CSCE 313
Introduction to Computer Systems

Texas A&M University • Fall 2026

LAB 1
Debugging with GDB

Student Companion Guide

Step-by-step walkthrough, explanations,
and the reasoning behind every command

This guide accompanies the official lab handout. Where the two differ, the handout governs the grade — but read the notes here first.

August 31, 2026

Contents

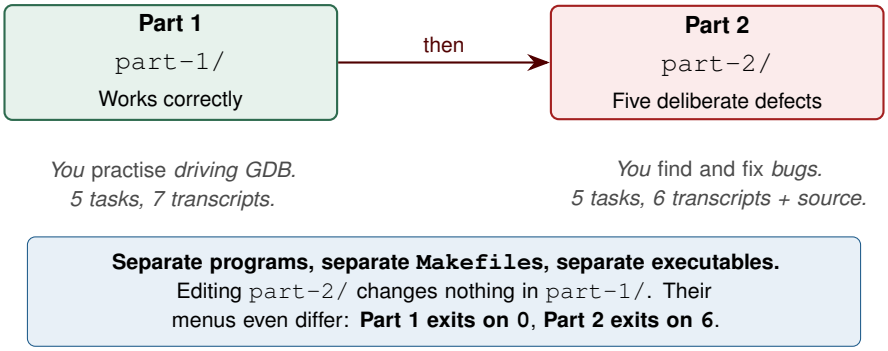
1

How to Use This Guide

This lab teaches you to use a debugger. Not to memorise commands — to *ask a running program questions and get answers*. That skill is the difference between "it crashed, I don't know why" and "it crashed, here is the line, here is the value, here is the fix."

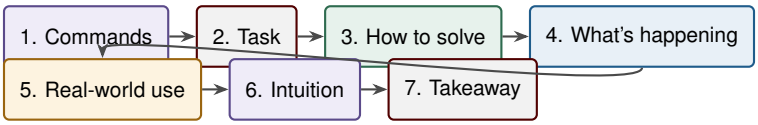
1.1 The shape of the lab

There are two programs, and they are **not** the same program.

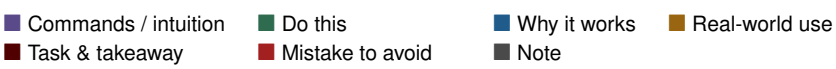


1.2 The structure of every task

Each task in this guide follows the same seven-step flow. Once you have read one, you know how to read them all.



Colour tells you what a box is for, so you can skim:



Two prompts, two languages — read this before anything else

Almost every confusing error in this lab is a command typed at the wrong prompt.

Prompt	You are talking to	Understands
`user@host:~$`	the **shell** (bash)	`ls`, `cd`, `make`, `script`, `gdb`, `exit`
`(gdb)`	**GDB**	`break`, `run`, `print`, `watch`, `quit`

If you type `script` at a `(gdb)` prompt you get `Undefined command: "script"`. If you type `break 35` at a shell prompt you get `break: command not found`. Before typing, glance at the prompt.

2

Setup and Pre-flight

Do this once, before Task 1. Five minutes here saves an hour later.

2.1 Get the repository

```
git clone https://github.com/CSCE-313-FA26/<your-repository>.git
cd <your-repository>
ls
```

You should see `part-1/`, `part-2/`, `deliverables/`.

2.2 Silence the debuginfod prompt, permanently

On its first run GDB asks whether to download debug info for system libraries. The handout says answer `n` every time. Do it once instead:

```
echo "set debuginfod enabled off" >> ~/.gdbinit
```

What `debuginfod` is, and why you don't want it

Your own code has debug symbols because it was compiled with `-g`. System libraries like `libc` and `libstdc++` ship *stripped* — their symbols live on a separate Ubuntu server, and `debuginfod` fetches them on demand.
That is genuinely useful when you are debugging *into* the standard library. For this lab you never need to, and leaving it on makes GDB stall on startup trying to reach the network. `~/.gdbinit` is a file GDB reads on every launch — the same idea as `.bashrc` for your shell.

2.3 Build

```
cd part-1
make
```

Expected output — one line, nothing else:

```
g++ -std=c++17 -g main.cpp bank.cpp -o banking-system
```

If it prints `'banking-system' is up to date`, it is already built. Any *warning* means something is wrong before you have even started.

Why `-g` is the flag the entire lab depends on

Without `-g`, the compiler emits machine code and nothing else. The binary knows there is a function at address `0x18d4`; it does not know that function was called `main`, that it came from `main.cpp`, or that the thing in register `x0` is called `choice`.

`-g` additionally writes **DWARF debug information** into the executable: a line table mapping machine addresses to source file and line, plus a description of every type, variable, and scope. GDB reads that table. Every single thing you are about to do — `break 35`, `print bank`, `ptype Account`, `backtrace` — is a lookup in it.

Strip it away and GDB still works, but it can only show you raw addresses and hexadecimal. That is what debugging a production binary without symbols feels like, and it is why shipping a matching "symbol file" is standard practice in industry.

The single most expensive mistake in this lab

GDB runs the executable, not your source.
If you edit a file and forget `make`, GDB happily loads the *previous* build. Your fix appears to do nothing. You conclude the fix is wrong, change it to something worse, and lose an afternoon. The `Makefile` lists `types.h` as a prerequisite, so editing headers does trigger a rebuild. If you ever doubt it, settle the question: `make clean && make`.

2.4 Recording transcripts — and the path trap

You submit plain-text transcripts, not screenshots. The `script` command records everything that appears in your terminal: the program's output, what you type, and GDB's replies.

The handout's §3.6 command does not work as written

The handout shows:

```
script -q deliverables/part-1-task-1.txt
gdb ./banking-system
```

Those two cannot both work from the same directory. `banking-system` lives in `part-1/`, but `deliverables/` is at the *repository root*. From inside `part-1/` you get:

```
script: cannot open deliverables/part-1-task-1.txt: No such file or directory
```

Use `../deliverables/` instead. Every command in this guide already does.

The pattern, every time:

```
$ script -q ../deliverables/part-1-task-1.txt # shell:  start recording
$ gdb ./banking-system # shell:  launch gdb
(gdb) ... # gdb:  do the task
(gdb) quit # answer y
$ exit # shell:  STOP recording
```

Forgetting `exit` truncates the file

`script` stops recording when the *shell* it launched exits — not when GDB quits. Miss the final `exit` and your transcript ends mid-session. Always verify:

```
tail -3 ../deliverables/part-1-task-1.txt
```

The last line must read `Script done on ...`. No `Script done`, no credit.

One task, one fresh GDB session

Leftover breakpoints from a previous task change what you see. If you reuse a session you will get:

```
Note: breakpoint 1 also set at pc 0xaaaaaaaaa18d4.
Breakpoint 5 at 0xaaaaaaaaa18d4: file main.cpp, line 30.
```

That `Note:` is GDB telling you that you now have two breakpoints at the same address — both will fire, and your `info breakpoints` output will show seven entries where the grader expects three. Quit and relaunch between tasks. (`delete` with no arguments clears all breakpoints if you really need to stay in one session.)

3

Part 1 — Learning to Drive GDB

`part-1/` works correctly. Nothing here is broken. The point is to build fluency with the tool on a program that behaves, so that in Part 2 you can concentrate on the bug instead of on the debugger.

```
cd part-1
make
```

The program you are debugging

`part-1` is about 120 lines across three files. You need to know four things about it:

`types.h`	`struct Account {int id; double balance; bool active;}` and `class Bank`, which holds `Account accounts[10]` and a `Account* current_account`.
`bank.cpp`	`Bank::Bank()`, `Bank::login(int)`, `Bank::logout()`.
`main.cpp`	`print_menu()` and `main()`, which is a `while (running)` loop around a `switch (choice)`.
The menu	**1** Login, **2** Deposit, **3** Withdraw, **4** View Balance, **5** Upload, **6** Download, **7** Logout, **0** Exit.

The line numbers this lab uses:

Location	Source	Why the lab picks it
`main.cpp:30`	`int main() {`	where `break main` lands
`main.cpp:31`	`Bank bank;`	the object is constructed here
`main.cpp:32`	`bool running = true;`	the loop flag is initialised here
`main.cpp:35`	`print_menu();`	top of the loop — everything is in scope
`main.cpp:44`	`running = false;`	the line that ends the loop
`main.cpp:76`	`if (amount > 0) {`	one line before the balance changes
`bank.cpp:7`	`bool Bank::login(int id) {`	first line of `login`

3.1 Task 1 — Inspecting an object and its type

Commands used in this task

`break N`	`b N`	Stop at line `N` of the current default file
`run`	`r`	Start the program under GDB
`ptype T`	—	Print the full definition of a type
`print e`	`p`	Print the value of a variable or expression
`quit`	`q`	Leave GDB

What this task asks you to do

Stop the program at the top of the menu loop. Ask GDB what an `Account` looks like as a *type*, then what `bank.accounts[0]` holds as a *value*.

How to solve it

From a **shell** prompt inside `part-1/`:

```
script -q ../deliverables/part-1-task-1.txt
gdb ./banking-system
```

At the `(gdb)` prompt:

```
(gdb) break 35
(gdb) run
(gdb) ptype Account
(gdb) print bank.accounts[0]
(gdb) quit
```

Answer `y` to *Quit anyway?*, then at the shell: `exit`.

SUBMIT: `deliverables/part-1-task-1.txt`

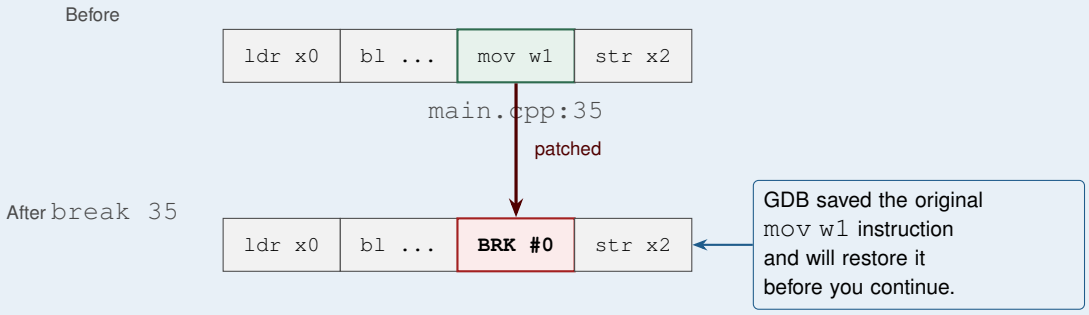
What you should see

```
Breakpoint 1 at 0x18fc: file main.cpp, line 35.
Starting program: .../part-1/banking-system

Breakpoint 1, main () at main.cpp:35
35 print_menu();
(gdb) ptype Account
type = struct Account {
   int id;
   double balance;
   bool active;

   Account(void);
   Account(int);
}
(gdb) print bank.accounts[0]
$1 = {id = -1, balance = 0, active = false}
```

Your address will differ from `0x18fc` — it depends on your CPU architecture and compiler. The line numbers and values should match exactly.

How a breakpoint actually works

This is worth understanding once, because it explains half the mistakes people make with a debugger.



1. You say `break 35`. GDB looks up line 35 in the DWARF *line table* and finds the address of its first instruction.
2. GDB reads the byte(s) at that address, saves them, and writes a **trap instruction** in their place (`BRK` on ARM, `int3` on x86).
3. The CPU reaches it, traps into the kernel, the kernel signals GDB, and GDB hands you the prompt.
4. On `continue`, GDB puts the original instruction back, single-steps over it, re-patches, and lets go.

Two consequences you will meet:

- **The patch is applied to the executable, using line numbers read from your source.** If those two have drifted apart because you edited without rebuilding, GDB will confidently break in the wrong place.
- **If line 35 had no code** — a blank line, a comment, a bare declaration — GDB slides forward to the next line that does, and silently reports *that* number. Always read the echo (`... line 35.`), not what you typed.

`ptype` versus `print`: static versus live

`ptype Account`

Reads the **type description** baked into the binary by `-g`.

Works **before** `run` — it needs no process.

`print bank.accounts[0]`

Reads **bytes out of a live process** and formats them using that type.

Needs `run` first, or: *No symbol in current context.*

`whatis` is `ptype`'s shallow cousin: `whatis bank.accounts[0]` answers `Account` and stops. `ptype` opens the type up.

Why `id = -1` and not 0

`Bank::Bank()` never mentions `accounts`. It does not have to: `accounts` is `Account accounts[10]`, and C++ default-constructs every element, running the default constructor (`id(-1), balance(0), active(false)`) ten times.

The choice of `-1` is deliberate design. `0` is a *valid account ID* in this program — you log in as user 0 in Task 4. If "empty" were marked by `0`, you could not tell an unused slot from account zero. `-1` is a **sentinel**: a value chosen from outside the range of legitimate data precisely so it cannot be mistaken for real data.

Where you will use this for real

`break` + `print` is the bread and butter of every bug you will ever chase. The specific situations:

- **"The function returns the wrong answer."** Break at the top, print the arguments. Half the time the inputs were already wrong and the function is innocent.
- **Reading unfamiliar code.** `ptype` on a class you did not write tells you its real layout in seconds — faster than chasing `#include`s through a codebase.
- **Sentinel values in production.** `-1` for "no ID", `SIZE_MAX` for "not found", `NaN` for "no measurement", `-1` from POSIX system calls. Recognising a sentinel on sight tells you a value is a *status*, not data.
- **Log-based debugging does not scale.** A `printf` tells you what you thought to ask before you ran it. A breakpoint lets you ask anything, after the fact, without rebuilding.

The intuition

A debugger is a **pause button plus an inspector** for a running program.
Normally a program is a black box: you feed it input, you get output, and everything in between happens too fast to see. `break` freezes it at a moment you choose. `print` lets you reach inside and read any variable at that instant. `ptype` is the blueprint; `print` is the photograph of one particular object built from it.
You are not reading the code and imagining what it does. You are watching it.

One extra command, worth knowing now

```
(gdb) ptype /o Account
/* offset | size */ type = struct Account {
/*      0 |    4 */ int id;
/* XXX  4-byte hole  */
/*      8 |    8 */ double balance;
/*     16 |    1 */ bool active;
/* XXX  7-byte padding  */
                       /* total size (bytes): 24 */
                     }
```

The `/o` modifier prints byte offsets and explicitly names the padding. Keep this output — Task 3 asks you to read those same 24 bytes by hand, and this is the answer key.

Task 1 troubleshooting

Message	Cause and fix
`No symbol "bank" in current context`	You ran `print` before `run`. Locals need a live stack frame.
`No symbol table is loaded`	You launched bare `gdb`. Use `gdb ./banking-system`.
Breakpoint lands on a different line	Bare `break 35` resolves against GDB's *current default file*. Use `break main.cpp:35`.
Menu prints before the breakpoint	You are in `part-2/`. Check `pwd`.
`script: cannot open ...`	The §3.6 path trap. Use `../deliverables/`.

Takeaway

`-g` puts a map in the binary; GDB reads that map. `break` patches a trap instruction at an address the map gave it. `ptype` asks about types and works before the program starts; `print` asks about values and needs a live process. A freshly constructed `Account` reads `{id = -1, balance = 0, active = false}`, and `-1` is a sentinel, not an accident.

3.2 Task 2 — Several breakpoints, and watching a variable

Commands used in this task

`break f`	`b f`	Break on entry to function `f`, after its prologue
`break file:N`	—	Break at line `N` of a *named* file
`info breakpoints`	`i b`	List every breakpoint and watchpoint
`watch e`	—	Stop the instant expression `e` *changes value*
`continue`	`c`	Resume until the next stop
`delete N`	—	Delete breakpoint `N` (bare `delete` = all)

What this task asks you to do

Two transcripts. First, set breakpoints three different ways and read GDB's breakpoint table. Then set a **watchpoint** on the loop-control variable and let the program run until choosing *Exit* changes it.

This is two `script` sessions, not one

The handout shows one continuous GDB session producing two files. `script` cannot do that — recording stops only when the *shell* exits, and your shell is busy running GDB.
So: two separate recordings, and you **re-set the breakpoints** in the second one. A fresh GDB process knows nothing about the previous one.
That is also why the second half cannot start from a truly empty session: with no breakpoints, `run` never stops, the program sits at the menu, and you never get a `(gdb)` prompt to type `watch` at.

Part 2-1 — three breakpoints

How to solve it

```
script -q ../deliverables/part-1-task-2-1.txt
gdb ./banking-system
```

```
(gdb) break main
(gdb) b 35
(gdb) b bank.cpp:7
(gdb) info breakpoints
(gdb) quit
```

`y`, then `exit`.

SUBMIT: `deliverables/part-1-task-2-1.txt`

```
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000aaaaaaaa18d4 in main() at main.cpp:30
2 breakpoint keep y 0x0000aaaaaaaa18fc in main() at main.cpp:35
3 breakpoint keep y 0x0000aaaaaaaa2484 in Bank::login(int) at bank.cpp:7
```

Exactly three rows, numbered 1–3, with no `already hit` lines. Anything else means you reused a session — quit and start over.

Three ways to name a place, and why they differ

Form	Example	What GDB does
by symbol	`break main`	Find the symbol, **skip its prologue**
by line	`b 35`	Line 35 of the *current default file*
by file+line	`b bank.cpp:7`	Line 7 of an *explicitly named* file

`break main` is not `break 30`. A function breakpoint deliberately lands *after the prologue* — the handful of instructions that set up the stack frame and store the arguments. GDB skips them so that arguments and locals are readable when you stop. Break at the literal first instruction and `print` would show garbage.

Why `bank.cpp:7` reports `Bank::login(int)`. GDB resolved line→address, then asked which function owns that address, and printed its **demangled** signature. The symbol actually stored in the binary is `_ZN4Bank5loginEi`: `4Bank`, `5login`, then `i` for the `int` parameter. C++ encodes parameter types into symbol names so that overloads can coexist in one symbol table — which is precisely why naming a function by file and line is useful. If `login` had two overloads, `break Bank::login` would be ambiguous; a line number is not.

See the prologue point for yourself — run this separately

This is not part of the deliverable, but it is the best 30 seconds in Part 1.

```
(gdb) break main
(gdb) run
(gdb) print bank.accounts[0]
$1 = {id = -5628, balance = 2.3363274982906867e-307, active = 48}
(gdb) print running
$2 = 247
(gdb) next
31 Bank bank;
(gdb) next
32 bool running = true;
(gdb) next
34 while (running) {
(gdb) print bank.accounts[0]
$3 = {id = -1, balance = 0, active = false}
(gdb) print running
$4 = true
```

`break main` stops **before `Bank bank;` runs**. The stack frame is allocated, but nothing has been written into it. So you are reading **leftover bytes** from whatever used that stack region last.
`active = 48` and `running = 247` are `bool`s holding a byte that is neither 0 nor 1 — GDB prints the raw number because the value is not a legal `bool`. `balance = 2.34e-307` is a denormal double, the classic signature of "these bytes were never a floating-point number."
Your numbers will be different from these. That is the point: uninitialised memory is whatever happened to be there. This is exactly why the lab breaks at line 35 and not at `main`, and it is the best argument for initialising your variables that this course will give you.

Part 2-2 — the watchpoint

How to solve it

```
script -q ../deliverables/part-1-task-2-2.txt
gdb ./banking-system
```

```
(gdb) break main
(gdb) b 35
(gdb) b bank.cpp:7
(gdb) run
(gdb) watch running
(gdb) c
(gdb) c
(gdb) c
```

On the **third** `c` the menu prints and waits — type `0` and press Enter. The watchpoint fires. Then `quit`, `y`, `exit`.

SUBMIT: `deliverables/part-1-task-2-2.txt`

`c`	Stops at	Why
1st	watchpoint	Line 32 runs `running = true`, overwriting the garbage
2nd	Breakpoint 2	`main.cpp:35`, top of the loop
3rd	*menu*	Program waits for input — type `0`
—	watchpoint	Line 44 runs `running = false` ← **the deliverable**

```
(gdb) watch running
Hardware watchpoint 4: running
(gdb) c
Continuing.

Hardware watchpoint 4: running
Old value = 247
New value = true
main () at main.cpp:34
34 while (running) {
(gdb) c
Breakpoint 2, main () at main.cpp:35
(gdb) c
=== Banking System Menu ===
...
Enter choice: 0

Hardware watchpoint 4: running
Old value = true
New value = false
main () at main.cpp:45
45 break;
```

That first firing is not an error

You set the watchpoint while `running` still held stack garbage, so GDB faithfully reports `247 → true` when line 32 initialises it. Leave it in your transcript.
If your garbage byte happened to already be `1`, that write changes nothing and the first firing **will not happen at all**. Also fine — pure luck of the stack. Either way, keep pressing `c` until you reach the menu.

Breakpoint asks about *code*; watchpoint asks about *data*

Breakpoint — "stop *here*"
You must already know *which line* to suspect.
Implemented by patching a trap instruction into the code.

Watchpoint — "stop *when this changes*"
You need only know *which variable* is wrong.
Implemented by the CPU's **debug registers** watching an address.

Notice what you never had to do: you never had to know that **line 44** was the line that ends the loop. You asked a question about data and GDB found the code for you. That inversion is the whole idea, and it is how you find a variable being clobbered by code you have not read.
Where it stopped matters too. The report says `main.cpp:45`, not 44. Line 44 is `running = false;`. A watchpoint fires *after* the store completes — the hardware traps on the write, so by the time you have control the new value is already in memory. GDB can still show you both

because it cached the old value when you set the watchpoint.

Hardware versus software watchpoints — check which you got

Look at the line right after `watch running`:

`Hardware watchpoint 4:`	**Good.** The CPU's debug registers trap on writes to that address. Effectively free.
`Watchpoint 4:`	**Software fallback.** GDB single-steps the *entire program*, re-reading the value after every instruction. `c` may take minutes.

x86 always gives hardware. On ARM (Apple Silicon under VMware Fusion) it depends on whether the guest sees the debug registers — on a standard Ubuntu Server 26.04 arm64 VM it does, and `info breakpoints` will show the type as `hw watchpoint`. If yours says otherwise, the task still works, just slowly.

Scope: watchpoints can expire

`running` is a local in `main`'s frame. When `main` returns GDB prints:

```
Watchpoint 4 deleted because the program has left the block in
which its expression is valid.
```

Normal, not an error. Contrast Task 5, where you watch `bank.accounts[0].balance` — an expression that stays valid, so the watchpoint survives.

Where you will use this for real

Watchpoints are the tool for "who is corrupting my data?" — the hardest class of bug there is, because the symptom appears nowhere near the cause.

- **A field changes and nobody admits to it.** A config value resets, a counter jumps, a pointer becomes null. In a 200k-line codebase you cannot read every write. Watch the address; the debugger names the culprit.
- **Memory corruption from somewhere else entirely.** A buffer overflow in module A silently rewrites a variable in module B. A watchpoint on B's variable catches A red-handed. You will see exactly this shape of bug in Part 2 Task 3.
- **Race conditions.** Watch a shared variable; when it changes unexpectedly, `backtrace` tells you which thread did it.
- **Hardware and embedded work.** Watchpoints are how you catch a device register being written by the wrong driver.

`info breakpoints` matters more than it looks. In a long session you accumulate breakpoints and forget them. Stale breakpoints are the #1 listed way to lose points on this lab, and in real work they are how you end up debugging a program that no longer behaves like the one you shipped.

The intuition

A breakpoint is a **tripwire across a path**: it fires when execution walks past a place you chose. A watchpoint is a **burglar alarm on a specific object**: it fires when that object is touched, no matter who touched it or from where.
When you know the *where*, use a breakpoint. When you only know the *what* — "this value is wrong and I have no idea who wrote it" — use a watchpoint. Most hard bugs are the second kind, which is why this is the single most under-used feature in GDB.

Takeaway

Three ways to name a location: symbol (prologue-skipped), bare line (default file), file+line (explicit). `info breakpoints` is your inventory — keep it clean. A watchpoint inverts the question from "stop here" to "stop when this changes," finds the line for you, and reports old and new values after the write lands. Watchpoints on locals expire with their scope.

3.3 Task 3 — Stepping, and reading the same bytes three ways

Commands used in this task

`step`	`s`	Run one source line, **entering** calls that have debug info
`next`	`n`	Run one source line, **stepping over** calls
`finish`	—	Run until the current function returns
`list`	`l`	Show source around the current line
`x/NFU a`	—	Examine memory: `N` units, format `F`, unit size `U`

What this task asks you to do

Two transcripts. First, step *into* `print_menu()`, look at the source around you, and run back out. Then read one `Account` object three different ways — as raw bytes, as words, and as a structure — and compare.

Part 3-1 — step, list, finish

How to solve it

```
script -q ../deliverables/part-1-task-3-1.txt
gdb ./banking-system
```

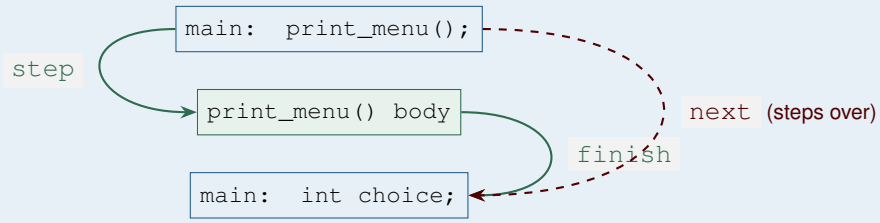
```
(gdb) b 35
(gdb) run
(gdb) step
(gdb) list
(gdb) finish
(gdb) quit
```

`y`, then `exit`.

SUBMIT: `deliverables/part-1-task-3-1.txt`

`step` versus `next` — and the trap in `step`

Both advance exactly one *source line*. The difference is what happens when that line contains a function call.



The trap: `step` only descends into functions **compiled with debug info**. All of `libstdc++` ships without it, so `step` on a line containing `std::cout « ...` behaves exactly like `next` and appears to do nothing interesting. Students conclude `step` is broken. It is not — it is refusing to drop you into assembly with no source to show.
It works here because `print_menu()` is your own `-g` code.
`finish` runs to the end of the current frame and prints `Run till exit from #0 ...`, plus `Value returned is $1 = ...` for a non-void function. It is the escape hatch for when you `step`ped somewhere you did not mean to go.

Part 3-2 — three views of 24 bytes

How to solve it

Continuing in the *same* GDB session is fine here, but start a new `script` recording. Cleanest is a fresh session:

```
script -q ../deliverables/part-1-task-3-2.txt
gdb ./banking-system
```

```
(gdb) b 35
(gdb) run
(gdb) x/24xb &bank.accounts[0]
(gdb) x/6xw &bank.accounts[0]
(gdb) print bank.accounts[0]
(gdb) quit
```

SUBMIT: `deliverables/part-1-task-3-2.txt`

Decoding `x/NFU`

Part	Means	Values used here
`N`	how many units	`24`, `6`
`F`	format	`x` hex, `d` decimal, `s` string, `c` char, `i` instruction
`U`	unit size	`b` byte (1), `h` half (2), `w` word (4), `g` giant (8)

So `x/24xb` is "24 bytes, in hex" and `x/6xw` is "6 four-byte words, in hex." $24 \times 1 = 6 \times 4 = 24$ bytes **— the same region, read at two different granularities.**

What the bytes mean

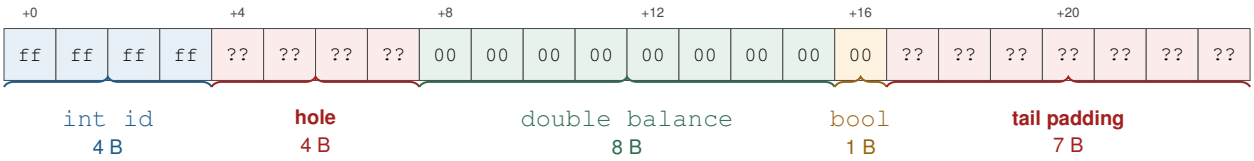


Figure 3.1: The 24 bytes of one `Account`. `ff ff ff ff` is `id = -1` in two's complement. The eight zero bytes are `balance = 0.0`. The bytes marked `??` are **padding** — never written by the constructor, so they hold whatever was there before, and they differ on every machine and every run.

Why the compiler left two holes

Hardware reads memory fastest when a value sits at an address that is a multiple of its size — this is **alignment**. A `double` wants an 8-byte boundary.

- `id` takes offsets 0–3.
- `balance` cannot start at offset 4 (not a multiple of 8), so the compiler inserts a **4-byte hole** and starts it at 8.
- `active` takes offset 16.
- The struct's own alignment is 8 (its strictest member), so its size must be a multiple of 8. Hence **7 bytes of tail padding**, total 24.

The tail padding is not waste — it is what makes `accounts[1]` start at a correctly aligned address too. Without it, every second element of the array would be misaligned.
Reorder the members (`double`, `int`, `bool`) and the same three fields fit in 16 bytes instead of 24 — a 33% saving. On a struct you allocate a million times, that is real memory. This is a genuine optimisation technique, and `ptype /o` is how you find candidates.

Two things students get wrong here

1. "My padding bytes are different from my partner's." Correct. Both of you are right. Padding is never initialised, so it holds stack residue. *That is the lesson* — if you ever compare two structs with `memcmp`, the padding will make identical objects compare unequal.
2. "I can't see the endianness the handout mentions." Also correct. With `id = -1` (`ff ff ff ff`, a palindrome) and `balance = 0.0` (all zeros), there is nothing to see. To actually observe byte order, log in and deposit first, then look at a non-zero balance:

```
(gdb) x/8xb &bank.accounts[0].balance
0x...: 0x00 0x00 0x00 0x00 0x00 0x00 0x59 0x40
```

100.0 as an IEEE-754 double is `0x4059000000000000`. On a **little-endian** machine the *least* significant byte is stored first, so it appears reversed: the `40` and `59` come last. Every x86 and every ARM64 machine you will use is little-endian; network protocols are big-endian, which is why `htons()` and `ntohl()` exist.

Where you will use this for real

`x` is for when the structured view is lying to you or is not available.

- **Parsing binary formats.** Network packets, file headers, firmware images. `x/32xb` on a buffer is how you check whether the bytes on the wire match the spec.
- **Struct layout mismatches.** Two components compiled with different packing or alignment settings will disagree about where a field lives. The structured view shows plausible nonsense; the raw view shows the truth.
- **Corrupted objects.** When a pointer is wild, `print` shows garbage interpreted as a type. `x` shows you the actual bytes, which often reveals what *did* get written there — an ASCII string in the middle of a struct tells you a `strcpy` overran.
- **Cache and memory optimisation.** Field reordering to shrink hot structs is standard in game engines, databases, and kernels.
- **`x/i $pc`** disassembles at the program counter — how you debug with no source at all.

The intuition

The same bytes have no inherent meaning. **A type is an interpretation you impose on memory.**
`x/24xb` is the negative: raw, uninterpreted. `x/6xw` is the same negative at a coarser grain. `print` is the developed photograph — GDB applying the `Account` type to decide that bytes 0–3 are a signed integer and bytes 8–15 are a float.
The structured view is easier to read but *hides things*: the padding vanishes, the byte order vanishes, and if the object is corrupt the structured view will confidently show you garbage that looks like data. When you stop trusting the structured view, drop to `x`.

Takeaway

`step` enters calls that have debug info; `next` steps over; `finish` runs out. `x/NFU` reads memory at any granularity. One `Account` is 24 bytes: 4 for `id`, a 4-byte alignment hole, 8 for `balance`, 1 for `active`, 7 bytes of tail padding. Padding is uninitialised and differs between machines — that is expected, not a bug.

3.4 Task 4 — Reading a call stack

Commands used in this task

`backtrace`	`bt`	Show the chain of calls that led here
`bt full`	—	Same, plus every frame's local variables
`frame N`	`f N`	Switch context to frame `N`
`up` / `down`	—	Move one frame toward `main` / away from it
`info locals`	—	Locals in the currently selected frame

What this task asks you to do

Break on `Bank::login`, drive the program there through the menu, and read the call stack that got you there.

How to solve it

```
script -q ../deliverables/part-1-task-4.txt
gdb ./banking-system
```

```
(gdb) b bank.cpp:login
(gdb) run
```

The menu appears. Choose `1`, then enter user ID `0`. GDB stops on entry to `Bank::login`. Then:

```
(gdb) backtrace
(gdb) bt full
(gdb) quit
```

SUBMIT: `deliverables/part-1-task-4.txt`

"I ran it and nothing stopped"

`login` is not called until you interact with the menu. `run` starts the program, which prints the menu and waits. You must choose `1` and enter `0`. Students who expect the breakpoint to fire immediately assume the `bank.cpp:login` syntax failed. It did not.

What a stack frame is

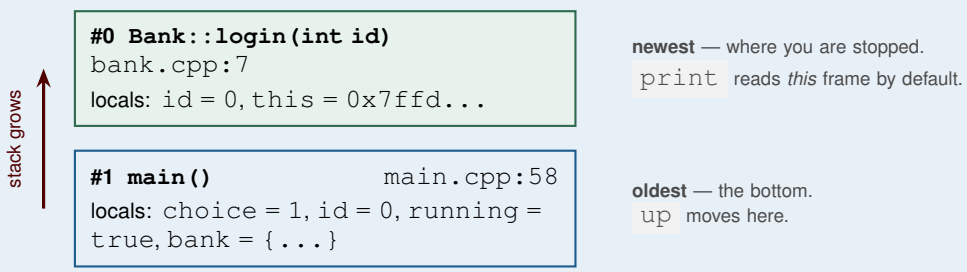


Figure 3.2: A call stack. Each frame holds one function call's arguments, local variables, and the return address that says where to resume.

Every time a function is called, the CPU pushes a **frame** onto the stack holding its arguments, its locals, and the return address. When it returns, the frame is popped. `backtrace` walks that chain from newest to oldest.
Frame #0 also has a `this` pointer, because `login` is a member function. Try `print *this` and `print this->accounts[0]`.
`up` and `down` change what `print` means. After `up`, `print id` refers to `main`'s local `id`, not `login`'s parameter. They happen to hold the same value here, which makes it a good place to notice that they are different variables.

Where you will use this for real

`bt` is the first thing you type on any crash, forever.

- **Every crash report you will ever read** is a backtrace. Java stack traces, Python tracebacks, browser console errors, iOS crash logs, the kernel oops — same idea, same reading skill.
- **"This function got a bad argument — who called it?"** A library function fails on nonsense input. `bt` names the caller that passed it.
- **Production postmortems.** A *core dump* is a frozen image of a crashed process. `gdb ./prog core` plus `bt` tells you where it died, hours later, on a machine you never logged into.
- **Performance profiling.** A profiler is essentially a tool that samples the backtrace thousands of times a second and counts which frames appear most.
- **`bt` on a hang.** Attach to a stuck process, `bt`, and you see exactly which call it is blocked in.

The intuition

A backtrace answers **"how did I get here?"** — the trail of breadcrumbs from `main` to the current instruction.
If `print` tells you *what* is wrong, `bt` tells you *who did it*. A bad value in a function is rarely that function's fault; usually someone passed it in. The stack is the record of who.
In Part 2 you will use this twice: once to find the line that dereferences a null pointer, and once to see the same frame repeated tens of thousands of times.

Takeaway

Every call pushes a frame: arguments, locals, return address. `bt` lists them newest-first; `bt full` adds the locals. `up` / `down` change which frame `print` reads. `bt` is the universal first move after any crash.

3.5 Task 5 — Interrupting a running program

Commands used in this task

`Ctrl+C`	—	Interrupt the running program, return to the `(gdb)` prompt
`watch e`	—	(again) stop when `e` changes
`next`	`n`	Advance one line

What this task asks you to do

Take control of a program that is *already running* — no breakpoints set in advance. Then watch a balance and step until it changes.

How to solve it

```
script -q ../deliverables/part-1-task-5.txt
gdb ./banking-system
```

This task interleaves **GDB commands** with **input to the program**. Only type a GDB command after you see a `(gdb)` prompt; if the last thing on screen is `Enter choice:` or `Enter amount...`, the *program* is listening.

You type	Read by	What happens
`run`	GDB	menu prints
`1` then `0`	**program**	`Logged in as user 0`, menu again
`Ctrl+C`	—	`SIGINT`; you get a `(gdb)` prompt
`b main.cpp:76`	GDB	breakpoint set
`c`	GDB	resumes; still waiting at `Enter choice:`
`2`	**program**	`Enter amount to deposit:`
`100.00`	**program**	**Breakpoint 1, main.cpp:76** — prompt returns
`watch bank.accounts[0].balance`	GDB	`Hardware watchpoint 2: ...`
`next` (repeat)	GDB	fires: `Old value = 0 / New value = 100`
`quit`, `y`, `exit`		

Typing `watch` right after `c` ends the program

```
(gdb) c
Continuing.
watch bank.accounts[0].balance ← no (gdb) prompt -- this went to the program
[Inferior 1 (process 124730) exited normally]
(gdb) watch bank.accounts[0].balance
No symbol "bank" in current context.
```

After `c` the program resumes the `read()` it was blocked in — it is still waiting for your **menu choice**. The `watch` line went to `std::cin » choice`. Extracting an `int` from "watch..." fails, a failed extraction sets `choice` to `0`, and `0` is Exit. The process is gone before you ever get a real prompt.

You must choose `2` and enter `100.00` *first*; the breakpoint at line 76 is what gives you the prompt to set the watchpoint at.

SUBMIT: `deliverables/part-1-task-5.txt`

The most common way to lose points on this lab

Your transcript **must contain** the watchpoint firing:

```
Hardware watchpoint 2: bank.accounts[0].balance
Old value = 0
New value = 100
```

Stopping your `next` sequence before this appears is the single most common error. Keep pressing `next` until you see the words **Old value**. If you find yourself back at the menu prompt without having seen it, something went wrong — start the task over.

What `Ctrl+C` actually does

`Ctrl+C` sends **SIGINT** to the foreground process group. Without a debugger, the default handler kills the program — that is why it stops runaway commands in a normal terminal. Under GDB, GDB intercepts the signal and stops the inferior *wherever it happens to be*. And where it happens to be, at that moment, is **blocked inside a `read` system call waiting for your keyboard input**.
So if you type `bt` right after interrupting, you will see library internals before you see `main`:

```
#0 0x... in __GI___libc_read () at ../sysdeps/unix/sysv/linux/read.c:26
#1 0x... in std::__basic_file<char>::xsgetn(...)
#2 0x... in std::basic_filebuf<...>::underflow()
...
#7 0x... in main () at main.cpp:38
```

That is expected, not a bug. You are seeing the guts of `std::cin » choice`. The frames with no source line are library code compiled without `-g`. On ARM the top frame is `__internal_syscall_cancel (...) at ./nptl/cancellation.c:40` followed by `warning: ./nptl/cancellation.c: No such file or directory` — also expected; it means libc's *source* is not installed, which you do not need.

Watchpoints fire during `next`, even over calls

A common wrong assumption: "`next` steps *over* calls, so if the write happens inside a function I will miss it."
You will not. A watchpoint is checked by the hardware on every write to that address, regardless of which function performs it or whether you stepped over it. GDB is watching the *data*, not the line.
In Part 1 the write is on line 77 (`bank.current_account->balance += amount;`) directly in `main`, so it fires within a step or two.

If `Ctrl+C` does not reach GDB

The signal must go to GDB, not to your terminal emulator or editor. In VS Code's integrated terminal over Remote-SSH it is occasionally swallowed. A plain `ssh` session from a real terminal is the reliable fallback. If it still misbehaves, `interrupt` typed at the `(gdb)` prompt does the same job.

Where you will use this for real

Attaching to a program that is already misbehaving is a core production skill.

- **The hang.** A server stops responding. You cannot restart it — restarting destroys the evidence. `gdb -p <pid>` attaches to the live process, `bt` shows which call it is stuck in: a deadlocked mutex, a socket read that never returns, an infinite loop.
- **The slow leak.** A process is at 40 GB after three days. You cannot reproduce it locally. Attach, inspect, detach.
- **Intermittent bugs.** The failure happens once an hour. You cannot sit at a breakpoint for an hour; you wait for the symptom, then interrupt.
- **`detach`** lets you walk away and leave the process running — so you can inspect a production service without killing it.

Interrupting is also how you answer "what is it *doing* right now?" for a program that appears to be doing nothing.

The intuition

Everything before this task was **planned** debugging: you knew where to look, so you set a breakpoint in advance. This task is **reactive** debugging: the program is misbehaving *now*, you had no plan, and you seize control mid-flight.
Real bugs rarely announce themselves in time for you to set a breakpoint. `Ctrl+C` is how you get from "something is wrong" to "I am inside it, looking around" without restarting and losing the state that made it go wrong.

Takeaway

`Ctrl+C` sends SIGINT; GDB catches it and stops the program wherever it is — usually blocked in a `read`, so the first backtrace is full of library frames. You can set breakpoints and watchpoints after interrupting and continue. Watchpoints fire on writes regardless of `next` versus `step`. **Your transcript must show `Old value` / `New value`.**

End of Part 1 — verify before moving on

```
ls -l ../deliverables/
for f in ../deliverables/part-1-*.txt; do
  echo -n "$f: "; tail -1 "$f" | grep -c "Script done"
done
```

You should have seven files, each ending in `Script done`: `part-1-task-1`, `-2-1`, `-2-2`, `-3-1`, `-3-2`, `-4`, `-5`. A `0` from that loop means a truncated transcript — redo it.

4

Part 2 — Finding and Fixing Five Defects

```
cd part-2
make
pwd # must end in /part-2 -- check this every time you start GDB
```

Read this before you start

`part-2` is a different program, not a broken `part-1`

	part-1	part-2
`MAX_ACCOUNTS`	10	**1,000,000**
`Bank::accounts`	`Account accounts[10]` (inline array)	`Account*` from `new Account[...]`
`Bank` itself	`Bank bank;` on the stack	`new Bank()` on the heap
`Account`	24 bytes, 3 fields	**40 bytes**, 5 fields
Deposit logic	inline in `main`	`Bank::deposit()` method
Exit key	**0**	**6**

Read the menu the program prints. Do not assume.

Fix in order — they compound

1. Until Task 1 is fixed, every deposit crashes, so Tasks 3 and 4 cannot be reached at all.
2. `MAX_TRANSACTIONS` ships at **2**. **Task 3 depends on that value.** Task 4 tells you to raise it to 100. Raise it early and Task 3 will not reproduce.
3. If Task 1 is fixed but Task 2 is not, each level of the runaway recursion also *allocates* a transaction array. With `MAX_TRANSACTIONS = 100` that is roughly 240 MB of heap churn before the stack gives out — your VM may thrash instead of crashing cleanly.

The one idea behind four of the five bugs

`Account` owns a raw pointer, `Transaction* transactions`. In C++, a class that owns raw memory must say what happens when it is **created**, **copied**, and **destroyed**. That expectation has a name: the **Rule of Three**.

The compiler will tell you this outright. From `part-2/`, on the shipped code:

```
g++ -std=c++17 -Weffc++ -c bank.cpp -o /dev/null
```

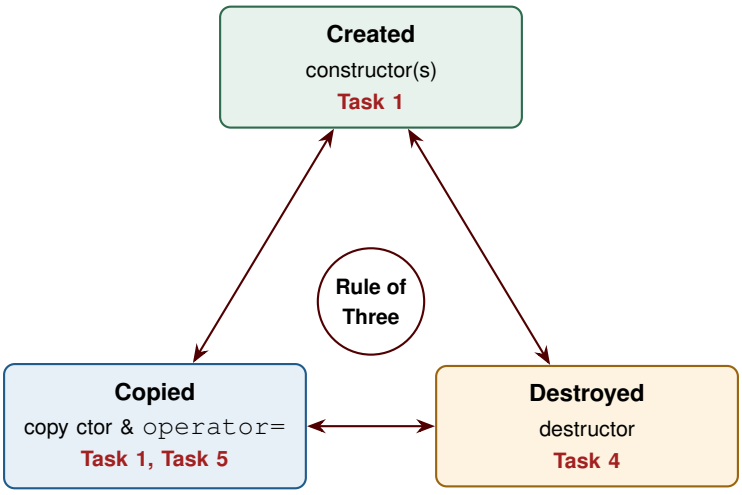


Figure 4.1: Declare one of these three and you almost certainly need all three. Tasks 1, 4, and 5 are the three corners of the same missing design.

```
warning: 'class Account' has pointer data members [-Weffc++]
warning: but does not declare 'Account(const Account&)' [-Weffc++]
warning: 'Account::transactions' should be initialized in the
      member initialization list [-Weffc++]
```

Task 2 is the odd one out — it is a logic bug, not an ownership bug.

The defects are labelled in the source

Every defect carries a comment naming its task, e.g. `// Task 3: Buffer overflow potential`. `grep -n "Task" part-2/*` finds all five in one command.
Do not let that be your answer. The grade is in the transcripts — reproducing the failure, capturing the evidence, and explaining the mechanism. Knowing *where* the bug is teaches you nothing about *why* it fails, and the "why" is what transfers to your own code.
One of those comments is also **wrong**. See Task 5.

4.1 Task 1 — Uninitialized transaction array

Commands used in this task

`run`	`r`	Run until it crashes
`backtrace`	`bt`	Find the line that crashed and who called it
`print e`	`p`	Inspect the pointer that was dereferenced

What this task asks you to do

Log in as user 0 and deposit 100. It segfaults. Find out why the `transactions` pointer is not valid at that moment — and find **both** places that need fixing.

How to solve it

```
script -q ../deliverables/part-2-task-1.txt
gdb ./banking-system
```

```
(gdb) run
```

Choose `1`, user ID `0`, then `2`, amount `100`. It crashes.

```
(gdb) backtrace
(gdb) print transactions
(gdb) print transactionCount
(gdb) quit
```

SUBMIT: `deliverables/part-2-task-1.txt`

What you should see

```
Program received signal SIGSEGV, Segmentation fault.
Account::addTransaction (this=..., amount=100, desc=0x... "deposit")
   at types.h:59
59 transactions[transactionCount].accountId = id;
(gdb) backtrace
#0 Account::addTransaction (...) at types.h:59
#1 Bank::deposit (this=..., amount=100) at bank.cpp:42
#2 main () at main.cpp:62
(gdb) print transactions
$1 = (Transaction *) 0x0
```

`0x0` is the whole answer: the pointer is **null**, not garbage.

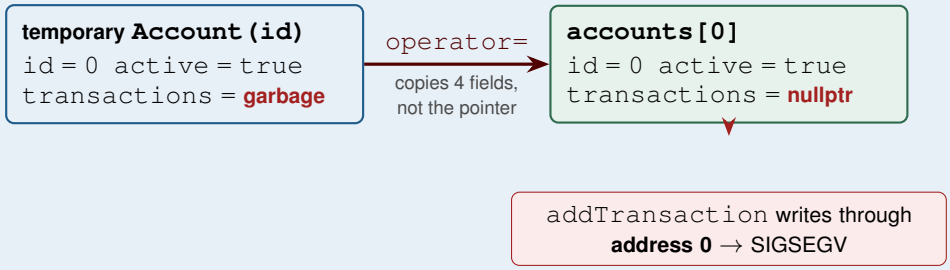
Why it is null, step by step — this is subtler than it looks

The obvious story is "the constructor never set it, so it holds garbage." That story is *wrong*, and following it will send you to the wrong fix.
`Bank::login` does:

```
accounts[id] = Account(id); // bank.cpp:20
```

That is **copy-assignment of a temporary**, not construction. Trace it:

1. `accounts[id]` already exists — `new Account[MAX_ACCOUNTS]` default-constructed it, and `Account()` *does* set `transactions = nullptr`.
2. The temporary `Account(id)` is built. **Its** `transactions` is genuinely indeterminate garbage — `Account(int)` never touches it.
3. `operator=` runs. It calls `delete[] transactions` on the *target* — which is `nullptr`, so that is legal and does nothing. It then copies four fields and **never assigns `transactions`**.
4. So `accounts[id].transactions` is *still* `nullptr`.



The handout's hint — *"there is more than one such path in types.h"* — means exactly this: an owning pointer must be established in **every constructor** and **every assignment**.

The fix

Two places in `types.h`:

```
37 Account(int i): id(i) {
38    active = true;
39    balance = 0;
40    transactionCount = 0;
41    transactions = new Transaction[MAX_TRANSACTIONS]; // Task 1 (a)
42 }
```

```
44 Account& operator=(const Account& other) {
45    if (this != &other) {
46       delete[] transactions;
47       id = other.id;
48       balance = other.balance;
49       active = other.active;
50       transactionCount = other.transactionCount;
51       transactions = new Transaction[MAX_TRANSACTIONS]; // Task 1 (b)
52    }
53    return *this;
54 }
```

Rebuild and confirm the deposit works:

```
make
./banking-system
```

For the curious — this minimal fix is still latently wrong

`operator=` copies `other.transactionCount` but hands you an *empty* array. The object now claims a count it cannot back up.
It does not matter here, because the temporary always has `transactionCount == 0`. A genuinely correct copy-assignment would deep-copy each element, including `strcpy`ing every `description`. That is the bug you will ship in your own code if you only learn the minimal fix.

Why a null dereference is a *good* bug

Reading or writing address `0x0` always faults, because the operating system deliberately leaves the first page of every process's address space unmapped. That is not an accident — it is

a safety net, so that the extremely common mistake of using a null pointer fails loudly and immediately instead of silently corrupting whatever happened to live at address 0.
Compare with the *garbage* pointer case: had `Account(int)`'s indeterminate pointer been the one dereferenced, it might have pointed at valid mapped memory. The write would have succeeded, corrupted something unrelated, and the program would have failed much later, somewhere else entirely. Task 3 is exactly that scenario.
A crash at `0x0` is the friendliest memory bug you can get.

Where you will use this for real

- **Null-pointer crashes are the most common crash in production C/C++.** The workflow you just used — run, crash, `bt`, print the pointer — is the standard first response, and it usually takes under a minute.
- **The real lesson is initialisation discipline.** Modern C++ answers this with `std::vector` and smart pointers, which cannot be left uninitialised. Rust answers it by refusing to compile. Both exist because of exactly this bug.
- **Partial-copy bugs are everywhere.** Add a field to a class and forget to add it to the copy constructor, and you get a subtly half-copied object. This is a real and frequent source of production bugs, and it is why `= default` and compiler-generated copies are preferred wherever possible.
- **`0x0` in any crash report** — Java NPE, Go nil dereference, iOS `EXC_BAD_ACCESS` at 0 — means the same thing: something was never given a value.

The intuition

An object is not "created" when memory is set aside for it. It is created when every one of its fields has been given a meaning.
`Account(int)` looked complete — it set the id, the balance, the flag, the count. It missed one field, and that field was the one that owned memory. An object with a dangling or null owning pointer is a **half-built object**: it passes every type check, prints plausibly, and detonates the moment anyone uses the part that was never finished.
The Rule of Three is the discipline that prevents half-built objects.

Takeaway

The pointer is `nullptr`, not garbage — because `operator=` copies the other fields and leaves `transactions` at whatever the target already had. An owning pointer must be set in *every* constructor and *every* assignment. Address `0x0` always faults by design, which makes null dereferences the easiest memory bug to diagnose.

4.2 Task 2 — Runaway recursion in `login`

Commands used in this task

`break Class::f`	—	Break on entry to a method
`continue`	`c`	Resume; here, watch the argument grow each time
`delete N`	—	Remove breakpoint `N` so the crash can actually happen
`bt -12`	—	Show the **last** twelve frames (the *oldest*)

What this task asks you to do

Log in with user ID **1**. The program dies almost immediately. Watch the recursion happen at a breakpoint, then remove the breakpoint, let it crash, and read the bottom of the stack. Note that user **0** does *not* trigger it. That is the clue.

How to solve it

```
script -q ../deliverables/part-2-task-2.txt
gdb ./banking-system
```

```
(gdb) break Bank::login
(gdb) run
```

Choose `1`, enter user ID `1`. You stop on entry. **That is the only input you give the program.** From here you type nothing but `continue` at the `(gdb)` prompt — the *program* supplies the next `id` itself, because line 21 calls `login(id + 1)`:

```
(gdb) continue
(gdb) continue
(gdb) continue
```

Each stop is a *new call nested inside the previous one*, and reports a larger `id` — 1, then 2, then 3. You are watching the recursion happen one level at a time. (If the menu reappears between `continue` s and says *Already logged in*, you are running **part-1** — check the path GDB printed after `Starting program:`. Part 1's `login` has no recursion and starts at `bank.cpp:7`; Part 2's starts at `bank.cpp:13`. With the right binary and the breakpoint set, control never returns to `main` until the crash.) Now remove the breakpoint and let it run:

```
(gdb) delete 1
(gdb) continue
(gdb) bt -12
(gdb) quit
```

SUBMIT: `deliverables/part-2-task-2.txt`

You cannot reach the crash with the breakpoint still set

The breakpoint stops execution at *every* level, and there are roughly 105,000 levels. You cannot `continue` through them by hand.

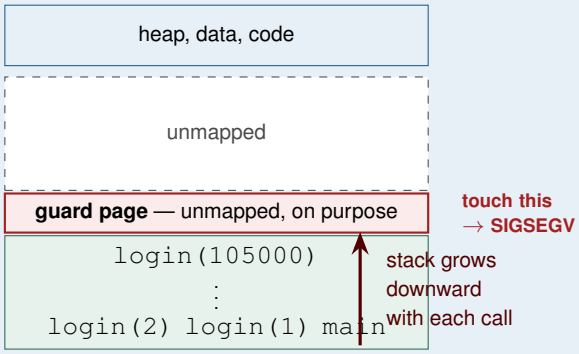
`delete 1` first. This is listed as mistake #5 on the handout: *not continuing far enough for the backtrace to show the recursion.*

The defect

```
13 bool Bank::login(int id) {
14    if (id < 0 || id >= MAX_ACCOUNTS) {
15       return false;
16    }
17    // Task 2: Infinite recursion potential
18    if (!accounts[id].active) {
19       accounts[id] = Account(id);
20       if (id > 0) login(id + 1); // bank.cpp:21 - no base case
21    }
22    current_account = &accounts[id];
23    return true;
24 }
```

Why it never stops, and why user 0 is safe

There is **no base case**. The only terminating condition is `id >= MAX_ACCOUNTS`, and `MAX_ACCOUNTS` is 1,000,000. The stack runs out long before that — around 105,000 frames. `if (id > 0)` is exactly why user 0 is safe: `login(0)` finds `id > 0` false and never recurses. Every non-zero id recurses forever.



The stack grows toward lower addresses. Below it the kernel leaves a **guard page** deliberately unmapped. When the recursion pushes far enough to touch it, the CPU faults and the process gets SIGSEGV.

It is a segfault, not a "stack overflow error"

Students who have written Java or Python expect a named `StackOverflowError`. **There is no such thing at the OS level.** You get SIGSEGV — the same signal as a null dereference in Task 1.
That is why `bt` matters so much: the *signal* does not distinguish these two completely different bugs. The *stack* does. Thousands of identical frames is the unmistakable signature of runaway recursion.

Why `bt -12` and not `bt`

Plain `bt` would try to print 105,000 frames. `bt -12` shows the **last** twelve — the *oldest*, nearest `main`:

```
#104988 0x... in Bank::login (this=0x..., id=12) at bank.cpp:21
#104989 0x... in Bank::login (this=0x..., id=11) at bank.cpp:21
...
#104998 0x... in Bank::login (this=0x..., id=2) at bank.cpp:21
#104999 0x... in Bank::login (this=0x..., id=1) at bank.cpp:21
#105000 0x... in main () at main.cpp:58
```

The `id` counts *down* toward 1 as you go older, and `main` sits at the bottom. **Do not expect a particular depth** — it depends on your stack limit (`ulimit -s`) and frame size. What matters is the repeating frame pattern and that it does not terminate.

The fix

Delete line 21. That is the entire fix.

Do not "fix" this by adding a bound

The instinct is to add `if (id > 0 && id < 100) login(id + 1);`. Resist it.
Ask the right question: *what is this call supposed to accomplish?* `login(id)` logs in user `id`. Recursing into `login(id+1)` creates an account for a user nobody asked about. It is not a broken feature — it has no purpose at all. Bounding it just makes the pointless work finite.
Recognising "this code should not exist" is a real and undervalued debugging skill. The best fix is often a deletion.

Where you will use this for real

- **Every recursive function needs a base case**, and forgetting one is a rite of passage — tree traversals with a cycle, parsers on malformed input, `toString()` that calls itself.
- **Stack overflow as a security boundary.** Deep recursion driven by attacker-controlled input (a deeply nested JSON or XML document) is a real denial-of-service vector. Parsers impose depth limits precisely because of this.
- **Reading a repeating backtrace** is the diagnostic. Whether it is GDB, a Java stack trace, or a browser console, thousands of identical frames means the same thing.
- **Deleting the breakpoint to reach the crash** is a general technique: breakpoints *change* program behaviour by stopping it. Sometimes you must get out of the way to let the failure happen.
- **Tail-call optimisation and iteration.** Converting recursion to a loop is the standard fix when depth is unbounded — it is why production tree walks often use an explicit stack.

The intuition

Recursion is a promise: *"I will call myself on a smaller problem, and eventually the problem gets small enough to answer directly."* The base case is what makes it smaller-and-eventually-done.

Here the problem gets *bigger* every time — `id + 1` moves away from the only stopping condition. The stack is a finite resource being spent at one frame per call with nothing ever returning, so it is not a question of *whether* it dies, only how fast.
The deeper point: **the signal tells you nothing; the stack tells you everything.** Task 1 and Task 2 both produce SIGSEGV and could not be more different. `bt` is what separates them.

Takeaway

No base case, and the recursion moves *away* from the terminating condition. `id > 0` is why user 0 survives. Delete the breakpoint before continuing or you will never reach the crash. `bt -12` reads the oldest frames. It is SIGSEGV, not a named stack-overflow error — the repeating frames are the diagnosis. The fix is to delete the line.

4.3 Task 3 — Buffer overflow in `addTransaction`

Commands used in this task

`break Class::f`	—	Break on each call to `addTransaction`
`print e`	`p`	Read `transactionCount` and `MAX_TRANSACTIONS` each hit
`continue`	`c`	Advance to the next deposit
`info breakpoints`	`i b`	Shows `already hit N times` — a free counter

What this task asks you to do

With Task 1 fixed, deposits work. Deposit repeatedly and the program eventually dies. Watch `transactionCount` climb past `MAX_TRANSACTIONS`, then add the bounds check.

Critical: you need *four* deposits, not three

The handout says the third deposit is the first out-of-range write. That is true. But the program **does not die on the third deposit**. Measured on the shipped code with `MAX_TRANSACTIONS = 2`:

Deposits	Result	What happened
1, 2	fine	Writes land inside the array
3	**exits cleanly**, `rc=0`	Writes *past* the array — no crash, no message, and it still reports "Deposit successful"
4	`double free or corruption (out)`, `rc=134`	The next allocation discovers the corrupted heap metadata

If you deposit exactly three times you will see **nothing at all** and conclude the bug is not there. **Deposit at least four times.**

How to solve it

Confirm `MAX_TRANSACTIONS` is still **2** in `types.h`. Then:

```
script -q ../deliverables/part-2-task-3.txt
gdb ./banking-system
```

```
(gdb) break Account::addTransaction
(gdb) run
```

Log in as user `0`, then deposit. At each breakpoint hit:

```
(gdb) print transactionCount
(gdb) print MAX_TRANSACTIONS
(gdb) continue
```

Repeat for **four** deposits. `transactionCount` reads `0, 1, 2, 3` while `MAX_TRANSACTIONS` stays `2`. The fourth deposit aborts.

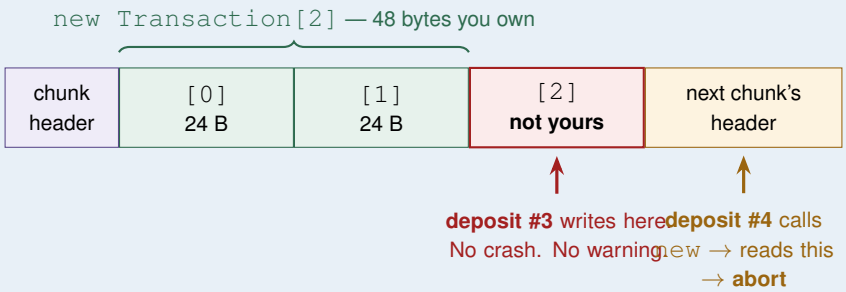
SUBMIT: `deliverables/part-2-task-3.txt`

The defect

```
57 bool addTransaction(double amount, const char* desc) {
58     // Task 3: Buffer overflow potential ← no bounds check
59     transactions[transactionCount].accountId = id; // types.h:59
60     transactions[transactionCount].amount = amount;
61     transactions[transactionCount].description = new char[strlen(desc) + 1];
62     strcpy(transactions[transactionCount].description, desc);
63     transactionCount++;
64     return true; ← always true, even when full
65 }
```

Why the crash happens one deposit *after* the mistake

`new Transaction[2]` gives you 48 bytes of heap (2×24). The heap allocator places **bookkeeping metadata** immediately around each block: the size of the block, flags, and links used by `free`.



Deposit 3 writes 24 bytes past the end of your 48. Those bytes land on the *next chunk's header*. Nothing checks them at write time — C++ does no bounds checking, and the CPU sees an ordinary, legal write to mapped memory. The deposit reports success.

Deposit 4 calls `new char[...]` for its description. Now the allocator walks its data structures, reads the header that deposit 3 scribbled on, finds it inconsistent, and aborts:

```
double free or corruption (out)
Aborted (core dumped)
```

Read the message carefully. It says "double free." Nothing was freed twice. The allocator can only report "my metadata is wrong" — it cannot know that the culprit was a stray write from an unrelated array one operation earlier. **Heap corruption messages name the victim, never the attacker.** That gap is the entire point of this exercise.

The fix

```
57 bool addTransaction(double amount, const char* desc) {
58    if (transactionCount >= MAX_TRANSACTIONS) return false; // Task 3
59    transactions[transactionCount].accountId = id;
60    ...
61 }
```

The scaffolding for `false` is already there

Look at `bank.cpp:39-44`. `Bank::deposit` already checks the return value and propagates a failure, with a comment explaining why: otherwise `main` prints "Deposit successful" for a deposit that was never credited.
The `bool` return type was always meant to mean something. Your bounds check is what finally gives it meaning. After the fix, the fifth deposit prints "Deposit failed" instead of corrupting the heap — which is the correct behaviour for a full transaction log.

Where you will use this for real

Buffer overflow is the most consequential bug class in the history of computing. Not an exaggeration — Morris worm (1988), Code Red, SQL Slammer, and Heartbleed were all out-of-bounds memory access.

- **As a security vulnerability.** An attacker who controls what spills past the end can overwrite a return address and redirect execution. This is the foundation of stack-smashing attacks, and it is why modern systems ship ASLR, stack canaries, and non-executable stacks.
- **The delay is the danger.** A bug whose symptom appears one operation later — or one *hour* later, in a different subsystem — is orders of magnitude harder to find than one that crashes on the spot. Teams burn days on "random" crashes that trace back to a single off-by-one.
- **Never trust the error message's noun.** "double free", "corrupted top size", "malloc(): invalid size" all mean *the heap is inconsistent*, not *the thing named is broken*. Tools like AddressSanitizer exist specifically to catch the write instead of the aftermath.
- **This is why `std::vector` exists.** `.at()` bounds-checks, `.push_back()` grows. Rust and Go bounds-check by default. The entire memory-safe-language movement is a response to this one bug.

The intuition

Think of the array as a parking space you rented and the heap as a crowded lot. Writing past the end is parking across the line into the neighbour's space. Nothing stops you at the moment you do it — there is no gate, no alarm. The neighbour finds out later, when they come back to their car.
Two lessons follow. **First: the absence of a crash is not evidence of correctness.** Deposit 3 "worked." **Second: the place where a program dies is not the place where it went wrong.** Your instinct will be to inspect whatever code was running at the abort. That code is the victim. The culprit already ran and finished.

Takeaway

No bounds check, so deposit 3 writes past a 48-byte block onto the next chunk's header. Nothing crashes — deposit 3 reports success. Deposit 4's allocation reads the corrupted metadata and aborts with a message naming the wrong problem. You need **four** deposits to see it. The fix is one line, and the `bool` return was waiting for it.

4.4 Task 4 — Memory leak in the transaction descriptions

Commands used in this task

`print e`	`p`	Print a `Transaction` or its `description` pointer
`p/x e`	—	Print in hexadecimal — here, the *addresses*
`x/s addr`	—	Examine memory as a null-terminated string
`x/32xb a`	—	32 bytes in hex — the raw transaction array

What this task asks you to do

Every `Transaction` holds a `char* description` allocated with `new char[]`. Nothing ever frees it, and the `transactions` array itself is never released either. Observe the allocations piling up, then fix the ownership.

How to solve it

First raise the limit so there is room to observe several transactions — `types.h:11`:

```
const int MAX_TRANSACTIONS = 100;
```

Rebuild with `make`. Then:

```
script -q ../deliverables/part-2-task-4.txt
gdb ./banking-system
```

```
(gdb) break Account::addTransaction
(gdb) run
```

Log in as user `0` and deposit. **The breakpoint stops at the entry of `addTransaction`, before this deposit's slot is filled** — so inspect a slot at the hit *after* the one that filled it:

Hit	Do
1	`print transactions[0]` — expect an empty slot, `description = 0x0` (see below). `continue`.
2	`print transactions[0],print transactions[0].description, x/s transactions[0].description` — now filled. `continue`.
3	`continue`
4	the eight commands below — `transactions[0..2]` are all filled

At the fourth hit, compare all three:

```
(gdb) print transactions[0].description
(gdb) print transactions[1].description
(gdb) print transactions[2].description
(gdb) p/x &transactions[0]
(gdb) p/x transactions[0].description
(gdb) p/x transactions[1].description
(gdb) p/x transactions[2].description
(gdb) x/32xb &transactions[0]
(gdb) quit
```

SUBMIT: `deliverables/part-2-task-4.txt`

"Why is `transactions[0]` empty at the first hit?"

```
Breakpoint 1, Account::addTransaction (this=..., amount=40, ...) at types.h:61
61 if (transactionCount >= MAX_TRANSACTIONS) return false;
(gdb) print transactions[0]
$1 = {accountId = 0, amount = 0, description = 0x0}
(gdb) x/s transactions[0].description
0x0: <error: Cannot access memory at address 0x0>
```

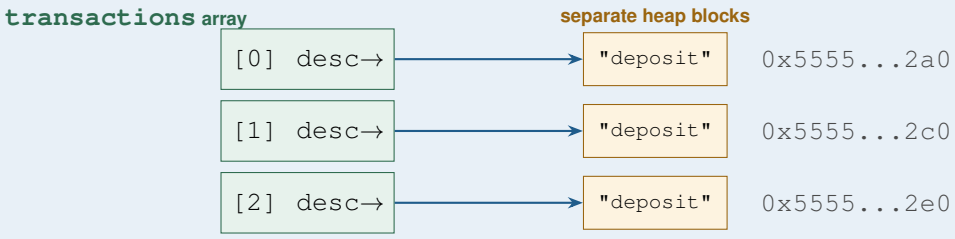
You are stopped on the *first line* of the function. The stores to `transactions[0]` are the lines that follow, and none has run yet. The `40` is still in the `amount` parameter. `description` is null, so `x/s` tries to read a string at address 0 — the unmapped first page, exactly what segfaulted in Task 1.

The zeros are luck, not a guarantee: `new Transaction[100]` does not initialise plain structs. You see zeros because the kernel hands out fresh heap pages pre-zeroed. Same lesson as the `247` in Part 1.

(Your line number will be 61, not 59, if you have already added the Task 3 bounds check — it pushes everything down two lines.)

What you are looking at

Each `description` sits at its **own** heap address, and none of them is ever released.



Nothing frees any of this. The array leaks, and every string in it leaks. The destructor `~Account()` is empty.

Note the honest limit of what GDB can show you here. You are watching allocations *happen*. You cannot see that they are never *released* — a debugger has no way to prove a negative about the future. That is why §5.6 exists and why the leak-check transcript, not this one, is what actually proves Task 4. The address listing is evidence; LeakSanitizer is proof.

The defect — two leaks, not one

```
69 ~Account() {
70     // Task 4: Memory leak - no deallocation ← empty destructor
71 }
```

And a second, easier-to-miss one in `operator=`:

```
44 Account& operator=(const Account& other) {
45     if (this != &other) {
46         delete[] transactions; ← frees the array, NOT the strings inside it
47         ...
```

Writing only a destructor still leaks

`delete[] transactions` releases the array of `Transaction` structs. It does **not** release the `char` buffers those structs point at — `Transaction` has no destructor, and `delete[]` only runs destructors, it does not chase pointers.
So you must free the strings *first*, then the array. And you must do it in **both** places: the destructor and `operator=`. Factor it into a helper so you cannot get them out of step.

The fix

```
void freeTransactions() {
    if (transactions) {
        for (int i = 0; i < transactionCount; i++)
            delete[] transactions[i].description; // each string
        delete[] transactions; // then the array
        transactions = nullptr;
    }
}

~Account() { freeTransactions(); }
```

and in `operator=`, replace the bare `delete[] transactions;` with `freeTransactions();`.

`delete` versus `delete[]` — they are not interchangeable

`new char[n]` must be released with `delete[]`, never `delete`. `new T` must be released with `delete`, never `delete[]`. Mixing them is undefined behaviour — and Task 5 is what that looks like when it goes wrong.

A double-free you might expect — and why it does not happen

A reasonable worry: `Account` has no *copy constructor*. If an `Account` were copied, both copies would hold the same `transactions` pointer, and the second destructor would free it twice.
In *this* program that never happens. The only copy is `accounts[id] = Account(id)`, which is copy-*assignment*, and your Task 1 fix makes `operator=` allocate its own fresh array. The temporary and the target own separate memory, so each destructor frees only its own. Verified clean under AddressSanitizer.
The missing copy constructor is still a genuine latent bug — `-Weffc++` flags it, and it would bite the instant anyone passed an `Account` by value. **The compiler is warning you about a landmine you happen not to have stepped on.** That is worth taking seriously rather than dismissing.

Where you will use this for real

- **Leaks kill long-running processes.** A one-off tool can leak freely — the OS reclaims everything at exit. A server that leaks 200 bytes per request is fine in testing and dead in three days. Leaks almost always surface in production first.
- **"Whoever allocates, frees" is an ownership contract.** Most real leaks are not forgotten `free` calls; they are two components each assuming the other owns the memory.
- **RAII is the C++ answer.** Tie the lifetime of a resource to the lifetime of an object, so the destructor runs automatically — including when an exception unwinds the stack. `std::string` and `std::vector` would have made this entire class impossible to get wrong.
- **Not just memory.** File descriptors, socket handles, database connections, mutex locks — all leak the same way, and running out of file descriptors takes a server down as surely as running out of RAM.
- **Leak detection is a build flag, not an afternoon.** `-fsanitize=leak` in CI catches these before they ship.

The intuition

Every `new` is a **borrow** with an implied promise to return it. A leak is a broken promise — and the borrowed memory is not merely unused, it is **unusable**, because the allocator still believes it is in service.
The reason leaks are hard is that they are the **absence** of an event. A crash is something happening; a leak is something *not* happening, forever, quietly. You cannot see it by looking at any single moment of the program — which is exactly why GDB can only hint at it here and why you need a tool that tracks every allocation over the whole run.

Takeaway

Two leaks: each `description` string and the `transactions` array itself. `delete[]` on the array does not free the strings inside it. Fix both the destructor *and* `operator=`, via a shared helper. GDB can show allocations happening but can never prove they were not released — that is what LeakSanitizer is for.

4.5 Task 5 — Invalid delete in `logout`

Commands used in this task

`break Class::f`	—	Break on `Bank::logout`
`print e`	`p`	Compare `current_account` with the array base
`x/x addr`	—	Examine one word at that address

What this task asks you to do

Log in, then log out. The program aborts and the C library prints a diagnostic naming the problem. Work out what allocated the object `current_account` points at, and whether `delete` is entitled to free it.

How to solve it

```
script -q ../deliverables/part-2-task-5.txt
gdb ./banking-system
```

```
(gdb) break Bank::logout
(gdb) run
```

Choose `1`, user ID `0`, then `5` to log out. At the breakpoint:

```
(gdb) print current_account
(gdb) x/x current_account
(gdb) continue
```

It aborts. **Put the abort message in your transcript.** Then `quit`.

SUBMIT: `deliverables/part-2-task-5.txt`

What you should see

```
(gdb) continue
Continuing.
free(): invalid pointer

Program received signal SIGABRT, Aborted.
```

The comment in the source is wrong

`bank.cpp:29` says `// Task 5: Double free potential`. **It is not a double free.** Nothing is freed twice. The handout is careful about this and the code contradicts it.

`free(): invalid pointer`	This address was **never returned by `malloc`**. The allocator has no record of it.
`double free or corruption (...)`	This chunk is already on the free list, or its header is inconsistent.

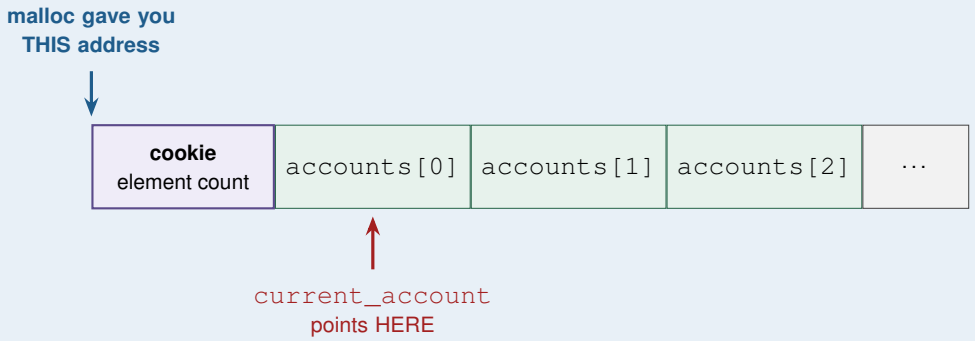
Being able to tell these apart is a real diagnostic skill — and Task 3 already showed you that the second message can appear when nothing was double-freed either.

The defect

```
28 void Bank::logout() {
29     // Task 5: Double free potential
30     delete current_account; // bank.cpp:30
31     current_account = nullptr;
32 }
```

`current_account` was set in `login` as `&accounts[id]` — a pointer *into* an array that came from `new Account[MAX_ACCOUNTS]`.

Two rules broken at once



1. **Wrong form.** Memory from `new[]` must be released with `delete[]`. Using plain `delete` is undefined behaviour even when the address is right.
2. **Wrong address.** `new[]` for a type with a non-trivial destructor stores a **cookie** — the element count — in a few bytes *before* the first element, so the destructor loop knows how many objects to destroy. So the address `malloc` handed out is a few bytes *below* `&accounts[0]`. Even logging in as user 0 gives an address the allocator does not recognise. Log in as user 5 and you are pointing 200 bytes into the middle of a live allocation.

Hence `free(): invalid pointer` — glibc looked up the address, found no chunk header where one should be, and refused.

And the deeper problem: `logout` does not own this memory. `Bank` allocated the array in its constructor and frees it in `~Bank()`. `current_account` is an *observer* — a non-owning pointer that merely refers to something someone else owns. Freeing through an observing pointer is

the mistake, and it is the same Rule of Three failure from a third angle.

The fix

Delete the `delete`:

```
28 void Bank::logout() {
29    current_account = nullptr; // just stop referring to it
30 }
```

Logging out means "I am no longer looking at this account," not "destroy this account."

Where you will use this for real

- **Owning versus non-owning pointers is the central question of resource management.** Modern C++ makes it explicit in the type: `unique_ptr<T>` owns, `shared_ptr<T>` co-owns, raw `T*` and `T&` observe. Rust encodes it in the borrow checker.
- **Freeing a pointer you do not own is a security bug, not just a crash.** If the allocator *does* accept it, that region can be reallocated while another part of the program still holds a pointer to it — a use-after-free, one of the most commonly exploited vulnerability classes in browsers and kernels.
- **Read the allocator's message precisely.** `invalid pointer`, `invalid size`, `corrupted top size`, `double free` are *different diagnoses*. Treating them as one generic "heap error" throws away the allocator's most useful clue.
- **Interior pointers.** Holding a pointer into the middle of a buffer is common and fine — iterators, string views, packet parsers. What is never fine is *freeing* through one.

The intuition

A pointer says *where*. It says nothing about *who is responsible*. `current_account` and `accounts` can hold the same address while having completely different jobs — one is the owner, one is just looking.
`delete` is not "forget about this." It is "give this back to the allocator." You can only give back what you were given, exactly as you were given it: the same address, and the matching form (`delete` for `new`, `delete[]` for `new[]`).
`logout` confused *stop referring* with *destroy*. Setting the pointer to `nullptr` was always the whole job.

Takeaway

`current_account` is a non-owning pointer into an array owned by `Bank`. `delete` on it breaks two rules at once: wrong form (`new[]` needs `delete[]`) and wrong address (the `new[]` cookie sits before the first element). glibc reports `free(): invalid pointer` — which is **not** the double-free message, despite what the source comment claims. The fix is to remove the `delete`.

5 Cross-Checking with the Compiler and Sanitizers

GDB is a microscope: it shows you what a program is doing *once you know roughly where to look*. Two other tools work the other way round — they find the problem and hand you the location.

GDB
You pick the place.
Shows any value at any moment.
Needs a hypothesis.

`-Weffc++`
Static — never runs the program.
Finds *design* faults.
Free. Instant.

ASan / LSan
Dynamic — instruments every access.
Finds *real* faults, with a stack trace.
≈2× slower.

5.1 Ask the compiler about your class design

From `part-2/`, on the code as it shipped:

```
g++ -std=c++17 -Weffc++ -c bank.cpp -o /dev/null
```

```
warning: 'class Account' has pointer data members [-Weffc++]
warning: but does not declare 'Account(const Account&)' [-Weffc++]
warning: 'Account::transactions' should be initialized in the
      member initialization list [-Weffc++]
```

Those warnings describe Tasks 1, 4, and 5 between them, and they cost zero seconds of runtime. **The compiler knew about three of your five bugs before the program ever ran.**

The intuition

A warning is not a nag. It is a senior engineer reading your code and saying "this shape of class is usually broken in a specific way." `-Weffc++` does not know your `transactions` pointer is null — it knows that a class with a raw owning pointer and no copy constructor is *the* classic broken shape, and it is right often enough to be worth listening to.
Build with warnings on. Treat them as errors. The bugs they catch are the ones that would otherwise cost you an afternoon in GDB.

5.2 Ask the runtime to check every memory access

```
g++ -std=c++17 -g -fsanitize=address,leak -o banking-asan main.cpp bank.cpp
./banking-asan
```

Run it the same way you did in Task 1 — log in as 0, deposit 100. Instead of a bare *Segmentation fault*:

```
ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000
   #0 ... in Account::addTransaction(double, char const*) types.h:59
   #1 ... in Bank::deposit(double) bank.cpp:42
   #2 ... in main main.cpp:62
```

`0x000000000000` is the giveaway: a null pointer, which is exactly the uninitialized `transactions` array from Task 1.

Line-number drift

> The handout quotes `types.h:56`; the shipped code reports **59**. Minor drift between handout revisions. Do not worry if your number differs by a line or two — match on the *function name*, not the number.

5.3 The graded leak check (deliverable 13)

This is the only step that *proves* Task 4 rather than suggesting it. Looking at addresses in GDB shows allocations happening; it cannot show that they were never released. LeakSanitizer states it outright.

How to solve it

Capture **one** transcript showing LeakSanitizer twice — before and after your Task 4 fix, in the same session.

```
script -q ../deliverables/part-2-task-4-leakcheck.txt

# 1. BEFORE: revert your Task 4 fix (empty destructor, bare delete[])
g++ -std=c++17 -g -fsanitize=address,leak -o banking-asan main.cpp bank.cpp
./banking-asan # log in, deposit twice, exit

# 2. AFTER: restore your fix, rebuild, run exactly the same steps
g++ -std=c++17 -g -fsanitize=address,leak -o banking-asan main.cpp bank.cpp
./banking-asan # log in, deposit twice, exit

exit
```

SUBMIT: `deliverables/part-2-task-4-leakcheck.txt`

Expected, with Tasks 1, 2, 3, 5 fixed and `MAX_TRANSACTIONS = 100`:

```
BEFORE:
==679==ERROR: LeakSanitizer: detected memory leaks
SUMMARY: AddressSanitizer: 4816 byte(s) leaked in 4 allocation(s).

AFTER:
(no leak summary at all)
```

Where "4 allocations" comes from

Logging in as user 1 with two deposits leaks exactly four blocks:

Count	What	Bytes
2	`Transaction[100]` arrays — one from the temporary `Account(1)`, one from `operator=`	2400 each
2	the `"deposit"` strings	8 each
4		**4816**

Do not expect a particular byte count — it scales with `MAX_TRANSACTIONS`. But *4 allocations* is stable and a good sanity check. What matters for the grade is that the summary is **present** before and **absent** after.

Where you will use this for real

- **Sanitizers are standard practice, not a teaching toy.** Chrome, Firefox, Android, and the Linux kernel all run continuous ASan builds. Google's fuzzing infrastructure is essentially "run the program on random input under ASan and see what it says."
- **They find bugs that never crash.** Task 3's overflow does not crash on deposit 3; ASan reports it *at the moment of the write*, with the allocation site and the write site both named.
- **The cost is affordable in CI, not in production.** Roughly 2× slowdown and 3× memory — fine for tests, not for shipping.
- **The workflow to adopt:** write with warnings on, test under sanitizers, and reach for GDB when you need to understand *why* rather than *where*.

Takeaway

`-Weffc++` is free and static; it caught three of five bugs by reading the class design. ASan classifies the fault and hands you a stack trace. LSan is the *only* tool here that can prove a leak, which is why deliverable 13 exists. GDB remains the tool for understanding a bug you have already located.

6 Deliverables and Final Checks

6.1 The fifteen items

#	Path	Contents
1	`deliverables/part-1-task-1.txt`	`ptype` and the first account
2	`deliverables/part-1-task-2-1.txt`	the three breakpoints
3	`deliverables/part-1-task-2-2.txt`	the watchpoint firing
4	`deliverables/part-1-task-3-1.txt`	step, list, finish
5	`deliverables/part-1-task-3-2.txt`	the three views of memory
6	`deliverables/part-1-task-4.txt`	backtrace and `bt full`
7	`deliverables/part-1-task-5.txt`	interrupt and watchpoint output
8	`deliverables/part-2-task-1.txt`	crash and backtrace
9	`deliverables/part-2-task-2.txt`	recursive frames
10	`deliverables/part-2-task-3.txt`	overflow and counters
11	`deliverables/part-2-task-4.txt`	descriptions and memory
12	`deliverables/part-2-task-5.txt`	abort and pointer
13	`deliverables/part-2-task-4-leakcheck.txt`	LeakSanitizer before and after
14	`part-2/bank.cpp`	your corrected source
15	`part-2/types.h`	your corrected source

Names are checked mechanically — a mis-named file is a missing file

If you recorded into `../deliverables/` with the names above, they are already right. Verify with `ls deliverables/` before you push.

6.2 Pre-submission checklist

Run all of this before you push

```
cd ~/lab1/csce-313-fa26-lab-1-xoumyax

# 1. All 13 transcripts exist and none is truncated
for f in deliverables/part-*.txt; do
  printf "%-45s %s\n" "$f" \
    "$(tail -1 "$f" | grep -q 'Script done' && echo OK || echo TRUNCATED)"
done

# 2. part-2 builds clean from scratch
cd part-2 && make clean && make

# 3. Every path runs without crashing
```

```
printf '1\n0\n2\n100\n2\n50\n4\n5\n6\n' | ./banking-system # login 0
printf '1\n1\n2\n100\n4\n5\n6\n' | ./banking-system # login 1 (Task 2)

# 4. No leaks
g++ -std=c++17 -g -fsanitize=address,leak -o banking-asan main.cpp bank.cpp
printf '1\n1\n2\n10\n2\n20\n6\n' | ./banking-asan # expect silence
```

Then commit and push everything.

6.3 The eight ways people lose points

	Mistake	Prevention
1	Not restarting GDB between tasks — stale breakpoints in the output	One task, one session. Watch for `Note: breakpoint N also set`.
2	Transcripts not named exactly as required	`ls deliverables/` and compare to the table.
3	Forgetting `exit`, leaving the transcript truncated	`tail -1` every file; look for `Script done`.
4	Part 1 Task 5: stopping before the watchpoint fires	Keep pressing `next` until you see `Old value`.
5	Part 2 Task 2: not continuing far enough	`delete 1` *before* `continue`, then `bt -12`.
6	Editing source without rebuilding	`make` after every edit. When in doubt, `make clean && make`.
7	Part 2 Task 3: only three deposits, so nothing happens	**Four** deposits minimum.
8	Running `part-1/banking-system` for a Part 2 task	Read the path after `Starting program:`; `pwd` before `gdb`.

7

Quick Reference

7.1 Every command in this lab

Command	Short	What it does
Starting and stopping		
`gdb ./banking-system`		Start GDB on the executable
`run`	`r`	Start the program
`continue`	`c`	Resume until the next stop
`quit`	`q`	Leave GDB
`Ctrl+C`		Interrupt a running program
Breakpoints		
`break N`	`b N`	Line `N` of the current default file
`break file.cpp:N`		Line `N` of a named file
`break Class::function`		Function entry, after the prologue
`break file.cpp:function`		Function named by file
`info breakpoints`	`i b`	List breakpoints and watchpoints
`delete N`		Delete breakpoint `N` (bare `delete` = all)
Moving		
`next`	`n`	One line, stepping *over* calls
`step`	`s`	One line, stepping *into* calls with debug info
`finish`		Run until the current function returns
`list`	`l`	Show source around the current line
Looking		
`print expr`	`p`	Print a variable or expression
`p/x expr`		Print in hexadecimal
`ptype T`		Print the definition of a type
`ptype /o T`		. . . with byte offsets and padding
`whatis expr`		Print just the type of an expression
`x/NFU addr`		Examine memory: `N` units, format `F`, size `U`
`info locals`		Locals in the current frame
Watching and stacks		
`watch expr`		Stop when `expr` changes value
`backtrace`	`bt`	Show the call stack
`bt full`		. . . with each frame's locals
`bt -N`		The *last* `N` frames (oldest)
`frame N` / `up` / `down`	`f N`	Select a frame

7.2 Which tool for which symptom

Symptom	Reach for	Why
Crash, and you want the line	`run` then `bt`	The stack names the line and the caller
A value is wrong and you know where	`break` + `print`	Freeze and inspect
A value is wrong and you *don't* know where	`watch`	The debugger finds the writer
Thousands of identical frames	runaway recursion	Missing base case
`SEGV on 0x0`	null pointer	Something never initialised
`free(): invalid pointer`	freeing a non-heap or interior pointer	Wrong owner, or wrong form
`double free or corruption`	heap metadata is inconsistent	Often an out-of-bounds write *earlier*
Memory grows without bound	`-fsanitize=leak`	Only LSan can prove a leak
Class owns a raw pointer	`-Weffc++`	Rule of Three check, free and instant

7.3 What to remember six months from now

The five ideas this lab is really about

1. **A type is an interpretation of bytes.** The bytes do not change; `print` and `x` are two lenses on the same memory. When the structured view stops making sense, drop to raw.
2. **Ask about data, not just about code.** Breakpoints need a hypothesis about *where*. Watchpoints only need to know *what*. The hardest bugs are the ones where you have no idea where.
3. **The place a program dies is not the place it went wrong.** Task 3's crash is one full operation after the mistake. Task 5's message names the wrong problem entirely. Read the evidence, do not trust the label.
4. **Ownership is a design decision the compiler cannot guess.** Who allocates, who frees, who is merely looking. Tasks 1, 4, and 5 are three faces of one unanswered question — the Rule of Three.
5. **Use the tool that finds bugs, then the tool that explains them.** Warnings and sanitizers locate; GDB explains. Running the first two costs a compiler flag; skipping them costs afternoons.

Getting help

Public questions: the course Discord — most setup problems are already answered there.
Private questions: `davidkebo@tamu.edu`. Office hours and help sessions: see the syllabus.
The Canvas inbox is **not** monitored.

You may discuss this lab conceptually with classmates.
The implementation you submit must be your own work.