

last update: 11th February 2024

Session: 2023-24

COMP281 Lecture 6

Principles of C and Memory Management

Phil Jimmieson

Last Lecture

- Functions

Today

- Arrays
 - Declaration & Initialisation
 - Accessing array elements
 - Passing arrays to functions
 - 2D-arrays
- Debugging

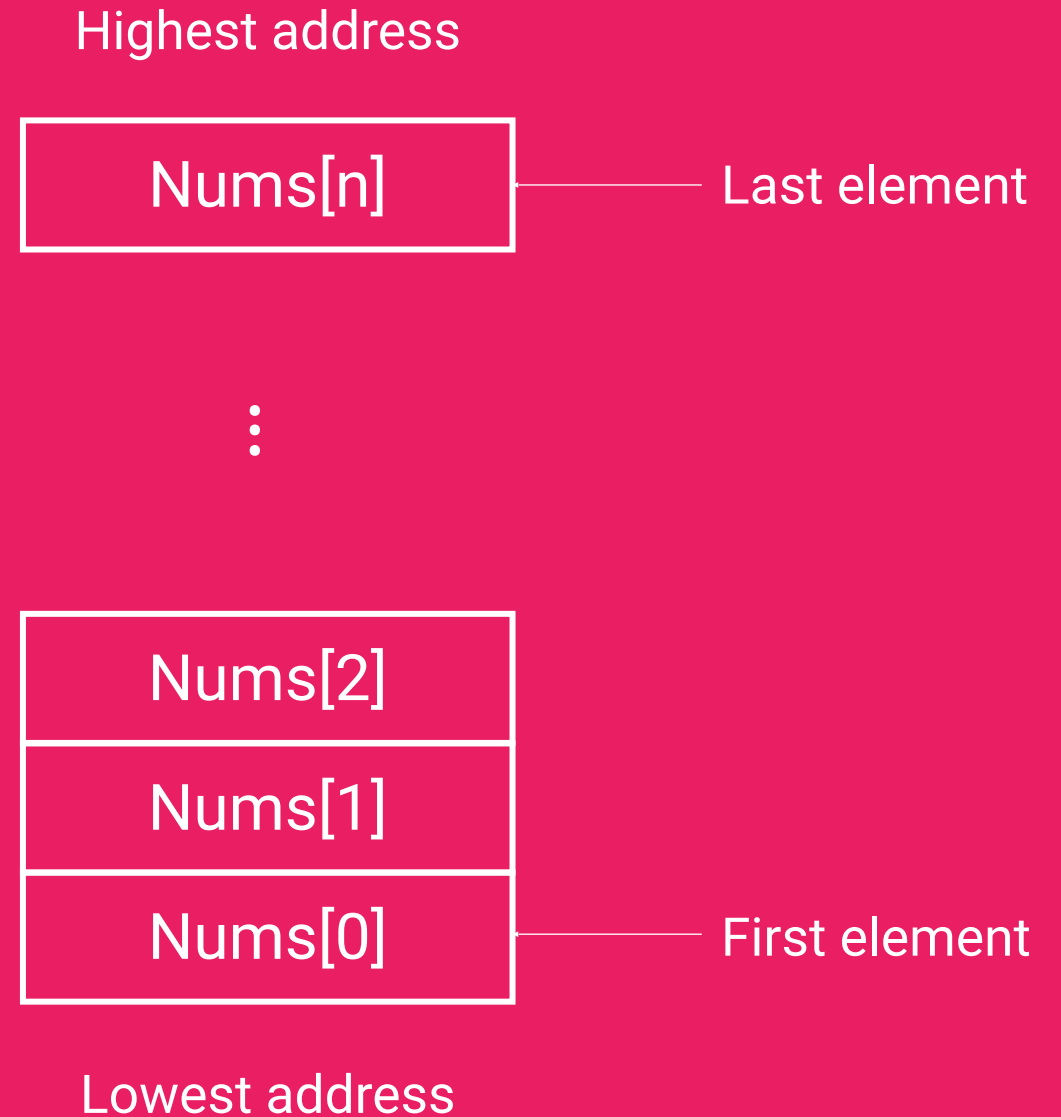
Arrays

What is an Array?

What is an Array?

- A kind of data structure that stores a fixed-size collection of elements of the same type.
- A linear sequence of elements, e.g.
 - array of `ints`
 - array of `chars` (string)
- Instead of declaring individual variables, such as `num0`, `num1`, ..., you declare an array variable such as `nums` and use `nums[0]`, `nums[1]`, ..., to represent individual variables.
- A specific element in an array is accessed by an index.

- An array consist of contiguous **memory** locations.
- The highest address corresponds to the last element.
- The lowest address corresponds to the first element.



Declaring an Array

```
type arrayName [ arraySize ];
```

To declare an array, you need to specify

- The type of the elements
 - type can be any valid C data type
- The number of elements required by an array – arraySize
 - arraySize must be an integer constant greater than zero

For example: `float income[3];`

* `income` is a variable array which is sufficient to hold up to 3 floats.

Initialising an Array

- One by one

```
income[0] = 198.76;
```

```
income[1] = 236.89;
```

```
income[2] = 218.54;
```

- Using a single statement

```
float income[3] = {198.76, 236.89, 218.54};
```


Initialising an Array

An array can be:

- Initialised
- Partially initialised
- Not initialised
- Uninitialised
 - i.e. garbage

Initialising an Array

Examples

```
int my_array[10];
```

```
    /* not initialised */
```

```
int my_array[5] = { 1, 2, 3, 4, 5 };
```

```
    /* initialised */
```

```
int my_array[] = { 1, 2, 3, 4, 5 };
```

```
    /* OK, initialised */
```

```
int my_array[4] = { 1, 2, 3, 4, 5 };
```

```
    /* warning */
```

```
int my_array[10] = { 1, 2, 3, 4, 5 };
```

```
    /* OK, partially initialised */
```

Initialising an Array

Note on partial initialisation

```
int my_array[10] = { 1, 2, 3, 4, 5 };
```

- rest of array initialised to 0

```
int my_array[10];
```

- entire array uninitialised
- contains garbage

Initialising an Array

Explicit initialisation of an array

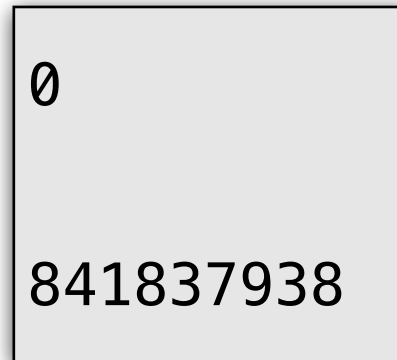
```
int i;  
int my_array[10];  
for (i=0; i<10; i++)  
{  
    income[i] = 2 * i;  
}
```

* This is the most flexible approach

Initialising an Array

Something bad could happen

```
int my_array[10];  
  
/* what happens here? */  
printf("%d\n", my_array[0]);  
  
/* what happens here? */  
printf("%d\n", my_array[99]);
```



- No checking!
- C is an **unsafe** language!

```
array.c:6:42: warning: array index 99 is past the end of the array (which contains 10 elements)
[-Warray-bounds]
    /* what happens here? */ printf("%d\n", my_array[99]);
                                   ^~~~~~
array.c:4:4: note: array 'my_array' declared here
    int my_array[10];
    ^
1 warning generated.
```

Initialising an Array

```
int my_array[5];  
my_array = { 4, 1, 7, 3, 5 };
```


Initialising an Array

Note, the following is illegal

```
int my_array[5];
```

```
my_array = { 4, 1, 7, 3, 5 } /* wrong */
```

- The { 4, 1, 7, 3, 5 } syntax is *only* usable when **declaring** a new array, but not for reassigning the contents of the array.

```
int my_array[5] = { 4, 1, 7, 3, 5 } /* ok */
```

```
int my_array[ ] = { 4, 1, 7, 3, 5 } /* ok */
```

Accessing Array Elements

An element is accessed by indexing the array name. this is done by placing the index of the element within square brackets after the name of the array. For example

```
float salary = income[1];
```

This *statement* will take the 2nd element from the array and *assign* the value to salary variable.

Example 1

```
#include <stdio.h>

int main(void) {
    int n[10];      /* n is an array of 10 integers */
    int i, j;       /* initialize elements of array n to 0 */
    for(i=0; i<10; i++) {
        n[i] = i + 100; /* set element at location i to i + 100 */
    } /* output each array element's value */
    for(j=0; j<10; j++){
        printf("Element[%d] = %d\n", j, n[j]);
    }
    return 0;
}
```

Example 1

```
#include <stdio.h>

int main(void) {
    int n[10];      /* n is an array */
    int i, j;       /* initialize el
for(i=0; i<10; i++) {
    n[i] = i + 100; /* set elemen
}                  /* output each c
for(j=0; j<10; j++){
    printf("Element[%d] = %d\n",
}
return 0;
}
```

result

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

Two-dimensional arrays

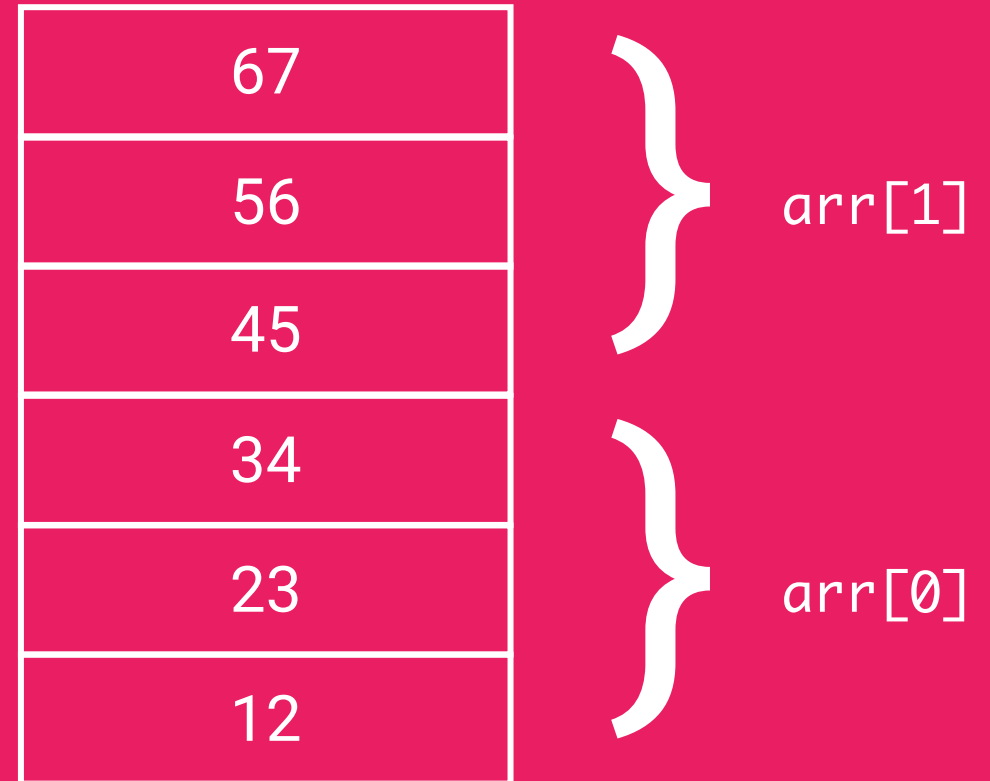
```
int arr[2][3]; /* not arr[2, 3] */
int i, j;
int sum = 0;
arr[0][0] = 12;
arr[0][1] = 23;
arr[0][2] = 34;
arr[1][0] = 45;
arr[1][1] = 56;
arr[1][2] = 67;
for ( i=0; i<2; i++ ) {
    for ( j=0; j<3; j++ ) {
        sum += arr[i][j];
    }
}
printf("sum = %d\n", sum);
```

Two-dimensional arrays

Two-dimensional arrays can be split into component one-dimensional arrays

```
int arr[2][3];  
    /* initialise */  
    /* arr[0] is array of 3 ints */  
    /* arr[1] is another array of 3 ints */
```


How is `arr` laid out in **memory**?



Two-dimensional arrays

Initialising a two-dimensional array

```
int my_array[2][3];  
    /* not initialised */  
int my_array[2][3] = { {12, 23, 34}, {45, 56, 67} };  
    /* ok */  
int my_array[2][3] = { 12, 23, 34, 45, 56, 67 };  
    /* warning with -Wall */
```

Two-dimensional arrays

Initialising a two-dimensional array

```
int my_array[2][2];
```

```
int my_array[][] = { {12, 23, 34}, {45, 56, 67} };
```

```
int my_array[][3] = { {12, 23, 34}, {45, 56, 67} };
```

Two-dimensional arrays

Initialising a two-dimensional array

```
int my_array[2][];
```

```
    /* invalid */
```

```
int my_array[][] = { {12, 23, 34}, {45, 56, 67} };
```

```
    /* invalid */
```

```
int my_array[][3] = { {12, 23, 34}, {45, 56, 67} };
```

```
    /* ok */
```

Two-dimensional arrays

Initialising a two-dimensional array

```
int my_array[][3] = { 12, 23, 34, 45, 56, 67 };  
/* warning with -Wall */
```

Note

- All, but leftmost, dimension must be specified
- Compiler can compute leftmost dimension
- OK to specify leftmost dimension as well

Two-dimensional arrays

Accessing two-dimensional array elements

```
int val = arr[2][3];
```

This takes the 4th element from the 3rd row of the array.

Example 2

```
#include <stdio.h>

int main(void) {
    /* an array with 5 rows and 2 columns*/
    int myIntArray[5][2] = { {0,0},{1,2},{2,4},{3,6},{4,8} };
    int i, j;    /* output each array element's value */
    for(i=0; i<5; i++) {
        for(j=0; j<2; j++) {
            printf("myIntArray[%d][%d] = %d\n",i,j,myIntArray[i][j]);
        }
    }
    return 0;
}
```

Example 2

```
#include <stdio.h>

int main(void) {
    /* an array with 5 rows and 2 columns */
    int myIntArray[5][2] = { {0,0},
                              {1,1},
                              {2,2},
                              {3,3},
                              {4,4} };
    int i, j;    /* output each array element */
    for(i=0; i<5; i++) {
        for(j=0; j<2; j++) {
            printf("myIntArray[%d][%d] = %d\n", i, j, myIntArray[i][j]);
        }
    }
    return 0;
}
```

result

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

How to pass an array to functions?

Passing an array to functions

What does this do?

```
void foo(int i) {  
    i = 12;  
}
```

```
/* later... */
```

```
int i = 10;
```

```
foo(i);
```

```
/* what is i now? */
```

Passing an array to functions

What does this do?

```
void foo(int i) {  
    i = 12;  
}
```

```
/* later... */  
int i = 10;
```

```
foo(i);  
/* what is i now? */
```

- Current value of `i` is copied into function argument `i`
- Passing a value to a function as argument does not change the value
- We say that C is a “call-by-value” language

Passing an array to functions

What does this do?

```
void foo(int i) {  
    i = 12;  
}
```

```
/* later... */
```

```
int i = 10;
```

```
foo(i);
```

```
/* what is i now? */
```

- Current value of `i` is copied into function argument `i`
- Passing a value to a function as argument does not change the value
- We say that C is a “call-by-value” language
- **But arrays are "different"!**

Passing an array to functions

What does this do?

```
void foo(int i) {  
    i = 12;  
}
```

```
/* later... */
```

```
int i = 10;
```

```
foo(i);
```

```
/* what is i now? */
```

- Current value of `i` is copied into function argument `i`
- Passing a value to a function as argument does not change the value
- We say that C is a “call-by-value” language
- **But arrays are "different"!**
 - (actually, not really, but it seems like they are; need pointers for full explanation...)

Passing an array to functions

Arrays passed to functions *can* be modified

```
void foo(int arr[])  
{  
    arr[0] = 12; /* modifies the array */  
}  
/* later... */  
int my_array[5] = { 1, 2, 3, 4, 5 };  
printf("%d\n", my_array[0]);  
foo(my_array);  
printf("%d\n", my_array[0]); /* it changed! */
```

1

12

Passing an array to functions

Last array dimension in declaration is ignored for one-dimensional arrays

```
void foo2(int arr[5])    /* same as arr[] */  
{  
    arr[0] = 12;  
}
```

Same as

```
void foo(int arr[])  
{  
    arr[0] = 12;  
}
```

Passing a 2D array to functions

- Two-dimensional (or higher-dimensional) arrays can also be passed to functions
- However, we must specify all array dimensions except for the leftmost one (which is optional)
 - Same rule as for initialising 2D arrays

Passing a 2D array to functions

```
int sum_2d_array(int arr[][3], int nrows)
{
    int i, j;
    int sum = 0;

    for (i = 0; i < nrows; i++)
    {
        for (j = 0; j < 3; j++)
            {sum += arr[i][j];
        }
    }

    return sum;
}
```

Passing a 2D array to functions

```
int sum_2d_array(int arr[][3], int nrows)
{
    int i, j;
    int sum = 0;

    for (i = 0; i < nrows; i++)
    {
        for (j = 0; j < 3; j++)
            {sum += arr[i][j];
            }
    }

    return sum;
}
```

Better to use a constant,
rather than the number 3
here.

Passing a 2D array to functions

```
#define MAX_LEN 3
int sum_2d_array(int arr[][MAX_LEN], int nrows)
{
    int i, j;
    int sum = 0;

    for (i = 0; i < nrows; i++)
    {
        for (j = 0; j < MAX_LEN; j++)
            {sum += arr[i][j];
        }
    }

    return sum;
}
```

Passing a 2D array to functions

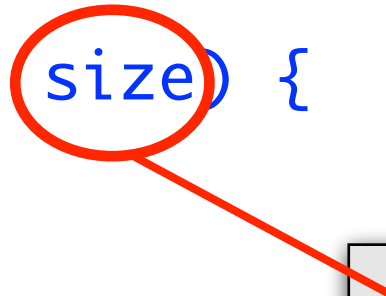
- Also ok to specify leftmost dimension:

```
int sum_2d_array(int arr[2][3], int nrows)
{
    /* same as before */
}
```

- Compiler still ignores leftmost dimension
 - may need to pass it in as an extra argument e.g. as `nrows` here

Example 3

```
double getAverage(int arr[], int size) {  
    int i;  
    double avg;  
    double sum=0;  
    for(i=0; i<size; ++i) {  
        sum += arr[i];  
    }  
    avg = sum/size;  
    return avg;  
}
```



Note that this is just a variable that we've created, not a language keyword.

Example 3

```
#include <stdio.h>

/* function declaration */
double getAverage(int arr[], int size);

int main() {
    /* an int array with 5 elements */
    int balance[5] = { 1000, 2, 3, 17, 50 };
    double avg;
    /* pass "pointer" to the array as an argument */
    avg = getAverage(balance, 5);
    /* output the returned value */
    printf("Average value is: %f\n", avg);
    return 0;
}
```

Example 3

```
#include <stdio.h>
/* function declaration */
double getAverage(int arr[], int size);
int main() {
    /* an int array with 5 elements */
    int balance[5] = { 1000, 2, 3, 17, 20 };
    double avg;
    /* pass pointer to the array as argument */
    avg = getAverage(balance, 5);
    /* output the returned value */
    printf("Average value is: %f\n", avg);
    return 0;
}
```

result

Average value is: 214.400000

Debugging

CodeGrade says “No”

- Before you start debugging, read the notifications!
 - There may be useful information there.
 - e.g. is it a compile time error or a run-time error?
(and do you know the differences?!)

CodeGrade says “No”

- Before you start debugging, read the error message
 - There may be useful information there.
 - e.g. is it a compile time error or a run-time error?
(and do you know the differences?!)

For example

- **Compile time:**
 - Syntax errors
 - Type-checking errors
 - (rarely) compiler crashes
- **Run time:**
 - Division by zero
 - Dereferencing a null pointer
 - Running out of memory
 - Trying to open a file that isn't there

CodeGrade says “No”

- Things to do
 - Read the problem again.
 - Are you sure you have interpreted it correctly?

CodeGrade says “No”

- Test it yourself: it's very likely (99%) that the fault is with your program.
 - Did you try different test cases?
 - What assumptions did you make about the test cases?
 - DON'T assume the test case(s) give you all of the possibilities
 - Did you try.... large numbers, small numbers, negative numbers, ..., etc.?

CodeGrade says “No”

- Check the output
 - Does the output match exactly?
 - Don't print anything else to the screen!
 - “replicate CodeGrade” by piping a test file into your program! (see previous lecture)

Prepare `test.in`

a text file with input in the format specified in assignment

```
% ./a.out < test.in > test.out
```

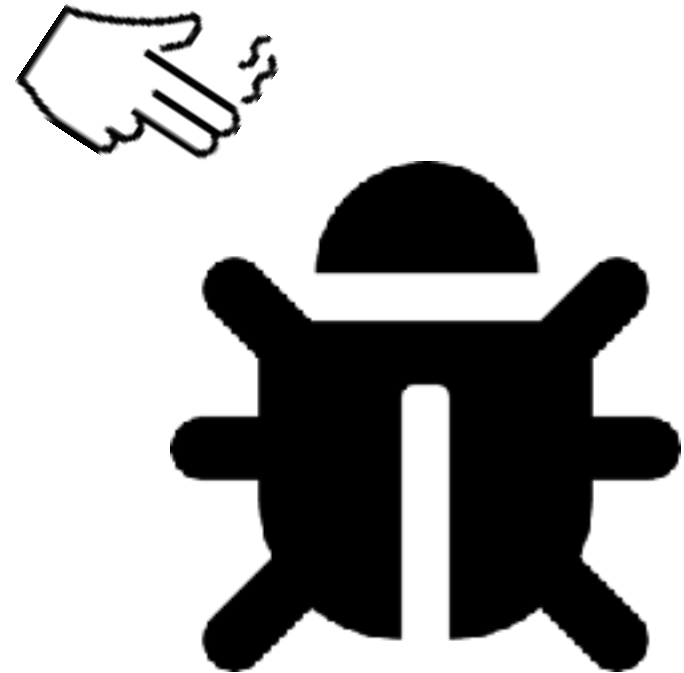
`test.out` will contain all the output from your program

Debugging for runtime errors

- No matter how experienced a coder you are, any program (software) you develop cannot be completely free of bugs.



Debugging for runtime errors



- No matter how experienced a coder you are, any program (software) you develop cannot be completely free of bugs.
- Identifying bugs and fixing them is one of the most important tasks in the software development cycle.

Debugging for runtime errors

- LLDB – “a next generation, high-performance debugger”
- GDB - the GNU project debugger
 - debuggers let you peek inside a program while the program is executing, and help identify exactly where the problem is.
 - Four main kinds of things to help. You catch bugs in the act:
 - Start your program, specifying anything that might affect its behaviour.
 - Make your program stop on specified conditions.
 - Examine what has happened, when your program has stopped.
 - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Debugging for runtime errors

- LLDB – “a next generation, high-performance debugger”
 - Install on linux
 - `brew install llvm`
- GDB – the GNU project debugger
 - Provided as part of the cygwin package (and also with Visual Studio)

Example 4

```
#include <stdio.h>

int main(void) {
    int i, j;
    char name[ ] = "matrix";
    i = j = 0;
    for ( ; i < 3; ++i) {
        for ( ; j < 4; ++j)
            printf("(%d, %d)\t", i, j);
        printf("\n");
    }
    printf("This is a %dx%d %s.\n", i, j, name);
    return 0;
}
```

Expecting output

```
(0, 0) (0, 1) (0, 2) (0, 3)
(1, 0) (1, 1) (1, 2) (1, 3)
(2, 0) (2, 1) (2, 2) (2, 3)
This is a 3x4 matrix
```

Example 4

```
#include <stdio.h>

int main(void) {
    int i, j;
    char name[ ] = "matrix";
    i = j = 0;
    for ( ; i < 3; ++i) {
        for ( ; j < 4; ++j)
            printf("(%d, %d)\t", i, j);
        printf("\n");
    }
    printf("This is a %dx%d %s.\n", i, j, name);
    return 0;
}
```

The output we get

```
(0, 0) (0, 1) (0, 2) (0, 3)
```

```
This is a 3x4 matrix
```

What's wrong?

Example 4

1. Compile the file with the **-g** flag to the gcc compiler

```
gcc -g 4.c -o debugging
```


Example 4 (using lldb)

2. Enter the debugging mode

`lldb debugging`

Getting the following response

```
(lldb) target create "debugging"  
Current executable set to 'debugging' (x86_64).  
(lldb)
```

Example 4 (using lldb)

3. Adding breakpoints

Example using line number:

```
b 8
```

Example using function name

```
b main
```

The name of the source file could also be added before the line or function name (separated by a colon).

```
b 4.c:8
```

```
b 4.c:main
```

Example 4 (using lldb)

4. Running the file in the debug mode

run

```
Process 30063 launched: '/Users/leishi/Public/code/debugging' (x86_64)
Process 30063 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
   frame #0: 0x0000000100000eb1 debugging`main at 4.c:8
    5      char name[10] = "matrix";
    6      i = j = 0;
    7
->  8      for ( ; i < 3; ++i) {
    9          for ( ; j < 4; ++j)
   10              printf("(%d, %d)\t", i, j);
   11              printf("\n");
Target 0: (debugging) stopped.
(lldb)
```

Now the debugging stops at line 8, because we defined a breakpoint at **b 8**

Example 4 (using lldb)

4. Running the file in the debug mode

Now we have one of the following options to keep debugging:

- **next** or **n**: Steps to the next line.
- **step** or **s**: Steps into current's line function.
- **continue** or **c**: Continue until the next breakpoint.
- **finish**: Finishes executing the current function and then pauses.

Example 4 (using lldb)

5. Breakpoints

List all existing breakpoints

`br list`

```
Current breakpoints:  
1: file = '/Users/leishi/Public/code/4.c', line = 8, exact_match = 0,  
locations = 1  
    1.1: where = debugging`main + 65 at 4.c:8, address =  
debugging[0x0000000100000eb1], unresolved, hit count = 0
```

Deleting the '0 line' breakpoint

`br del 1`

Deleting all breakpoints

`br del`

Example 4 (using lldb)

6. Showing variables

To show specific variables use `print` or `p` followed by the variable name
`p i`

To show all arguments and local variables
`frame variable`

7. Modifying variables

The variables can be updated using `expr [variable name] = [value]`, e.g.,
`expr i = 2`

Example 4 (using lldb)

8. Showing current line

You can use the command `f` to see the current line where the debugger is stopped in.

`f`

9. Using backtrace

Your program may have died deep inside a function that has been called more than once, and you need to know which sequence of nested functions calls led to the failure. The answer is `bt` (backtrace).

`bt`

Example 4 (using lldb)

10. Moving up and down within the frames

Sometimes, you may need to move up to a higher frame using the **up** command to debug there. The **down** command moves you back down a frame towards where you started.

The up and down commands let you move up and down the calling stack of nested functions so you can issue debug commands about the function that's “active” at each level.

up

down

Example 4 (using gdb)

2. Enter the debugging mode

`gdb debugging`

Getting the following response (on Windows)

```
$ gdb debugging.exe
GNU gdb (GDB) (Cygwin 7.10.1-1) 7.10.1
Copyright (C) 2015 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-pc-cygwin".
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 debugging.exe...done.
(gdb) |
```

Example 4 (using gdb)

3. Adding breakpoints

Example using line number:

```
b 8
```

Example using function name

```
b main
```

The name of the source file could also be added before the line or function name (separated by a colon).

```
b 4.c:8
```

```
b 4.c:main
```

Example 4 (using gdb)

4. Running the file in the debug mode

r

```
(gdb) b 8
Breakpoint 1 at 0x10040110d: file debugging.c, line 8.
(gdb) r
Starting program: /cygdrive/c/Users/phil/Documents/C programming/debugging.exe
[New Thread 11796.0x1ffc]
[New Thread 11796.0x15a4]
[New Thread 11796.0x3138]
[New Thread 11796.0x2164]

Breakpoint 1, main () at debugging.c:8
8                               printf("(%d, %d)\t", i, j);
(gdb) |
```

Now the debugging stops at line 8, because we defined a breakpoint at b 8

Example 4 (using gdb)

4. Running the file in the debug mode

Now we have one of the following options to keep debugging:

- **n**: Steps to the next line.
- **s**: Steps into current's line function.
- **c**: Continue until the next breakpoint.
- **f**: Finishes executing the current function and then pauses.

Example 4 (using gdb)

5. Breakpoints

List all existing breakpoints

info **b**reak

```
(gdb) info break
Num      Type           Disp Enb Address            What
1        breakpoint     keep y   0x000000010040110d in main at debugging.c:8
          breakpoint already hit 3 times
2        breakpoint     keep y   0x00000001004010ed in main at debugging.c:4
          breakpoint already hit 1 time
3        breakpoint     keep y   0x000000010040110d in main at debugging.c:8
          breakpoint already hit 3 times
```

Deleting the 2nd breakpoint

d 2

Deleting all breakpoints (with a prompt from gdb)

d

Example 4 (using gdb)

6. Showing variables

To show specific variables use **p** followed by the variable name
p i

To show all arguments and local variables
info locals

7. Modifying variables

The variables can be updated using set var [variable name] = [value], e.g.,
set var i = 2

Example 4 (using gdb)

8. Showing current line

You can use the command **frame** to see the current line where the debugger is stopped in.

frame

9. Using backtrace

Your program may have died deep inside a function that has been called more than once, and you need to know which sequence of nested functions calls led to the failure. The answer is **backtrace**

backtrace

Example 4 (using gdb)

10. Moving up and down within the frames

Sometimes, you may need to move up to a higher frame using the **up** command to debug there. The **down** command moves you back down a frame towards where you started.

The up and down commands let you move up and down the calling stack of nested functions so you can issue debug commands about the function that's “active” at each level.

up

down

Summary

Today

- Arrays
- Debugging

Next

- Pointers