EEE101: C Programming & Software Engineering I

Lecture 11: Software Development Issues

Dr. Rui Lin/Dr. Mark Leach
Office: EE512/EE510
Email: rui.lin/mark.leach@xjtlu.edu.cn
Dept. of EEE XJTLU

Outline of Today’s Lecture (10)

- Programming Issues:
  - Debugging your code
  - “Make” file
  - Version Control
  - Software Testing

Debugging your code

Firstly compilation errors: these must be located and corrected before you can run your program.

Incorrect results/outputs or exceptions: these mean there is a problem with your algorithm design and so the debugging begins:

The old way

- Use a pencil and paper, execute (calculate the result of each instruction, line by line.
- Slow…but effective (a good test of your programming ability)

The new way

- Use a software tool (e.g. Visual Studio) to step through the code line by line until the error is seen or the exception happens

Keep a pencil and paper handy either way ☺

Debugging Techniques (1/2)

Commenting out code

Sometimes you know there is a bug, but it's hard to find.

Suppose your program fails with an exception and aborts.

Commenting-out single lines of code (changing them to comments) can sometimes help locate bugs

```
int main(){
          drawBlock1();
/*        drawBlock2(); */
          drawBlock3();
          }
```

Change each line into a comment one by one and run

If the error goes away, the fault probably lies in that function.

Debugging Techniques (2/2)

Insert print statements controlled by a variable

```
int main(){
        int debug = 0;
        if(debug) printf("Now at step a");
        /*lots of code*/
        if(debug) printf("Now at step b");
        /*lots of code*/
        if(debug) printf("Now at step c");
        /*lots of code*/
        }
```

Changing the variable debug to 1 will print al of the statements.

Can determine where a programme crashes, or print values at specific points.

Software Debugging Tools

Most compilers provide common debugging tools including:

Single stepping

Allows your program to be evaluated line by line. Coupled with an output or watch window, you can choose to see various variable values as statements are executed

Breakpoints

Allow a program to be executed to a particular point and then single stepped from that point

makefile, Version Control & Testing

The rest of todays topics are introduced from an industrial perspective. The purpose of delivering this information is to make you aware that these are issues and so are of interest.

“Make” Your Files (1/2)

- **What are “make” and “makefile”?**

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

- **Why do we need make?**

To compile a project consisting of multiple files, each time you modify your code you need to type all of the file names each time. **“make”** is a tool which handles this automatically.

- **How does make work?**

After editing one or many files in a project, execute **make**. The **make** program follows instructions in the **makefile**, comparing dates on the source code files to the dates on the corresponding target files. The compiler is invoked if a source file date is more recent than that of the 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 (1/2)

- **A simple example of using make and makefile**

```
#a comment
hello.exe: hello.c
	mycompiler hello.c
```

- **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.c.
  There may be multiple dependencies and multiple rules. Many make programs require that all the rules begin with a tab.

Version Control (1/3)

Imagine you are working in a software development team of more than 100 engineers for a new Operating System. With each engineer working on specific modules that must work together. The concept of ADT's is being applied but all modules will impact on all other modules when integrated.

It is possible while you are working that a problem you need to solve may disappear or be different because of other peoples work on other modules.

Version control software helps to monitor changes. Common functionality of version control include:

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

Version Control (2/3)

- **How does a centralized version control software package work? All software is stored at a server called the software repository and all programmers are clients on the server. Each client computer copies the files from the repository (working copy) and after completing coding, their work is deposited 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 compilable 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)

Testing is a very important part of any product development process to assure the quality of the product. Testing, however, cannot identify all defects within the product (software in our case). Instead, it compares the state and behaviour of the product against **oracles**—principles or mechanisms by which someone might recognize a problem. Note that not only can functional requirements but also non-functional 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: does it do what it's supposed to do and do what it needs to do.

It's 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, debugging activities take place and the fault fixed, the testing process then resumes. Testing often continues after product release on the market.

Software Testing (2/4)

Testing methods are roughly divided into

- Static testing: reviews, walkthroughs, or inspections.
- Dynamic testing: executing programmed code with a given set of test cases.

Testing traditionally takes a box approach:

White box testing: tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. An internal view 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 determines appropriate outputs. White box testing checks whether or not the software does what it is supposed to do.

Black box testing: treats the software as a "black box", examining functionality without any knowledge of internal implementation. black box testing checks whether or not the software does what it needs to do.

Grey box testing: involves having knowledge of internal data structures and algorithms for purposes of designing tests, for execution of those tests is at the user, or black-box level. This as a mixture of white and black box testing.

Software Testing (3/4)

Some of the techniques involved are:

- **White box**
  - API (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?

The End…

but the labs are still going ☺