ECE/ME/EMA/CS 759
High Performance Computing for Engineering Applications

Conclusion, Quick Overview of C Programming

`gdb`

Logging in to Euler

The **`Eclipse`** IDE

The Hardware/Software Interplay

September 9, 2013

"Be who you are and say what you feel, because those who mind don't matter and those who matter don't mind."
Dr. Seuss

Before We Get Started…

- Last time
  - Brief overview of C
  - Most important concepts covered: pointers and memory layout

- Today
  - Wrap up quick overview of C Programming
  - Super quick intro to `gdb` (debugging tool under Linux)
  - Learn how to login and use Euler, the CPU/GPU cluster
  - Basic tidbits about how computers are organized and how they work

- First assignment made available on the class website later today
  - HW01 due on Mo, September 17, at 11:59 PM (Learn@UW cutoff time)
  - Post related questions to the forum

Function Types

Another less obvious construct is the "**pointer to function**" type.
For example, qsort: (a sort function in the standard library)

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *) );
```

The last argument is a comparison function

```
/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type) (const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

const means the function is not allowed to modify memory via this pointer.

size_t is an unsigned int

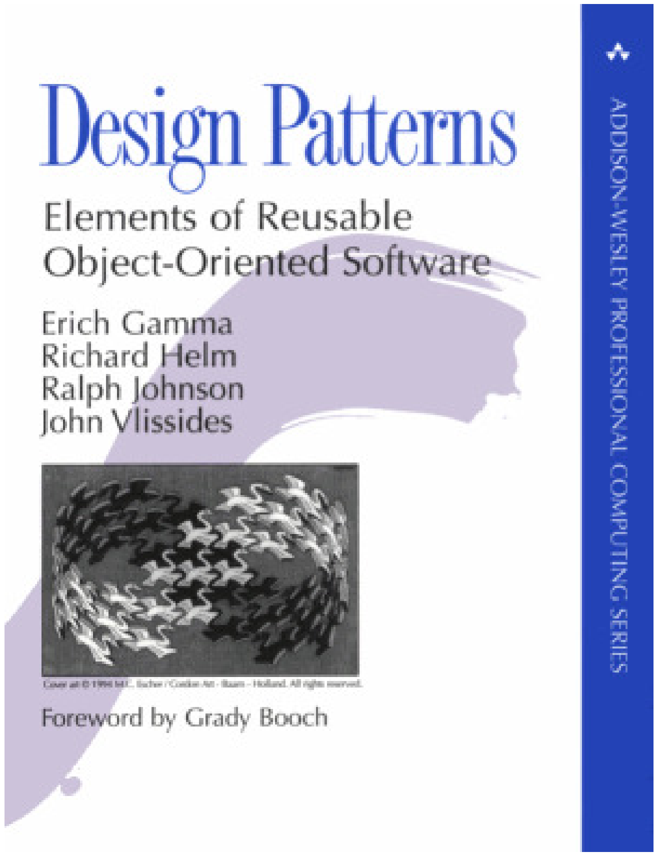
void * is a pointer to memory of unknown type.

Why is Software Development Hard?

- Complexity: Every conditional ("if") doubles the number of paths through your code, every bit of state doubles possible states
  - Recommendation: reuse code with functions, avoid duplicate state variables
- Mutability: Software is easy to change.. Great for rapid fixes… And rapid breakage…
  - Recommendation: tidy, readable code, easy to understand by inspection, provide *plenty* of meaningful comments.
- Flexibility: Problems can be solved in many different ways. Few hard constraints, easy to let your horses run wild
  - Recommendation: discipline and use of design patterns

Design Patterns

- A good book if you are serious about programming


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Quiz: Usage of Pointers & Pointer Arithmetic

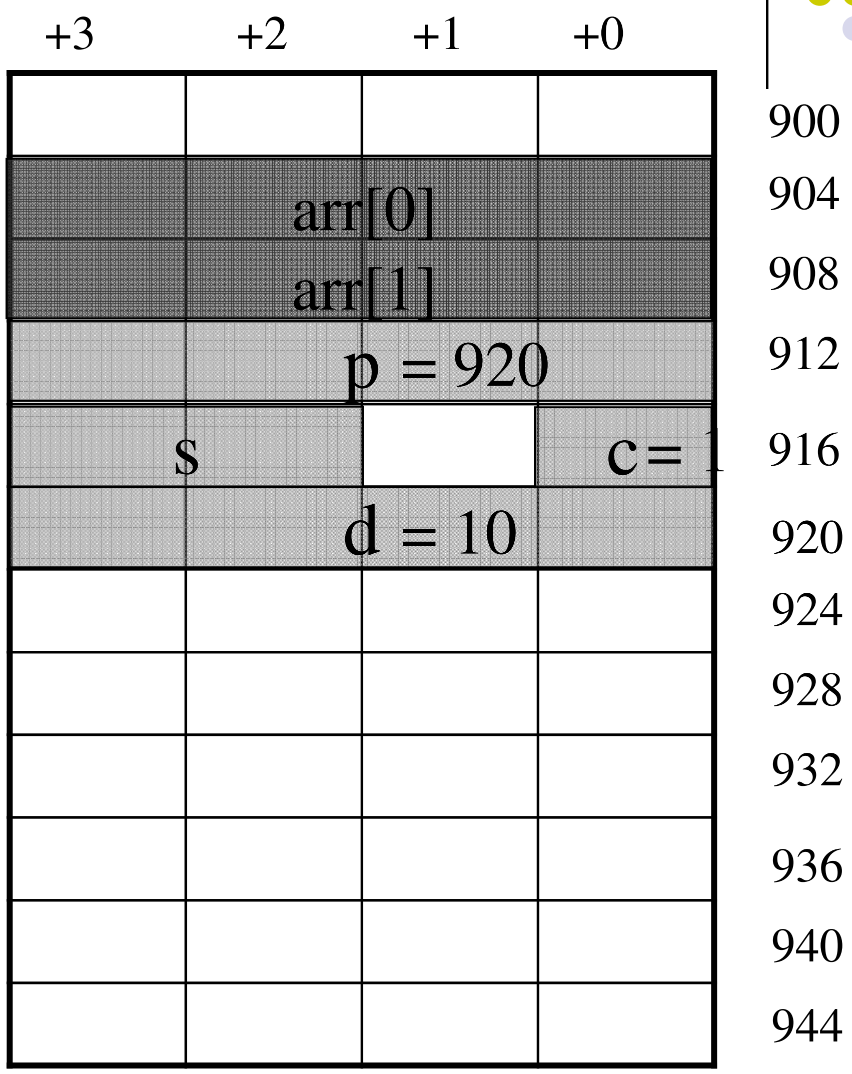
```
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];

    p = &d;
    *p = 10;
    c = (char)1;

    p = arr;
    *(p+1) = 5;
    p[0] = d;

    *( (char*)p + 1 ) = c;

    return 0;
}
```



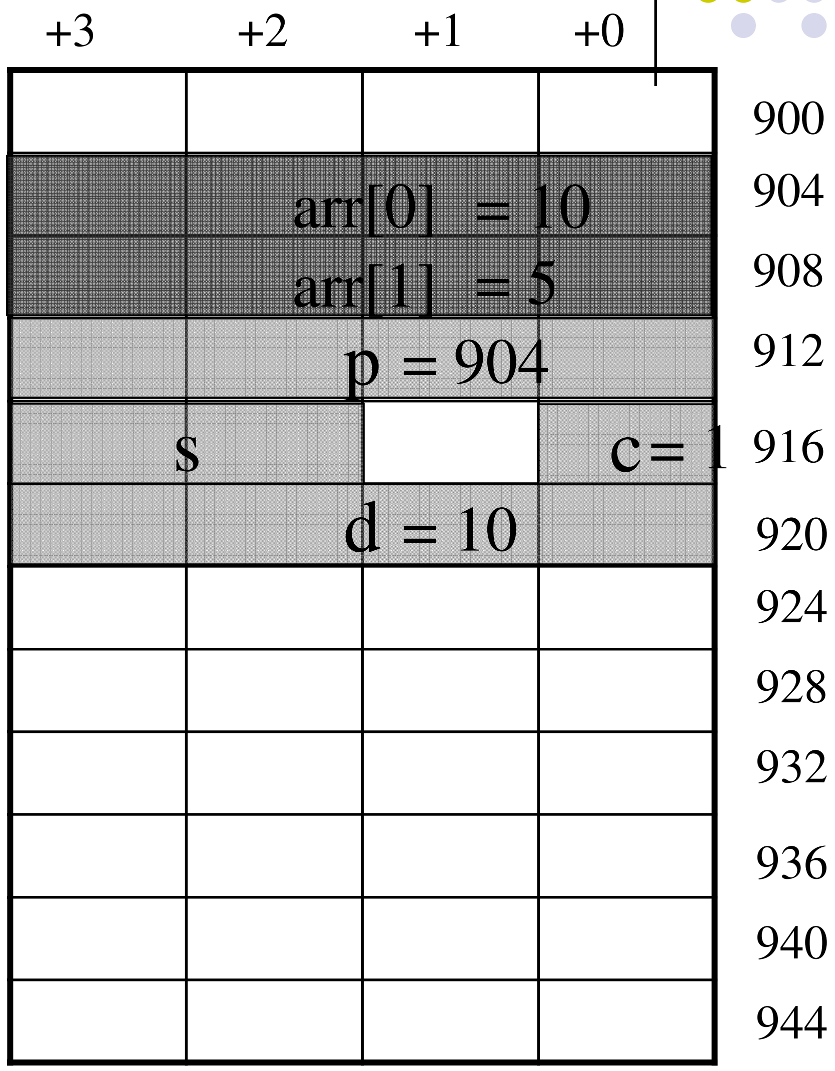
Q: What are the values stored in arr? [assume little endian architecture]

Quiz [Cntd.]

```
p = &d;
*p = 10;
c = (char)1;

p = arr;
*(p+1) = 5;  // int* p;
p[0] = d;

*( (char*)p + 1 ) = c;
```

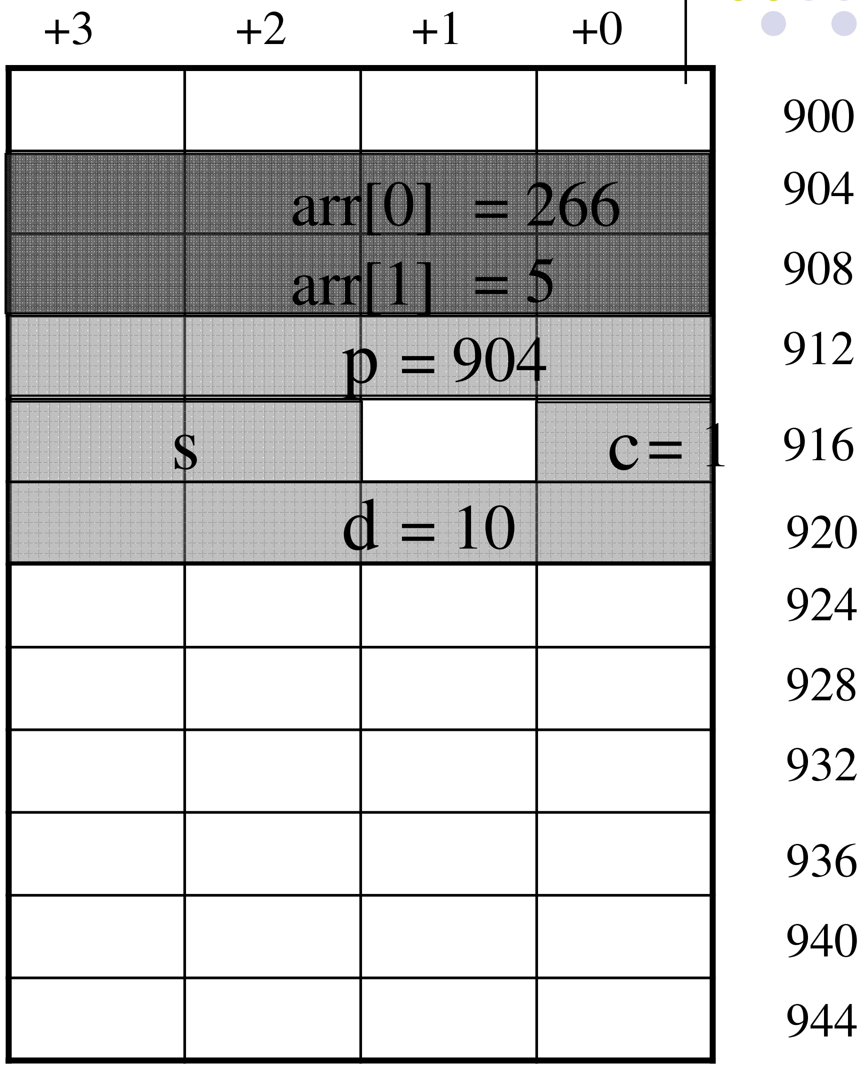


Question: arr[0] = ?

Quiz [Cntd.]

```
p = (int*) malloc(sizeof(int)*3);
p[2] = arr[1] * 3;
s = (short)( *(p+2) );
free( p );
p=NULL;
```

Assumption: we have the same memory layout like in the previous example



Quiz [Cntd.]

```
p = (int*) malloc(sizeof(int)*3);
p[2] = arr[1] * 3;
short s = (short)( *(p+2) );
free( p );
P = NULL;
```

Q: what will be the value of “s”, and why?

Q: what if you say p[2]=0 after you free the memory?

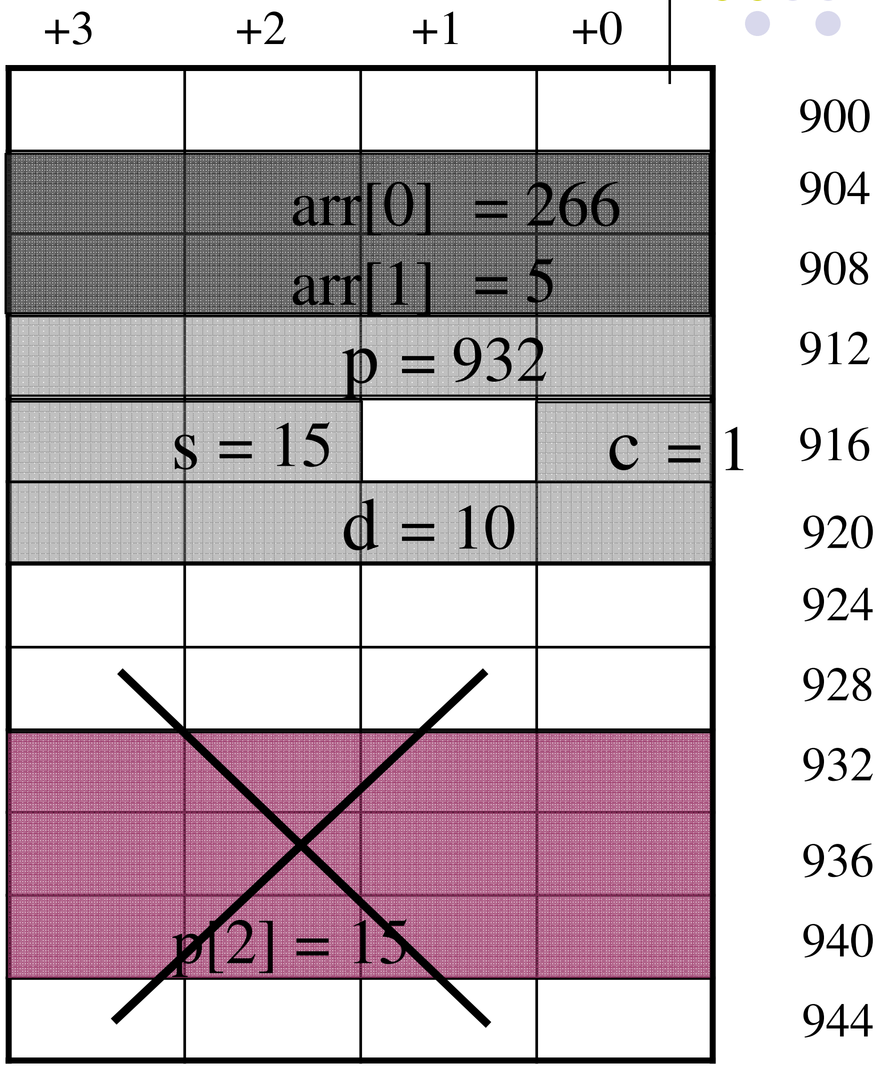
A: code runs, weird stuff might happen

Q: what if you say p[2]=0 right after you set p to NULL?

A: code crashes (no compile time warning)

Q: what if you do not call free(p)?

A: memory leak.



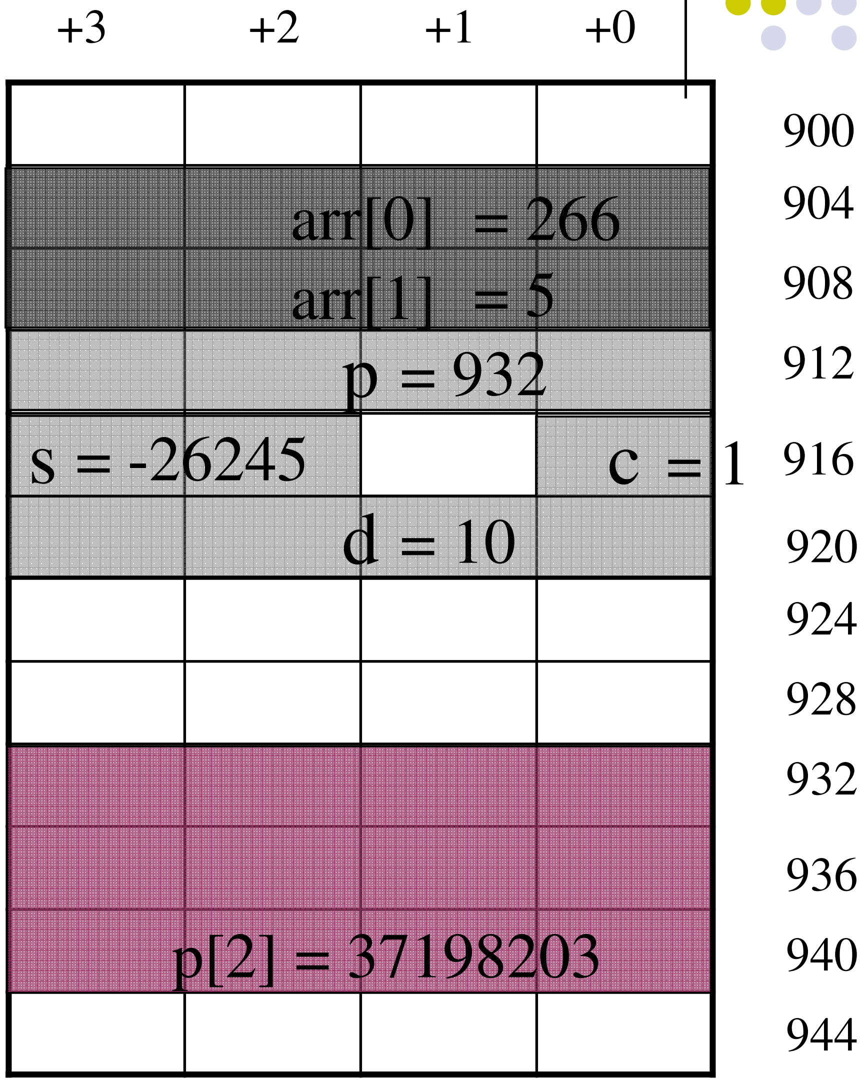
Quiz [Cntd.]

```
int dummy = 12399401
p = (int*) malloc(sizeof(int)*3);
p[2] = dummy * 3;
s = (short)( *(p+2) );
free( p );
```

Q: what is the value of "s" now, and why?

A:

p,3	932	int *
[0]	-842150451	int
[1]	-842150451	int
[2]	37198203	int
s	-26245	short



Debugging on Euler

[with gdb]

Slides on gdb include material from Purdue University

gdb: Intro

- `gdb`: a utility that helps you debug your program
- Learning `gdb` is a great investment, boosts productivity
- A debugger will make a good programmer a better programmer
- In ME759, you should go beyond sprinkling “`printf`” here and there to try to debug your code
  - Avoid: Compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link, compile-link,

Compiling a Program for gdb

- You need to compile with the “-g” option to be able to debug a program with gdb.
- The “-g” option adds debugging information to your program

```
gcc -g -o hello hello.c
```

Running a Program with `gdb`

- To run a program called **`progName`** with **`gdb`** type

  ```
  >> gdb progName
  (gdb)
  ```

- Then set a breakpoint in the main function

  ```
  (gdb) break main
  ```

- A breakpoint is a marker in your program that will make the program stop and return control back to gdb

- Now run your program

  ```
  (gdb) run
  ```

- If your program has arguments, you can pass them after run.

Stepping Through your Program

- Your program will start running and when it reaches “main()” it will stop:

```
(gdb)
```

- You can use the following commands to run your program step by step:

```
(gdb) step
```

It will run the next line of code and stop. If it is a function call, it will enter into it

```
(gdb) next
```

It will run the next line of code and stop. If it is a function call, it will not enter the function and it will go through it.

Example:

```
(gdb) step
(gdb) next
```

Printing the Value of a Variable

- The command

  ```
  (gdb) print varName
  ```

  … prints the value of a variable

E.g.

```
(gdb) print i
$1 = 5
(gdb) print s1
$1 = 0x10740 "Hello"
(gdb) print stack[2]
$1 = 56
(gdb) print stack
$2 = {0, 0, 56, 0, 0, 0, 0, 0, 0, 0}
(gdb)
```

Setting Breakpoints

- A breakpoint is a location in a program where the execution stops in **gdb** and control is passed back to you
- You can set breakpoints in a program in several ways:

```
(gdb) break functionName
```

Set a breakpoint in a function. E.g.

```
(gdb) break main
```

```
(gdb) break lineNumber
```

Set a break point at a line in the current file. E.g.

```
(gdb) break 66
```

It will set a break point in line 66 of the current file.

```
(gdb) break fileName:lineNumber
```

It will set a break point at a line in a specific file. E.g.

```
(gdb) break hello.c:78
```

```
(gdb) break fileName:functionName
```

It will set a break point in a function in a specific file. E.g.

```
(gdb) break subdivision.c:paritalSum
```

Watching a Variable

- Many times you want to keep an eye on a variable that for some reason assumes a value that is not in line with expectations

- To that end, you can “watch” a variable and have the code break as soon as the variable is read or changed

- You can set breakpoints in a program in several ways:

```
(gdb) watch varName
```

Program breaks whenever **`varName`** gets written by the program

```
(gdb) rwatch varName
```

Program breaks whenever **`varName`** gets read by the program

```
(gdb) awatch varName
```

Program breaks whenever **`varName`** gets read/written by the program

```
(gdb) info watchpoints
```

You get a list of all watchpoints, breakpoints, and catchpoints in your program

Example:

[watching a variable]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

Example: Watching Variable “s”

- Below is a copy-and-paste from gdb, for our short program

```
(gdb) awatch s
Hardware access (read/write) watchpoint 2: s
(gdb) continue
Continuing.
Hardware access (read/write) watchpoint 2: s

Old value = 0
New value = 15
main () at pointerArithm.cpp:15
15              free( p );
```

Regaining the Control

- When you type

  ```
  (gdb) run
  ```

  the program will start running and it will stop at a breakpoint

- If the program is running without stopping, you can regain control again typing ctrl-c

- When you type

  ```
  (gdb) continue
  ```

  the program will run until it hits the next breakpoint, or exits

Where Are You?

- The command

  (gdb) where

  Will print the current function being executed and the chain of functions that are calling that fuction.

  This is also called the backtrace.

  Example:

```
(gdb) where
#0  main () at test_mystring.c:22
(gdb)
```

Seeing Code Around You...

- The command **`list`** shows you code around the location where the execution is “break-ed”

```
(gdb)list
```

It will print, by default, 10 lines of code.

There are several flavors:

```
(gdb)list lineNumber
```

…prints code around a certain line number

```
(gdb)list functionName
```

…prints lines of code around the beginning of a function

```
(gdb)set listsize someNumber
```

…controls the number of lines showed with **`list`** command

Exiting **gdb**

- The command “quit” exits **gdb**.

```
(gdb) quit
The program is running.  Exit anyway?
  (y or n) y
```

Debugging a Crashed Program

- Also called “postmortem debugging”
- When a program segfaults, it writes a **core file.**

```
bash-4.1$ ./hello
Segmentation Fault (core dumped)
bash-4.1$
```

- The core is a file that contains a snapshot of the state of the program at the time of the crash
  - Information includes what function the program was running upon crash

Example:
[Code crashing]

```
#include <iostream>

int main(){
  int arr[2]={266,5};

  int * p;
  short s;

  p = (int*) malloc(sizeof(int)*3);

  p[2] = arr[1] * 3;

  s = (short)( *(p+2) );

  free( p );

  p=NULL;

  p[0] = 5;
  return 0;
}
```

This is why it’s crashing…

Running gdb on a Segmentation fault

- Here’s what gdb says when running the code…

```
[negrut@euler CodeBits]$ gdb badPointerArithm.out
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
Copyright (C) 2010 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".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out...done.
(gdb) run
Starting program: /home/negrut/ME964/Spring2012/CodeBits/badPointerArithm.out
warning: the debug information found in "/usr/lib/debug//lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).

warning: the debug information found in "/usr/lib/debug/lib64/libc-2.12.so.debug" does not match
"/lib64/libc.so.6" (CRC mismatch).


Program received signal SIGSEGV, Segmentation fault.
0x0000000000400641 in main () at pointerArithm.cpp:19
19                      p[0] = 5;
(gdb)
```

Debugging – Departing Thoughts

- Debug like a pro (don’t use **`printf`**…)
- **`dbg`** can save you tons of time
- If you want to have a GUI slapped on top of **`gdb`**, use “**`ddd`**” on Euler
- Under Windows, VisualStudio has an excellent debugger