0. Introduction to Systems Programming

This section is about *advanced* techniques for coding in computer systems. however, as always,
It's concepts, not code.

Therefore, it is recommended to first read the other sections, and then come back to this section *only* when needed for learning how to implement some of the general concepts described elsewhere.

1. A-INTERMEZZO-C-pointers.pdf
2. B1-INTERMEZZO-non-local-flow-of-control.pdf
3. B2-INTERMEZZO-C-code-setcontext.pdf
4. B3-INTERMEZZO-C-code-LD_PRELOAD.pdf
5. C-INTERMEZZO-GDB-fine-art-of-debugging.pdf

INTERMEZZO A: Introduction to C Pointers

Gene Cooperman

A Introduction to C Pointers

Understanding pointers is a fundamental milestone in learning C. Understanding pointers is also required for understanding the man pages of system calls. At its core, a pointer is simply a variable that stores the **memory address** of another variable.

A.1 Why Does C Have Pointers?

Before we study how pointers work, it is worth asking *why* the C language has them at all. The answer comes from the history of the computer industry in the 1970s.

A.1.1 A Little Business History

In the 1970s, the computer world was split in two. **IBM** was a quasi-monopoly in **digital computers**, and its operating systems were written in **assembly language**. **AT&T**, through its Bell Labs research arm, was a *regulated* monopoly whose business was building **analog telephone switches**. Two researchers at Bell Labs, Ken Thompson and Dennis Ritchie, wanted to create an operating system of their own, a project that could potentially open the door for AT&T to build digital computers.

There was an obstacle. IBM's assembly-language-based operating system required *hundreds* of programmers to write and maintain. Thompson and Ritchie did not have hundreds of programmers, and they did not want an operating system that large. Therefore, at Bell Labs they made a different decision: they would build a **small** operating system, written in a **new programming language**. That language became **C**. For C to serve as the language of an operating system, it would need **pointers to memory** (a bridge between the traditional assembly-language style of programming and the new high-level language).

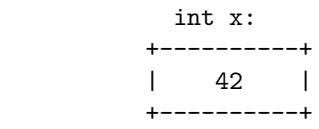
A.1.2 Designing the Language in Stages

Ritchie and Thompson built this new language in stages, each one refining the last. They began with an existing language called **BCPL**, created by Martin Richards. Thompson took the ideas of BCPL and produced a smaller, stripped-down language he called **B**, compact enough to run on their early, tiny machine. Working with B revealed its limits: B had no data types, and it could not take advantage of newer hardware that could address memory one *byte* at a time. Ritchie then evolved B, adding data types (such as a distinct `char` type), pointers, and arrays, into the language we know today: **C**. He wrote the first real C compiler, and its readability and performance were tested on actual programs. (Later, in 1978, Brian Kernighan and Ritchie wrote the book that served as the language's definition for years, which is why C's early form is often called "K&R C.")

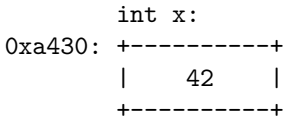
The point of this history is that pointers were a deliberate design decision. They are the feature that lets a small, readable, high-level language still reach down and touch memory directly, the low-level work that an operating system requires.

A.2 From a Beginner's "Box" to an Address

If you have taken an introductory programming course, you have probably seen a variable drawn as a **box** with a label and a value inside. For a variable `x` holding the value `42`, the picture looks like this:



But every box lives *somewhere* in the computer's memory, and that location has an **address**. Let us say this box happens to be located at address **0xa430**:



C needs a way to talk about the *address* of the box, not just the value inside it. That is what the `&` operator is for: `&x` means **"the address of `x`"**. Therefore, `&x` is `0xa430` in our example.

The `&` symbol is called an **ampersand**. Why this symbol? The designers looked at the keyboard for a character whose name would remind us of its job. The name **"ampersand"** begins with **"a"**, just like **"address of."** That is the mnemonic: **`&` = ampersand = address of.**

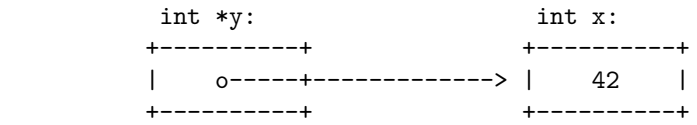
A.2.1 Pointing One Variable at Another

Now suppose we want a *second* variable, `y`, whose job is to **point at** `x`. We declare and initialize it like this:

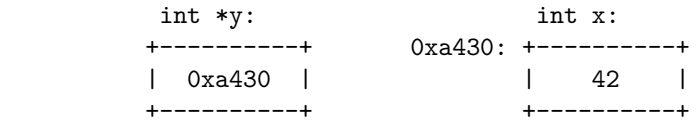
```
int x = 42;
int *y = &x;
```

The type `int *` means **"a pointer to an `int` box."** We store into `y` the value `&x`, which is the address of `x`.

Conceptually, we can draw `y` as a box with an **arrow** that points at the `x` box:



But that arrow is just a picture for our understanding. Internally, the `y` box does not hold an arrow; it holds a **number**, namely the address of `x`. Here is how a pointer is *actually* stored:



The value inside `y` is `0xa430`, the address of the `x` box. An arrow and an address are two ways of drawing the same fact.

A.2.2 Dereferencing: Following the Pointer

Finally, we need a way to **follow** a pointer to reach the value it points at, and possibly to change that value. This is called **dereferencing**, and it uses the `*` operator:

```
int x = 42;
int *y = &x;    // The value of y will be 0xa430, the address of "x".
int z = *y;     // "*y" dereferences y, and z gets the value 42.
*y = 17;        // "*y" on the left-hand side: the "x" box now contains 17.
z = *y;         // Now the "z" box contains 17.
```

Read these lines carefully, because the last two show the two faces of dereferencing:

- On the **right-hand side**, `*y` means *"go to the box `y` points at and read its value."* Therefore, the `z` box gets 42 (`int z = *y;` copies 42 into `z`).
- On the **left-hand side**, `*y` means *"go to the box `y` points at and store into it."* Therefore, `*y = 17;` changes the value inside `x` itself, even though we never mentioned `x` by name.

This is the power a pointer gives us: through `y`, we can reach in and modify `x` indirectly.

A.3 Pointers and Structs

One of the most common data structures in C is a **pointer to a struct**. Suppose we define a struct with three `int` fields and make a pointer to one:

```
struct S { int a; int b; int c; };
struct S s;             // an actual variable of type "struct S"
struct S *ptr = &s;     // ptr holds the address of s
int x = (*ptr).a;       // dereference ptr, then take field "a"
```

To read the field `a` through the pointer, we must first **dereference** `ptr` with `*ptr` to reach the struct, and then use `.a` to select the field. But the notation `(*ptr).a` is awkward: the parentheses are required (because `.` binds more tightly than `*`), and the mix of `*` and `.` is easy to get wrong.

Therefore, C added a cleaner notation, the **arrow operator** `->`:

```
int x = ptr->a;   // ptr->a  means exactly the same thing as (*ptr).a
```

The two expressions are identical in meaning. Whenever you have a pointer to a struct, use `ptr->field` rather than `(*ptr).field`, as it is the idiom you will see in essentially all C code, including the man pages for system calls.

A.3.1 Building a Linked List

The arrow operator becomes especially useful once a struct contains a pointer to *another struct of the same type*. This is exactly how a **linked list** is built. We define a node with two fields: an `int` called `value`, and a pointer to the next node called `next`:

```
struct node {
  int value;          // the data stored in this node
  struct node *next;  // a pointer to the next node (or NULL at the end)
};
```

We can build a small list of two nodes (`10` and `20`) one step at a time, filling in each node's fields after declaring it:

```
struct node *head;
struct node n1;
struct node n2;
head = &n1;
n1.value = 10;
n1.next = &n2;
n2.value = 20;
n2.next = NULL;
```

Note that `n1` and `n2` are struct *variables*, not pointers, so we use the dot operator (`n1.value`) to reach their fields. We use `&n2` when storing into `n1.next`, because `next` is a *pointer* field and needs an address.

In memory, the nodes form a chain, each `next` field pointing at the following node until we reach `NULL`:

```
   head:
  +-----------+     +-----------+     +-----------+
  |     o-----+---> | value 10  |     | value 20  |
  +-----------+     | next  o---+---> | next  o---+---> NULL
                    +-----------+     +-----------+
```

After this, `head->next->value` will have the value `20`.

A word about `NULL`. In C it is usually defined as `void *NULL = 0x0;`, that is, the address `0x0`. On most systems the address `0x0` is reserved and unreachable, so `NULL` means **"pointer to nothing"**. The special type `void *` is a pointer type that is interchangeable with all other pointer types, which is why `NULL` can be assigned to any pointer. For example, `int *x = NULL;` simply means that the `x` box will contain the address `0`.

A.3.2 An Example: Why Pointers Were Needed

To see why an operating-system language truly needs pointers, imagine we are writing code to control a hardware **device**. The device exposes a small block of memory whose fields tell us how to access it. In C we describe that block with a `struct`:

```
struct device_registers {
  unsigned int status;   // read: is the device ready?
  unsigned int control;  // write: command the device
  unsigned int data;     // read/write: the actual bytes
};
```

The hardware designer tells us that this device's registers live at a fixed memory address, say, `0xfe00`. No ordinary variable lives there; the address is fixed by the physical wiring of the machine. How, then, can our C program reach it? With a **pointer**, set to point directly at that address:

```
struct device_registers *dev = (struct device_registers *) 0xfe00;

if (dev->status == 1) {   // read the device's status field
  dev->data = 'A';        // write a byte into the device
  dev->control = 1;       // command the device to act
}
```

Without pointers, there would be no way to say *"treat this specific address as a* ***`struct`*** *and read and write its fields."* That is exactly the kind of operation that once required assembly language, and it is exactly what pointers let C do while still looking like a readable, high-level program. As we will see, it is the same mechanism that lets your programs talk to the operating-system kernel through system calls.

A.4 Pointers, Arrays and Strings

In C, an **array is a constant pointer** to its first element. Consequently, C does not have a native string type; it uses the type `char *` as a pointer to the beginning of an array of characters. By convention, an array of `char` representing a string must end with the **null character (`\0`)** to indicate the end of the string.

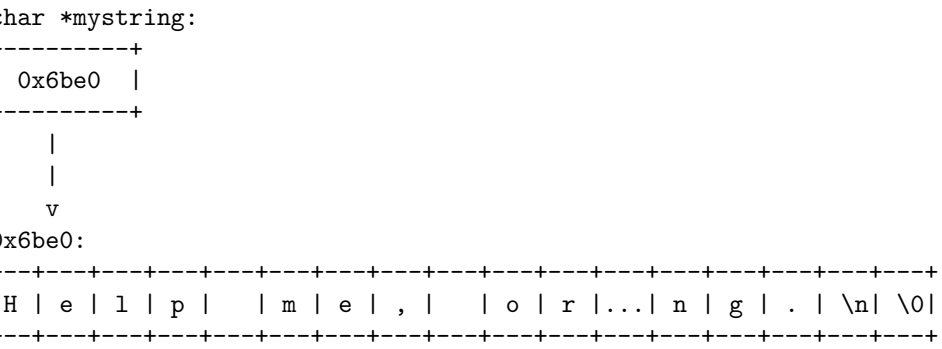
```c
#include <stdio.h>
int main() {
  char *mystring = "Help me, or I'm leaving.\n";
  // ALTERNATIVE:  char mystring[] = "...\n";
  printf("%s\n", mystring);
  printf("%s\n", mystring + 12);
  printf("%s\n", &mystring[12]);
  mystring[7] = '\0';
  printf("%s\n", mystring);
  return 0;
}
```

A.4.1 Expected Output

```
Help me, or I'm leaving.
I'm leaving.
I'm leaving.
Help me
```

A.4.2 Memory Diagram of the C String

This diagram shows the `mystring` pointer pointing to the array starting at address **0x6be0**.



A.4.3 An Array Name is a Constant Pointer

Finally, note that we could have used either of:

```c
const char *mystring = "Help me, or I'm leaving.";
char mystring[] = "Help me, or I'm leaving.";
```

In this form, the diagram would no longer show a mystring box in memory. Now, mystring is a constant. Constants have a fixed value known only to the compiler, and the value of the constant cannot be changed at runtime.

A.4.4 Understanding Pointer Arithmetic and the Null Character as Terminator

1. **Pointers of `char *`:** The expression `mystring + 12` calculates a new address by moving 12 character-sized boxes forward from the start of the array.
2. **Addressing Indices:** The expression `&mystring[12]` is functionally identical to `mystring + 12`, as both return the memory address of the character at index 12. Similalry, `mystring[12]` returns a `char`, and the expression is equivalent to `(*mystring+12)`.
3. **The Null Character (\0) as Terminator:** By setting `mystring[7] = '\0';`, the code inserts a "stop sign" into the array. When `printf` processes the string, it stops immediately upon hitting the null character, even though the rest of the characters are still in memory.
4. **Pointers of `int`:** Now, let's consider an array of int: `int mynum[100]`. In this case, the expression `&mynum[12]` is functionally identical to `mynum + 12`, as both return the memory address of the character at index 12. Similalry, `mynum[12]` returns an `int`, and the expression is equivalent to `(*mynum+12)`.

With this insight, we can now discuss **pointer arithmetic**. What happens if you add an integer to a pointer? What happens if you subtract two pointers? The answer to these questions depends on the type of the pointer. Is it a pointer to a char (`char *`), a pointer to an int (`int *`), or something else? In particular, how many bytes does a `char` occupy and how many bytes does an `int` occupy.

In addition to the operators, '*' and '&', C provides a third operator, `sizeof`. In a typical C compiler for a typical CPU, we will usually find that `sizeof(char)==1` and `sizeof(int)==4`, where the size is measured in bytes. This means that a "char box" occupies 1 byte and an "int box" occupies 4 bytes.

Nevertheless, `&mynum[7]-&mynum[6]==(myunum+7)-(mynum+6)==1`. Intuitively, we are operating on pointers to `int`. When you add one to a pointer to `int`, then we are now pointing to the *next* "int box". An "int box" has size 4. So, we are now pointing to an address that is 4 bytes beyond the original address, but as a pointer to `int`, we have incremented the pointer to 1. Similarly, if we subtract two pointers to `int`, the result is the number of "int boxes" between the first pointer and the second pointer.

You can verify this with:

```c
#include <stdio.h>
// Adding and subtracting pointers to int.
int main() {
  int mynum[100];
  int *ptrA = mynum+6;
  int *ptrB = mynum+7;
  // Subtracting these two pointers yields the int, 1.
  printf("ptrB-ptrA: %ld\n", ptrB-ptrA);
  // Again subtracting these two pointers yields the int, 1.
  printf("(mynum+7)-(mynum+6): %ld\n", (mynum+7)-(mynum+6));
  // And again subtracting these two pointers yields the int, 1.
  printf("&mynum[7]-&mynum[6]: %ld\n", &mynum[7]-&mynum[6]);
  // Adding a pointer and the int 1 yields a new pointer that is 4 bytes higher.
  printf("ptrA: %p; ptrA+1: %p\n", ptrA, ptrA+1);
  // Again, the second pointer is 4 bytes higher.
  printf("&mynum[6]: %p; &mynum[7]: %p\n", &mynum[6], &mynum[7]);
  return 0;
}
```

And finally, C provides a **cast**, so that you can change the type. So, `(char *)(mynum+7)-(char *)(mynum+6)` is equal to 4, not 1. We can fit 4 "char boxes" between the two pointers that we subtracted.

A.5 Pointers, Functions and IN-OUT parameters

A function argument of type `int` is always an **IN parameter**: the function call passes a value *into* the function, and the function cannot change the caller's original variable. When a function argument is of a *pointer* type, however, we gain more options. Such an argument can be an **IN parameter**, an **OUT parameter**, or an **IN-OUT parameter**, depending on how the function uses it during a call: whether the pointer is used to read a value *into* a local variable of the function (IN), to write a local value of the function *out* to the memory pointed to by some external variable (OUT), or both (IN-OUT).

As a first example, consider a function that swaps the values of two variables. Both `a` and `b` are IN-OUT parameters: the function reads each value *in*, and writes a new value back *out* through the same pointer.

```
void swap(int *a, int *b) {
  int temp = *a;   // read the value pointed to by a (IN)
  *a = *b;         // write a new value through a (OUT)
  *b = temp;       // write a new value through b (OUT)
}
```

For example:

```
int x = 3;
int y = 5;
swap(&x, &y);   // Now, x is 5 and y is 3.
```

Some languages like C++ have function arguments that are "reference variables". If `swap` were defined in C++ using reference variables, then we would call `swap(x, y)`. But the compiler transforms the swap function internally to something like the pointer-based swap function with IN-OUT parameters, as above.

The following function has an IN-OUT parameter, since the value `*myvariable` is passed in, and the value `*myvariable + 1` is passed out. Study this function, and see if you can understand the logic of the output.

```
#include <stdio.h>
void addOne(int *myvariable) {
  *myvariable = *myvariable + 1;
}
// Three ways to increment a variable, including pointers.
int main() {
  int x = 17;
  int *y = &x;
  x = x+1;
  printf("x: %d\n", x);
  *y = *y + 1;
  printf("x: %d\n", x);
  addOne(&x);
  printf("x: %d\n", x);
  return 0;
}
```

A.5.1 Expected Output

```
x: 18
x: 19
x: 20
```

A.6 System Calls and IN/OUT Parameters

A **system call** is a controlled entry point into the operating system kernel. User programs cannot directly access hardware or perform privileged operations—they must ask the kernel to do it on their behalf.

Common syscalls include:

- `read` - Read data from a file or input device
- `write` - Write data to a file or output device
- `open` - Open a file
- `close` - Close a file
- `time` - Get the current time
- `malloc` - Allocate a block of memory
- `free` - Release a block of memory back to the system

A.6.1 Pointers in System Calls

Many syscalls require pointers as arguments. This is because:

1. **Efficiency**: Passing a pointer to a large buffer is cheaper than copying the entire buffer
2. **Kernel access**: The kernel needs to know where in your program's memory to read from or write to
3. **Variable-length data**: Strings and buffers have varying sizes

A.6.1.1 IN Parameters vs OUT Parameters

When a syscall takes a pointer argument, it can be used in two ways:

IN parameter: The pointer points to data that the syscall will *read*. Your program fills in the data before the syscall, and the kernel reads it.

OUT parameter: The pointer points to a buffer where the syscall will *write* results. Your program provides empty space, and the kernel fills it in.

Some parameters are **IN-OUT**: the kernel both reads the initial value and writes back a result.

A.6.2 Example 1: The `write` Syscall (IN Parameter)

The `write` syscall outputs data to a file descriptor. Its signature in C is:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd` - **IN parameter**: File descriptor (1 = stdout, 2 = stderr)
- `buf` - **IN parameter**: Pointer to the data to write
- `count` - **IN parameter**: Number of bytes to write
- Returns: Number of bytes actually written

The `buf` parameter is an IN parameter because the kernel *reads* from this memory location.

```
char message[] = "Hello, World!\n";
// Pass the address of our data using & (or array name decays to pointer)
int bytes_written = write(1, message, 14);
// The kernel reads FROM the memory at &message[0]
// Our data flows: message -> kernel -> screen
```

Notice that we pass the *address* of `message` to the kernel. The kernel will read 14 bytes starting at that address.

A.6.3 Example 2: The `read` Syscall (OUT Parameter)

The `read` syscall inputs data from a file descriptor. Its signature in C is:

```
ssize_t read(int fd, void *buf, size_t count);
```

- `fd` - **IN parameter**: File descriptor (0 = stdin)
- `buf` - **OUT parameter**: Pointer to buffer where data will be stored
- `count` - **IN parameter**: Maximum number of bytes to read
- Returns: Number of bytes actually read

The `buf` parameter is an OUT parameter because the kernel *writes* to this memory location.

```
char buffer[100];  // Reserve space for input (initially empty/undefined)
// Pass the address of our buffer.  An array name is also a pointer.
int bytes_read = read(0, buffer, 100);
// The kernel writes TO the memory at buffer (also could be: &buffer[0])
```

Before the syscall, `buffer` is uninitialized space. After the syscall, the kernel has filled it with whatever the user typed.

A.6.4 Example 3: The `time` Syscall (OUT Parameter)

The `time` syscall gets the current time. Its signature in C is:

```
time_t time(time_t *tloc);
```

- `tloc` - **OUT parameter**: Pointer to where the time should be stored (can be NULL)
- Returns: The current time (seconds since January 1, 1970)

This syscall demonstrates an interesting pattern: the return value and the OUT parameter contain the same information.

```
time_t tloc;
// Pass the address of time_val using &
time_t result = time(&tloc);
// The kernel writes TO the memory at &time_val
// After the call:
//   - result contains the current time
```

Using the `&` operator, we pass the *address* of `tloc`. The kernel then uses this address to write the current time into our variable.

A.6.5 Example 4: The `malloc` and `free` Syscalls (Dynamic Memory)

The `malloc` syscall asks the system for a block of memory and *returns a pointer* to it; `free` gives that block back. Their signatures in C are:

```
void *malloc(size_t size);
void free(void *ptr);
```

- `malloc` takes the number of bytes wanted and returns a `void *` pointing to the new block (or `NULL` if none is available).
- `free` takes the pointer that `malloc` returned and releases that block.

Recall the linked list example from the earlier section. There, `n1` and `n2` were ordinary variables, and their memory vanished when the function returned. With `malloc`, we can instead create nodes that outlive the function. Starting from `head`, we allocate each node on demand:

```c
struct node *head = malloc(sizeof(struct node));  // head points to n1
head->value = 10;
head->next = malloc(sizeof(struct node));  // head->next (n1.next) points to n2
head->next->value = 20;
head->next->next = NULL;
```

Here `sizeof(struct node)` gives `malloc` the number of bytes one node needs. When we are finished with the list, we return each block with `free`:

```c
free(head->next);   // free n2 first
free(head);         // then free head
```

Note the order: we free `head->next` *before* `head`, because once `head` is freed we can no longer follow it to reach the second node.

A.6.6 Summary: Pointer Usage Patterns in Syscalls

Syscall	Pointer Parameter	Type	Purpose
`write`	`buf`	IN	Kernel reads data from your buffer
`read`	`buf`	OUT	Kernel writes data into your buffer
`time`	`tloc`	OUT	Kernel writes time value
`malloc`	(return value)	OUT	Kernel returns a pointer to new memory
`free`	`ptr`	IN	Kernel reads the pointer to release its memory

A.6.7 Key Principles

1. **Syscalls are the interface to the OS** - They're how your program requests services from the kernel
2. **Pointers are essential for syscalls** - They allow the kernel to access your program's memory
3. **IN parameters point to data you provide** - The kernel reads from these locations
4. **OUT parameters point to space you allocate** - The kernel writes results to these locations
5. **Always allocate enough space for OUT parameters** - Buffer overflows cause crashes and security vulnerabilities
6. **Check return values** - Syscalls can fail; the return value typically indicates success or error

Understanding syscalls and pointer parameters reveals how all programs interact with the operating system. Every time you call `printf()` or `scanf()` in C, the library ultimately makes syscalls like `write` and `read`, passing pointers exactly as we've shown here!

INTERMEZZO B1: Non-Linear Flow of Control

Gene Cooperman

B1 INTERMEZZO: Non-Linear Flow of Control

Objective: To understand how program execution flow can deviate from the standard sequential path using three primary mechanisms: Signal Handling, Context Switching, and Pre-execution Initialization.

B1.1 The Foundation: Reviewing the Call Stack

Standard program execution is **linear** and **synchronous**. When a function is called, the system manages memory using the call stack.

- **The Call Frame:** Every pending function call creates a memory area called a "call frame" or "stack frame."
- **Contents:** A frame holds local variables, arguments, and most critically, the **return address** (the instruction pointer where execution should resume).
- **Linearity:** The stack grows predictably (e.g., `main` calls `A`, `A` calls `B`). Under normal (linear) control flow, execution returns to the calling function (LIFO order).

B1.2 Example 1: Signal Handlers (Asynchronous Events)

Signal handlers allow the Operating System (OS) to interrupt the program at any moment to communicate an event (a signal sent by the operating system), such as an integer divide-by-zero, timer expiry, or user interrupt.

- **Trigger:** An **asynchronous** event originating from the OS kernel or hardware.
- **Comparison to Exceptions:** A signal handler processes this asynchronous event in a similar philosophical manner to **exception handlers** seen in higher-level languages like Java and C++. When an asynchronous fault occurs, the signal handler is invoked, much like a `catch` block is invoked when an exception is thrown.
- **Mechanism:**
  1. Execution of the process is paused immediately by the operating system.
  2. The OS forces the creation of a **new call frame** on top of the existing stack structure.
  3. This call frame belongs to the registered **Signal Handler function**.
- **Non-Linearity:** This is a non-linear flow because execution jumps from any arbitrary instruction to the handler function, without a corresponding `call` instruction in the program's source code.

Here is a small example of an asynchronous signal handler using the Linux syscall `alarm` (conceptually similar to the Java class `Timer`).

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <stdlib.h>

void alarm_handler(int sig) {
    char msg[] = "\nalarm_handler() was called.\n";
    write(1, msg, sizeof(msg) - 1);  // printf is not async-safe for signal handlers
    _exit(1);                        // exit is not async-safe for signal handlers
}

int main() {
    signal(SIGALRM, alarm_handler);
    alarm(3);
    while (1); // infinite loop
    return 0;
}
```

B1.3 Example 2: Context Switching (`getcontext` and `setcontext`)

This mechanism allows user-level code to freeze the state of the CPU and later restore it, enabling programmatic jumps between arbitrary points in execution.

B1.3.1 What is a "Context"?

A "context" (represented by `ucontext_t` in C) is a data structure containing the snapshot of the CPU's state at a single moment. It is the current state of the CPU (or the state of the CPU core running the current thread), and especially the current state of the registers, including the **Program Counter (Instruction Pointer)** and **Stack Pointer**.

B1.3.2 The Flow: Implementing Try-Catch using Signals

The `try-catch` syntax for exception handlers in Java or C++ could have been implemented as a signal handler combined with a call to `setcontext`.

1. **The "Try" Block:** At the beginning of a high-level `try` block, the runtime performs a call to **`getcontext()`**. This saves the current, stable CPU state into a context variable (the "checkpoint").
2. **The Fault/Exception:** If code inside the `try` block triggers a fatal error, the OS sends a **signal**.
3. **The Signal Handler (The "Catch"):** The pre-registered **signal handler** function is executed.
4. **The Unwind:** Inside the signal handler, the non-linear action occurs: a call to **`setcontext()`** using the checkpoint saved in step 1.

- **Effect:** This instantly **unwinds the stack** (discarding the signal handler's frame and the frames that led to the fault), and execution resumes at the exact instruction immediately following the original `getcontext()` call, thus returning to an earlier call frame on the stack.

Function	Action in a Try/Catch implementation	Effect on Flow
`getcontext()`	**Save:** Executed at the start of the `try` block.	Execution continues linearly.

Function	Action in a Try/Catch implementation	Effect on Flow
Signal Handler	**Interrupt:** Invoked on fault (e.g., integer divide by zero).	Interrupts linear flow, pushes new frame.
`setcontext()`	**Restore:** Called inside the signal handler to jump back to the `getcontext` location.	Execution **teleports** (non-local jump) back to the saved state.

B1.4 Example 3: Constructor Functions (Pre-execution Initialization)

This non-linear flow occurs before the program's explicit starting point (`main`).

- **The Rule:** If a global object or variable is initialized by a **constructor function**, this must occur even **before the normal execution of the "main" function**.
- **Mechanism:** The **Dynamic Linker** (the program loader) handles this flow by executing the constructors in all loaded libraries before jumping to the program's main entry point.

B1.4.1 Forcing Non-Linear Flow with `LD_PRELOAD`

This scenario can be forced by, for example, setting the environment variable, `LD_PRELOAD=/full_path/mylib.so` before executing a program.

- If `mylib.so` has a global variable or object initialized by a constructor function, then the constructor function will execute even before the function `main`, due to the use of `LD_PRELOAD`.

B1.4.2 Flow of Control Summary

The flow is dramatically altered from the expected: **Loader (Linker) → Library Constructor → Main function**.

B1.5 Summary Table

Mechanism	Trigger	Stack Behavior	Flow Type
Signal Handler	Asynchronous OS Event	Pushes new frame on top	Interrupt
setcontext	Explicit Function Call	Overwrites CPU state (Non-local jump)	Jump/Restore
Constructors	Program Load / Linker	Execution occurs before the defined entry point (`main`).	Pre-execution

---

B1.6 Going Deeper

Mechanism	man page
Signal Handler	`man 2 signal` (register a signal handler function); `man 2 kill` (send signal) [production code usually prefers the more complex `sigaction`]
setcontext	`man 3 setcontext` (includes setcontext and getcontext); requires at beginning of file: `#define _GNU_SOURCE`
Loader/linker	`man ld.so` (and search on `LD_PRELOAD`)
Constructors (used with `LD_PRELOAD`)	Java: built into spec for classes C: add `__attribute__((constructor))` after return type of the function

B1.6.1 Example Code:

See `B2-INTERMEZZO-C-code-setcontext` and
`B3-INTERMEZZO-C-code-LD_PRELOAD`, later in this file, for example C code.

INTERMEZZO B2: C Code Example: Context Switching for Exception Emulation

Gene Cooperman

B2 C Code Example: Context Switching for Exception Emulation

This example demonstrates how to combine **Signal Handling** (Example 1) with `setcontext` (Example 2) to achieve a non-local jump that simulates a high-level exception (`try/catch` block) by "rewinding" the program flow.

B2.1 File: `context_exception.c`

```c
#include <stdio.h>
#include <stdlib.h>
#include <ucontext.h>
#include <signal.h>
#include <unistd.h>

// The "checkpoint" variable to store the CPU state (registers, PC, stack pointer)
static ucontext_t checkpoint_context;

// A flag to simulate the logic flow of a try/catch block
// 'volatile' ensures the compiler doesn't optimize away checks on this variable.
volatile int exception_caught = 0;

// This is our Signal Handler (the low-level "catch" mechanism)
void exception_handler(int sig) {
    printf("\n[Signal Handler] Caught signal %d (SIGFPE).\n", sig);
    printf("[Signal Handler] Resolving error and restoring context...\n");

    // 1. Update the state to indicate we handled the error
    exception_caught = 1;

    // 2. Non-Local Return: JUMP
    // We overwrite the current CPU state with the state saved earlier.
    // Execution will immediately resume at the point where getcontext was called.
    setcontext(&checkpoint_context);

    // The program will NEVER reach code immediately after setcontext().
}
```

```c
int main() {
    // Register the signal handler for SIGFPE (Floating Point Exception / div by zero)
    signal(SIGFPE, exception_handler);

    printf("[Main] Starting program.\n");

    // SAVE THE CONTEXT (The "Try" block entry point)
    // getcontext saves the current registers and stack ptr into 'checkpoint_context'
    // Execution proceeds immediately after this line (getcontext returns 0).
    getcontext(&checkpoint_context);

    if (exception_caught == 0) {
        // --- FIRST PASS (Linear Flow) ---
        printf("[Main] Context saved. About to attempt dangerous operation...\n");

        // --- DANGER ZONE ---
        // This causes an immediate interrupt/signal (SIGFPE)
        int a = 10;
        int b = 0;
        int result = a / b;

        // This line is unreachable in the first pass
        printf("[Main] Result: %d\n", result);
    }
    else {
        // --- SECOND PASS (Non-Linear Flow) ---
        // Execution jumps directly here because setcontext restored the IP
        // to the getcontext line, but the variable 'exception_caught' is now 1.
        printf("[Main] Recovered! Execution resumed at the checkpoint.\n");
    }

    printf("[Main] Program finishing normally.\n");
    return 0;
}
```

B2.2 Lecture Talking Points

1. **The Checkpoint (`getcontext`)**:

- This function saves the entire CPU state (including the instruction pointer pointing to the line *after* `getcontext`) into `checkpoint_context`. This is the **linear flow**.

2. **The Interruption (Signal)**:

- The division by zero (`a / b`) causes the OS to stop the program, push a frame for `exception_handler` onto the stack, and execute the handler. This is the **asynchronous, stack-building** phase.

3. **The Teleport (`setcontext`)**:

- Inside the handler, `setcontext(&checkpoint_context)` is the critical non-linear operation.
- It **overwrites** the CPU's registers with the saved state.
- This causes execution to immediately jump back to the `getcontext` call site, but this time, the `exception_caught` flag is set to handle the jump correctly. The stack is effectively **unwound** without

the standard `ret` instruction.

INTERMEZZO A: C Code Example: Context Switching for Exception Emulation

Gene Cooperman

B3 Complete Lecture: Non-Linear Flow of Control (ASCII)

B3.1 The Foundation: Reviewing the Call Stack

Standard program execution is **linear** and **synchronous**. When a function is called, the system manages memory using the call stack.

- **The Call Frame:** Every pending function call creates a memory area called a "call frame" or "stack frame."
- **Contents:** A frame holds local variables, arguments, and most critically, the **return address** (the instruction pointer where execution should resume).
- **Linearity:** The stack grows predictably (e.g., `main` calls `A`, `A` calls `B`). Execution always returns to the function directly below it on the stack (LIFO).

B3.2 Example 1: Signal Handlers (Asynchronous Events)

Signal handlers allow the Operating System (OS) to interrupt the program at any moment to communicate an event (e.g., divide-by-zero, timer expiry, user interrupt).

- **Trigger:** An **asynchronous** event originating from the OS kernel or hardware.
- **Mechanism:**

1. The CPU is paused immediately.
2. The OS forces the creation of a **new call frame** on top of the existing stack structure.
3. This frame belongs to the registered **Signal Handler function**.

- **Non-Linearity:** This is a non-linear flow because execution jumps from any arbitrary instruction to the handler function, without a corresponding `call` instruction in the program's source code.

B3.3 Example 2: Context Switching (`getcontext` and `setcontext`)

This mechanism allows user-level code to freeze the entire state of the CPU and later restore it, enabling programmatic jumps between arbitrary points in execution.

B3.3.1 What is a "Context"?

A "context" (represented by `ucontext_t` in C) is a data structure containing the snapshot of the CPU's state at a single moment. It includes:

- **Program Counter (PC) / Instruction Pointer:** The address of the next instruction.
- **Stack Pointer (SP):** The current top of the stack.
- **Registers:** The values stored in all General Purpose Registers.

B3.3.2 The Flow: Emulating Exceptions

Using `setcontext` is key to implementing exceptions in low-level runtime environments.

Function	Action	Effect on Flow
`getcontext()`	**Save:** Saves the current CPU state (registers, PC, SP) to a context variable. (The "Try" block checkpoint).	Execution continues linearly.
`setcontext()`	**Restore:** Loads a previously saved context back into the CPU.	Execution **teleports** back to the instruction immediately following the original `getcontext()` call.

B3.3.3 C Code Example: Context-Based Exception

```c
#include <stdio.h>
#include <ucontext.h>
#include <signal.h>

static ucontext_t checkpoint_context;
volatile int exception_caught = 0;

void exception_handler(int sig) {
    printf("\n[Signal Handler] Caught SIGFPE.\n");
    // Non-Local Return: Instantly jump back to the saved context.
    exception_caught = 1; // Set flag to exit the "if" block
    setcontext(&checkpoint_context);
}

int main() {
    signal(SIGFPE, exception_handler);

    // Save the "Try" context (the checkpoint)
    getcontext(&checkpoint_context);

    if (exception_caught == 0) {
        printf("[Main] Context saved. Attempting illegal operation...\n");
        int a = 10;
        int b = 0;
        int result = a / b; // Causes SIGFPE signal
        printf("[Main] Result: %d\n", result);
    }
    else {
```

```
        // Execution jumps here from setcontext
        printf("[Main] Recovered! Flow restored to the checkpoint.\n");
    }

    return 0;
}
```

B3.4 Example 3: Constructor Functions (Pre-execution Initialization)

This non-linear flow occurs before the program's explicit starting point (`main`).

- **The Rule:** Any global object or variable initialized by a constructor function **must** execute *before* the `main` function starts.
- **Mechanism:** The **Dynamic Linker** handles this flow by executing the constructors in all loaded libraries before jumping to the program's main entry point.

B3.4.1 Forcing Non-Linear Flow with `LD_PRELOAD`

Using the environment variable `LD_PRELOAD`, we force the Dynamic Linker to load a custom shared library (`mylib.so`) and execute its constructor before `main`.

B3.4.2 C Code Example: Constructor Injection

B3.4.2.1 `hack.c` (The Shared Library)

```
// hack.c
#include <stdio.h>

// The compiler attribute that marks this function for pre-execution.
__attribute__((constructor))
void my_init_hook() {
    printf("\n[Constructor Hook] --> Code running BEFORE main() starts!\n");
}
```

B3.4.2.2 `main.c` (The Target Program)

```
// main.c
#include <stdio.h>
int main() {
    printf("[Main] --> Now executing the main function.\n");
    return 0;
}
```

B3.4.2.3 Execution Guide

```
# 1. Compile the library and the program
gcc -shared -fPIC -o libhack.so hack.c
gcc -o normal_program main.c

# 2. Run with the non-linear flow enforced
LD_PRELOAD=./libhack.so ./normal_program
```

Output:

```
[Constructor Hook] --> Code running BEFORE main() starts!
[Main] --> Now executing the main function.
```

B3.5 Summary Table (ASCII)

Mechanism	Trigger	Stack Behavior	Flow Type
Signal Handler	Asynchronous OS Event	Pushes new frame on top	Interrupt
setcontext	Explicit Function Call	Overwrites CPU state (Non-local jump)	Jump/Restore
Constructors	Program Load / Linker	Execution occurs before the defined entry point (`main`).	Pre-execution

INTERMEZZO C: GDB and the Fine Art of Debugging

Gene Cooperman

C GDB and the Fine Art of Debugging

C.1 The Core Conflict: Manifestation vs. Root Cause

In debugging, we deal with two distinct entities:

1. **The Manifestation:** This is the symptom. It's the "Segmentation Fault," the "Assertion Failed" message, or the incorrect value printed at the end of a calculation.
2. **The Root Cause:** This is the actual logic error—the uninitialized variable, the "off-by-one" index, or the race condition—that occurred perhaps thousands of lines of code or millions of CPU cycles before the manifestation appeared.

The **Art of Debugging** is the process of tracing the manifestation backward through time and logic to find the root cause. Your goal is to narrow the gap between the moment the error happens and the moment the program tells you something is wrong.

C.2 The Three Levels of Debugging

C.2.1 Level 1: Print Statements (The Quick Look)

- **Method:** Inserting `printf()` or `cout` to track state.
- **The Limitation:** For long-running programs, this produces a "wall of text." It can also cause **Heisenbugs**, where the act of printing changes the program's timing and hides the bug.

C.2.2 Level 2: Assertions and Defensive Programming

- **Method:** Defining what the state *must* be using `assert()`.
- **Goal:** Force the program to stop **immediately after the root cause occurs**.
- **Defensive Syscalls:** Always check return codes.

  Example:

```
if (signal(SIGUSR2, handler) == SIG_ERR) {
    perror("signal"); // Explains WHY it failed
    exit(1);
}
```

C.2.3 Level 3: Using a Good Debugger like GDB

The guide in the next section provides an overview of essential GDB commands and setup instructions for different operating systems. Using even a few of these commands effectively can save hours of manual debugging.

C.3 The Basics of GDB (GNU DeBugger)

C.3.1 Platform Setup

If you are using a current Mac (macOS), **GDB is not readily available** (in particular, it is unsupported on Apple Silicon). You have two options:

- **Virtualization:** Create a Linux virtual machine on your Mac to use GDB.
- **Native Alternative (LLDB):**
  1. Install command line tools: `xcode-select --install`.
  2. Invoke **lldb** instead of gdb (e.g., `lldb a.out`).
  3. Refer to the GDB to LLDB command map for command translations.

C.3.2 Getting Started

- Invoking GDB
  To start GDB with specific program arguments:

  Command: `gdb --args <executable> <args>`

  Example: `gdb --args ./a.out arg1 arg2`

- Starting and Stopping Execution
  - **Set Initial Breakpoint:** `break main`
  - **Launch Program:** `run`
  - **Exit:** `quit`.

C.3.3 Essential Commands

- **Where Am I? (Inspection)**
  - **Threads:** `info threads` (List all threads if this process has more than one thread).
  - **Switch Thread:** `thread 1` (Inspect thread number 1).
  - **Backtrace:** `where` or `backtrace` or `bt` (Displays the call stack).
  - **Frames:** `frame 2` or `f 2` (Move to a specific stack call frame).
  - **Source Code:** `list` or `l` (Displays the code around the current line).
- **Printing & Examining**
  - **Check Type:** `ptype <variable>` (e.g., `ptype argv[0]`).
  - **View Value:** `print <variable>` (e.g., `print argv[0]`).
  - **Listing:** Try `list myfunction` or `list myfile:17` for line number 17.
  - **Finding function names:** `info function SUBSTRING` for `SUBSTRING` a part of a function name. (Useful for discovering fully qualified names in C++.)
- **Breakpoints**
  - **`break`**: Try `break myfunction` or `break myfile:17` for line number 17.
  - **`info breakpoints`**: List all breakpoints.
  - **`disable`**: Try `disable 2` to disable breakpoint number 2.

- `enable`: Try `enable 2` to re-enable breakpoint number 2.
- **Execution Control**
  - `next`: Move to the next line of code.
  - `step`: Step inside a function call.
  - `until`: Continue until reaching a higher line number (useful to escape a "for loop").
  - `finish`: Finish the current function and return to the caller.
  - `continue`: Resume execution until the next breakpoint is hit.

(**HINT:** Most of these commands can take a numeric argument for executing multiple times.)

C.3.4 Advanced Debugging & Tips

- **Special Scenarios**
  - **Infinite Loops:**
    - Execute `run` in GDB, and then type **control-c** (`^C`) to interrupt the program and talk to GDB. Once interrupted, use `where` or `print` to find the loop's location.
  - **Multiprocessing:** Use `set follow-fork-mode child` to switch debugging to the child process after a `fork()` call. The default is to continue to debug the parent process.
- **Productivity**
  - **Auto-complete:** Use the **TAB** key.
  - **Navigation: Cursor keys** work for command history.
  - **Documentation:** Use `help <command>` (e.g., `help continue`).
  - **Browsing versus Command Mode:** Switch back and forth between a mode for browsing the source code, and the normal command mode.
    (**NOTE:** `^X` is the notation for `control-x`, i.e., pressing the control key, and then typing `x`.)
    - To switch to source code browsing mode, do any of:
      `tui enable` or `layout src` or simply `^Xa`
      The cursor keys will then be used to browse the source code.
    - Do ^Xa again to switch back to command mode.
    - While in browsing mode, try `focus cmd` or `focus src` or `^Xo` to choose whether the cursor keys should focus on browsing mode or the normal GDB command history.

C.4 Mastering the GDB Debugger

Advanced GDB use is a fine art that solves problems where other methods fail. After you have played with the basics of GDB, try returning here to understand why GDB is more powerful than most other debuggers.

- **Conditional Breakpoints:** `break [loc] if [cond]` — Stop only when a specific state is met.
- **Watchpoints:** `watch [var]` — Stop the instant a memory location changes.
- **GDBinit Scripts:** Automate your environment setup.

C.5 (Optional) Lab Exercise: Hunting the "Phantom" Write

C.5.1 The Scenario

You are working on a security-sensitive module. A local variable `isAdmin` is being overwritten by "phantom" data. There are no lines of code that explicitly assign a new value to `isAdmin`, yet its value changes during the execution of a loop.

C.5.2 The Buggy Code (`corrupt.c`)

```c
#include <stdio.h>

struct Session {
    int port_buffer[5];
    int isAdmin; // This should stay 0 (False)
};

int main() {
    struct Session user_session;
    user_session.isAdmin = 0;
    printf("Before loop: isAdmin = %d\n", user_session.isAdmin);
    // Simulating data processing
    for (int i = 0; i <= 5; i++) {
        user_session.port_buffer[i] = i + 100;
    }
    printf("After loop: isAdmin = %d\n", user_session.isAdmin);
    if (user_session.isAdmin != 0) {
        printf("SECURITY ALERT: Administrative privileges corrupted!\n");
    }
    return 0;
}
```

C.5.3 The Lab Steps

Step 1: Compilation Compile with debug symbols (`-g`) and disable optimizations (`-O0`) so the compiler doesn't reorder your code.

```
gcc -g -O0 corrupt.c -o corrupt
./corrupt
```

Observe the output. Why did isAdmin change to 105?

Step 2: Start the Debugger Launch the program inside GDB:

```
gdb ./corrupt
```

Step 3: Set the Watchpoint We need to stop the program the instant memory is modified.

1. Type `start` to stop at the beginning of `main`. (`start` is a shortcut for setting a temporary breakpoint on `main` and then doing `run`.)
2. Type `watch user_session.isAdmin`. (A watchpoint on a local variable is automatically deleted when that variable goes out of scope. Here we stay inside `main`, so the watchpoint remains valid for the whole program.)

Step 4: Execution and Analysis Type `continue`. GDB will pause execution and show you the exact line responsible for the change.

C.6 GDB Command Cheat Sheet

Category	Command	Description
Start	`run [args]`	Start execution from the beginning.

Category	Command	Description
Control	`next` / `step`	Skip over / Step into functions.
Inspect	`print [var]`	Show current value of a variable.
Track	`watch [var]`	Stop the world when value changes.
Logic	`break [line] if [cond]`	Conditional breakpoint.
UI	`tui enable`	Show source code window (split view).

C.7 `.gdbinit` Automation Template

Save this as `.gdbinit` in your project folder to automate your setup.

But `.` in `.gdbinit` means this is a "hidden" file. Better yet, I prefer to save this as `gdbinit`, and then do:
`gdb -x gdbinit ./a.out`
If your `./a.out` command takes arguments, then use:
`gdb -x gdbinit --args ./a.out ARG1 ...`

```
# USAGE:  gdb -x THIS_FILE --args EXECUTABLE ARGS ...
# Stop GDB from interrupting gdbinit with questions
set breakpoint pending on
set pagination off
set confirm off

# Stop if about to exit prematurely
break _exit

# Never forget all the GDB commands that you used to get to the bug.
# Afterward, you can go:  gdb -x gdbinit_history
set history save on
set history filename gdbinit_history

# ============================================================================
# === If you're copying this gdbinit file, skip the optional parts, below. ===

# OPTIONALLY, add a new GDB command (useful only for this exercise).
define lab_setup
    start
    watch user_session.isAdmin
    printf "Environment ready. Watchpoint set.\n"
end

# OPTIONAL PRINT STYLES
# Print arrays with one element per line (in case of array of struct)
set print pretty on
# Print actual type of an object (if using C++)
set print object on
# If you would like GDB to to enable colors in output
set style enabled on

# OPTIONALLY, set it to execute immediately:
```

```
break main
run
```

C.8 (Optional) The Mystery Challenge: "The Broken Link"

A linked list of 1000 nodes is corrupted at node 742.

The Discovery Workflow:

1. **Run** until it crashes to find the "victim" ID (742).
2. **Restart** and set a **conditional breakpoint**: `break mystery.c:30 if curr->id == 742`.
3. **Continue** to jump straight to the relevant iteration.
4. **Watch** the pointer: `watch curr->next`.
5. **Continue** to see the exact function that changes the address to `0xBAD105`.

C.9 (Optional) The Art of Debugging Quiz

Instructions: Select the best answer for each question.

1. A program crashes at line 100 due to a NULL pointer assigned at line 20. Which is the manifestation?

- **A)** The NULL pointer assignment at line 20.
- **B)** The Segmentation Fault at line 100.
- **C)** The programmer's failure to initialize.
- **D)** The lack of an assert statement.
- **Hint:** *Think about which event is the observable symptom of the failure.*

2. What is the primary drawback of using print statements in large systems?

- **A)** They cannot display pointer values.
- **B)** They produce too much output, hiding relevant info.
- **C)** They automatically resolve the bug.
- **D)** They slow the program down to a crawl.
- **Hint:** *Consider the "wall of text" problem mentioned in the lecture.*

3. An 'assert' statement is primarily used to:

- **A)** Automatically fix logic errors.
- **B)** Force the program to stop as close to the root cause as possible.
- **C)** Speed up execution.
- **D)** Convert C code to assembly.
- **Hint:** *Think about the goal of "stopping early" in the debugging process.*

4. Why use `perror()` after a failed system call?

- **A)** To hide the error from the user.
- **B)** To provide a human-readable description of why the OS call failed.
- **C)** To restart the system call.
- **D)** To bypass the debugger.
- **Hint:** *Recall the example:* `if (signal(...) == SIG_ERR) { perror("signal"); }`.

5. Which GDB feature is best for finding 'who' corrupted a specific variable?

- **A)** Backtrace

- **B)** Conditional Breakpoint
- **C)** Watchpoint
- **D)** Next command
- **Hint:** *Think about the tool used in the "Phantom Write" lab.*

6. To skip 7,499 iterations of a loop and stop only at 7,500, you should use:

- **A)** `step 7500`
- **B)** `watch i == 7500`
- **C)** `break main.c:50 if i == 7500`
- **D)** `run 7500`
- **Hint:** *This saves you from hitting "continue" thousands of times.*

7. The purpose of a `.gdbinit` script is to:

- **A)** Compile the code.
- **B)** Automate setup, breakpoints, and custom commands.
- **C)** Prevent the program from crashing.
- **D)** Format the hard drive.
- **Hint:** *Consider the "fine art" of scripting your tools for efficiency.*

8. In GDB, the `step` command differs from `next` because:

- **A)** `step` enters function calls; `next` does not.
- **B)** `step` is for hardware; `next` is for software.
- **C)** `next` enters function calls; `step` does not.
- **D)** They are the exact same command.
- **Hint:** *Think about what happens when you reach a line like `printf(...)`. Do you want to go inside the library source code?*

9. Why compile with the `-g` flag?

- **A)** To make the program faster.
- **B)** To include debug symbols mapping machine code to source.
- **C)** To disable all print statements.
- **D)** To compress the executable.
- **Hint:** *Without this, GDB will only show you assembly code and memory addresses.*

10. What is a 'Heisenbug'?

- **A)** A bug that only occurs in German software.
- **B)** A bug that changes behavior when you try to observe it.
- **C)** A bug that is permanently fixed.
- **D)** A bug found only in assembly code.
- **Hint:** *This was mentioned in the section about why print statements can be problematic.*

C.10 Quiz Answer Key & Rationales

Q#	Answer	Rationale
1	**B**	The crash is the symptom (manifestation).
2	**B**	High volume of text creates "noise."

Q#	Answer	Rationale
3	**B**	Early detection prevents error cascading.
4	**B**	Translates OS error codes into English.
5	**C**	Watchpoints monitor memory writes specifically.
6	**C**	Conditional breaks skip irrelevant states.
7	**B**	Scripting saves time and effort during repetitive sessions.
8	**A**	`step` goes deeper into the stack.
9	**B**	Without symbols, GDB cannot link binary to source lines.
10	**B**	Observation (like `printf`) alters timing and memory.