Hands On With the C/C++ IDE

Andrew Overholt, Bernhard Merkle
Hands On With The C/C++ IDE

In this tutorial, attendees will be led through focused 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
- Fedora 14 pre-installed with Eclipse CDT Linux Tools
- Available for
  - Oracle Virtual Box
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
Tutorial Setup 101

Image: Oracle VM VirtualBox Manager interface with options for Ubuntu and Fedora virtual machines.
Tutorial Setup 101
Glossary and architecture

- Linux Tools
- Project
- Other C/C++ plugins
- C/C++ Development Tooling (CDT)
- Eclipse Platform
- Native toolchain
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
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
Structure of a typical Unit Testing Framework
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
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
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();
```cpp
#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();
}
```
#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));
...
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
#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();
}
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
Objective

– Allow to create a string with an 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() $\rightarrow$ ECLIPSECON
  - e.toLower() $\rightarrow$ eclipsecon
  - e.trim() $\rightarrow$ 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

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

```c
a * b + c;
...
(a * b) + c;
...
```
```c
a * (b + c);
```
```c
a * (f() + g());
```
```c
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++

```
-W effc++ (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-
```
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:

```cpp
// a poorly designed String class
class String {
public:
    String(const char *value);
    ~String();
    ...
    // no copy ctor or operator=
private:
    char *data;
};
```
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; }
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)

void doNothing(String localString) {}

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

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

Example:

// 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 numTanks; // 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`

```cpp
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 ANalysys

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;
}
```

1 error, 13 warnings, 0 others

- **Possible assignment in condition**
  - File: `hello.c`
  - Line: 19
- **Bad function name “Aara1” (pattern /^[a-z]$/)**
  - File: `hello.c`
  - Line: 14
- **Catch clause uses reference in declaration of exception**
  - File: `foo.cc`
  - Line: 26
- **Class ‘a’ has virtual method ‘pre’ but non-virtual destructor ‘~a’**
  - File: `foo.cc`
  - Line: 16
- **Statement has no effect**
  - File: `foo.cc`
  - Line: 21
- **Statement has no effect**
  - File: `foo.cc`
  - Line: 22
- **Statement has no effect**
  - File: `hello.c`
  - Line: 20
- **Suggested parenthesis around expression**
  - File: `foo.cc`
  - Line: 27
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 console/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
  – Suppressions
Linticator: Project Configuration

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

- New
- Go Into
- Open in New Window
- Copy
- Paste
- Delete
- Move
- Rename
- Build Configurations
- Make Targets
- Index
- Linticator
- Enable Linticator
- Properties
- Alt+Enter
Linticator: Overview

Source Annotations

Message Explanations

Messages Overview
Lindicatior: Problems View + Message Explanation

![Screenshot of Lindicatior showing problems view and message explanation.](image)

**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
Linticator: Suppress Message

```c
int f() { return 42; }
int main() {
    f();
    return 0;
}
```

Options to suppress messages:
- Ignore message "Ignoring return value" for this function
- Ignore message 522 at this location
- Ignore message 534 at this location

Inhibit Lint Messages...

Inhibit Messages
Configure the inhibition options for the messages.

<table>
<thead>
<tr>
<th>Message</th>
<th>Global</th>
<th>File</th>
<th>Call</th>
<th>Func</th>
<th>Sym</th>
</tr>
</thead>
<tbody>
<tr>
<td>522: Highest operation, function 'f', lacks side-effects</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>534: Ignoring return value of function 'f(void)'</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Run Linticator after configuring inhibition options.
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
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 feature is based on the idea of John Lakos found in his book Large-Scale C++ Software Design (5th guideline)

Example

```
/* main.cpp */

#include "Y.h"

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

```
/* Y.h */

#include "X.h"

/* more code */
```

```
/* 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 */

#include "Y.h"
#include "Z.h"

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

```cpp
/* X.h */
class X { };
```

```cpp
/* Y.h */
class Y { };
```

```cpp
/* Z.h */
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
Includator: Find unused files

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

```
//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

CUTE:  http://www.cute-test.com/
Linticator:  http://www.linticator.ch
Includator:  http://includator.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.