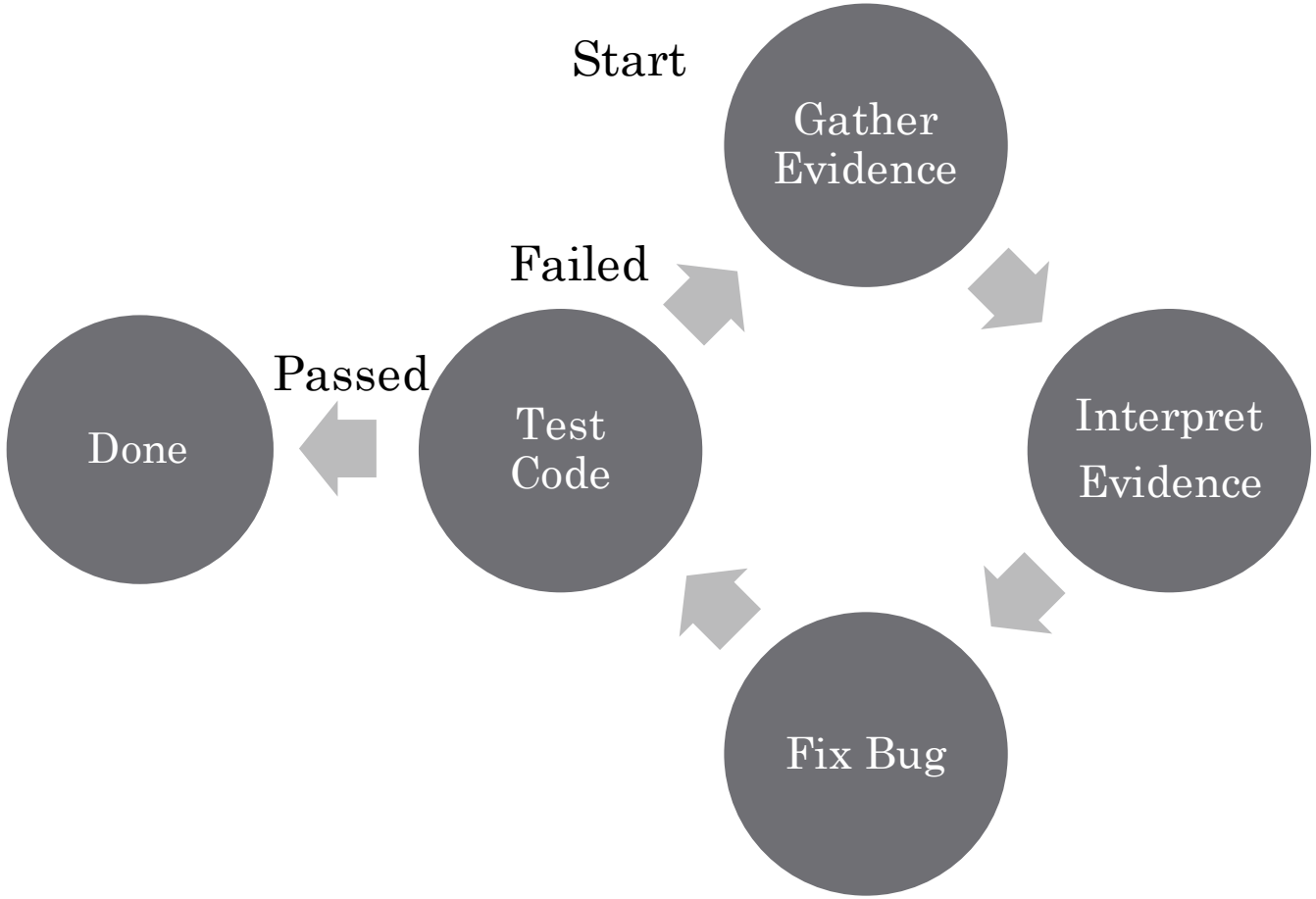
Software Testing

Debugging Techniques

Debugging

- the process of locating and fixing bugs in programs.
- one of the programmers least liked activities.
- It can be frustrating, labor intensive, and sometimes downright painful.
- His programmers mature, they realized that it is often better to write the code carefully and avoid bugs that it is to go through the pain or finding and fixing the bugs.

Debugging Process



Evidence Gathering

- inserting print statements
- Creating a log file
- Using an interactive debugger

Buffered Output

- One of the difficulties of a print statement is that output is usually buffered.
- This means that when a program terminates suddenly, output can be left in a buffer
- This can cause the programmer to believe that the code was in a different position than it really was when it terminated.
- To get around this problem, you should flush the output streams regularly

Types of Errors

- Syntactic errors
  - You code does not follow rules of the language
  - Detected and reported at compile time
  - Easy to fix

- Semantic errors
  - Logic errors in the running program
  - More difficult to find and fix
  - Requires the use of debugging techniques

- Requirements errors
  - We are building the wrong program
  - Clarify requiremeents

Syntactic Errors

- Errors
  - These are severe and stop compilation
  - Need to be corrected before continuing

- Warnings
  - Do not stop compilation
  - Tell the programmer that something unusual has been found
  - Often points to something that will become a semantic error
  - Should NOT be ignored
  - Programmers should strive for a program with no warnings

Linker Errors

- A linker error occurs because
  - Something was not defined,
  - Something was defined twice

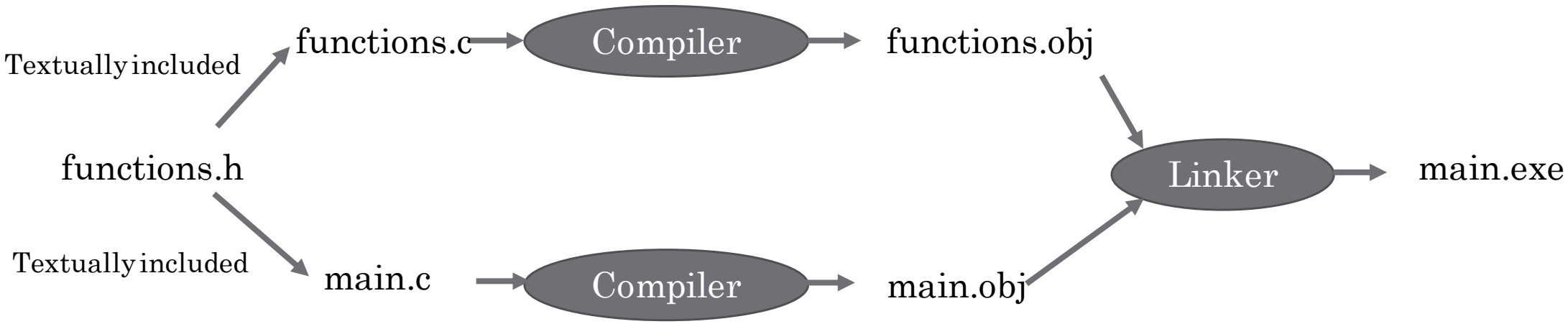
- This declares a function

```
int square(const int n);
```

- This defines the function

```
int square(const int n)
{
return n * n;
}
```

Separate Compilation



Extern Declarations

Declaration

```
#ifndef HEADER_H
#define HEADER_H


// declares type of n
extern int n;

#endif
```

Definition

```
#ifndef HEADER_H
#define HEADER_H


// allocates memory for n
int n;

#endif
```

Debugging Techniques

- When debugging you need to know:
  - Where the program was executing when the bug occurred,
  - Whether the point of termination of the program was where the bug occurred
  - If the program produced incorrect results, where is the bug ?
  - How did this line of code ever get executed?
  - What are the values of the variables as the program executes ?

Debugging Clues

- As you gain experience, you will find that there are certain giveaways as to what might be the source of the bug.
- Bugs which totally stump novice programmers are often solved in a couple of minutes by experienced programmers.
- Below I will go through some of the most common reasons for bugs and how to identify them.

Segmentation Faults

- Modern computers and operating systems have the concept of protected memory.
- If your program attempts to access memory that has been not allocated for your program then it generates a segmentation fault.
- The segmentation fault normally results in the termination of the program.
- The most common reasons for a segmentation fault are:
  - subscripts which run off the end of an array,
  - A pointer containing an invalid address,
  - A NULL pointer,
  - An uninitialized variable,
  - Accessing memory which has been deallocated.

Bus Error

- similar to the segmentation fault in that it is in illegal memory access.
- segmentation fault address is valid memory that is not allocated to your program
- whereas a bus error access is memory which does not exist.

Random Behaviour

- your program behaves differently every time it is run
- almost always a result of uninitialized variables

Program Executes The Wrong Code

- your program is executing code that there is no way it should have been executing
- part of the stack trace might be missing
- symptoms that there is a much larger problem in your program.
- your program took a random jump and ended up in some code that it shouldn't have ended up here.
- This could be
  - a bad pointer or
  - it could be some strange problem with the program.
  - one of the most difficult and challenging bugs to identify.

My Variable Just Changed it's Value

- if a variable seems to have changed its value on its own,
  - then some other piece of code must have changed the value.
- Often it is the result of running off the end of an array or
- using a bad pointer which accessed valid memory within your program.
- Compilers generally layout memory linearly as the variables are declared.
- Take a look at where you declared the variable whose value is being changed.
- If you find it is declared right before or after an array then check to make sure that the bounds of those arrays are not being exceeded.
- Use the watch feature on your debugger to notify you when the variable changes.

My Program Works on One Compiler and Fails on Another

- programs behave differently on different compilers and operating systems is because
  - different compilers in different operating systems do different things

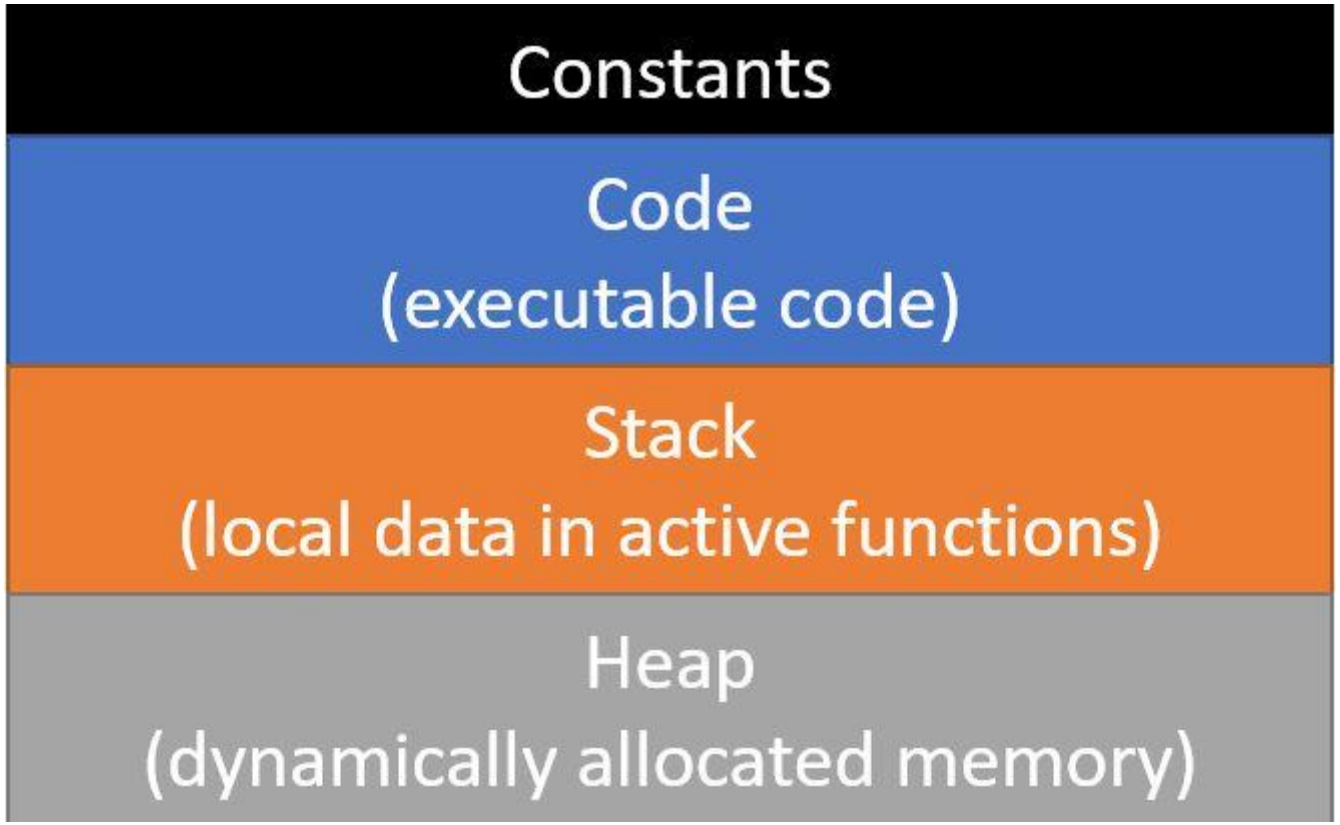
- Windows zeroes memory before starting a program, UNIX does not
- Memory for arrays and structs laid out differently
- Different word sizes on different computers
- Root cause is probably exceeding the bounds of an array or a bad pointer
- Might be an uninitialized variable, but less likely

My Program Just Stopped!

- You might have caught an exception or signal and done nothing,
- If you use a try/catch make sure you do not do nothing when an exception is raised.

Memory Layout

Application memory is broken into 4 main parts:



Managed Memory

- Some languages, like Java and C#, offer *managed memory*.
- This means that there is a *new* operator to allocate a new object but no *delete* operator
- These languages keep a count of the number of references there are in a program to each dynamically allocated variable.
- When that count drops to zero, no one is using the variable, it is marked available for reuse.
- A separate thread runs a *garbage collector* which is responsible for gathering unused variables
- Some of these languages allow you to give hints to the garbage collector to run and reclaim memory

Out of Stack Space

- you've called so many functions that you have literally used up all the memory allocated for the runtime stack.
- Possible reasons:
  - storing a very large number of local variables or large arrays as local variables inside the functions
  - a case of infinite recursion.

Out of Heap Memory

- Occurs because
  - you allocated too much memory (rare), or
  - you have a *memory leak*
- Other Heap Problems
  - You deleted memory twice
  - You used a pointer to deleted memory

- A memory leak is when you allocate memory but you never free it.

Numeric Problems

- **Incorrect Results** - might used integer arithmetic and lost the fractional part of a number.
- **Results start OK but then change sign** - Numbers are stored such that if a number gets too big, it wraps around and becomes negative. The problem is that you are using too small a type to store the number.
- **Inf** - this means that the result of the calculation is infinity. Look for division by zero or other arithmetic problem
- **NaN** - Not a Number. Often due to divide by zero but might be for another reason. Regardless, the value is not a valid number
- **Prints a strange value** - Check your format codes to make sure you are not printing an int as a float or similar type problem in a printf.

Non-Zero Return Code

- This is difficult to debug since it could be caused by many things.
- Consider:
  - divide by zero
  - out of memory

- You really need to narrow down the cause of the bug.
  - Try running the code and stopping at various points until you hit the bug.
  - This might give a clue as to where it is occurring.
  - Repeatedly narrow it down until you find the line on which it occurs.