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 ≤ 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.