Software Development 1: Introduction

1. **Introduction**
   - **software development and software engineering**
   - **software requirements and software design**
   - **abstraction**
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. Testing
6. Debugging

Software Development

Software development refers to the process of creating a *correctly functioning* **software system**.

A *software system* consists of one or more **computer programs** along with needed configuration/data files, documentation, etc.

Development obviously involves **implementing** the program(s) in suitable **programming language(s)**.

However, the development process also generally involves other important elements as well:

- **requirements specification**
- **design**
- **testing**
- **documentation**

Software Engineering

Software engineering is the area of computer science that deals with formal *methodologies* for *software systems development*.

A fairly large number of *SE methodologies* are in use, including:

- waterfall model
- test-driven development
- agile development
- extreme programming
- rapid application development (RAD)
- scrum development
- lean development
- dynamic systems development method (DSDM)

Most of these methodologies were designed for the development of *large-scale* software systems, by **teams** of developers.

Incremental Software Development

An important characteristic that many SE methodologies share is that the software system is developed **incrementally**–i.e., in **stages**/**phases**.

In some, development proceeds from highly *abstract* versions of the entire system to increasingly more *concrete* versions.

In others, development proceeds by incrementally adding working components.

Incremental development may be part of an **iterative process**, where the implementation and even design are successively refined.

Incremental development will be a key topic in these lectures!

Software Requirements Specification

For most applications, it makes little sense to start building a software system until one understands precisely how it is supposed to function.

Important elements of a **software requirements specification** are the **user interface** design and the **input-output behavior** (functional requirements).

Other important elements can include *performance* requirements (e.g., time required to process amount of data), *error handling* behavior, and the *operating system(s)* and computer/networking *hardware* the software must function on.

The SRS may also specify a particular *programming language* if the new system must interface with an existing system.

Software Design

Program **implementation** (writing of code) does *not* commence the instant a software requirements specification is delivered.

Instead, there is a **design** process that determines:

- the **logic** necessary to achieve the spec's goals
- the decomposition into **programs** and **compilation units**
- the decomposition into **modules** (**functions/methods**)
- the **libraries** or calls to be used for particular functionality

Failing to spend adequate time developing a good design typically results in a time-consuming development process due to numerous bugs that are difficult to eliminate.

Even if poorly-designed software is eventually made to work correctly, it will likely be hard to understand and costly to maintain.

Abstraction (in Design and Implementation)

Abstraction is a key mechanism for dealing with the *complexity* of programming.

Abstraction means: *limiting the amount of detail that we must consider at any one time.*

The primary abstraction mechanism for *control/logic* is the use of **functions/methods** (which give a name to a block of code).

Functions/methods also allow us to avoid **code duplication**.

The primary abstraction mechanisms for *data* are **composite data types** (which allow a set of primitive data elements to be treated as a unit).

Abstraction is what enables **incremental software development**.

Abstraction (contd.)

To better understand the role of abstraction in simplifying the development process, consider that we could implement any (non-recursive) program as a single `main`.

Of course the `main` may be huge and difficult to understand, and may also contain significant amounts of (largely) duplicate code.

Breaking the code down into a set of functions/methods permits us to limit the length and complexity of any one module, as well as to avoid code duplication.

Abstraction (contd.)

Consider a `main` containing multiple **code fragments**:
(a code fragment is a sequence of several lines of code)

```
int main()
{
  ...variable declarations...
  ...code fragment A1...
  ...code fragment B...
  ...code fragment C1...
  ...code fragment A2...
  ...code fragment B...
  ...code fragment C2...
}
```

(Notation like `A1` and `A2` indicate the code is largely the same, differing only in the values of some variable(s).)

Abstraction (contd.)

We can simplify `main` by defining functions/methods for each code fragment, containing the fragment's code, along with some appropriate parameters:

```
int main()
{
  ...variable declarations...
  func_A(1,...);
  func_B(...);
  func_C(1,...);
  func_A(2,...);
  func_B(...);
  func_C(2,...);
}

function func_A(x,...)
{
  ...code fragment A(x)...
}
...
```


Abstraction (contd.)

This could nearly literally be our entire new `main`—much shorter and thus much easier to understand, than the original.

It would be an *abstraction* of the original `main`, because we are ignoring the code necessary to implement each of the functions.

(Of course this does depend on the programming language allowing any necessary data to be exchanged among the functions/methods, since their code will no longer be in a shared **scope** as it was in the original `main`.)

Another advantage of this sort of abstraction is that the functions can be implemented (and even tested) *independently* from each other and from `main`.

Software Development 2: Incremental Devel.

1. Introduction
2. **Incremental Software Development**
   - **incremental development**
   - **top-down vs. bottom-up development**
   - **stubs and drivers**
3. Incremental Development Example
4. Software Bugs
5. Testing
6. Debugging

Incremental Software Development

Perhaps the most important piece of advice for being successful developing complex programs is:
develop the program incrementally, i.e., in stages!

Program bugs can interact in complex ways, so the difficulty of debugging code goes up *rapidly* as the number of bugs increases.

Debugging a program with twice as many bugs as another will typically be vastly more than twice as hard.

Working *incrementally* means adding code for a small amount of new functionality, then testing and debugging your program, before moving on to add code for additional functionality.

Incremental Software Development (contd.)

Not only does this approach limit the *number* of bugs in your developing program, it helps limit *where* they might be (most likely in the new code).

Many students mistakenly believe that working incrementally will be too time consuming.

Instead, they type in the entire program, and only then begin trying to get it to compile and work properly.

Unless you are an experienced programmer working with techniques you are familiar with, this approach is highly likely to lead to failure and frustration!

Please don't work that way—particularly when working with an unfamiliar language and/or new programming techniques.

Top-Down vs. Bottom-Up

There are two standard approaches for incremental development:
- **top-down development**
- **bottom-up development**

In *top-down development*, we start with an *abstract* version of our overall logic, then successively refine it until all of the logic has been implemented.

This generally involves the successive addition of *functions/methods* at *"lower levels"* of abstraction.

Lower abstraction levels means functions that come closer to doing what ultimately needs to be done to complete functionality, making less use of use of other functions we have written.

At the lowest level, we will have functions that make use of only system functionality (language operators and library calls).

Top-Down vs. Bottom-Up (contd.)

In *bottom-up development*, we start by implementing the lowest (abstraction) level functions first, then add successively higher-level functions that make use of these functions.

Only at the very last step will we have code that carries out (even abstractly) the entire functionality the system is supposed to have.

In general, either a top-down or bottom-up approach can be used with any project, though sometimes one may make more sense.

The choice of programming language can also be a factor in your choice.

Top-Down vs. Bottom-Up (contd.)

Languages like Lisp and Python have **interactive environments**, where you can run individual lines of code.

Interactive environments help support *bottom-up development*.

By contrast, compiled-only languages like C/C++/Java can require significantly more effort to develop bottom-up.

The reason for this is that with bottom-up development you will have many functions that must be tested separately (since you lack an overall abstract program).

Interactive environments make this relatively simple: just compile each function when completed and then run it to test it.

Without an interactive environment, **driver** programs will be required for bottom-up development.

Stubs and Drivers

Stubs and **drivers** are concepts that arise in the context of top-down or bottom-up incremental program development.

A **stub** is code that substitutes for the code necessary to provide some functionality, prior to that code being written.

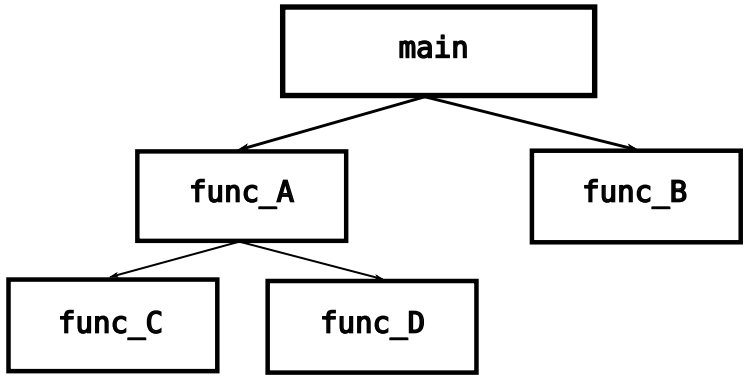
In *top-down development*, having an *abstract* version of the program basically means that the program calls *stub* functions prior to the real functions being implemented.

A **driver** is code that sets up data necessary and then calls an implemented function/method.

In *bottom-up development*, *drivers* allow components to be tested prior to the program as a whole being completed to the point where it can call the components.

Stubs and Drivers Illustrated

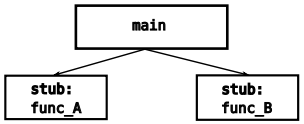
Suppose the function call design for a program is as follows:



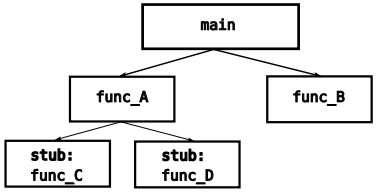
(Vectors indicate function calls.)

Stubs and Drivers Illustrated (contd.)

Top-down development would start as so:



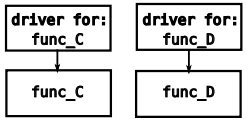
Proceed as follows:



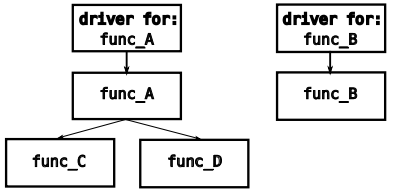
And finally end up with the complete design.

Stubs and Drivers Illustrated (contd.)

Bottom-up development would start as so:



Proceed as follows:



And finally end up with the complete design.

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.

Software Development 4: Bugs

1. Introduction
2. Incremental Software Development
3. Incremental Development Example
4. **Software Bugs**
   - **compilation errors**
   - **software bugs**
   - **programming bugs**
   - **taxonomy of programming bugs**
5. Testing
6. Debugging

Compilation Errors

The very first step in getting a program to run correctly is getting it to **compile**, so you have an **executable** you can test.

Compilation errors mean that the most basic element of a program, its **syntax**, is wrong.

Syntax errors are so basic, they are not generally considered as **software bugs**.

Experienced programmers simply do not end up with compiler errors (except as the result of typos or silly oversights).

If a program does not yet compile, you cannot carry out any testing on it, so you can say *absolutely nothing about its correctness!*

Compilation Warnings

When a compiler gives *warnings* rather than errors, you will end up with an executable that you can test.

However, you should not be mislead: most warnings from GCC indicate the existence of *serious errors in your program*.

It is generally pointless to begin testing a program that has compiler warnings.

Instead, treat the warnings as if they were errors, and fix your code to eliminate them.

Bottom line: students often place great value on their being able to get a program to compile, but successful compilation means *nothing* for program correctness.

Software Bugs

A **software bug** is a program *flaw* that causes the program to (sometimes) behave differently than the way it is supposed to.

For example, a program might produce the wrong outputs for certain inputs, or it might crash with certain inputs.

The primary sources of **software bugs** are:

- **programming errors** (errors in program source code)
- **design errors** (errors in developing the system design)

Design errors can be the most difficult to catch, since they can result from misunderstanding the requirements specification, so tests can also be incorrect.

Programming Bugs

There are many different types of **programming error bugs**.

Most programming bugs might be classified as **logic errors**.

That is, they represent errors in the **program logic**:
the program takes incorrect actions (with particular data).

This is not terribly useful however; we need a much finer-grained categorization of programming errors to guide testing and debugging.

To this end, several people/groups have created **taxonomies** of programming bugs.

While there are some fairly popular taxonomies, there is certainly no universally accepted standard.

Programming Bug Taxonomy

We will discuss the following simple taxonomy:

- **control errors** – incorrect actions taken for some inputs/values
- **processing errors** – incorrect operations to produce results
- **memory errors** – problems with memory access, types, heap management
- **coding errors** – problems due to coding errors (syntactically valid, semantically invalid code)
- **interface errors** – incorrect calls to library routines, etc.
- **error handling errors** — failure to detect/handle errors and invalid inputs/values

Control Errors

Control errors are errors in the *actions* taken in response to particular inputs/values.

I.e., they are errors in the **control flow** of a program.

They result from errors in the branching and looping code of a program, from the omission of required statements, or the insertion of unwanted statements.

E.g., given an input of 5 the program should take code branch A but it actually takes branch B, or given this input it should loop six times calling a function but loops just five times.

Control errors will not be caught by compilers, since they require understanding the actions required to meet the spec.

Control Errors (contd.)

Many common "types" of control errors have been identified:

- incorrect predicates (e.g., `&&` vs. `||`, `<` vs. `>`)
- infinite loops or recursion
- off-by-one errors
- fencepost errors
- non-independent "cases"
- unreachable code or code paths
- race conditions
- deadlock

Processing Errors

Processing errors are errors in the operators/functions/methods used to compute results.

While also errors in "program logic," it is useful to distinguish between errors of control flow and errors of computation.

Examples of processing errors include:

- incorrect operators or functions being used
- overflow or underflow problems in numeric computations
- issues due to roundoff in floating point computations
- divide-by-zero or other invalid numeric operations
- inappropriate casts and type conversions

Memory Errors

Memory errors involve problems due to allocation, deallocation, access, and typing of variables and other program memory.

Examples of memory errors include:

- buffer overflows
- out-of-bounds array accesses
- memory leaks
- dangling pointers
- wrong data type (e.g., size)
- NULL pointer derefernces
- returning pointer to stack-allocated variable from function

Coding Errors

Coding errors are programming bugs that result from coding *mistakes* (code does not mean what the programmer meant it to mean) and *omissions*.

A mistake might mean that a programmer wrote:

```
if (ch = fgetc() != NULL)
```

when they meant/needed to write:

```
if ((ch = fgetc()) != NULL).
```

Forgetting the extra set of parens in the first version will lead to `ch` being assigned either 0 or 1 rather than the ASCII code for the next character.

This will not be caught by a C compiler since the incorrect code is syntactically valid: it still assigns an integer to `ch`, just not the correct integer.

Coding Errors (contd.)

A number of coding errors are fairly common in C:

- **operator precedence** problems
- swapping comparison (==) with assignment (=)
- forgetting to initialize variables
- missing/extra semicolons
- missing braces (particularly with nested if-then-else's)
- missing header includes or function prototypes

Interface Errors

Interface errors are calls to library routines, system calls, or even user functions, which are incorrect in some way.

Syntactically invalid calls (e.g., wrong number or types of parameters) will be caught by the compiler, so are not considered here.

Interface errors are syntactially valid calls that are nevertheless incorrect due to some misunderstanding about the inputs or outputs of a function.

A common type of example using the C `strcpy` function:

```
char str1[] = "test string";
char *str2;
strcpy(str2,str1);
```

The error is that `strcpy()` does no memory allocation (the user must do that *prior* to calling it).

Interface Errors (contd.)

`fgets()` is another common source of interface errors:

```
char line[200];
printf("Next line: %s",fgets(line,200,fptr));
```

This demonstrates several problems (or potential problems):

- will get entire next line only if it is $\leq$ 198 characters
- may not get newline included (so not printed as a line)
- `fgets()` may return `NULL` rather than a string

Error Handling Errors

Error handling errors refer to a program failing to detect and handle errors, or failing to ensure that inputs/values are valid.

It is obvious that a program must handle valid inputs correctly.

For a program to be *robust*, however, it must also deal with incorrect inputs and unexpected errors (e.g., file permissions problems) in an appropriate manner.

Failing to detect and handle incorrect inputs or other errors can lead to **vulnerabilities** that can be **exploited** to attack computer systems.

Even when there are not serious consequences, failing to properly detect and handle errors can result in a program that is very annoying for users when problems occur.

Error Handling Errors (contd.)

C does not have an **exception** mechanism.

Instead, errors are indicated by particular *return values* from functions.

Virtually every call to a library function or system call must be *wrapped* in code to check if the call failed and take some appropriate response if it did:

```
if ((fd = open(argv[1],O_RDONLY)) == -1) {
  perror"Error opening file argument");
  exit(EXIT_FAILURE); }
```

assert()

C's `assert()` macro can be used to validate input values or result values: `void assert(scalar expression)`

E.g., `assert(strlen(str1) < COPY_BUFF_LEN);`

Failure of an `assert()` condition causes an error message to be printed and the program **aborted**.

Assertion checking can be disabled at compile time.

Software Development 5: Testing

1. Introduction
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. **Testing**
   - **software testing**
   - **test cases and test suites**
   - **adequate test suites**
   - **error testing**
   - **automating testing**
   - **limits of testing**
6. Debugging

Software Testing

Testing is a critical component of software development.

Software testing involves execution of a software system or one of its components, for the purpose of evaluating whether the unit under test meets the *software requirements specification*.

This generally means evaluating whether the unit under test satisfies one of more of the following properties:

- accepts the required range of invocations and inputs
- produces the correct input-output behavior
- has desired/acceptable user interface
- handles input/user errors appropriately
- acceptable performance (CPU time, memory, etc.)
- runs in intended CPU/OS environments

Software Testing (contd.)

A range of **software testing methodologies** have been developed.

See *"software testing"* on Wikipedia for more information.

The manner in which testing enters into the development process depends on the development approach being used.

This is particularly true when software is being built by a *team* of developers.

Large software projects frequently employ people whose only job is testing the code being developed.

Testing will then tend to be highly automated, typically supporting both **unit testing** and **regression testing**.

Software Testing Terms

We will concentrate on testing issues appropriate for any developer.

Here are a few key testing terms one should know:

- **test case** – data/call that has particular characteristics
- **test suite** – set of all test cases for program
- **test script** – program to automate use of test suite
- **test harness** – all testing tools and data for a program
- **black box** – testing without code knowledge (API only)
- **white box** – testing while aware of code (e.g., branches)
- **code coverage** – refers to lines of program code tested
- **edge/boundary case** – case with min/max value for a parameter
- **corner case** – usually means error case (beyond boundaries)
- **regression** – bug introduced as result of fixing another bug

Test Cases

A **test case** involves a single *invocation* of a program and/or a particular *input sequence* for an interactive program.

Program invocation involves:
- the combination of options, if any
- the number of arguments, if any
- the characteristics of each of the arguments
- whether call is valid or invalid

Input sequences can involve:
- input values (e.g., particular filenames or numeric values)
- input characteristics (e.g., length of or characters in inputs)
- single vs. multiple input iterations
- termination of input sequence/iteration
- valid or invalid inputs

Test Suites

The set of all *test cases* for a unit is known as the **test suite**.

The goal of a *test suite* should be to ensure that the unit under test works *"as it is supposed to."*

In other words, that it does *not* contain any *bugs!*

Developing a test suite that is adequate to meet this goal is not easy with programs of any complexity.

Experience has shown that students tend to do a very poor job of meeting this goal.

They develop only one or two *test cases*, and consider themselves done when their program works on this small number of cases.

Adequate Test Suites

In general, the only way to be certain a program is bug free would be to test it on *every possible input* (and be able to check its output is correct).

That is obviously *impossible* for all but a trivial ("toy") program.

A program that takes files as input, for example, must be able to deal with an effectively *infinite* number of possible inputs.

Even when a program takes only a large finite number of possible inputs, there would seldom be any way to check outputs.

Doing so would require a program that is known to be correct, but then why would you need another program?

Adequate Test Suites (contd.)

Luckily, there are a few relatively straightforward methods that can help one design a set of test cases that provide reasonable assurance a program is (fairly) bug free.

The testing design approaches we will cover are:

- **equivalence class testing**
- **boundary value testing**
- **code coverage testing**

Equivalence Class Testing

Equivalence class testing is also known as: **equivalence testing**, **equivalence partitioning**, and **equivalence class partitioning**.

The basic idea is that the set of all possible program inputs can be **partitioned** into *subsets*, such that if the program correctly handles *one* input from a subset, it should handle *all* the rest.

In other words, while there may be a large/infinite number of inputs, there will be only a *finite* number of **classes** of inputs.

By testing at least one input from each class, you will have provided (some) coverage of all possible inputs.

The concept of input classes is related to the mathematical notion of **equivalence classes**.

Equivalence Class Testing (contd.)

With simple programs, it may be fairly easily to identify input classes—e.g., see the Wikipedia article "equivalence partitioning."

Identifying input classes is also easier when doing *white box* or *grey box* testing—i.e., when one has access to program code.

Consider the following function:

```
int compute(int x)
{
  if (x <= 0)
    ...code fragment A...
  else if(x > 0 && x < 100)
    ...code fragment B...
  else
    ...code fragment C...
}
```

There are *three* input classes, easily identifiable from the three if-else cases (code fragments A, B, C).

Boundary Value Testing

Boundary value testing (or **analysis**) is related to *equivalence class testing*.

The difference is that instead of testing with *any* value from an *input class*, **boundary/edge case** inputs are chosen.

The reason is that boundary/edge case values are more likely to reveal bugs than would values in the "middle" of an input class.

Identifying boundary values will be easier with *white box* testing, by looking at conditional and loop conditions.

E.g., edge cases for `x` in the above function would be:
(class 1) `0`; (class 2) `1` and `99`; (class 3) `100`.

Code Coverage Testing

Code coverage testing refers to testing that aims to test every one of particular "elements" of program code.

A *test suite's* coverage may be evaluated at various levels:

- **functions** – each function in the program gets called
- **calls** – each function call gets executed
- **statements** – each statement in the program gets executed
- **decisions/branches** – each alternative branch of each control structure gets executed (condition made both true and false)
- **conditions** – each Boolean subexpression gets evaluated to true and false (not just condition overall)
- **paths** – each path (sequence of branches) gets executed
- **loops** – each loop body gets executed zero, one, and multiple times

Error Testing

One area that students often ignore is testing their code to make certain it detects and handles invalid inputs and errors.

In other words, they fail to test that their program does not contain **error handling errors**.

One aspect of error testing is calling the program with **corner cases**—i.e., values that are beyond the boundaries for valid values.

For example, if your program need handle input file names of at most 100 characters, you need to test with file names of 101+ characters to make certain invalid inputs are caught and dealt with appropriately.

Error Testing (contd.)

Unfortunately, testing some errors can be difficult because the error conditions cannot be easily produced.

E.g., `fgetc()` will return `EOF` for both file-end and an *error*.

How can one force an error return from `fgetc()` (with valid code)?

Getting an i/o error would require something like having the file on an external storage device and pulling the plug in the middle of looping reading—not really practical.

Often it is easiest to temporarily invert the error check condition in the program—to force the error handling code to be run, to verify that it properly handles errors.

Automating Testing

All but the simplest programs will require a significant number of test cases.

Since any modification to the program could introduce a **regression**, all tests should ideally be rerun after any program changes.

Doing this by manually rerunning the program (and checking its output/behavior) could become extremely tedious and time consuming.

Luckily, it is relatively easy to provide basic automation of testing for many programs via **shell scripts**.

The simplest possible approach is to simply place a sequence of the program calls for testing into a script file.

Automating Testing (contd.)

E.g., a simple testing script might look like:

```
#!/bin/bash
./myprog ...arguments for test case #1...
./myprog ...arguments for test case #2...
./myprog ...arguments for test case #3...
...
```

Create such a file, make it *executable* ("`chmod +x script`") and you can then rerun all your tests by simply doing "`./script`".

Of course you must be able to see if your program is doing the right thing, so it is good to have it *pause* between tests to scrutinize output.

An easy way to do this is to put a `read` command between each `myprog` call, making you type "Enter" between tests.

Automating Testing (contd.)

If you want something a bit more sophisticated, insert lines like this: `echo -n "Hit Enter to continue"; read`

One can even automate the checking of a test cases's output via `diff` or `sdiff`.

With each test, **redirect** program output to a file:

```
./myprog ...arguments for test case #1... > output1
```

Assuming you have the correct output in a file `correct1`, simply then `diff` the two files:

```
if ! diff -q outpu1 correct1 > /dev/null; then
  echo "Program failed on test #1"
fi
```


Automating Testing (contd.)

Some programs are **interactive**—i.e., they take input from the terminal during execution.

These programs can also be automated, by using **input redirection**:

```
./myprog ...arguments for test case #1... < input1
```

and even:

```
./myprog ...arguments for test case #1... < input1 > output1
```

`input1` is a file that contains the sequence of lines that would have been typed as interactive inputs to the program.

Another tool for testing interactive programs is **Expect** (and later relatives).

The Limits of Testing

Testing is not a substitute for thoughtful design and careful implementation.

For all but the simplest programs, *"adequate"* testing is not guaranteed to reveal every bug.

In fact, some bugs are impossible catch with certainty no matter how much testing is done.

For example, bugs due to **uninitialized variables** may manifest themselves only when the computer memory contains particular values.

This is something you have little control over when testing, so such bugs may remain uncaught even after running many tests.

The Limits of Testing (contd.)

Particularly difficult to test are **concurrent programs**: programs consisting of multiple **processes** or **threads** executing "in parallel."

Concurrent programs may contain **race condition** errors, which manifest only under particular process/thread scheduling orders.

Again, this is something one has little/no control over in testing.

Bottom line: don't simply assume testing will catch any bugs so you don't need to be careful in design and implementation!

Software Development 6: Debugging

1. Introduction
2. Incremental Software Development
3. Incremental Development Example
4. Software Bugs
5. Testing
6. **Debugging**
   - **debugging vs. testing**
   - **testing and programming errors**
   - **tracing**
   - **tracing tools**
   - **instrumentation**
   - **other debugging tools**

Debugging vs. Testing

Testing aims to discover *if* software functions properly or not.

If testing reveals that the software does *not* function properly, this means it contains *bugs* (*errors*).

Debugging is the process of locating the bugs/errors in the code and eliminating them.

The patched program must obviously be thoroughly *retested!*

As we discussed, the primary sources of **software bugs** are:

- **programming errors** (errors in program source code)
- **design errors** (errors in developing the system design)

(We will here assume the specification is correct, but with some software systems, testing may reveal deciencies in the spec!)

Testing and Design Errors

A **design error** occurs when the program code functions as it was designed to, but fails to work properly in some tests.

The discovery of design errors requires changes to the design of the program, and subsequent changes to the program code.

Design errors can lead to very substantial source code changes being required, if major errors were made in the original design.

One the other hand, design errors can be as simple as an overlooked case, requiring no more than an additional `if-else` or `switch` case.

Program design is beyond the scope of this lecture.

Testing and Programming Errors

Programming errors occur when the program source code does not behave according to the program *design*.

We earlier looked at a simple **programming bug taxonomy**.

Programming errors/bugs lead to three common outcomes for test cases that *fail:*

- The program **terminates normally** but produces incorrect output or fails to take correct actions (e.g., prompting).
- The program **terminates abnormally** (i.e., **crashes**).
- The program "**hangs**" (i.e., does not terminate until killed).

Tracing

Tracing refers to printing out (or recording) information about what is happening during a program run.

Tracing is the primary method for locating software bugs.

The primary targets for tracing are:

- program statements executed (branches and loops)
- variable values
- function/method calls (invocations and returns)
- I/O (e.g., file and socket reads/writes)

Tracing tools can provide particular kinds of traces.

Tracing can also be done by **instrumenting** the code—i.e., adding instructions to collect or print out info.

Tracing (contd.)

Tracing tools include **debuggers** and tracing utilities such as `ltrace` and `strace`.

Instrumentation can be done via certain tools such as `gprof` (in conjunction with special GCC flags).

Instrumentation can also be done *manually*, by simply inserting print statements into the code.

C also provides some very basic "tracing" capability for variable values via its `assert()` macro, which validates values.

Tracing: Program Statements

One reason a program may behave improperly is that it does not execute the correct code statements under certain conditions.

In order to identify such errors, it is useful to have a trace of the program statement that get executed under the problem-causing conditions.

Tracing of program statements can be done with debuggers, manual instrumentation, and instrumentation tools like `gprof`.

Often we do not need a *complete* trace of program execution, only information about whether particular branches were taken, whether particular functions were called, and/or how many times certain loop bodies were executed.

Both debuggers and manual instrumentation can provide focused statement tracing.

Tracing: Program Statements (contd.)

Tracing statements to find a control flow error is a first step in debugging the error.

In order to fix the error, the reason for the incorrect control flow must be identified and remedied.

Sometimes such errors can be found by **inspection** of the code responsible for the erroneous control decisions.

Sometimes the reasons will not be obvious, and will require tracing the values being used in the control decisions.

Tracing: Variable Values

Several types of software bugs can lead to variables having incorrect values.

Incorrect variable values may directly result in output errors, or may produce control flow errors that ultimately lead to output errors or other incorrect behavior.

Both debuggers and manual instrumentation can be used to examine—or trace the evolution of—the values stored in program variables.

C's `assert()` macro can be used to identify values that end up outside of the valid range (and stop program execution).

Tracing: Function Calls

Virtually all programs are broken down into functions/methods, so tracing function calls is one way to trace basic control flow.

Generally we will also want to trace the *arguments* given in function calls, as well as any *return values* from the calls.

Debuggers can be used to get this information, though they generally require the user to select each function to be traced, so it is not fast to simply trace all program functions.

Debuggers also allow you to examine the function **call stack** (by doing what is generally called a **backtrace**).

This makes it possible to see the sequence of calls (with arguments) that caused a certain function to be executed, without having to trace each function.

Tracing: Function Calls (contd.)

Stack backtraces are particularly useful when programs *crash*.

Debuggers can use **core dump** files to identify the sequence of function calls (with arguments) that lead to a crash.

The tracing utilities `ltrace` and `strace` allow easy tracing of calls to *library functions* and *system calls* that are made by a program.

Manual instrumentation can also be used to trace function calls.

Tracing: I/O

In order to locate certain program bugs, it is often critical to be able to trace the data that is being read/written by the program.

For example, a program may appear to be making incorrect computations on numbers read from a file, when the problem is actually due to bad assumptions about the file format.

Often, program *"hangs"* are actually due to programs waiting for I/O, which can easily be seen by tracing I/O operations.

Since all I/O is ultimately mediated by the OS, all I/O ultimately involves **system calls** such as `read()` and `write()`.

Because of this, the `ltrace` and `strace` tools are generally a good choice for tracing I/O.

Tracing Tools: GDB

The most popular **debugger** for Linux is the **GNU Debugger** or **GDB** (command `gdb`).

GDB is a *command-line* only tool.

Several *GUI front-ends* have been built for it and several C/C++ IDEs use GDB as their underlying debugger.

The GNU Project's GUI front-end for GDB is **Data Display Debugger** (**DDD**).

GDB/DDD are covered in the **Development Tools** lecture #5.

GDB/DDD allow you to trace statement executions, function calls, and variable values.

Tracing Tools: GDB (contd.)

Unfortunately, using GDB to simply trace calls to some function is not exactly straightforward:

1. Start GDB on your program:
   `gdb ./myprog` (remember to compile with the `-g` flag)
2. Set `myfunc()` as a *breakpoint*:
   `br myfunc`
3. Now set commands to be run at breakpoint:
   ```
   silent
   bt 1
   c
   end
   ```
4. Run program:
   `run`

(Note: GDB can be *"scripted"* via a *commands file*.)

Tracing Tools: ltrace/strace

The tracing utilities `ltrace` and `strace` are also covered in the **Development Tools** lecture #5.

`ltrace` allows you to trace the (dynamic) **library function calls** and **system calls** made by your program as it runs.

`strace` traces only *system calls*.

Note that they cannot be used to trace calls to the functions defined in your own program.

They are particularly useful in programs that make heavy use of library functions or system calls.

Since all program **I/O** ultimately involves system calls, `ltrace`/`strace` are useful for *tracing I/O*.

Tracing Tools: gprof

The **GNU prof** program, `gprof`, can provide information about function calls in a program execution.

Specifically, it can show which functions were called and how many times, and provide a basic call "graph."

To use `gprof`, call `gcc` with the `-pg` flag, e.g.:
```
gcc -Wall -pg -o prog prog.c
```

Now, when you run `prog` it will produce a file `gmon.out` containing function call info for `gprof`.

Use `gprof` to display the info:
```
gprof -b ./prog gmon.out
```

Google will find numerous `gprof` tutorials online.

Manual Instrumentation

Manual instrumentation of a program is as simple as inserting *print statements* (e.g., `printf()`'s) into the program source code.

Printing different messages at different points in the code allows statement execution tracing, printing variable values at different points allows variable value tracing.

While students often dismiss manual instrumentation and immediately turn to debuggers, manual instrumentation can often work better (easily focused, edit-once-run-many).

The major drawback of manual instrumentation is that it requires modifying source code and disrupting normal program output.

Luckily, there are relatively simple ways to minimize these problems.

Manual Instrumentation (contd.)

The key approach for minimizing manual instrumentation issues is to set the instrumentation code up so that it can be easily *turned on and off*.

Accomplishing this by using **C preprocessor directives** that support **conditional compilation** allows instrumentation code to be completely eliminated from executables when desired.

Furthermore, the instrumentation code can be included/not based on options in `gcc` commands.

Thus, once properly instrumented, the instrumentation code can be activated or deactivated, simply by recompiling.

Manual Instrumentation (contd.)

Control of instrumentation code is accomplished via preprocessor (object) **macros** and the ability to define such macros using `gcc`'s `-D` option.

E.g., we can *define* the macro `DEBUG` by calling `gcc` as:

```
gcc -DDEBUG ...
```

E.g., we can define `DEBUG` with *value 1* by calling `gcc` as:

```
gcc -DDEBUG=1 ...
```

Programs can test `DEBUG` by including directives like:

```
#ifdef DEBUG
```

or

```
#if DEBUG > 0
```


Manual Instrumentation (contd.)

The most basic way to include controllable instrumentation code is by inserting code like the following into your program:

```
#ifdef DEBUG
  fprintf(stderr,"Value of x at point 3 is: %d\n",x);
#endif
```

Debug "levels" can be supported like:

```
#if DEBUG >= 1
  fprintf(stderr,"Value of x at point 3 is: %d\n",x);
#endif
```

(Note that debug output is to `stderr`, so as not to interfere with standard output, and allow *redirection* of debugging output.)

Manual Instrumentation (contd.)

A bit more sophisticated approach is to include something like the following in source files (or use a header file `DTRACE.h`):

```
#if DEBUG
  #define DTRACE(args...) fprintf(stderr, args)
#else
  #define DTRACE(args...)
#endif
```

Tracing statements can then be added to the program with single lines like: `DTRACE("Value of x at point 3 is: %d\n",x);`

(Note: the `DTRACE(args...)` notation is C99!)

Other Debugging Tools

There are a variety of other tools that can be useful in debugging:

- **static code analysis tools**
- **simulators**

Static analysis tools analyze program source code to provide information useful for debugging or to identify potential code errors.

Many of these tools are commercial/proprietary.

FOSS examples include: `ctags`, `cxref`, `cflow`, "lint" versions.

It is best to Google something like "static code analysis tools" to find current information.

Other Debugging Tools (contd.)

One of the most popular *simulator* tools is **Valgrind**.

Valgrind is a very useful tool for locating **memory errors**.

Valgrind is covered in the **Development Tools** lecture #5.