Software Development 3: Incremental Example

1. Introduction
2. Incremental Software Development
3. **Incremental Development Example**
   - **top-down example, withs stubs**
   - **bottom-up example, with drivers**
4. Software Bugs
5. Testing
6. Debugging

Top-Down Example, with Stubs

To understand **top-down development** and **stub code**, we will consider the `firstline.c` example handout program, which is to print out the first line in a file given as a command line argument.

In thinking about how this program is to work, we might come up with the following logic:

1. check program arguments
2. open file (for reading)
3. read first line from file
4. print line out (to standard output)

Let us now look at a sequence of passes at this program, which start off being extremely abstract, but eventually become a fully working program.

(Note that we will not explicitly show header includes, etc.)

Top-Down Example, Step 1

As an extremely simple first pass, we might use **stub statements**, which simply print out what should happen:

```
int main(int argc, char *argv[])
{
  printf("Testing arguments.\n");

  printf("Opening file.\n");

  printf("Getting first line of file.\n");

  printf("Printing first line of file.\n");

  return EXIT_SUCCESS;
}
```

We can now compile this file and test it.

Top-Down Example, Step 2

Next, let's add the code in to actually test arguments:

```
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "Usage: firstline FILE\n");
    return EXIT_FAILURE;
  }

  printf("Opening file.\n");

  printf("Getting first line of file.\n");

  printf("Printing first line of file.\n");

  return EXIT_SUCCESS;
}
```

Again, compile and test by calling with different numbers of "file" arguments to be sure it works.

Top-Down Example, Step 3

Next, let's add the code to open the argument file:

```
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "Usage: firstline FILE\n");
    return EXIT_FAILURE;
  }

  FILE *fpntr;
  if ((fpntr = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "Error opening file %s: %s\n", file_pathname, strerror(errno));
    return EXIT_FAILURE;
  }

  printf("Getting first line of file.\n");

  printf("Printing first line of file.\n");

  return EXIT_SUCCESS;
}
```

Again, compile and test; we may not be able to be completely certain code is correct, but by testing with valid and invalid file arguments, we should get a good idea if it is right.

Top-Down Example, Step 4a

Next we need to work on reading the first line from the file.

For this, we want to call a *function* rather than **inlining** code.

As a first step, we will write a simple **stub function**, which has the required syntax (so it can be called from `main`), but which simply *simulates* reading the line.

The desired *prototype* for the function is:

```
char *get_line(FILE *fpntr)
```

I.e, we pass the `FILE*` handle for the open file and get back the first line as a valid C string.

Top-Down Example, Step 4b

For now, we will write a function that does nothing but print out that we reached it, and return a fixed *test string*.

This will allow us to test the interface between `main` and `get_line()`:

```
char *get_line(FILE *fpntr)
{
  printf("In function get_line()\n");

  return "Test line from get_line()\n";
}
```

Top-Down Example, Step 4c

We revise `main` to call this function and print the returned string:

```
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "Usage: firstline FILE\n");
    return EXIT_FAILURE;
  }

  FILE *fpntr;
  if ((fpntr = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "Error opening file %s: %s\n", file_pathname, strerror(errno));
    return EXIT_FAILURE;
  }

  char *first_line
  if ((first_line = get_line(fpntr)) == NULL) {
    perror("Error reading line");
    exit(EXIT_FAILURE);
  }

  printf("First line in file %s:\n%s", file_pathname, first_line);

  return EXIT_SUCCESS;
}
```

Top-Down Example, Step 5a

At this point, all that remains to complete `firstline.c` is to replace the `get_line()` *stub* with code that actually reads the first line in the file.

Suppose we modify `get_line()` to be:

```
char *get_line(FILE *fpntr)
{
  char line_buff[MAX_LINE_LENGTH];

  if (fgets(line_buff, MAX_LINE_LENGTH, fpntr) != NULL)
    return line_buff;
  else
    return NULL;
}
```

What happens if you now compile your program (with `-Wall`)?

Top-Down Example, Step 5b

What happens is that you get a *warning* like:

```
warning: function returns address of local variable
     return line_buff;
     ^
```

Since this is just a warning, you can run the code, but what may well happen is that your line is "messed up" when printed out.

As the compiler tells you, returning a pointer to a local variable results in your program containing a **logic error**, since the local variable memory will be reused after return from the function.

While the compiler tells you that the problem is within your newly refined function, because that is all you had changed, this should have been obvious to you.

Top-Down Example, Step 5c

There are two ways to fix this problem:

1. declare `line_buff` to be `static`
2. switch to using **dynamic memory** (`malloc()`)

We will use the `static` approach for simplicity:

```
char *get_line(FILE *fpntr)
{
  static char line_buff[MAX_LINE_LENGTH];

  if (fgets(line_buff, MAX_LINE_LENGTH, fpntr) != NULL)
    return line_buff;
  else
    return NULL;
}
```

The code should now compile and run properly on all possible test cases—we are done!

Top-Down Example, Notes

It is critical that you understand that you *must* do *adequate* **testing** *at each step* in the top-down refinement process.

Your goal is to be as certain as possible that your current code is correct before proceeding to further refine it.

If (undetected) *bugs* are left at any stage, this will make the debugging job much harder at later stages.

(What it means to do adequate **testing** will be discussed later.)

Top-Down Example, Notes (contd.)

Another critical point to note is that we have included **error checking** code as we go.

Students sometimes want to wait until they "have the program working" before adding this code; they view having to write it as somewhat of an annoyance.

Please resist that impulse!

Error checking code can help you catch bugs in your program *during development*.

Also, it is much easier to test the error checking code as part of your incremental development process.

Including appropriate error checking code is a critical element of building **reliable software**.

Bottom-Up Example with Drivers

To understand **bottom-up development** and **driver code**, we will again consider the `firstline.c` example program.

This is a very simple program, with only a single level of function call, so bottom-up development is not too useful.

Nonetheless, we can demonstrate its basic ideas.

In bottom-up development, we start by implementing the lowest abstraction level functions, and work up to the top-level `main`.

Here, we have only a single function, `get_line()`, so we start there.

Bottom-Up Example, Step 1

Let us implement a working `get_line()` as we first did earlier:

```
char *get_line(FILE *fpntr)
{
  char line_buff[MAX_LINE_LENGTH];  #intentional bug!

  if (fgets(line_buff, MAX_LINE_LENGTH, fpntr) != NULL)
    return line_buff;
  else
    return NULL;
}
```

While we have implemented `get_line()`, we cannot be certain it is correct because we have not yet tested it.

However, since we haven't implemented our `main` for firstline.c, we do not have any way to run `get_line()`.

(If we had an *interactive environment*, we could simply call it!)

Bottom-Up Example, Step 2a

What we do is write a **driver**: a simple `main` that sets up data needed to call and test `get_line()`, and then calls it.

`get_line()` requires that we pass it an open file handle (a `FILE*`).

The goal is to fully test our function to try make to make certain it is working perfectly.

The only way to do this will be to call it on a variety of test files.

This can be done in two ways:

- have the driver take a filename as a command-line argument and use scripts to run through the set of test files
- build the set of test files into the driver (have it successively open a fixed set of files and call `get_line()`)

Bottom-Up Example, Step 2b

We will demonstrate the first of these approaches:

```
int main(int argc, char *argv[])
{
  if (argc != 2) {
    fprintf(stderr, "Usage: test-getline FILE\n");
    return EXIT_FAILURE;
  }

  FILE *fpntr = fopen(argv[1], "r");
  char *first_line = get_line(fpntr);

  if (first_line != NULL)
    printf("%s", first_line);
  else
    printf("get_line() returned NULL\n");

  return EXIT_SUCCESS;
}
```

Bottom-Up Example, Step 2c

Points about this driver:

- it does not include *error checking* code for the `fopen()` call because we are testing `get_line()` not this driver, plus we control how the driver is called (so always call it properly)
- `get_line()` can return two possible results:
  (1) a line (string) or (2) `NULL`
- we must call the driver with test files that result in both types of returns
- printing the returned line without any additional text makes it easier to automate testing

Bottom-Up Example, Step 3a

At this point, we have a driver for `get_line()`, so must develop a set of test files to feed to the driver.

The test files must be adequate to validate that `get_line()` works under all conditions.

Adequate testing will require we test:

- both types of possible returns from `get_line()`
- different line characteristics (length, blank, EOF-terminated)
- different file characteristics (empty, single line, multiple)
- that a valid C string is always returned

(What it means to do adequate **testing** will be discussed later.)

Bottom-Up Example, Step 3b

Testing should reveal the problem with returning a pointer to a local variable that we discussed earlier.

One that is fixed and all tests rerun successfully, we will have validated that `get_line()` works as it is supposed to.

In a system consisting of multiple functions, we would follow the same procedure for all the lowest level functions.

After the lowest-level functions had been validated, we would shift our focus up to the functions that call these lowest level functions, and so forth.

If the lowest-level functions were properly validated/tested, bugs should be confined to the new next-higher-level functions.

Bottom-Up Example, Step 4

Eventually, we will work our way up to writing `main`.

When working bottom-up, it is best to limit the amount of code that must go into `main`; it should consist primarily of calls to our (already validated) functions.

If these functions have been properly tested, they should be correct.

Errors that occur when testing `main` should then be due to bugs in the code for `main` only.

At this point we are done with drivers, we simply run the overall **test suite** for the complete program.