Lecture 6: Memory, part 4
C & Memory

• C allows you to directly access memory
• There is no “babysitter” that monitors your memory accesses
  – Unless you wander outside a “segment,” in which case your program crashes
• This helps performance
• But makes programming difficult – no seat belts
• HOWEVER: there are programs that will monitor your memory accesses and let you know when you have done bad things – valgrind & gdb
This week’s lab...

• If you “do memory wrong,” then your program will crash

• How to fix?

• Special program: gdb (GNU debugger)
  – gdb can give you lots of hints about why your program crashed!
NO CLASS TUESDAY

• Instead YouTube lecture
• Posted this weekend
• Will be entirely about how to do Project 2D
• (2C discussed later this lecture)
What should you be doing?

• Due today: 2B
• 2C may be posted tonight if we make it far enough in lecture
• Reading the textbook, if you have not done it already
  – Chapter 2.1, 2.2, 2.3, 2.4., 2.5, and 2.6.1
  – Chapter 4.1 (but not 4.1.2), 4.2-4.4
  – will be an upcoming quiz on details in the book (things I have not lectured on will be fair game)
Review
Pointers

• Pointer: points to memory location
  – Denoted with ‘*’
  – Example: “int *p”
    • pointer to an integer
  – You need pointers to get to heap memory

• Address of: gets the address of memory
  – Operator: ‘&’
  – Example:
    ```
    int x;
    int *y = &x;  // this example is pointing to an automatic variable, not a dynamic variable
    ```
What is an array?

- Block of contiguous memory
- If elements each have size $N$ bytes and there are $M$ elements, then $N \times M$ contiguous bytes
- Let $A$ be address of the beginning of the array
- Then $A[0]$ is at “$A$”
- And $A[1]$ is at “$A+N$”
- $A[2]$ is at “$A+2 \times N$”
- and so on...
C02LN00GFD58:212 hank$ cat A.c
#include <stdio.h>

int main()
{
    int A[5];
    int i;

    printf("A is at %p\n", A); /* Note I can pass in A as a pointer */
    for (i = 0; i < 5; i++)
    {
        printf("A[%d] is at %p\n", i, &(A[i]));
    }
}
C02LN00GFD58:212 hank$ gcc A.c
C02LN00GFD58:212 hank$ ./a.out
A is at 0x7fff5e8b5bd0
A[0] is at 0x7fff5e8b5bd0
A[1] is at 0x7fff5e8b5bd4
A[2] is at 0x7fff5e8b5bd8
A[3] is at 0x7fff5e8b5bdcc
A[4] is at 0x7fff5e8b5be0
Dynamic memory works differently

- You allocate it, it stays around until you de-allocate it or the program ends
- Important: you need a way to keep track of memory
  - If not, the memory will be “leaked.”
- So we need a way of managing dynamic memory.
- The concept for doing this in C is **POINTERS**
Dynamic Memory Allocation

• Special built-in function to allocate memory from heap: `malloc`
  – Interacts with Operating System
  – Argument for malloc is how many bytes you want

• Also built-in function to deallocate memory: `free`
free/malloc example

Enables compiler to see functions that aren’t in this file. More on this next week.

```c
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```

sizeof is a built in function in C. It returns the number of bytes for a type (4 bytes for int).

don’t have to say how many bytes to free ... the OS knows
Automatic vs Dynamic

• Automatic memory lives only for its current scope
• Dynamic memory lives until you free it, or until the program ends
This is just fine...

Hanks-iMac:Downloads hank$ cat scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int *X = malloc(sizeof(int)*2);
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}

Hanks-iMac:Downloads hank$ gcc scope.c
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
This is not fine...

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

int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}
```

Hanks-iMac:Downloads hank$ gcc bad_scope.c
bad_scope.c:9:12: warning: address of stack memory associated with local variable 'X' returned [-Wreturn-type]
    return X;

1 warning generated.
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
And here it goes wrong...
Pause and do an example and film it
Memory Segments

• Your program is divided into 4 segments:
  – Code segment: where the machine code instructions are
  – Data segment: where the global variables live … and other things
  – “Stack”: where automatic memory lives
  – “Heap”: where dynamic memory lives
  – Note: stack & heap are data structures and we will learn more about them later.

If you access memory addresses outside your segments, you get a “segmentation fault” … which causes a crash.
And in some cases it doesn’t have to hit the heap. Just take more memory than intended.
Character Strings
ASCII Character Set

<table>
<thead>
<tr>
<th>ASCII Code Chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>NUL</td>
</tr>
<tr>
<td>DLE</td>
</tr>
<tr>
<td>!</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>@</td>
</tr>
<tr>
<td>P</td>
</tr>
<tr>
<td>`</td>
</tr>
<tr>
<td>p</td>
</tr>
</tbody>
</table>

There have been various extensions to ASCII ... now more than 128 characters
Many special characters are handled outside this convention

image source: granneman.com
Some Special Characters

```c
#include <stdio.h>

int main()
{
    char A = 'A';
    char B = 'B';
    char NL = '\n';
    char tab = '\t';
    char backslash = '\\';
    char NULL_character = '\0';
    printf("A = %d, B = %d, NL = %d, tab =%d, backslash = %d, NULL char = %d\n",
           A, B, NL, tab, backslash, NULL_character);
    printf("A = %c, B = %c, NL = %c, tab =%c, backslash = %c, NULL char = %c\n",
           A, B, NL, tab, backslash, NULL_character);
}
```

New: `%c` prints character
character strings

• A character “string” is:
  – an array of type “char”
  – that is terminated by the NULL character

• The C library has multiple functions for handling strings
building a character string

C02LN00GFD58:212 hank$ cat hello_world.c
#include <stdio.h>

int main()
{
    char H[12];
    H[0] = 'H';
    H[1] = 'e';
    H[2] = 'l';
    H[3] = 'l';
    H[4] = 'o';
    H[5] = ' ';
    H[6] = 'W';
    H[7] = 'o';
    H[8] = 'r';
    H[9] = 'l';
    H[10] = 'd';
    H[11] = '\0';
    printf("String is %s\n", H);
}

C02LN00GFD58:212 hank$ gcc hello_world.c
C02LN00GFD58:212 hank$ ./a.out
String is Hello World
building a character string

C02LN00GFD58:212 hank$ cat hello_world2.c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
}
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world

Compiler automatically adds a \0 for you!
```c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
    H[5] = '\0';
    printf("H is %s\n", H);
}
```

```
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world
H is hello
```
Did you notice

• Characters in C are single quotes: ‘A’
• Strings in C are double quotes: “hello world”
• A string of a single character is still double quotes: “A”
  – (And this is actually “A\0”)
```c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
}
```
Character strings example

```c
128-223-223-72-wireless:330 hank$ cat string.c
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is "\%s" and str2 is "\%s"\n", str, str2);

    str[5] = '\0';

    printf("Now str is "\%s" and str2 is "\%s"\n", str, str2);
}
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
```
Useful C library string functions

- strcpy: string copy
- strncpy: string copy, but just first N characters
- strlen: length of a string

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

int main()
{
    char str[12] = "hello world";
    char str2[6], str3[7];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello
```
Useful C library string functions

- `strcpy`: string copy
- `strncpy`: string copy, but just first N characters
- `strlen`: length of a string

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

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```

What happened here?
# More useful C library string functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Searching:</th>
<th>Other:</th>
<th>Macros</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copying:</strong></td>
<td><strong>memchr</strong> Locate character in block of memory</td>
<td>memset Fill block of memory</td>
<td><strong>NULL</strong> Null pointer</td>
<td><strong>size_t</strong> Unsigned integral type</td>
</tr>
<tr>
<td>memcpy</td>
<td>(function)</td>
<td>(function)</td>
<td>(macro)</td>
<td>(type)</td>
</tr>
<tr>
<td>memmove</td>
<td>Move block of memory (function)</td>
<td>strerror Get pointer to error message string (function)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcpy</td>
<td>Copy string (function)</td>
<td>strlen Get string length (function)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>strncpy</td>
<td>Copy characters from string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Concatenation:</strong></td>
<td><strong>strcat</strong> Concatenate strings (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcat</td>
<td>Append characters from string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Comparison:</strong></td>
<td><strong>memcmp</strong> Compare two blocks of memory (function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcmp</td>
<td>(function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcmp</td>
<td>Compare two strings (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcoll</td>
<td>Compare two strings using locale (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strncmp</td>
<td>Compare characters of two strings (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strxfrm</td>
<td>Transform string using locale (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Searching:</strong></td>
<td><strong>strchr</strong> Locate first occurrence of character in string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strchr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strcspn</strong></td>
<td>Get span until character in string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strcspn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strpbrk</strong></td>
<td>Locate characters in string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strpbrk</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strrchr</strong></td>
<td>Locate last occurrence of character in string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strrchr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strspn</strong></td>
<td>Get span of character set in string (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strspn</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strstr</strong></td>
<td>Locate substring (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>strstr</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>strtok</strong></td>
<td>Split string into tokens (function)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
argc & argv

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- "./a.out hank childs"
- \( \rightarrow \) argc == 3
C02LN00GFD58:212 hank$ cat argv.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for (i = 0 ; i < argc ; i++)
    {
        printf("Argument %d is %s\n", i, argv[i]);
    }
}
C02LN00GFD58:212 hank$ gcc argv.c
C02LN00GFD58:212 hank$ ./a.out hank childs loves C and vi
Argument 0 is ./a.out
Argument 1 is hank
Argument 2 is childs
Argument 3 is loves
Argument 4 is C
Argument 5 is and
Argument 6 is vi
Project 2C

- Assigned today
- Due Friday Oct 25
- Parsing floating point numbers (e.g., -3.68)
- Lots of hints in prompt

CIS 212: Project #2C
Assigned: October 16, 2019
Due: October 25, 2019
(which means submitted by 6am on October 26, 2019)
Worth 6% of your grade

Please read this entire prompt.

Assignment:
1) Write a C function that parses a string that represents a floating point number (example: “3.68”) and turns it into a real number (i.e., 3.68). (If confused by this, see the starter code.)
2) Quite a few hints are below.

This project will be graded by running the Unix script “proj2C_checker” on the VirtualBox. proj2C_checker is available on the course website.

If your program fail the checker script on the VirtualBox, then you should expect less than half credit.

There are various corner cases in implementing this. For example, parsing 3.23487293489 will break most implementations since “23487293489” is bigger than 2^31. I am not expecting you to figure out all of these corner cases. If you pass “proj2C_checker,” then you will get full credit (provided you actually have implemented parsing).

If the proj2C_checker program shows any difference, you will get less than half credit.

What should you upload?: Just a single file, which is your C source code.
Enums, structs, typedef, union
Enums

• Enums make your own type
  – Type is “list of key words”

• Enums are useful for code clarity
  – Always possible to do the same thing with integers

• Be careful with enums
  – … you can “contaminate” a bunch of useful words
enum example

C keyword “enum” – means enum definition is coming

```c
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

```c
int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
```
enums translate to integers ... and you can set their range

```c
#include <stdio.h>

enum StudentType {
    HighSchool = 105,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};

int main() {
    printf("HighSchool = %d, GradStudent = %d\n", HighSchool, GradStudent);
}
```

```bash
hank$ gcc enum2.c
hank$ ./a.out
HighSchool = 105, GradStudent = 110
```
But enums can be easier to maintain than integers

```c
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};
```

```c
int AverageAge(enum StudentType st) {
    if (st == HighSchool) return 16;
    if (st == Freshman) return 18;
    if (st == Sophomore) return 19;
    if (st == Junior) return 21;
    if (st == Senior) return 23;
    if (st == PostBacc) return 24;
    if (st == GradStudent) return 26;

    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Simple Data Types

- float
- double
- int
- char
- unsigned char

All of these are simple data types
Structs: a complex data type

• Structs: mechanism provided by C programming language to define a group of variables
  – Variables must be grouped together in contiguous memory
• Also makes accessing variables easier ... they are all part of the same grouping (the struct)
C keyword “struct” – means struct definition is coming

struct syntax

```
struct Ray
{
    double origin[3];
    double direction[3];
};
```

This struct contains 6 doubles, meaning it is 48 bytes

Declaring an instance

```
int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

“.” accesses data members for a struct
Nested structs

```c
struct Origin {
    double originX;
    double originY;
    double originZ;
};

struct Direction {
    double directionX;
    double directionY;
    double directionZ;
};

struct Ray {
    struct Origin ori;
    struct Direction dir;
};
```

```c
int main() {
    struct Ray r;
    r.ori.originX = 0;
    r.ori.originY = 0;
    r.ori.originZ = 0;
    r.dir.directionX = 0;
    r.dir.directionY = 0;
    r.dir.directionZ = 0;
}
```

- Accesses `dir` part of `Ray`
- Accesses `directionZ` part of `Direction` (part of `Ray`)
typedef

- typedef: tell compiler you want to define a new type

```c
struct Ray
{
    double origin[3];
    double direction[3];
};

int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

saves you from having to type “struct” every time you declare a struct.
Other uses for typedef

• Declare a new type for code clarity
  – typedef int MilesPerHour;
    • Makes a new type called MilesPerHour.
    • MilesPerHour works exactly like an int.

• Also used for enums & unions
  – same trick as for structs ... typedef saves you a word
So important: struct data member access is different with pointers

typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}

typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray *r = malloc(sizeof(Ray));
    r->origin[0] = 0;
    r->origin[1] = 0;
    r->origin[2] = 0;
    r->direction[0] = 1;
    r->direction[1] = 1;
    r->direction[2] = 1;
}

Pointers: use “->”
Instances (i.e., not pointers): use “.”
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int    y;
    char   z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This data structure has 4 bytes
Unions

Why are unions useful?

```c
128-223-223-72-wireless:330 hank$ cat union.c
#include <stdio.h>

typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3;   /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = %f, as u.y = %d\n", u.x, u.y);
}
```

128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
Unions Example

typedef struct
{
   int firstNum;
   char letters[3];
   int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
   char letters[3];
   int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
   int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
   CA_LICENSE_PLATE ca;
   OR_LICENSE_PLATE or;
   WY_LICENSE_PLATE wy;
} LicensePlate;
Unions Example

typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;

typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
Bonus Slides
Important note on debugging: buffering and printf

• Important: printf is buffered

• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed

• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

One solution: fflush(stdout)
Let’s pause and do a video on fflush
mv: Unix command for renaming a file

```
c02ln00gfd58:212 hank$ mkdir tmp
c02ln00gfd58:212 hank$ cd tmp
c02ln00gfd58:tmp hank$ ls
c02ln00gfd58:tmp hank$ touch a
c02ln00gfd58:tmp hank$ ls a
c02ln00gfd58:tmp hank$ mv a b
c02ln00gfd58:tmp hank$ ls b
c02ln00gfd58:tmp hank$
```