Program Architecture

CS 2212 INTERMEDIATE PROGRAMMING/C AND C++

Course Facts and Mechanics
C Programming style.

Objectives

- Specifications
- Test plan
- Starting a new program
- Coding style

Test Plan

Before you write even one line of code, construct a test plan. This is a list of inputs with the expected output or error comment that should be given for each. The inputs on this list must cover the

following categories:

- Easy to process; you can figure out the answer in your head.
- The minimal and maximal acceptable inputs, if those are defined.
- Error inputs (too small, too large). Each error outcome should be tested independently.
- Typical inputs for a real situation. This is used to get your output formatting right.

The Top-Down Idea

Now you are ready to start designing a solution. Here is the general procedure, which will be expanded below.

- Start a new project folder and a new main program file.
- In it, write down the obvious parts of the program.
- Inside main, write comments for each major phase of the program.
- If a program phase is brief (less than 3 lines), write the code.
- As you work on main, write declarations for any variables you need.
- Invent and name a function for anything that requires more than a couple of lines of work.

The Top-Down Idea, continued

A function stub is a copy of the function's prototype, with nothing in the body except a trace comment and a return statement, if necessary.

- After the end of main, write a stub for every function you have named.
- Compile and debug your stubby program.
- Choose the easiest function. Write a real definition for it, with appropriate local variable declarations, and debug again.
- Now or later, you may find that you guessed wrong about an initial function prototype. You may find you need more parameters. If so, correct the prototype and the function's top line.
- Continue writing one function at a time and debug it using your easy test cases until the program is done.

The Top-Down Idea, finished

- Pay attention to the program heading and your input prompts. Are they clear? Fix any problems.
- Look at your output. Is it neat and readable? Fix any problems you see.
- Perform a full test on the finished program.
- Use the mouse to capture the output and paste it into a file.
- Make sure the output is in the same folder as the program.
- Put your code through your IDE's re-indent process. Make sure the layout looks professional. (Eclipse, use CTRL-A, CTRL_I)

Specifications

To write a program, start with a specification that is as precise as possible. Lists the information and requirements that your client (or teacher) has set out. The parts of a specification are:

- **Goal:** a brief statement in English about the general job you are trying to do.
- **Inputs:** what data will the user supply? What types? Are there minimum and maximum allowable values?
- **Outputs:** What answers must you compute, and how should they be displayed? How many decimal places are needed?
- **Constants and formulas:** mathematical equations needed to derive the output from the input, and constant values used in those formulas, if any.
- **Other:** Any other requirements the client provides.

Black Box Testing

Views the system as a black box – you can only see the input and output, not the code itself. This is what repl.it does for us, you can see that your output is correct, but it doesn't prove your program was done correctly.

Black Box Testing:

- Given a specific input, what is the output?
- Tests the actions and functions of the system and makes sure they work.
- This category can also include regression testing among other things.
- Black Box testing should also include non-functional testing – stress testing.

White/Glass Box aka Functional Testing

Test Plan is designed to test each line of the code and all options. Write code to test each line of the program and all choices in a selection statement or loop.

- Design tests to make sure every part of the program is exercised.
- All paths are taken.
- All branches are examined.
- All loops are executed at their boundaries and within their operational bounds. Internal data validity is tested.
- Test the borders of all variables and limits
- Completely test every part of the program logic.

Difference between White Box Testing and Black Box Testing

Black Box Testing	**White Box Testing**
It is a testing method without having knowledge about the actual code or internal structure of the application	It is a testing method having knowledge about the actual code and internal structure of the application
It is a testing method having knowledge about the actual code and internal structure of the application	This type of testing is performed at a lower level of testing such as Unit Testing, Integration Testing
It concentrates on the functionality of the system under test	It concentrates on the actual code – program and its syntax's
Black box testing requires Requirement specification to test	White Box testing requires Design documents with data flow diagrams, flowcharts etc.
Black box testing is done by the testers	White box testing is done by Developers or testers with programming knowledge.

Security: Fuzz Testing or Fuzzing

Fuzzing or **fuzz testing** is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. Typically, fuzzers are used to test programs that take structured inputs. This structure is specified, e.g., in a file format or protocol and distinguishes valid from invalid input. **An effective fuzzer generates semi-valid inputs that are "valid enough" in that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program and are "invalid enough" to expose corner cases that have not been properly dealt with.**

For the **purpose of security**, **input that crosses a trust boundary is often the most interesting.** For example, it is **more important to fuzz code that handles the upload of a file by any user** than it is to fuzz the code that parses a configuration file that is accessible only to a privileged user.

Test Plan: Start with Black Box Test Cases

Before you write even one line of code, start to construct your test plan. This is a **list of inputs with the expected output or error comment** that should be given for each. The inputs on this list must cover the following categories:

- Easy to process; you can figure out the answer in your head.
- The minimal and maximal acceptable inputs, if those are defined.
- Error inputs (too small, too large). Each error outcome should be tested independently.
- Typical inputs for a real situation. This is used to get your output formatting right.

Test Plan: Add Structural Test Cases

Add test cases to the Black Box test plan to exercise every line of code.

- Test all borders of loops.
- Test every branch, path and section of code.
- Try to exercise every line of code and every option as much as possible.
- Valid and invalid input

Test Plan: Add Security Test Cases

Add test cases to the test plan to look for security vulnerabilities that can be exploited.

- Look for crashes and memory leaks
- Incorrectly handled corner cases in inputs or events
- Look for input that crosses the trust barrier
  - Uploads by any user
- Fuzzers can be Black/White or Grey Box

The Top-Down Design

Now you are ready to start designing a solution. Here is the general procedure, which will be expanded below.

- Start a new project folder and a new main program file.
- In it, write down the obvious parts of the program.
- Inside main, write comments for each major phase of the program.
- If a program phase is brief (less than 3 lines), write the code.
- As you work on main, write declarations for any variables you need.
- Invent and name a function for anything that requires more than a couple of lines of work.

The Top-Down Design, continued

A **function stub** is a copy of the function's prototype, with nothing in the body except a print statement “You reached X function” and a return statement, if necessary.

- After the end of main, **write a stub** for every function you have named.
- Compile and **debug** your stubby program – with incomplete functions.
- Choose the **easiest function**. Write a real definition for it, with appropriate local variable declarations, and **debug again**.
- Now or later, you may find that you guessed wrong about an **initial function prototype**. You may find you need more parameters. If so, **fix the prototype** and the **function's top line**.
- Continue writing one function at a time and debug it using your easy test cases until the program is done.

The Top-Down Design, finished

- Now pay attention to the program heading and your input prompts. Are they clear? Fix any problems.
- Look at your output. Is it neat and readable? Fix any problems you see.
- Perform a full test on the finished program.
- Use the mouse to capture the output and paste it into a file.
- Make sure the output is in the same folder as the program.
- Put your code through your IDE's re-indent process. Make sure the layout looks professional.

Questions?