# Debugging

Lecture 3

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

# Another C Program Example

The program below computes the average of the following integers: 2, 2, 3, 4 and 5:

**average.c**

```c
#include <stdio.h>

float Average( int *w, int n ) {
    int i;
    int avg;
    for ( i = 0; i < n; i++ )
        avg += w[i];

    return avg / n;
}

int main ( void ) {
    int x[5] = {2, 2, 3, 4, 5};
    int num = 5;
    float avg = Average( x, num );
    printf( "Average: %f\n", avg );
    return 0;
}
```

**Oops!** We were expecting the **average** to be **3.2**! **How do we approach this problem?**

**DIY!**

# Debuggers

- “(...) *a computer program that is used to test and debug other programs* (...).”

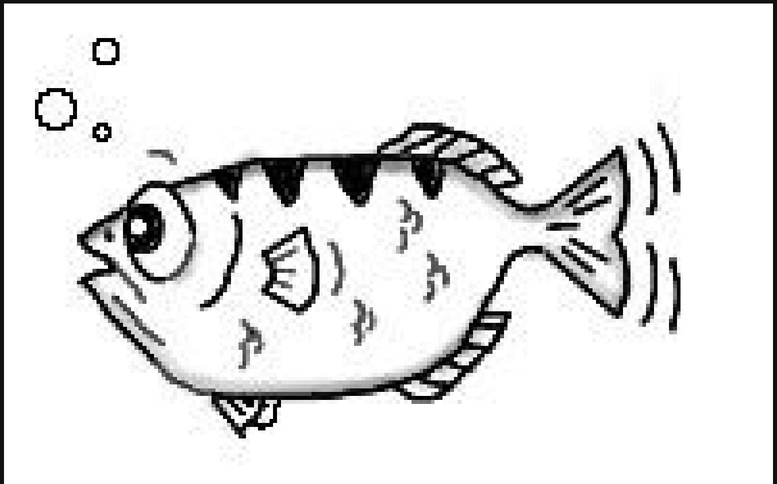
  Debugger, Wikipedia.

- Examples:
  - Microsoft Visual Studio Debugger.
  - LLDB.
  - GDB.

# GDB

## GDB, or the GNU Project Debugger

- Popular debugger tool used among Unix/Linux programmers.
- It comes, usually, pre–installed in several Linux distributions.



**The Archer Fish, the GDB mascot.**

# GDB

Features (as of version 7.10):

- Can be used to debug C and C++ programs.
- Partial support to some other languages.
- Text−based.
- “Normal”, temporary and conditional breakpoints.
- Single−stepping.
- Resume.
- Watchpoints.
- Variable inspection.
- Call stack inspection.
- Etc.

# GDB Usage Example

Back to our broken C program:

**average.c**

```
#include <stdio.h>

float Average( int *w, int n ) {
    int i;
    int avg;
    for ( i = 0; i < n; i++ )
        avg += w[i];

    return avg / n;
}

int main ( void ) {
    int x[5] = {2, 2, 3, 4, 5};
    int num = 5;
    float avg = Average( x, num );
    printf( "Average: %f\n", avg );
    return 0;
}
```

Recompile the program with the following command:

```
~$ gcc -g3 average.c -o average
```

**Inserts debugging information into the executable.**

# GDB Usage Example

- After the recompilation, invoke GDB on the executable file:

```
~$ gdb ./average
```

- From within the GDB environment, let's issue the following commands:
  - List the source code from within GDB:

```
(gdb) list 1, 100
```

or

```
(gdb) l 1, 100
```

  - Set a breakpoint at line 13:

```
(gdb) break 13
```

or

```
(gdb) b 13
```

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Run the program:

```
(gdb) run
```

or

```
(gdb) r
```

- Lets inspect the value of variable **x**:

```
(gdb) print x
```

or

```
(gdb) p x
```

- Execute line 13 (just one step):

```
(gdb) step
```

or

```
(gdb) s
```

- Lets inspect the value of variable **x** again:

```
(gdb) p x
```

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Which is the next line to be executed?

```
(gdb) frame
```

or

```
(gdb) f
```

- One more step:

```
(gdb) s
```

- Step over the call to **`Average()`**:

```
(gdb) next
```

or

```
(gdb) n
```

**Differently from `step`, `next` do not step into functions!**

- Inspect the value of variable **`avg`**:

```
(gdb) p avg
```

**It seems that the problem is within the `Average()` function!**

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Let's restart the program:

```
(gdb) r
```

- We've got stuck at line 13 again! First, let's print breakpoint information for the program:

```
(gdb) info breakpoint
```

or

```
(gdb) info b
```

- Now, delete breakpoint 1:

```
(gdb) delete 1
```

or

```
(gdb) d 1
```

- Step until we reach line 15.

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Now, let's step into the function **`Average()`**:

```
(gdb) s
```

→ **`step` steps into functions!**

- Let's check the value of the local variable **`avg`**:

```
(gdb) p avg
```

→ **`avg` was not properly initialized!**

- Initialize **`avg`** with 0!
- Now, recompile the program (**do not close GDB!**).
- Rerun the program, without breakpoints, and check the answer. → **It seems that we still have a problem!**

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Set a breakpoint at the function **Average()**:

```
(gdb) b Average
```

- Rerun the program (it will stop within **Average()**).
- Set a watchpoint for when the loop finishes (**i == n**):

```
(gdb) watch i == n
```

- Continue until next breakpoint / watchpoint.

```
(gdb) continue
```

or

```
(gdb) c
```

# GDB Usage Example

From within the GDB environment, let's issue the following commands (cont.):

- Check the value of variables **`avg`** and **`n`**.
- Step until we leave **`Average()`**.
- Check the value that was returned by the function.

Damn! The value is incorrect! So what????

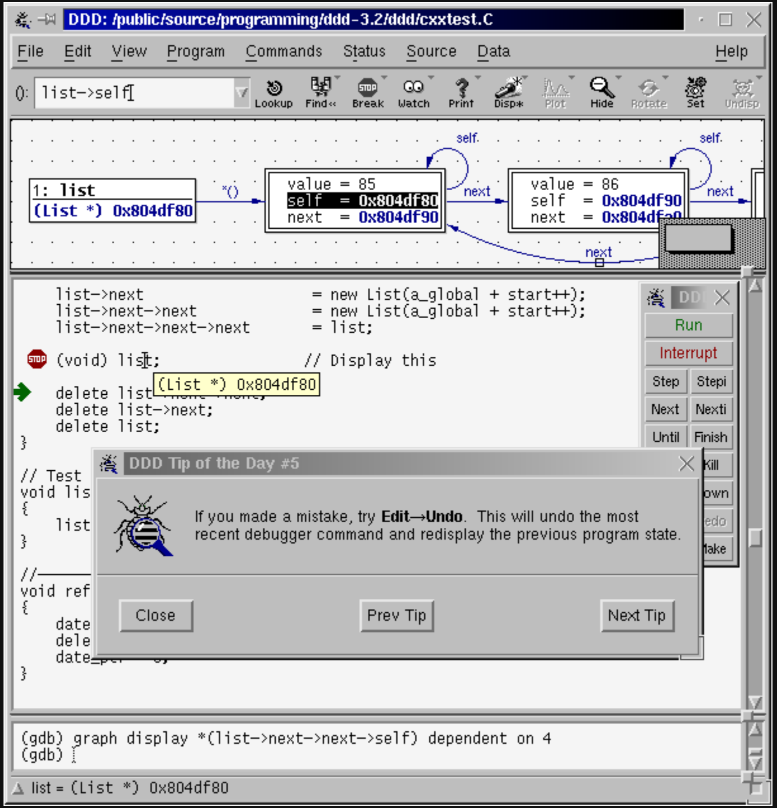
- The value returned by **`Average()`** is the result of a integer division! We have to cast one of the operators (e.g. **`(float) avg/n`** ) to force a float division!
- Apply cast, recompile, delete all breakpoints and rerun!

# GDB Summary

- GDB is a quite powerful debugger tool.
- However, GDB does no allow one to easily follow the source code during a debug section.
- Some **GDB front ends** were developed, most notably:
  - CTRL + x + a : splits the GDB screen in command and source code windows (buggy!).
  - cgdb: curses based GDB front end.
  - Eclipse: an IDE that may use GDB as its debugging tool.
  - DDD : the Data Display Debugger.

# The Data Display Debugger (DDD)

- It is a graphical interface to GDB.



```
value = 85
self  = 0x804df80
next  = 0x804df90
```

```
value = 86
self  = 0x804df90
next  = 0x804dfa0
```

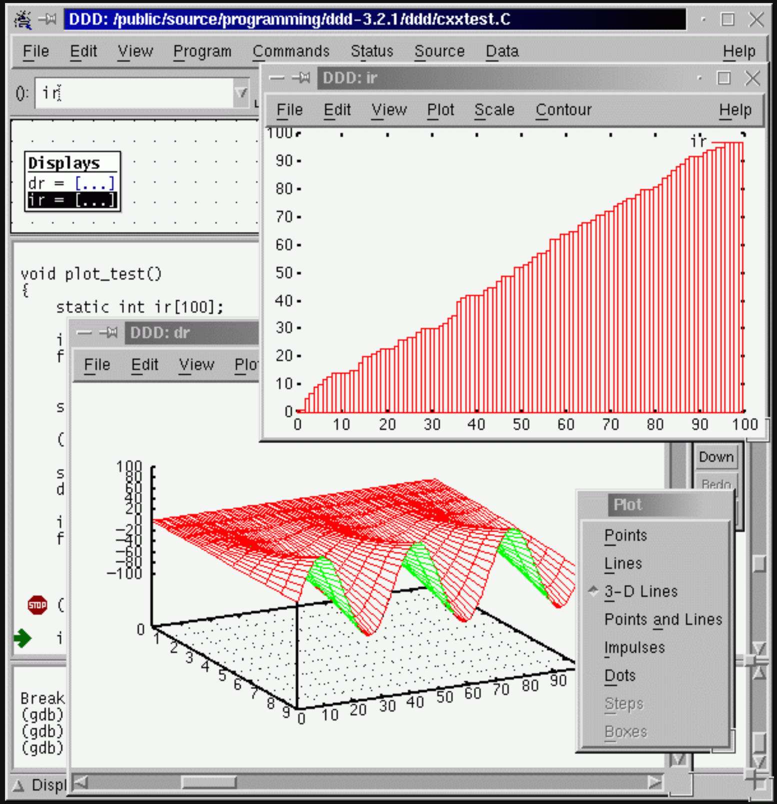
```
    list->next                  = new List(a_global + start++);
    list->next->next            = new List(a_global + start++);
    list->next->next->next      = list;

    (void) list;               // Display this
                                (List *) 0x804df80
    delete list
    delete list->next;
    delete list;
}

// Test
void lis
{
    list
}

//———
void ref
{
    date
    dele
    date
}
```

DDD Tip of the Day #5

If you made a mistake, try **Edit→Undo**. This will undo the most recent debugger command and redisplay the previous program state.

```
(gdb) graph display *(list->next->next->self) dependent on 4
(gdb)
```

```
list = (List *) 0x804df80
```



```
Displays
dr = [...]
ir = [...]
```

```
void plot_test()
{
    static int ir[100];
```

```
Break
(gdb)
(gdb)
(gdb)
```

- More on:
  - https://www.gnu.org/software/ddd

# Back to C Data Types... Pointers!

A **pointer** variable is a **memory location** into which **data** (i.e. a **memory address**) can be **stored**.

# Pointer Variable Example

Compile and run the following code from within GDB:

**example_23.c**

```
int main( void ) {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

x = 25; → $00000000000000000000000000011001_2$

- `(gdb) b 4` → Set a breakpoint at line 4.
- `(gdb) r` → Run (stops at line 4).
- `(gdb) p &x` → Print the address of `x`.
- `(gdb) p x` → Print the integer value stored at `x`.
- `(gdb) s`

byte →

|   | Addr. | Value |
|---|---|---|
|   | addr1 - 1 | . . . |
| x | addr1 + 0 | . . . |
|   | addr1 + 1 | . . . |
|   | addr1 + 2 | . . . |
|   | addr1 + 3 | . . . |

|   | Addr. | Value |
|---|---|---|
|   | addr2 + 0 | . . . |
|   | addr2 + 1 | . . . |
|   | addr2 + 2 | . . . |
|   | addr2 + 3 | . . . |
|   | addr2 + 4 | . . . |
|   | addr2 + 5 | . . . |
|   | addr2 + 6 | . . . |
|   | addr2 + 7 | . . . |

DIY!

# Pointer Variable Example

Compile and run the following code from within GDB:

**example_23.c**

```
int main( void ) {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

$00000000000000000000000000011001_2$

$0000000000000000011111111111111111111111111111111101110100000100_2$

| | |
|---|---|
| `(gdb) b 4` | Set a breakpoint at line 4. |
| `(gdb) r` | Run (stops at line 4). |
| `(gdb) p &x` | Print the address of `x`. |
| `(gdb) p x` | Print the integer value stored at `x`. |
| `(gdb) s` | Execute line 4 (`x = 25`). |
| `(gdb) p x` | Print the integer value stored at `x`. |
| `(gdb) p px` | Print the address stored at `px`. |
| `(gdb) s` | |

byte

**x**

| Addr. | Value |
|---|---|
| addr1 - 1 | ???????? |
| addr1 + 0 | 00011001 |
| addr1 + 1 | 00000000 |
| addr1 + 2 | 00000000 |
| addr1 + 3 | 00000000 |

**px**

| Addr. | Value |
|---|---|
| addr2 + 0 | . . . |
| addr2 + 1 | . . . |
| addr2 + 2 | . . . |
| addr2 + 3 | . . . |
| addr2 + 4 | . . . |
| addr2 + 5 | . . . |
| addr2 + 6 | . . . |
| addr2 + 7 | . . . |

# Pointer Variable Example

Compile and run the following code from within GDB:

example_23.c

```
int main( void ) {
     int x;
     int *px;
     x = 25;
     px = &x;

     return 0;
}
```

x = 25; → $00000000000000000000000000011001_2$

px = &x; → $0000000000000000011111111111111111111111111111111101110100000100_2$

```
(gdb) b 4    → Set a breakpoint at line 4.
(gdb) r      → Run (stops at line 4).
(gdb) p &x   → Print the address of x.
(gdb) p x    → Print the integer value stored at x.
(gdb) s      → Execute line 4 (x = 25).
(gdb) p x    → Print the integer value stored at x.
(gdb) p px   → Print the address stored at px.
(gdb) s      → Execute line 5 (px = &x).
(gdb) p px   → Print the address stored at px.
```

byte →

x

| Addr. | Value |
|---|---|
| addr1 - 1 | ???????? |
| addr1 + 0 | 00011001 |
| addr1 + 1 | 00000000 |
| addr1 + 2 | 00000000 |
| addr1 + 3 | 00000000 |

px

| Addr. | Value |
|---|---|
| addr2 + 0 | 00000100 |
| addr2 + 1 | 11011101 |
| addr2 + 2 | 11111111 |
| addr2 + 3 | 11111111 |
| addr2 + 4 | 11111111 |
| addr2 + 5 | 01111111 |
| addr2 + 6 | 00000000 |
| addr2 + 7 | 00000000 |

# Examining Data with GDB: `print`

- `print` is the most **common** way to **examine data**, and is based on **expression evaluation.**
- `print` is able to **format** the **output!**
- Back to the previous example: set a breakpoint at line 7 (`return 0`) and run.

**`example_23.c`**

```c
int main( void ) {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

**Experiment `print` with these arguments:**

```
(gdb) p x
(gdb) p /x x
(gdb) p /t x
```

- `(gdb) p x` → **Print the integer value stored at `x`.**
- `(gdb) p /x x` → **Print the value at `x` in hexa format.**
- `(gdb) p /t x` → **Print the value at `x` in binary format.**

# Examining Data with GDB: `x`

- The `x` command allows for **low-level** data examination.
- It prints the **contents** of **memory** positions in a specified **format**.

# Examining Data with GDB: x

Back to the previous example: set a breakpoint at line 7 (**return** 0) and run.

**example_23.c**

```
int main( void ) {
    int x;
    int *px;
    x = 25;
    px = &x;

    return 0;
}
```

|  | Addr. | Value |
|---|---|---|
|  | addr1 - 1 | ???????? |
| x | addr1 + 0 | 00011001 |
|  | addr1 + 1 | 00000000 |
| byte → | addr1 + 2 | 00000000 |
|  | addr1 + 3 | 00000000 |

|  | Addr. | Value |
|---|---|---|
| px | addr2 + 0 | 00000100 |
|  | addr2 + 1 | 11011101 |
|  | addr2 + 2 | 11111111 |
|  | addr2 + 3 | 11111111 |
|  | addr2 + 4 | 11111111 |
|  | addr2 + 5 | 01111111 |
|  | addr2 + 6 | 00000000 |
|  | addr2 + 7 | 00000000 |

Experiment **x** with these arguments:

|  |  |
|---|---|
| `(gdb) x &x` → | Print the value at address &**x** (last format). |
| `(gdb) x/t &x` → | Print the value at address &**x** in binary. |
| `(gdb) x/d &x` → | Print the value at address &**x** in decimal. |
| `(gdb) x/4tb &x` → | Print 4 bytes in binary starting at addr. &**x**. |
| `(gdb) x/8tb &px` → | Print 8 byt. in binary starting at addr. &**px**. |

# Scripting GDB

What if we would like to print the intermediary values of the summation below?

**summation.c**

```c
#include <stdio.h>

int Sum( int begin, int end ) {
    int i;
    int acc = 0;
    for ( i = begin; i <= end; i++ )
        acc += i;

    return acc;
}

int main ( void ) {
    int a = 1;
    int b = 5;
    int sum = Sum( a, b );
    printf( "Sum: %i\n", sum );
    return 0;
}
```

We can automate it with **GDB scripting**, thus avoiding code modifications!

# Scripting GDB

The following GDB script dumps on the screen all intermediary values generated during the summation computation:

**sumdebug.gdb**

```
set width 0
set height 0
set verbose off

b 8

commands 1
    silent
    printf "acc = %i\n", acc
    continue
end

b 10

commands 2
    silent
    printf "acc = %i\n", acc
    continue
end

run
```

**Invoking GDB**

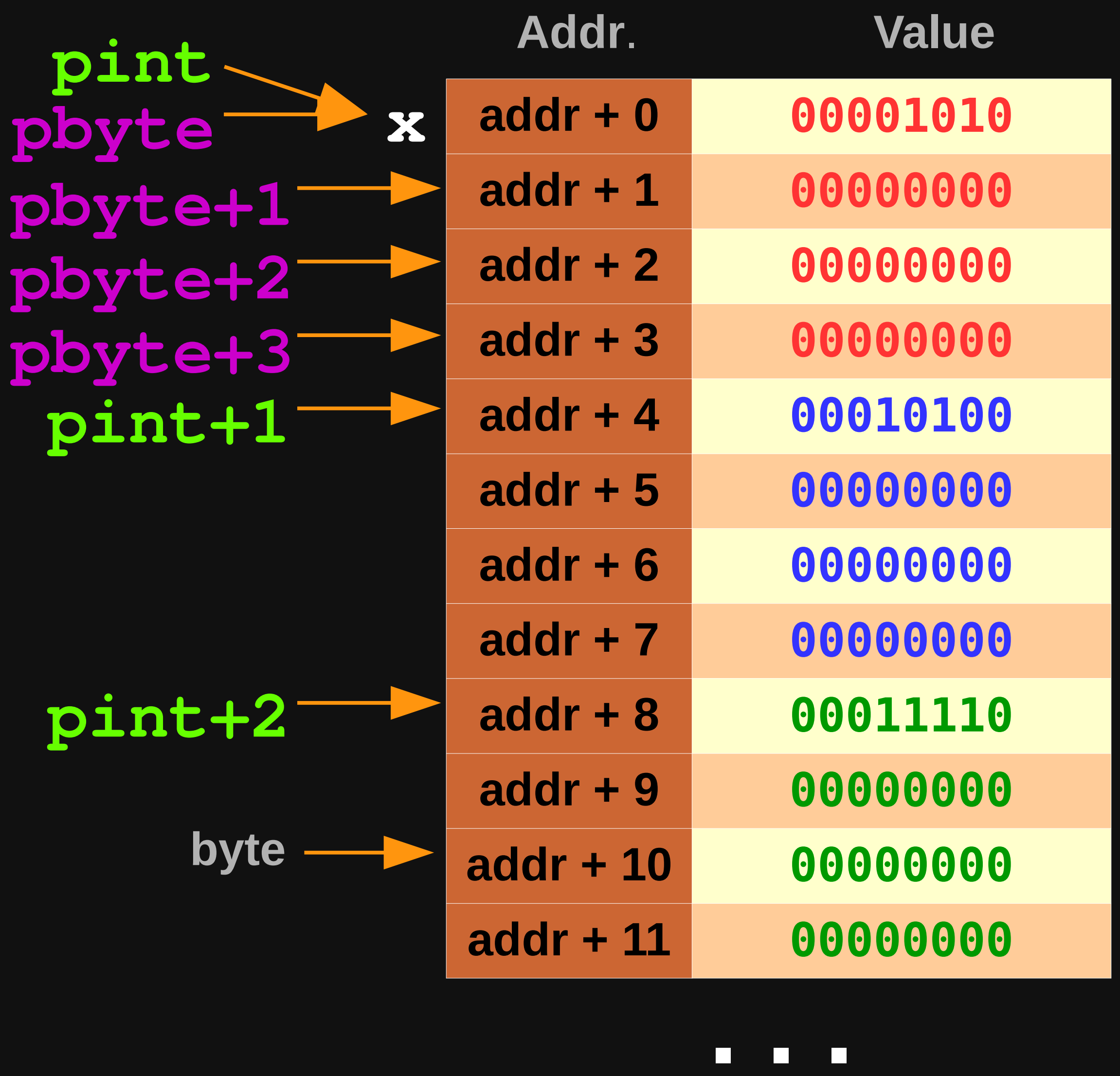
```
~$ gdb --batch --command=sumdebug.gdb a.out
```

DIY!

# Pointer Arithmetic

example_24.c

```c
int x[4] = { 10, 20, 30, 40 };

int main( void ) {
    int *pint = x;

    *pint = 0;
    *(pint + 1) = 0;
    *(pint + 2) = 0;
    *(pint + 3) = 0;

    char *pbyte = ( char* ) x;

    *pbyte = 255;
    *(pbyte + 1) = 255;
    *(pbyte + 2) = 255;
    *(pbyte + 3) = 255;

    return 0;
}
```



| Addr. | Value |
|---|---|
| addr + 0 | 00001010 |
| addr + 1 | 00000000 |
| addr + 2 | 00000000 |
| addr + 3 | 00000000 |
| addr + 4 | 00010100 |
| addr + 5 | 00000000 |
| addr + 6 | 00000000 |
| addr + 7 | 00000000 |
| addr + 8 | 00011110 |
| addr + 9 | 00000000 |
| addr + 10 | 00000000 |
| addr + 11 | 00000000 |

# Pointer Arithmetic

**example_24.c**

```c
int x[4] = { 10, 20, 30, 40 };

int main( void ) {
    int *pint = x;

    *pint = 0;
    *(pint + 1) = 0;
    *(pint + 2) = 0;
    *(pint + 3) = 0;

    char *pbyte = ( char* ) x;

    *pbyte = 255;
    *(pbyte + 1) = 255;
    *(pbyte + 2) = 255;
    *(pbyte + 3) = 255;

    return 0;
}
```

**Dereferencing can be equivalently rewritten with [ ]!**

**example_25.c**

```c
int x[4] = { 10, 20, 30, 40 };

int main( void ) {
    int *pint = x;

    pint[0] = 0;
    pint[1] = 0;
    pint[2] = 0;
    pint[3] = 0;

    char *pbyte = ( char* ) x;

    pbyte[0] = 255;
    pbyte[1] = 255;
    pbyte[2] = 255;
    pbyte[3] = 255;

    return 0;
}
```

**Looks familiar?**

# References

**Learning C with GDB**. Alan O' Donnell.

- https://www.recurse.com/blog/5-learning-c-with-gdb