

CS100 Recitation 4

GDB、Sanitizer 与 C 语言代码调试

Speaker Wenjie Liu

liuwj2026@shanghaitech.edu.cn



上海科技大学

ShanghaiTech University

成绩统计：完整代码（调试用错误版）


```c
#include <stdio.h>

int main(void) {
    int n = 0;
    int sum = 0;
    int count = 0;
    if (scanf("%d", &n) != 1 || n < 0 || n > 10000)
      {
        return 1;
    }
    for (int i = 0; i < n; ++i) {
        int score = 0;
        if (scanf("%d", &score) != 1 || score < 0
     || score > 100) {
            return 1;
        }
        if (score >= 60) {
            sum += score;
        }
        if (score > 60) {
            ++count;
        }
    }
    if (count == 0) {
        printf("No passing scores\n");
    } else {
        printf("%g\n", (double)sum / count);
    }
    return 0;
}
```


立志成才 报国裕民

成绩统计：程序运行了，结果就正确吗？


```
bash-4.4$ gcc grades.c -o grades
bash-4.4$ ./grades
3
61 89 99
83
bash-4.4$ ./grades
4
55 60 71 89
110
bash-4.4$
```

第一组：61、89、99
预期平均值 83，输出 83。

第二组：55、60、71、89
及格成绩为 60、71、89。
预期 $220/3 \approx 73.3333$，
实际输出**110**。

同一段代码，为什么一组正确，另一组错误？

目录

1 程序出错之后，从哪里开始?

2 GDB：让程序停下来看看

3 Sanitizer：这次运行哪里不合法?

4 C 语言程序的常见错误与调试

程序出错之后，从哪里开始？

HW1：完整代码（调试用错误版）

上海科技大学 ShanghaiTech University

依据 HW1 P3 要求编写的课堂调试变体：读整数至 0，输出总和与最大值。

```c
/* Classroom debugging variant of HW1 P3; not the
     original starter solution. */
#include <stdio.h>

int updateMaximum(int maximum, int value) {
    int result = maximum;
    if (value > result) {
        result = value;
    }
    return result;
}

int main(void) {
    int sum = 0;
    int maximum = 0;
    int hasValue = 0;
    int x = 0;
    while (scanf("%d", &x) == 1 && x != 0) {
        sum += x;
        if (!hasValue) {
            maximum = x;
        } else {
            maximum = updateMaximum(maximum, sum);
        }
        hasValue = 1;
    }
    if (hasValue) {
        printf("%d %d\n", sum, maximum);
    }
    return 0;
}
```


立志成才 报国裕民

HW1：编译通过，然后呢？

P3 的 starter 要求：读整数，遇到 0 结束，输出**总和与最大值**；不用数组或动态内存。

课堂运行截图

```
bash-4.4$ cd ./demo/HW1
bash-4.4$ gcc HW1_P3.c -o HW1_P3
bash-4.4$ ./HW1_P3
3 5 2 0
10 10
bash-4.4$
```

- 总和：$3+5+2=10$。
- 最大值：5。
- 预期输出：`10 5`。
- 错误版输出：`10 10`。

输入：`3 5 2 0`。
0 是结束标记，不是待统计的数据。

能编译、能运行，不代表实现了需求。先用小输入找出差异。

先说清楚：错在哪里？

现象	第一步要收集什么？
编译或链接失败	完整命令、第一条诊断、对应源码。
运行崩溃	能复现的输入、信号、调用栈。
输出错误	明确的预期结果、一个小反例。
一直不结束	输入是否读完、程序停在哪个循环或调用。

“我的程序有问题”还不够。先补全：**输入是什么？应该怎样？实际怎样？**

一次只验证一个猜想

对于输出 10 10 的 HW1 课堂变体，先问：

1. 是否依次读到了 3、5、2，并在 0 处停止？
2. 每轮结束后，总和是否依次为 3、8、10？
3. 最大值是否依次为 3、5、5？
4. 传给更新最大值函数的，是当前输入，还是别的量？

让猜想可以被验证

写下“这里应该是多少”，再观察“这里实际是多少”。
保留失败输入，每次修改后重新运行。

立志成才 报国裕民

从复现到验证：工具分别帮什么忙？



- **编译警告**：编译时发现可疑写法；逐条阅读。
- **日志与断言**：记录关键状态，检查必须成立的条件。
- **GDB**：观察变量、执行位置与调用栈。
- **Sanitizer**：在运行时检查特定类别的非法行为。

工具提供线索；预期结果与原因解释，需要你来确定。

立志成才 报国裕民

GDB：让程序停下来看看

GDB 是什么?

GDB（GNU Debugger)是 GNU 项目的调试器，本课使用它调试 C 语言编写的程序。

- **启动程序**：指定参数和输入，复现出错的运行过程。
- **暂停执行**：在某个位置、条件成立或变量改变时停下来。
- **查看现场**：观察变量、源码位置和函数调用关系。
- **控制执行**：逐步运行，找到实际行为第一次偏离预期的位置。

从源码到调试

gcc 将 C 源码编译为可执行程序；GDB 加载并调试这个程序。
VSCode 可以通过扩展调用编译器与调试器，提供图形化操作界面。

调试信息帮助 GDB 显示源码行号与变量；后面将介绍编译参数-g。

立志成才 报国裕民

GDB 与 GCC 有什么区别?

GCC 负责把源码编译成程序，GDB 负责观察和控制程序的运行。

- gcc 读取 C 源码，编译、链接成可执行程序。
- gdb 加载可执行程序，设置断点、查看变量、单步执行。

在终端中**先编译，再启动调试**:

```
mkdir -p build/demo
gcc -std=c17 -g -O0 -Wall -Wextra -Wpedantic -Werror \
    code/c/sum.c -o build/demo/sum
gdb ./build/demo/sum
```

修改源码后，需要先重新编译。

`-O0` 中的 O 是字母、0 是数字；`-Wall` 并非开启全部警告。

选项	含义
`-g`	生成调试信息，便于 GDB 定位源码和变量。
`-O0`	关闭优化，便于逐行调试。
`-Wall`	开启一组常用编译警告。
`-Wextra`	开启额外的编译警告。
`-Wpedantic`	提示部分不符合所选 ISO C 标准的写法。
`-Werror`	将警告视为错误，有警告时编译失败。

立志成才 报国裕民

安装 GDB：Ubuntu ／ WSL

在**Ubuntu ／ WSL 的 Ubuntu 终端**执行以下命令，需要当前账号拥有 sudo 权限：

```
sudo apt update
sudo apt install build-essential gdb
```

```
(base) liuwenjie@LWJ2025:~$ sudo apt install
[sudo: authenticate] Password:
Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 11
(base) liuwenjie@LWJ2025:~$ sudo apt install build-essential gdb
build-essential is already the newest version (12.12ubuntu2.26.04.2).
gdb is already the newest version (17.1-2ubuntu1).
Summary:
  Upgrading: 0, Installing: 0, Removing: 0, Not Upgrading: 11
(base) liuwenjie@LWJ2025:~$
```

- `apt update`：更新软件包索引。
- `build-essential`：GCC、Make 等构建工具。
- `gdb`：GDB 调试器。

截图中的 `is already the newest version` 表示相应工具已安装。

共享服务器可能使用其他发行版或没有 sudo 权限，安装前先确认环境。

没有 sudo 权限时怎么使用 GDB?

`is not in the sudoers file` 表示当前账号没有 sudo 权限。

- 在共享服务器上，使用管理员提供的工具，或已准备好的用户目录工具。
- 本项目当前演示机是**RHEL 8.10**；Ubuntu 的 `apt` 安装命令不适用。

当前工作区已准备好本地 GDB 与运行库，可以在 Bash 中执行：

```
cd /data/liuwenjie/Myfile/CS100/slides/r4
source build/tools/environment.sh
```

这里加载的是本工作区已有工具。每次打开新终端需重新加载；清理 `build/`后需重新准备这些工具。其他服务器的工具路径以管理员说明为准。

安装完成后，如何验证？

在**终端**中依次执行：

```
command -v gdb
gdb --version
gcc --version
gdb -q
```

`command -v` 显示 GDB 路径；
`--version` 显示版本信息。

出现`(gdb)`提示符后输入：

```
help
quit
```

`help` 显示命令帮助；
`quit` 退出并回到终端。

验证成功的标志

能找到路径、显示版本，并进入 GDB 执行 `help`，说明工具已可启动。
若提示 `command not found`，先检查安装位置或是否加载了本地工具环境。

这四行命令分别验证什么？

终端命令	含义与成功时的现象
`command -v gdb`	让 Shell 查找 `gdb` 命令，检查当前终端能否找到它；通常输出可执行文件路径。
`gdb --version`	显示 GDB 的版本信息并退出，确认调试器可以运行，也方便记录调试环境。
`gcc --version`	显示 GCC 的版本信息并退出，确认 C 编译器可以运行；后面用它编译示例源码。
`gdb -q`	以安静模式启动 GDB，省略欢迎语和版权信息；出现 `(gdb)` 后即可输入调试命令。

`-q` 中的 `q` 表示**quiet**；进入 GDB 后用 `help` 查看帮助，用 `quit` 退出。

工具已经安装但终端找不到时，检查 `PATH`，或重新加载本地工具环境。

立志成才 报国裕民

安装验证：路径、版本与交互界面

上海科技大学
ShanghaiTech University

查找路径、查看版本

```
(base) liuwenjie@LWJ2025:~$ command -v gdb
/usr/bin/gdb
(base) liuwenjie@LWJ2025:~$ gdb --version
GNU gdb (Ubuntu 17.1-2ubuntu1) 17.1
Copyright (C) 2025 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
(base) liuwenjie@LWJ2025:~$ gcc --version
gcc (Ubuntu 15.2.0-16ubuntu1) 15.2.0
Copyright (C) 2025 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

(base) liuwenjie@LWJ2025:~$
```

显示 GDB 路径，以及 GDB、GCC 的版本信息。

启动 GDB、查看帮助、退出

```
(base) liuwenjie@LWJ2025:~$ gdb -q
(gdb) help
List of classes of commands:

aliases -- User-defined aliases of other commands.
breakpoints -- Making program stop at certain points.
data -- Examining data.
files -- Specifying and examining files.
internals -- Maintenance commands.
obscure -- Obscure features.
running -- Running the program.
stack -- Examining the stack.
status -- Status inquiries.
support -- Support facilities.
text-user-interface -- TUI is the GDB text based interface.
tracepoints -- Tracing of program execution without stopping the program.
user-defined -- User-defined commands.

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) quit
(base) liuwenjie@LWJ2025:~$
```

gdb -q 进入，help 查看帮助，quit 退出。

立志成才 报国裕民

GDB 常用命令速查

命令示例	常用缩写	作用
`run`	`r`	启动或重新运行程序
`break main`	`b main`	在 main 函数设置断点
`continue`	`c`	继续运行
`next`	`n`	在当前调用层级单步执行
`step`	`s`	单步执行，尝试进入函数
`finish`	`fin`	运行到当前选定函数返回
`print value`	`p value`	查看变量或表达式的值
`list`	`l`	查看源码
`backtrace`	`bt`	查看调用栈
`help break`	`h break`	查看某条命令的帮助
`quit`	`q`	退出 GDB

这些命令在 (gdb) 中输入；终端中的 `gdb ./程序` 用于启动调试器。

立志成才 报国裕民

让编译结果适合调试


```
mkdir -p build/demo
gcc -std=c17 -g -O0 -Wall -Wextra -Wpedantic -Werror \
    code/homework/hw1-p3.c -o build/demo/hw1-p3
```

- `-std=c17`：显式指定本课使用的 C17 语言标准。
- `-g`：加入调试信息；`-O0`：本次演示关闭优化，便于逐行观察。
- `-Wall -Wextra -Wpedantic`：开启常用编译警告。
- 修改源码后**重新编译**，确认正在调试的是新生成的程序。

优化会改变源码与机器指令的对应关系；出现 `optimized out` 时检查构建参数。

立志成才 报国裕民

HW1：先看调用者与辅助函数

主循环中的调用：

```
18          sum += x;
19          if (!hasValue) {
20              maximum = x;
21          } else {
22              maximum = updateMaximum(maximum, sum);
23          }
24          hasValue = 1;
```

更新最大值：

```
4   int updateMaximum(int maximum, int value) {
5       int result = maximum;
6       if (value > result) {
7           result = value;
8       }
9       return result;
10  }
```

处理输入 5 时，x == 5、sum == 8。

传入辅助函数的 value 应该是多少？实际又是多少？

先把失败输入固定下来

- 输入文件：`code/input/sum-max.txt`，内容为 `3 5 2 0`。
- 正常运行：`./build/demo/hw1-p3 < code/input/sum-max.txt`。
- 预期 `10 5`，实际 `10 10`；保留这个反例。

下一步要验证的猜想

总和正确，但更新最大值时可能比较了错误的数据。
观察调用者的 `x / sum` 与被调用函数的 `value / result`。

小输入可以手算，也方便逐步跟踪。

立志成才 报国裕民

第一次启动 GDB

在终端启动调试器：

```
gdb ./build/demo/hw1-p3
```

进入（gdb）提示符后输入：

```
break updateMaximum
run < code/input/sum-max.txt
```

- `break updateMaximum`：在更新最大值的函数设置断点。
- `run < ...`：使用同一份输入启动程序。
- 程序停住后，GDB 显示当前源码位置。

HW1：从断点现场定位传参错误

```
bash-4.4$ cd ./demo/HW1
bash-4.4$ source /data/liuwenjie/Myfile/CS100/slides/r4/build/tools/environment.sh
bash-4.4$ gdb HW1_P3
GNU gdb (GDB) Rocky Linux 8.2-20.1.el8_10
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from HW1_P3...done.
(gdb) break updateMaximum
Breakpoint 1 at 0x400600: file HW1_P3.c, line 5.
(gdb) run < sum-max.txt
Starting program: /data/liuwenjie/Myfile/CS100/demo/HW1/HW1_P3 < sum-max.txt

Breakpoint 1, updateMaximum (maximum=3, value=8) at HW1_P3.c:5
5           int result = maximum;
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-251.el8.x86_64
(gdb) print maximum
$1 = 3
(gdb) print value
$2 = 8
(gdb) bt
#0  updateMaximum (maximum=3, value=8) at HW1_P3.c:5
#1  0x0000000000400662 in main () at HW1_P3.c:22
(gdb)
```

断点已命中，检查传入的数据

- `maximum = 3`：此前读到的最大值，符合预期。
- `value = 8`：当前输入是 5，函数却收到了累计总和 $3 + 5 = 8$。
- `bt`：当前在 `updateMaximum`，调用来自 `main` 第 22 行。

检查调用处：将 sum 改为 x

`updateMaximum(maximum, x)`

当前停在第 5 行，`result` 尚未初始化；先执行 `next` 再查看它。

立志成才 报国裕民

断点：只在你关心的位置停下来

命令	作用
`break updateMaximum`	在函数入口设置断点
`break code/homework/hw1-p3.c:22`	在指定文件行设置断点
`info breakpoints`	查看断点编号与位置
`disable 1` / `enable 1`	禁用／启用编号 1 的断点
`delete 1`	删除编号 1 的断点
`continue`	继续执行，直到再次停止

当前行通常是即将执行的源码行。读取变量时，要留意该行是否已经执行。

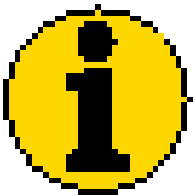
立志成才 报国裕民

观察变量：实际值符合预期吗？

在辅助函数中、`result` 初始化后停住，再使用：

```
print value
print result
display result
info locals
```

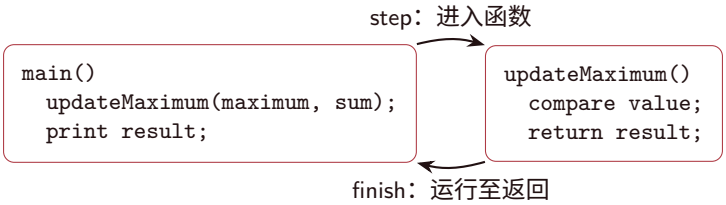
- `print (p)`：这一次查看一个变量或表达式。
- `display`：以后每次停止时自动显示表达式。
- `info locals`：查看当前栈帧中的局部变量。



截图：`main` 中的局部变量。

变量存在**作用域**；查看调用者的 `x` 和 `sum` 时，先切换到对应栈帧。

next、step 与 finish：函数调用处怎么走？



- `next`（`n`）：在当前调用层级向前走，通常跨过函数调用。
- `step`（`s`）：尝试进入有源码调试信息的被调用函数。
- `finish`：运行到当前选定函数返回之后。

遇到断点或信号可能提前停止。进入标准库内部时，可用 `finish` 返回。

立志成才 报国裕民

HW1 现场：step、next 与 finish 分别做了什么？

上海科技大学 ShanghaiTech University

```
(gdb) break HW1_P3.c:22
Breakpoint 1 at 0x400653: file HW1_P3.c, line 22.
(gdb) run < sum-max.txt
Starting program: /data/liuwenjie/Myfile/CS100/demo/HW1/HW1_P3 < sum-max.txt

Breakpoint 1, main () at HW1_P3.c:22
22                  maximum = updateMaximum(maximum, sum);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-251.el8.x86_64
(gdb) print x
$1 = 5
(gdb) print sum
$2 = 8
(gdb) print maximum
$3 = 3
(gdb) print hasValue
$4 = 1
(gdb) print value
No symbol "value" in current context.
(gdb) step
updateMaximum (maximum=3, value=8) at HW1_P3.c:5
5           int result = maximum;
(gdb) next
6           if (value > result) {
(gdb) print result
$5 = 3
(gdb) print value
$6 = 8
(gdb) finish
Run till exit from #0  updateMaximum (maximum=3, value=8) at HW1_P3.c:6
0x0000000000400662 in main () at HW1_P3.c:22
22                  maximum = updateMaximum(maximum, sum);
Value returned is $7 = 8
(gdb)
```

1 step：进入函数。

从 `main` 第 22 行进入 `updateMaximum` 第 5 行，收到 `maximum=3`，`value=8`。

2 next：执行初始化。

执行第 5 行后停在第 6 行；此时 `result=3`，`value=8`。

3 finish：运行到函数返回。

执行后续比较与返回，回到 `main`，显示返回值**8**。

若在第 22 行直接用 `next`，则跨过这次调用。截图中 `finish` 后仍显示第 22 行，可再 `next` 完成赋值并查看 `maximum`。

`value` 是辅助函数的形参，在 `main` 中不可直接查看。

立志成才 报国裕民

调用栈：是谁走到了这里？


```
bt
frame 1
print x
print sum
frame 0
```

#0 updateMaximum(3, 8)
当前函数

#1 main()
它的调用者

- bt：列出调用栈；frame 1：选中调用者的栈帧。
- 切换栈帧会改变你正在观察的局部变量上下文。
- **切换栈帧是在查看现场，不会让程序倒退执行。**

立志成才 报国裕民

崩溃或卡住时，也能查看现场

- 程序收到 SIGSEGV 等信号时，GDB 通常会停止。
- 使用 bt 查看调用链，再用 frame、print 检查参数与局部变量。
- 程序持续运行时，在 GDB 终端按 Ctrl+C 中断，再查看当前位置。
- 判断它是在等待输入、重复执行循环，还是停留在某次调用中。

崩溃位置是一条线索

如果停在标准库内部，沿调用栈找到自己的代码，检查传入的数据。
内存可能在更早的时候被破坏；结合 Sanitizer 继续追查。

条件断点：只观察传入 8 的这次调用


```
break updateMaximum if value == 8
run < code/input/sum-max.txt
next
print value
print result
```

- 处理第二个输入 5 时，总和已经变为 8；断点匹配 `value == 8`。
- `next` 执行初始化后，局部 `result == 3`，实参 `value == 8`。
- 用 `bt` 和 `frame 1` 回看调用者：**当前输入 x 明明是 5**。

条件表达式要使用断点位置可见的变量；重新演示前可删除旧断点。

观察点：这个变量什么时候变了？

在 updateMaximum 中、result 初始化为 3 后设置：

```
watch result
next
next
```

- watch result：值改变时停止，并显示新旧值。
- 单步执行比较与赋值，观察 result 从 3 变为 8。
- 函数正确比较了它收到的两个数；**调用者却传入了 sum 而不是 x**。

局部变量离开作用域后，相关观察点可能被删除；硬件观察点资源也有限。

现场演示一：找到第一次偏离预期的状态

1. 运行 HW1 课堂变体，确认 `3 5 2 0` 输出 `10 10`。
2. 在 `updateMaximum` 设置 `value == 8` 的条件断点。
3. 对照调用栈：调用者的 `x == 5`，函数收到的却是 `value == 8`。
4. 单步和观察点确认：这次调用把最大值从 3 更新为 8。
5. 把调用的第二个参数由 `sum` 改为 `x`，重新编译。
6. 重跑反例，再检查单个数据、重复最大值与 0 结束标记。

函数内部没有写错，也可能因为调用者传错参数而得到错误结果。

立志成才 报国裕民

Sanitizer：这次运行哪里不合法？

Sanitizer 是什么?

Sanitizer 是一组运行时错误检测工具。

编译时为程序加入额外检查；运行时检测到问题，就输出错误类别和位置等诊断信息。

ASan：AddressSanitizer

检查内存访问错误，例如：

- 数组越界访问。
- 内存释放后继续使用。

UBSan：UndefinedBehaviorSanitizer

检查特定未定义行为，例如：

- 有符号整数溢出。
- 整数除以零。

怎样使用?

编译时启用`-fsanitize=address,undefined`
$\longrightarrow$ 用测试输入运行程序 $\longrightarrow$ 根据报告定位并修复错误。

配合`-g` 生成调试信息，报告可以显示源码位置；只检查本次实际执行到的操作。

作业推荐命令里，为什么有 Sanitizer?

HW1 ／ HW2 的 README 给出了`-fsanitize=undefined` 和`-fsanitize=address`。

- **ASan**：帮助发现越界访问、释放后使用等内存错误。
- **UBSan**：帮助发现有符号溢出等特定未定义行为。
- **编译警告**检查可疑写法；运行时检查需要程序实际执行到出错操作。

对照作业要求选择实现

HW1 P3 不用数组或动态内存；HW2 P3 允许系数数组，禁止堆内存和 `math.h`。
本节先用固定系数数组和逐项累加，观察对应的两类报告。

HW2 P3：错误演示代码与检查构建

课堂调试用错误版：固定 $P(x) = 1 + 2x + 3x^2$，在 $x = 2$ 处求值，正确结果应为**17.000**。

```c
/* HW2 P3 classroom variant: a fixed polynomial, no heap and no
       math.h. */
#include <stdio.h>

double evaluate(const double *coef, int count, double x) {
    double value = 0.0;
    for (int i = count; i >= 0; --i) {
        value = value * x + coef[i];
    }
    return value;
}

int main(void) {
    const double coef[3] = {1.0, 2.0, 3.0};
    printf("%.3f\n", evaluate(coef, 3, 2.0));
    return 0;
}
```

完整演示副本：`demo/HW2/HW2_P3.c`。

在 `slides/r4/`中编译：

```
mkdir -p build/demo
gcc -std=c17 -g -O0 -Wall -Wextra -Wpedantic
      -Werror \
    -fsanitize=address,undefined \
    code/sanitizer/polynomial.c -o
      build/demo/polynomial-bounds
```

- `-g -O0` 便于定位源码与逐行观察。
- `-fsanitize=address,undefined` 启用 ASan 与 UBSan，链接时也需要。
- UBSan 报告错误后默认通常会继续执行；ASan 发现此类内存错误后通常会终止程序。

HW2 P3：解读这份 ASan 实测报告

终端实测报告（关键原文节选）：

```
==3800638==ERROR: AddressSanitizer: stack-buffer-overflow on address
      0x7ffe51589fa8 at pc 0x000000400af2 bp 0x7ffe51589f20 sp 0x7ffe51589f10
READ of size 8 at 0x7ffe51589fa8 thread T0
    #0 0x400af1 in evaluate /data/liuwenjie/Myfile/CS100/demo/HW2/HW2_P3.c:7
    #1 0x400c51 in main /data/liuwenjie/Myfile/CS100/demo/HW2/HW2_P3.c:14
Address 0x7ffe51589fa8 is located in stack of thread T0 at offset 56 in frame
    #0 0x400b41 in main /data/liuwenjie/Myfile/CS100/demo/HW2/HW2_P3.c:12

  This frame has 1 object(s):
    [32, 56) 'coef' <== Memory access at offset 56 overflows this variable
```

完整报告保留于课件源码；地址与进程号随运行变化。

1 错误类别

`stack-buffer-overflow`：栈上数组越界。

2 读取与位置

读取 8 字节，即本例一个 `double`；出错于第 7 行 `coef[i]`。

3 调用栈

`main` 第 14 行调用 `evaluate`，在其中触发错误。

4 数组范围

`[32, 56)` 共 24 字节；偏移 56 恰好越过末尾，对应 `coef[3]`。

循环从 count - 1 开始。
重新编译并运行，预期输出 17.000。

HW2 P3：多项式求值为什么越界了？


```
4  double evaluate(const double *coef, int count, double x) {
5      double value = 0.0;
6      for (int i = count; i >= 0; --i) {
7          value = value * x + coef[i];
8      }
9      return value;
10 }
```

课堂固定数据：

`coef = {1, 2, 3}`

$P(x) = 1 + 2x + 3x^2$

`count = 3`

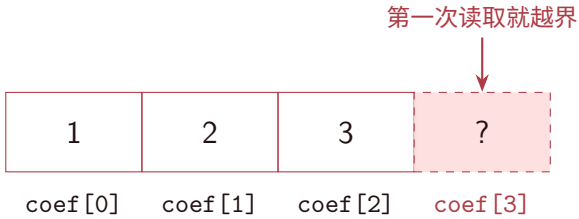
`x = 2`

合法结果为**17.000**。

从高次项向低次项递推：`value = value * x + coef[i]`。

第一次循环中，i 的值是否是合法下标?

三个系数，合法下标只有 0、1、2



- 初值 `i = count` 使第一次访问成为 `coef[3]`。
- 修复初值为 `count - 1`，依次访问下标 2、1、0。
- 本课堂变体使用栈上的固定数组；**没有引入堆分配或 VLA**。

第一次读懂 ASan 报告


```
ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 8
    #0 evaluate code/sanitizer/polynomial.c:7
    #1 main code/sanitizer/polynomial.c:14
[32, 56) 'coef' <== Memory access at offset 56 overflows this variable
```

1. **错误类别**：stack-buffer-overflow，栈上的越界访问。
2. **源码位置**：在 evaluate 函数读取 coef[i] 时出错。
3. **对象范围**：回到 main 中的固定数组，核对长度与循环初值。

此处为本课程序实测报告节选；省略地址、进程号和部分调用栈。

power：乘法在哪一轮超出范围？

固定计算 power(50000, 2)：

```c
#include <stdio.h>

int power(int x, int y) {
    int result = 1;
    for (int i = 0; i < y; ++i) {
        result *= x;
    }
    return result;
}

int main(void) {
    printf("%d\n", power(50000, 2));
    return 0;
}
```

第一次乘法得到 50000；第二次的数学结果为**2500000000**。

在 slides/r4/中启用 UBSan：

```
mkdir -p build/demo
gcc -std=c17 -g -O0 -Wall -Wextra -Wpedantic -Werror \
    -fsanitize=undefined \
    code/sanitizer/power.c -o build/demo/power-ubsan
./build/demo/power-ubsan
```

实测诊断（第 6 行）：

```
code/sanitizer/power.c:6:16: runtime error: signed integer
      overflow: 50000 * 50000 cannot be represented in type
      'int'
```

当前 32 位 int 上界是 2147483647，第二次乘法触发有符号溢出 UB。

立志成才 报国裕民

power：GDB 定位与修复验证

启动 `gdb -q build/demo/power-ubsan`，在乘法执行前停住：

```
break power.c:6 if i == 1
run
print i
print result
print x
bt
```

实测 `i=1`、`result=50000`、`x=50000`。

`bt` 显示当前在 `power`，由 `main` 调用；下一次乘法超出 `int` 范围。

修复乘积变量、返回类型与输出格式：

```c
#include <stdio.h>

long long power(int x, int y) {
    long long result = 1;
    for (int i = 0; i < y; ++i) {
        result *= x;
    }
    return result;
}

int main(void) {
    printf("%lld\n", power(50000, 2));
    return 0;
}
```

使用相同检查选项重新编译，输出**2500000000**，无 UBSan 诊断。

数组循环：最后一次迭代访问了哪里？

目标：让 6 个数组元素分别加 1。

```c
#include <stdio.h>

int main(void) {
    int data[6] = {0};
    for (int i = 0; i <= 6; ++i) {
        data[i]++;
    }
    printf("%d\n", data[0]);
    return 0;
}
```

循环包含 `i == 6`；此时访问 `data[6]`，越过数组末尾。

在 `slides/r4/`中单独启用 ASan：

```
mkdir -p build/demo
gcc -std=c17 -g -O0 -Wall -Wextra -Wpedantic -Werror \
    -fsanitize=address \
    code/sanitizer/array_loop.c -o build/demo/array-loop-asan
./build/demo/array-loop-asan
```

实测报告节选：

```
ERROR: AddressSanitizer: stack-buffer-overflow
READ of size 4
#0 0x400a3e in main code/sanitizer/array_loop.c:6
[32, 56) 'data' <== Memory access at offset 56 overflows this
      variable
```

自增先读后写，本次先报告读取 4 字节；`data` 范围为 `[32, 56)`，偏移 56 已越界。

数组循环：用条件断点验证修复

启动 `gdb -q build/demo/array-loop-asan`，在越界访问之前停止：

```
set print repeats 0
break array_loop.c:6 if i == 6
run
print i
print data
print sizeof data
```

实测 `i=6`，数组内容为 6 个 1，`sizeof data=24`。

前 6 轮已完成全部元素的更新；**第 7 次迭代多执行了一次。**

修复：将循环条件改为 `i < 6`。

```c
#include <stdio.h>

int main(void) {
    int data[6] = {0};
    for (int i = 0; i < 6; ++i) {
        data[i]++;
    }
    printf("%d\n", data[0]);
    return 0;
}
```

重新编译运行，输出**1**，无 ASan 诊断。

完整修复版：`code/fixed/array_loop.c`。

立志成才 报国裕民

没有报告，还需要检查什么？

- **测试覆盖**：本次没执行到的代码，运行时检查没有替你验证。
- **工具覆盖**：ASan 和 UBSan 各有检测范围，不能覆盖所有语言错误。
- **业务逻辑**：最大值更新用了总和、漏算常数项，都可能没有报告。
- **修复验证**：重新编译，重跑失败输入，再加正常与边界输入。

把工具放进同一个调试过程

Sanitizer 帮助定位非法操作，GDB 帮助观察状态与执行路径。
结合语言规则与题目要求，解释为什么出错、为什么修复有效。

C 语言程序的常见错误与调试

HW1 累加操作：函数里改了，外面为什么没变？

将求和操作放入一个 C 函数，用固定的小数据观察参数传递：

```c
void addValue(int sum, int x) {
    sum += x;
}
```

```c
int main(void) {
    int sum = 3;
    addValue(sum, 5);
    printf("%d\n", sum);
    return 0;
}
```

- 预期总和 8，实际仍为 3。
- 函数里的加法执行了；接下来应该观察哪一个 sum？

HW1 累加示例：为什么没改成功？怎样修复？

本次实测将加数改为 3：预期 6，实际 3。

```
(gdb) print sum
$1 = 3
(gdb) step
addValue (sum=3, x=3) at HW1_P3_sum.c:4
4           sum += x;
(gdb) next
5       }
(gdb) print sum
$2 = 6
(gdb) finish
main () at HW1_P3_sum.c:10
10          printf("%d\n", sum);
(gdb) print sum
$3 = 3
```

原因：修改的是形参副本。

函数内的 `sum` 变为 6；`finish` 回到 `main` 后，原变量仍为 3。

正确修改：传入地址，修改原变量。

```c
/* Classroom accumulation fix: add 3 through a
     pointer. */
#include <stdio.h>

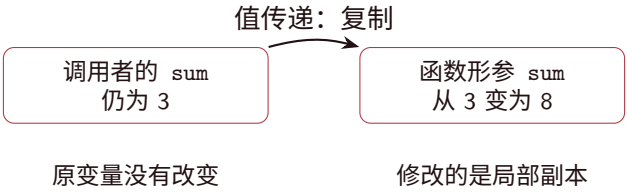
void addValue(int *sum, int x) {
    *sum += x;
}

int main(void) {
    int sum = 3;
    addValue(&sum, 3);
    printf("%d\n", sum);
    return 0;
}
```

`&sum` 传入原变量的地址，`*sum` 访问并修改它；输出**6**。

完整修复版：
`code/fixed/sum-three.c`。

C 的参数按值传递：怎样修改原变量？



- GDB 中在 `addValue` 内观察 `sum`，再 `finish` 回到调用者比较。
- 需要修改原变量时，可传入地址 `&sum`，用形参 `int *sum` 接收，再通过 `*sum` 修改。
- 另一种设计是返回新总和，由调用者接收返回值。

C 的参数都是值传递；传入指针时，复制的是地址值。

C17 中的行为：UB、实现定义与有定义

用整数、字符与数组示例，区分语言规则和题目要求。

类别	标准给了什么保证？	本课例子
UB：未定义行为 Undefined Behavior	标准不规定程序应如何表现；不保证报错。	有符号溢出、越界访问。
实现定义行为 Implementation-defined Behavior	实现作出选择，并在文档中说明。	普通 `char` 的符号属性。
有定义的行为 Well-defined Behavior	按明确的语言规则执行。	无符号回绕、合法整数除法。

有定义，也可能不符合题意

`3 / 2` 得到 1 符合 C 规则；若题目需要 1.5，仍然是逻辑错误。

这不是穷尽分类：还有**未指定行为**。**编译错误**需先处理，例如使用未声明标识符。

立志成才 报国裕民

有符号溢出与无符号回绕

上海科技大学
ShanghaiTech University

有符号 int：UB

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    int value = INT_MAX - 2;
    value += 3;
    printf("%d\n", value);
    return 0;
}
```

数学结果是 `INT_MAX + 1`，超出 `int` 范围；不能依赖回绕结果。

unsigned int：有定义

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    unsigned int value = UINT_MAX - 2;
    value += 3;
    printf("%u\n", value);
    return 0;
}
```

按该无符号类型的取值个数取模，输出**0**；这是语言规定的回绕。

UBSan 可报告左侧溢出；右侧通常没有诊断。**改变类型，就可能改变行为分类。**

立志成才 报国裕民

printf 格式必须匹配实参类型

格式与实参不匹配：UB

```
4     double value = 9.0;
5     printf("%d\n", value / 4);
```

value / 4 是 double，%d 却要求 int；格式串不会替你转换参数。

修复格式，输出2.25：

```
5     printf("%.2f\n", value / 4);
```

字符整数提升：有定义

```
4     char digit = '3';
5     printf("%c\n", digit + 4);
```

本课环境中 char 提升为 int，与%c 需要的参数类型匹配。

数字字符连续，'3' + 4 对应'7'，因此输出7。

-Wall 可提示左侧格式错误，-Werror 使本课构建失败；允许编译通过也不消除运行时 UB。

立志成才 报国裕民

char 的符号属性由实现定义


```c
#include <stdio.h>

int main(void) {
    char value = -1;
    printf("%d\n", value);
    return 0;
}
```

普通 char 与 signed char、unsigned char 是不同的类型。

本课 8 位 char 环境下，用 GCC 显式选择：

编译选项	输出
-fsigned-char	-1
-funsigned-char	255

- 默认选择由目标平台与编译器配置决定，并有文档说明。
- 这里的差异来自**实现定义的选择**，不是随机结果。

需要明确的符号属性时，直接使用 signed char 或 unsigned char。

数组越界与编译错误要分开

访问 data[6]：UB

```
4       int data[6] = {0};
5       for (int i = 0; i <= 6; ++i) {
6           data[i]++;
7       }
```

6 个元素的合法下标为 0 至 5；`i == 6` 时越界，Sanitizer 可帮助定位。

修复循环条件为 `i < 6`，每个元素增加一次，`data[0]` 为 1。

show 中找不到 number：编译错误

```
3   void show(void) {
4       printf("%d\n", number);
5   }
6
7   int main(void) {
8       int number = 17;
9       show();
10      return 0;
11  }
```

`main` 的局部变量不在 `show` 的作用域内；可通过形参传入：`show(number)`。

判断顺序：先检查声明、作用域与类型，再检查运行时边界；**有定义后仍要核对题意。**

小练习：HW2 多项式没有越界，结果仍然错了


```
5  double evaluate(const double *coef, int count, double x) {
6      double value = 0.0;
7      for (int i = count - 1; i > 0; --i) {
8          value = value * x + coef[i];
9      }
10     return value;
11 }
```

仍使用课堂固定数据：

$P(x) = 1 + 2x + 3x^2$

$x = 2$

预期 17.000，

实际8.000。

这次每个已访问下标都合法，应该检查什么？

立志成才 报国裕民

先写下调试计划，再动手改代码

1. **复现**：运行 `./build/demo/exercise`，对照预期 `17.000`。
2. **预测**：下标 2、1、0 都应被处理，中间结果应依次为 3、8、17。
3. **观察**：在 `evaluate` 中跟踪 `i` 和 `value`。
4. **追踪**：执行完下标 1 后，是否还有下标 0 这一轮？
5. **解释**：循环退出条件漏掉了哪个系数？

给自己一个检查点

已经访问的地址都合法，不代表算法完成了所有必要步骤。
修复之后，再用一个能手算的多项式检查你的理解。

练习解答：不能漏掉常数项


```
5  double evaluate(const double *coef, int count, double x) {
6      double value = 0.0;
7      for (int i = count - 1; i >= 0; --i) {
8          value = value * x + coef[i];
9      }
10     return value;
11 }
```

- 原来的 `i > 0` 跳过了 `coef[0]` 对应的最后一步。
- 修复为 `i >= 0`，递推结果由 8 变为 $8 \times 2 + 1 = 17$。
- 区分两类边界错误：**多访问一步可能越界；少执行一步可能只产生错误结果**。

修复版 `./build/demo/exercise-fixed` 输出 `17.000`。

立志成才 报国裕民

下次出错时，照这个顺序开始

1. 保存源码，记录构建命令，阅读编译警告。
2. 写出最小失败输入、预期输出与实际输出。
3. 运行 Sanitizer 构建；如有报告，定位并解释非法操作。
4. 对未解释的错误，用 GDB 观察关键状态，验证具体猜想。
5. 结合类型、边界、传递方式与生命周期解释原因。
6. 重新编译，重跑失败输入与边界测试，保留修复记录。

调试记录应该回答：证据是什么？原因是什么？为什么这个修复有效？

立志成才 报国裕民

一页速查：先选你需要的观察方式

想做什么？	使用什么？
从固定输入开始	`run < input.txt`
在指定位置停止	`break file.c:line`
继续／单步／进入／返回	`continue / next / step / finish`
查看当前值／自动显示	`print / display / info locals`
查看调用关系／选择现场	`bt / frame`
只观察某一轮	`break file.c:line if condition`
追踪值的变化	`watch expression`
检查本课内存与 UB 案例	`-fsanitize=address,undefined`

GDB 内可用 `help` 命令名查帮助；退出使用 `quit`。

课后复现与参考资料

- **构建案例**：`make demos`。
- **验证案例**：`make check-demos`。
- **GDB 演示脚本**：`code/commands/session.gdb`。
- **修复版与演示说明**：`code/fixed/`、作业来源说明与本目录 `README.md`。

- 工程师知识库：GDB 调试器基础使用——命令行调试完全指南。
- GDB 官方手册：断点、单步、观察点、调用栈。
- GCC 官方手册：Instrumentation Options。
- C 标准草案 N1570与GCC 字符符号选项说明。
- ASan 文档与UBSan 文档。

立志成才 报国裕民

Thank You

复现、观察、解释、验证