Explanation

Message: 533
function 'Symbol' should (not) return a value (see Location) -- 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.)
Linticator: Quickfix

```cpp
class Faulty {
    public:
    Faulty();
    virtual ~Faulty();

    void shouldBeVirtual() = 0;
```

1093: A pure specifier was given for function 'Faulty::shouldBeVirtual(void)' which was not virtual

2 quick fixes available:
- Declare function virtual
- Ignore message 1093 at this location
Linticator: Quickfix

```cpp
class Faulty {
    public:
        Faulty();
        virtual ~Faulty();

        virtual void shouldBeVirtual() = 0;
};
```
Linticator: Suppress Message

```c
int f() { return 42; }

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

Options to ignore specific messages can be configured in the Linticator dialog. This allows for selective suppression of warnings or errors, improving the developer's experience by reducing clutter in the IDE.
Includator

Project of IFS in Rapperswil, CH

- http://www.includator.ch

Features

- Find unused includes
- Directly include referenced files
- Organize includes
- Static code coverage
- Find unused files
Includator: Find unused includes

```c++
#include "A.h"
#include "B.h"
#include "C.h"

int main() {
    A a;
    B b;
    C c;
    return 0;
}
```
Includator: Directly include referenced files

This feature helps to automatically add include directives to a file under consideration, so that all files containing referenced declarations get included (directly). The features is based on the idea of John Lakos found in his book "Large-Scale C++ Software Design".

Example

```cpp
/* main.cpp */

#include "Y.h"

int main() {
  X x;
  return 0;
}
```

```cpp
/* Y.h */

#include "X.h"

/* more code */
```

```cpp
/* X.h */

class X { };

/* ... */
```

Here, the **Includator** makes the proposal to include file X.h directly into main.cpp independent of other, used or unused, types in Y.h.
Includator: Organize includes

This feature is similar to the one known form Eclipse JDT called Organize Imports. Its task is to find includes that should be added and/or includes that can be removed from a given file.

Example

```cpp
/* main.cpp */

1  #include "Y.h"
2  #include "Z.h"

3  int main() {
4    Y y;
5    X x;
6    return 0;
7  }

/* X.h */

1  class X { };  

/* Y.h */

1  class Y { };  

/* Z.h */

1  class Z { };  
```

Here, the Includator makes the proposal to to include file X.h and to remove the include of Z.h.
Includator: Static code coverage
Finding unused files means to look at all the include dependencies in a given C++ project and find header files which are not included at all. This situation can often arise after unused includes directives have been removed with the **Includator's find unused includes or organize includes** features.

**Example**

**Consider the following project structure:**

- **project:**
  - **main.cpp**
  - **X.h**
  - **Y.h**
  - **Z.h**

```cpp
//main.cpp

#include "X.h"
#include "Y.h"

int main() {
  X x;
  Y y;
  return 0;
}
```

Finding unused includes in the context of this project means to propose the deletion of file **Z.h**.
More information

Eclipse CDT: http://eclipse.org/cdt
Linux Tools Project: http://www.eclipse.org/linuxtools
Wascana: http://code.google.com/a/eclipselabs.org/p/wascana
CUTE: http://www.cute-test.com/
Linticator: http://www.linticator.ch
Includulator: http://includulator.ch/
Sconsolidator: http://www.sconsolidator.ch/
Conclusion

We hope you have enjoyed seeing some of the breadth and power of a few Eclipse C/C++ tools. All communities of developers writing these tools are active and always interested in feedback. Any level of participation is greatly appreciated and can be as easy as filing a bug, tweeting about a cool feature, or writing a blog post about how you set things up for your project.

Thank you.