

EE101 (2013/2014)

C Programming & Software Engineering

Lecture 11: Other Issues in the Software Development

Dr. Rui Lin/Dr. Mark Leach

Email: Rui.Lin/Mark.Leach@xjtlu.edu.cn

[Room EE512/EE510](#)

This Lecture

- Debugging Your Codes

----Issues associated with a large team/project:

- “Make” Your Files
- Version Control
- Software Testing

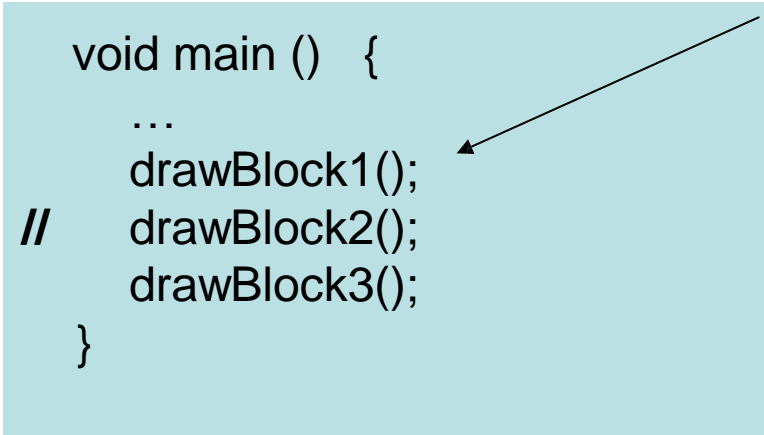
Debugging Techniques

- First **sort out** the compilation errors; compile and run the program.
- Getting a incorrect result or an exception usually means you have to **debug** your program...
- **The old way:**
 - Use **pencil and paper**, and play the role of the computer, executing each instruction line-by-line
 - This can be very effective, ... but also very slow!
- **The new way:**
 - Use a **debugging tool**, integrated development platform (IDP), to step through the source code as the program executes or to stop at the exception line
 - Visual Studio for example has some debugging capabilities
- **Either way:**
 - Always keep pencil and paper handy!

Debugging Techniques – Commenting-Out Code

- Sometimes, you know there's a bug but its hard to locate it !
- Suppose your program fails with an “exception” aborting your program.
- **Commenting-out** code (literally changing statements into comment lines) can sometimes help locate bugs:

```
void main () {  
    ...  
    drawBlock1();  
//    drawBlock2();  
    drawBlock3();  
}
```



- Change each line in turn into a comment line
- If error goes away, the fault *probably* lies in that function
- Can also use `/*...*/` to comment-out big sections of code

Debugging Techniques – Conditional Print Statements

- One Boolean variable can control multiple print statements
- ...Just need to change one statement and re-compile

```
int debug = 0; // change to true for debugging output
if (debug) printf ("Now at step A... x = %d \n" , x);
// lots of code goes in here...
if (debug) printf ("Now at step B... y = %d \n" , y);
// lots more code goes in here... etc.
```

Version Control, makefile and Testing

The topics on the next a few slides, version control, makefile and testing, will be introduced very briefly from an industrial point of view. In other words, you are not expected to master them (and it is not possible at this moment.). So, please do not be panic, if you do not understand them.

The purpose of adding these slides here is just to let you be aware of these issues. We just want you to know what's going on in the real world. It may be helpful when it comes to the moment that you are interviewed by a potential employer.

“Make” Your Files----(1/2)

- What are “make” and “makefile”?

“make”, which is initially developed on Unix but available everywhere in some form, is a program which manages all the individual files in a project by following the instructions in a text file called **makefile**.

- Why do we need **make**?

Assuming being a member of a team of hundreds of programmers and each of them has many files, you have to type all the file names you need in order to compile the whole project for each time you modify your code. This means you need to type hundreds of file names each time. So, what you need is a tool which handles this automatically. “**make**” is exactly what you need.

- How does **make** work?

When you edit some of the files in a project and then type **make**, the **make** program follows the instructions in the **makefile** to compare the dates on the source code files and to the dates on the corresponding target files. The compiler will be invoked if a source file date is more recent than that of target file. The **makefile** contains all the commands to put your project together. Learning to use **make** saves you a lot of time and frustration.

“Make” Your Files----(2/2)

- An simple example of using `make` and `makefile`

```
# A comment  
hello.exe: hello.cpp  
    mycompiler hello.cpp
```

- What does this say?

This says that **hello.exe** (the target) depends on **hello.c**. When **hello.c** has a newer date than **hello.exe**, `make` executes the “rule” **mycompiler hello.cpp**. There may be multiple dependencies and multiple rules. Many `make` programs require that all the rules begin with a tab.

Version Control ---(1/3)

- Why do we need [Version/Revision Control](#)?

Now, again imagine you are a member of a software development team of more than 100 engineers, you are assigned a few modules of a project and working parallelly with other colleagues. Although using the concept of ADT reduces the coupling between all the modules, all these modules still have impacts on each other because they are essentially different parts of the same project. This means it is possible that, after you started to work on a certain problem, the problem may disappear or be different because of the modifications done by someone else in another module.

- So, we may wonder whether we have any tools out there to help us?

Yes, version control software. ---example: two major software tools are: [Concurrent Version System \(CVS\)](#) and [SubVerSion \(SVN\)](#).

Many version control systems offer the following functionalities:

- File locking
- Version merging
- Baselines, labels and tags

Version Control --- (2/3)

- How does a centralized version control software package work?
All the software are stored at a server called software repository. The computers of all the programmers are clients respected to the server. Each client computer reads the files from the repository (working copy) and after completing the coding, they have to deposit their code back to repository. This process is called “commit”. Some of version control systems use lock-modify-unlock (obsoleted) mode and some systems, for example SVN, use copy-modify-merge mode to keep the entire project to be built properly at any moment.
- What is the relationship between version control and make?
Normally, **make** is integrated into version control system.
- Distributed version control
Distributed revision control systems (DRCS) take a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository. Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer

Version Control --- (3/3)

The following are two examples about what you do and see when you are using SVN (centralized) as your version control software.

- Have a local working copy using command **checkout**

```
$ svn checkout http://svn.example.com/svn/repo/trunk my-working-copy
```

```
A my-working-copy/README
```

```
A my-working-copy/INSTALL
```

```
A my-working-copy/src/main.c
```

```
A my-working-copy/src/header.h
```

```
...
```

```
Checked out revision 8810.
```

```
$
```

- Update your local working copy using command **update**

```
$ svn update
```

```
Updating '.':
```

```
U foo.c
```

```
U bar.c
```

```
Updated to revision 2.
```

```
$
```

Software Testing---(1/4)

- Overview of software testing
 - Testing is a very important part of any product development process to assure the quality of the product. Testing, however, cannot identify all the defects within the product (software in our case). Instead, it compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. Note that not only can functional requirements but also nonfunctional requirements such as [testability](#), [scalability](#), [maintainability](#), [usability](#), [performance](#), and [security](#) often be the source of defects.
 - The scope of software testing often includes examination of code and execution of that code in various environments and conditions as well as examining the aspects of code: [does it do what it is supposed to do and do what it needs to do](#).
 - It is also a dynamic process in the sense that it requires a product can be executed with given inputs and produce observable outputs to reveal the existence of the faults in the product under testing. Once a fault is detected, the debugging activities take place and once a fault is fixed, the testing process resumes. Testing may even continue after the product is released to the market.
 - The testing methods can be roughly divide into
 - Static testing: reviews, walkthroughs, or inspections.
 - Dynamic testing: executing programmed code with a given set of test cases.Together, they help improve software quality.

Software Testing---(2/4)

Methods: Testing traditionally take a box approach:

- White box
This tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing, an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. Therefore, white box testing checks **whether or not the software does what it is supposed to do**.
- Black box
This treats the software as a "black box", examining functionality without any knowledge of internal implementation. Therefore, black box testing checks **whether or not the software does what it needs to do**.
- Gray box
This involves having knowledge of internal data structures and algorithms for purposes of designing tests, while executing those tests at the user, or black-box level. You may think of this as a mixture of white and black box testing.

Software Testing---(3/4)

Some of the techniques involved are:

- White box
 - API testing (application programming interface) – testing of the application using public and private APIs
 - Code coverage – creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
 - Fault injection methods – intentionally introducing faults to gauge the efficacy of testing strategies
 - Mutation testing methods
- Black box
 - equivalence partitioning
 - boundary value analysis
 - all-pairs testing
 - state transition tables
 - decision table testing,
 - fuzz testing
 - model-based testing
 - use case testing
 - exploratory testing
 - specification-based testing.

Software Testing-(4/4)

- Testing levels:

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. These levels are

- unit testing,
- integration testing,
- system testing,
- acceptance testing.

- Testing process

- Traditional waterfall development model
- Agile or Extreme development model
- Top-down and bottom-up

- A typical testing cycle

- Requirements analysis-> Test planning-> Test development-> Test execution
-> Test reporting-> Test result analysis-> Defect Retesting-> Regression testing
-> Test Closure

- Automated testing and Manual testing

Questions?