Reading 6

By the C, by the C, by the beautiful C

- C is an imperitive and procedural language.
  - ↳ It is less abstracted then Python

- Use '#include' to impart libraries

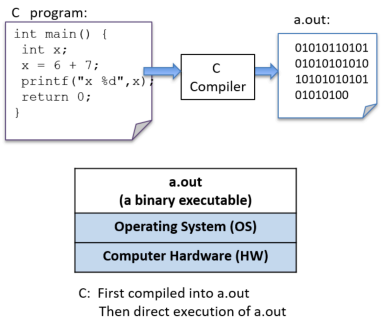
- The 'int main(void){}' returns 0 if program was completed without error.

- A C program must have a 'main' function and it must return an int.
  - ↳ The 'main' function is automatically run when the program executes.

- A python program is directly executed by the Python interpreter (a binary executable program that is run on the OS and hardware).

- A C compiler is a program that translates C source code into a binary executable form that the computer hardware can directly execute.

- Figure:



C: First compiled into a.out
Then direct execution of a.out

- Naming the executable: `$ gcc -o <output_executable_file> <input_source_file>`

- In C, all variables must be declared before they can be used.

- Variabe Types:
  - chars are single quoted
  - double quotes indicate a literal

  - Numerical:

Type name	Usual size	Values stored	How to declare
`char`	1 byte	integers	`char x;`
`short`	2 bytes	signed integers	`short x;`
`int`	4 bytes	signed integers	`int x;`
`long`	4 or 8 bytes	signed integers	`long x;`
`long long`	8 bytes	signed integers	`long long x;`
`float`	4 bytes	signed real numbers	`float x;`
`double`	8 bytes	signed real numbers	`double x;`

- C also provides unsigned versions of the integer numeric types.
  ↳ just add 'unsigned' key word before declaration

- Pre vs Post Increment

'++x': increment x first, then use its value

'x++': use x's value then increment it.

- Format string placeholders:

'%.g' → float or double

'%.d' → decimal value (int, short, chart)

'%.s' → string value

'%.c' → char value

- Reading from stdin Python vs C:

Python version	C version

```python
# Python input example


def main():


    num1 = input("Enter a number:")
    num1 = int(num1)
    num2 = input("Enter another:")
    num2 = int(num2)

    print("%d + %d = %d" % (num1, num2,
(num1+num2)))

# call the main function:
main()
```

```c
/* C input (scanf) example */
#include <stdio.h>

int main(void) {
    int num1, num2;

    printf("Enter a number: ");
    scanf("%d", &num1);
    printf("Enter another: ");
    scanf("%d", &num2);

    printf("%d + %d = %d\n", num1, num2,
(num1+num2));

    return 0;
}
```

- Boolean values in C:
  - 0 evaluates to false
  - nonzero evaluates to true

- C's short circuit operator evaluation stops evaluating a logical expression as soon as the result is known.

- do-while loops execute the body first and then check the condition!

```c
do {
    <body>
} while ( <boolean expression> );
```

- C does support for loops with a list of initialization and step parts:

```c
#include <stdio.h>

int main(void) {
    int i, j;

    for (i=0, j=0; i < 10; i+=1, j+=10) {
        printf("i+j = %d\n", i+j);
    }

    return 0;
}
```

- In C, everything done with a while loop can be done with a for loop and vice versa.
  - The same is NOT true for Python.
    - Ex: You can have an infinite for loop in C, but you cannot in Python

- In C, functions might take zero or more parameters as input and they return a single value of a specific type.

- The execution stack keeps track of the state of active functions in a program. Each function call creates a new stack frame (sometimes called an activation frame or activation record)

- Arrays and Strings

  - An array is an ordered collection of data elements of the same type and associates this collection with a single program variable.

  - To call a function that has an array parameter, pass the name of the array as the argument.

  - The name of the array is equivalent to the base address (address of item at index 0)

- Strings

  - C string must end with the null character ('\0'), to indicate the end of the string.
  - "hi" stores 'h', 'i', AND, '\0'

  - strlen() : returns length of string (not counting '\0')
  - strcpy(dest, src) : copies src to dest

· Structs

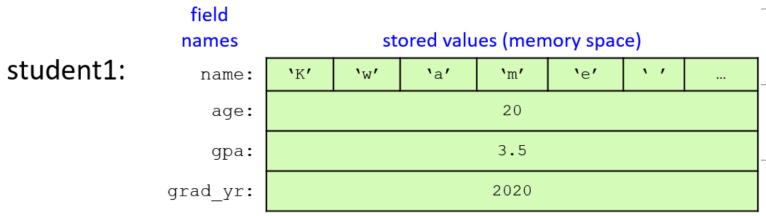
- A collection of data elements of different types.

- Declaration (use typedef for best practice):

```
struct studentT {
    char name[64];
    int age;
    float gpa;
    int grad_yr;
};
```

Expression	C type
student1	struct studentT
student1.age	integer (int)
student1.name	array of characters (char [])
student1.name[3]	character (char), the type stored in each position of the name array

- Only struct variable's fields are stored in memory.
- To the compiler field names are simply storage locations or offsets from the start of the struct variables memory.



- C struct types are lvalues, meaning they can occur on the left side of assignment.
  - lvalues are expressions that represent a memory storage location.

```
student2 = student1;  // student2 gets the value of student1
                      // (student1's field values are copied to
                      //  corresponding field values of student2)

strcpy(student2.name, "Frances Allen");  // change one field value
```

- When structs are passed to functions, they are passed by value.

• Pointer Variables

- A pointer variable stores the address of a memory location in which a value of a specific type can be stored.
  - ↳ usually used for "pass by pointer" parameters and dynamic memory allocation.

- Pointer variables store address values.
  - ↳ So usually assigning something with an address operator (&).
  - ↳ all pointers can be assigned NULL
  - ↳ * is the dereference operator
    - ↳ deferencing a pointer pointing to NULL will cause a segfault.

- Using pointers in function parameters passes an address by value BUT the address can be dereferenced to modify the value at that address

• Arrays in C:

- Use malloc() to dynamically allocate continuous memory on the heap.

- 2D arrays:

```
int   matrix[50][100];
short little[10][10];
```

```
int   val;
short num;

val = matrix[3][7];  // get int value in row 3, column 7 of matrix
num = little[8][4];  // get short value in row 8, column 4 of little
```

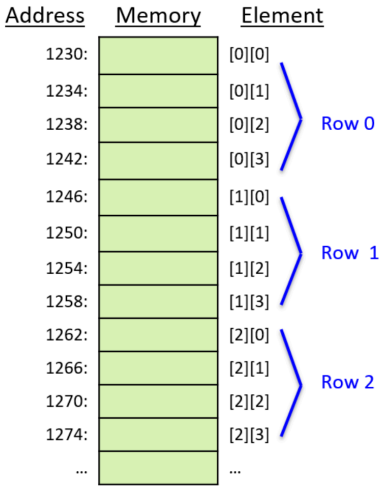
- For multi-dimensional array parameters, you must indicate that the parameter is a multidimensional array, but you can leave the size of the first dimension unspecified (for good general design)

• Ex:

```c
// a C constant definition: COLS is defined to be the value 100
#define COLS  (100)

/*
 * init_matrix: initializes the passed matrix elements to the
 *              product of their index values
 *   m: a 2D array (the column dimension must be 100)
 *   rows: the number of rows in the matrix
 *   return: does not return a value
 */
void init_matrix(int m[][COLS], int rows) {
    int i, j;
    for (i = 0; i < rows; i++) {
        for (j = 0; j < COLS; j++) {
            m[i][j] = i*j;
        }
    }
}

int main(void) {
    int matrix[50][COLS];
    int bigger[90][COLS];

    init_matrix(matrix, 50);
    init_matrix(bigger, 90);
    ...
```

• 2D array in memory:

```
int arr[3][4];
```



- 2D arrays of size $N \times M$ can be allocated in 2 ways:

1. Make a single call to `malloc`, allocating one large chunk of heap space to store all *N*x*M* array elements.
2. Make multiple calls to `malloc`, allocating an array of arrays. First, allocate a 1D array of *N* pointers to the element type, with a 1D array of pointers for each row in the 2D array. Then, allocate *N* 1D arrays of size *M* to store the set of column values for each row in the 2D array. Assign the addresses of each of these *N* arrays to the elements of the first array of *N* pointers.

- Method 1:

↳ memory efficient

↳ compiler treats this as 1D array

↳ so double indexing does NOT work

↳ instead you could use [i*M+j] to index

↳ M is column dimension
↳ i is row
↳ j is column

↳ Ex: Initializing:

```
// access using [] notation:
//   cannot use [i][j] syntax because the compiler has no idea where the
//   next row starts within this chunk of heap space, so the programmer
//   must explicitly add a function of row and column index values
//   (i*M+j) to map their 2D view of the space into the 1D chunk of memory
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        two_d_array[i*M + j] = 0;
    }
}
```

- Method 2:

↳ Stores the array as an array of N 1D arrays (one 1D array per row)

↳ Requires N+1 malloc() calls (one for the array of row arrays, N for the row's column arrays)

↳ Elements within a row are continuous, but elements are NOT continuous across rows of the 2D array.

↳ can use double bracket indexing.

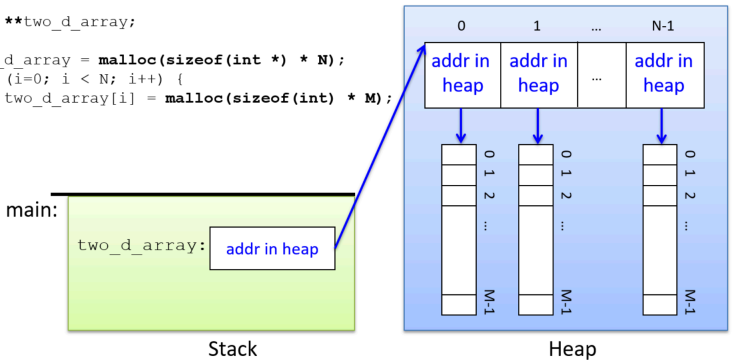
↳ allocation and element access are less efficient than method 1.

↳ Ex Allocating:

```
// the 2D array variable is declared to be `int **` (a pointer to an int *)
// a dynamically allocated array of dynamically allocated int arrays
// (a pointer to pointers to ints)
int **two_d_array;
int i;

// allocate an array of N pointers to ints
// malloc returns the address of this array (a pointer to (int *)'s)
two_d_array = malloc(sizeof(int *) * N);

// for each row, malloc space for its column elements and add it to
// the array of arrays
for (i = 0; i < N; i++) {
// malloc space for row i's M column elements
    two_d_array[i] = malloc(sizeof(int) * M);
}
```

Diagram of 2D array in memory:

```
int **two_d_array;

two_d_array = malloc(sizeof(int *) * N);
for (i=0; i < N; i++) {
    two_d_array[i] = malloc(sizeof(int) * M);
}
```



• I/O in C

- printf() placeholders for different number representations:
  - ↳ "%.x" → hexadecimal
  - ↳ "%.o" → octal

- scanf() example:

```
scanf("%d %g", &x, &y);
```

  - ↳ Individual input values must be seperated by at least one whitespace character.

- getchar() : reads value of next char for stdin
- putchar(myCh): writes the value of myCh to stdout

- Functions to change current position in file:

```
// to reset current position to beginning of file
void rewind(FILE *f);

rewind(infile);

// to move to a specific location in the file:
fseek(FILE *f, long offset, int whence);

fseek(f, 0, SEEK_SET);    // seek to the beginning of the file
fseek(f, 3, SEEK_CUR);    // seek 3 chars forward from the current position
fseek(f, -3, SEEK_END);   // seek 3 chars back from the end of the file
```

- Different placeholders for fscanf() to read from files:

```
%[...] string, up to first character not in brackets
%[0123456789] would read in digits
%[^...] string, up to first character in brackets
%[^\n] would read everything up to a newline
```

↳ Ex

```c
// read in a string consisting of only char values in the specified set (0-5)
// stops reading when...
//   20 chars have been read OR
//   a character not in the set is reached OR
//   the file stream reaches end-of-file (EOF)
fscanf(infile, "%20[012345]", array);

// read in a string; stop when reaching a punctuation mark from the set
fscanf(infile, "%[^.,:!;]", array);
```

Reading 6 Continued

- C constants:

```
#define N    (20)        // N:  alias for the literal value 20
#define PI   (3.14)      // PI: alias for the literal value 3.14
#define NAME ("Sarita")  // NAME: alias for the string literal "Sarita"
```

- C switch statement:

```
switch (<expression>) {

   case <literal value 1>:
        <statements>;
        break;          // breaks out of switch statement body
   case <literal value 2>:
        <statements>;
        break;          // breaks out of switch statement body
   ...
   default:             // default label is optional
        <statements>;
}
```

- An enumerated type is a way to define a group of related integer constants.
  - ↳ often used with switch statements.

- Ex Defining!

```
enum days_of_week {
   SUN,         // SUN will now be 0
   MON,         // MON will now be 1, and so on
   TUES,
   WED,
   THURS,
   FRI,
   SAT
};
```

```
enum days_of_week {
   SUN = 1,   // start the sequence at 1
   MON,       // this is 2 (next value after 1)
   TUES,      // this is 3, and so on
   WED,
   THURS,
   FRI,
   SAT
};
```

```
// an int because we are using scanf to assign its value
int val;

printf("enter a value between %d and %d: ", SUN, SAT);
scanf("%d", &val);

switch (val) {
  case FRI:
     printf("Orchestra practice today\n");
  case MON:
  case WED:
     printf("PSYCH 101 and CS 231 today\n");
     break;
  case TUES:
  case THURS:
     printf("Math 311 and HIST 140 today\n");
     break;
  case SAT:
     printf("Day off!\n");
     break;
  case SUN:
     printf("Do weekly pre-readings\n");
     break;
  default:
     printf("Error: %d is not a valid day\n", val);
};
```

•Typedef examples:

```c
typedef struct studentT {
  char name[MAXNAME];
  classYr year;     // use classYr type alias for field type
  float gpa;
} studentT;
```

```c
enum class_year {
  FIRST = 1,
  SECOND,
  JUNIOR,
  SENIOR,
  POSTGRAD
};

// classYr is an alias for enum class_year
typedef enum class_year classYr;

struct studentT {
  char name[MAXNAME];
  classYr year;     // use classYr type alias for field type
  float gpa;
};

// studentT is an alias for struct studentT
typedef struct studentT  studentT;

// ull is an alias for unsigned long long
typedef unsigned long long ull;

int main(void) {

  // declare variables using typedef type names
  studentT class[MAXCLASS];
  classYr yr;
  ull num;

  num = 123456789;
  yr = JUNIOR;
  strcpy(class[0].name, "Sarita");
  class[0].year = SENIOR;
  class[0].gpa = 3.75;
```

• Compiling

1. Precompiler Step: expands preprocessor directives:

```
$ gcc -E  myprog.c
$ gcc -E  myprog.c  > out
$ vim out
```

Shows immediate result after this step

2. Compile Step: source code to assembly

```
$ gcc -S  myprog.c
$ vim myprog.s
```

Shows immediate result after this step

3. Assembly Step: Assembly to binary

tells compiler to stop after this step

```
$ gcc -c  myprog.c

# disassemble functions in myprog.o with objdump:
$ objdump -d myprog.o
```

4. Linking Step: creates executable file

```
$ gcc myprog.c
$ ./a.out

# disassemble functions in a.out with objdump:
$ objdump -d a.out
```

• Debugging with GDB

- GDB users typically set breakpoints in their programs.
- when compiling, use the '-g' flag to enhance debugging

- running gdb!

```
$ gdb a.out
(gdb)          # the gdb command prompt
```

- Common gdb commands!

Command	Description
`break`	Set a breakpoint
`run`	Start program running from the beginning
`cont`	Continue execution of the program until it hits a breakpoint
`quit`	Quit the GDB session
`next`	Allow program to execute the next line of C code and then pause it
`step`	Allow program to execute the next line of C code; if the next line contains a function call, step into the function and pause
`list`	List C source code around pause point or specified point
`print`	Print out the value of a program variable (or expression)
`where`	Print the call stack
`frame`	Move into the context of a specific stack frame

- Ex

```
(gdb) break main

Breakpoint 1 at 0x8048436: file badprog.c, line 36.
```

The `run` command tells GDB to start the program:

```
(gdb) run
Starting program: ./a.out
```

If the program takes command line arguments, provide them after the `run` command (for example, `run 100 200` would run `a.out` with the command line arguments `100` and `200`).

- When you use 'next', a line will be displayed
  - ↳ this is where the program stopped, NOT the line that was executed

- If a function call is the line to be executed, then 'next' will execute the entire function.
  ↳ You can use 'step' to step into the function.

- You can set another breakpoint and then use cont to execute the program till that point, then you will be prompted again.

- If you are getting a segfault, you can use the 'run' the command.
  ↳ This will allow gdb to prompt you at the line of segfault.

- Keyboard shortcuts in GDB!
  ↳ 'p' → 'print'
  ↳ 'l' → 'list'
  ↳ 'n' → 'next'

- Also supports tab completion

- In depth GDB commands!

  - help

```
help <topic or command>   Shows help available for topic or command

help breakpoints    Shows help information about breakpoints
help print          Shows help information about print command
```

  - break

```
break <func-name>   Set breakpoint at start of function <func-name>
break <line>        Set breakpoint at line number <line>
break <filename:><line>  Set breakpoint at <line> in file <filename>

break main          Set breakpoint at beginning of main
break 13            Set breakpoint at line 13
break gofish.c:34   Set breakpoint at line 34 in gofish.c
break main.c:34     Set breakpoint at line 34 in main.c
```

- run

```
run <command line arguments>

run             Run with no command line arguments
run 2 40 100    Run with 3 command line arguments: 2, 40, 100
```

- step

```
step          Execute next line (stepping into a function)
step <count>  Executes next <count> lines of program code

step 10       Executes the next 10 lines (stepping into functions)
```

- next (executes entire function if function is called):

```
next             Execute the next line
next <count>     Executes next <count> instructions
```

- until

```
until <line>    Executes until hit line number <line>
```

- list

```
list                Lists next few lines of program source code
list <line>         Lists lines around line number <line> of program
list <start> <end>  Lists line numbers <start> through <end>
list <func-name>    Lists lines around beginning of function <func-name>

list 30 100         List source code lines 30 to 100
```

- frame (moving into different stack frame contexts)

```
frame <frame-num>   Sets current stack frame to <frame-num>
info frame          Show state about current stack frame

frame 3             Move into stack frame 3's context (0 is top frame)
```

- enable, disable, ignore, delete, clear:

```
disable <bnums ...>    Disable one or more breakpoints
enable  <bnums ...>    Enable one or more breakpoints
ignore  <bpnum> <num>  Don't pause at breakpoint <bpnum>
                         the next <num> times it's hit
delete  <bpnum>        Delete breakpoint number <bpnum>
delete                 Deletes all breakpoints
clear <line>           Delete breakpoint at line <line>
clear <func-name>      Delete breakpoint at function <func-name>

info break      List breakpoint info (including breakpoint bnums)
disable 3       Disable breakpoint number 3
ignore  2  5    Ignore the next 5 times breakpoint 2 is hit
enable  3       Enable breakpoint number 3
delete  1       Delete breakpoint number 1
clear   124     Delete breakpoint at source code line 124
```

- condition (stopping at breakpoints only when a condition is true):

```
condition <bpnum> <exp>    Sets breakpoint number <bpnum> to break
                           only when expression <exp> is true

break 28            Set breakpoint at line 28 (in function play)
info break          Lists information about all breakpoints
  Num Type           Disp Enb Address    What
   1   breakpoint    keep y   0x080483a3 in play at gofish.c:28

condition 1 (i > 1000)     Set condition on breakpoint 1
```

- print:

```
print <exp>     Display the value of expression <exp>

p i             print the value of i
p i+3           print the value of (i+3)
```

```
print    <exp>     Print value of the expression as unsigned int
print/x  <exp>     Print value of the expression in hexadecimal
print/t  <exp>     Print value of the expression in binary
print/d  <exp>     Print value of the expression as signed int
print/c  <exp>     Print ASCII value of the expression
print  (int)<exp>  Print value of the expression as unsigned int

print/x 123        Prints  0x7b
print/t 123        Print  1111011
print/d 0x1c       Prints 28
print/c 99         Prints 'c'
print (int)'c'     Prints  99
```

display:

```
display <exp>   Display value of <exp> at every breakpoint

display i
display array[i]
```

x (examing memory, similar to print but interprets argument as memory address):

```
x <memory address expression>

x  0x5678       Examine the contents of memory location 0x5678
x  ptr          Examine the contents of memory that ptr points to
x  &temp        Can specify the address of a variable
                 (this command is equivalent to: print temp)
```

whatis

```
whatis <exp>       Display the data type of an expression

whatis (x + 3.4)   Displays:  type = double
```

set

```
set <variable> = <exp>   Sets variable <variable> to expression <exp>

set x = 123 * y   Set var x's value to (123 * y)
```

info

```
help info       Shows all the info options
help status     Lists more info and show commands

info locals     Shows local variables in current stack frame
info args       Shows the argument variable of current stack frame
info break      Shows breakpoints
info frame      Shows information about the current stack frame
info registers    Shows register values
info breakpoints  Shows the status of all breakpoints
```

- Debugging with Valgrind:

  - Valgrind does not detect stack memory access errors at the same granularity as it does with heap memory, and it does not detect memory access errors with global data memory.

  - To view details of individual memory leaks, use the `--leak-check=yes` option.

  - Valgrind output preceeded by "==${PID}==" on every line

  - Most valgrind errors and warnings have the following format:

    1. The type of error and warning.
    2. Where the error occured (a stack trace at the point in the program's execution when the error occurs)
    3. Where heap memory around the error was allocated (usually the memory allocation related to the error.)

- Makefiles

  - The goal of makefiles is to compile whatever files need to be compiled, based on what files have changed.

  - Makefile Syntax:

```
targets: prerequisites
  command
  command
  command
```

  - The *targets* are file names, separated by spaces. Typically, there is only one per rule.
  - The *commands* are a series of steps typically used to make the target(s). These *need to start with a tab character*, not spaces.
  - The *prerequisites* are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called *dependencies*

  - Makefiles use timestamps to determine if something needs to be recompiled.

- Variables can only be strings
  - ↳ Reference them using either ${} or $().

- Calling 'make' will run the first rule by default, so you can do this!

```
all: one two three

one:
  touch one
two:
  touch two
three:
  touch three

clean:
  rm -f one two three
```

- When there are multiple targets for a rule, the commands will run for each target.
- `$@` is an automatic variable that contains the target name.

```
all: f1.o f2.o

f1.o f2.o:
  echo $@
# Equivalent to:
# f1.o:
#  echo f1.o
# f2.o:
#  echo f2.o
```

- The * wildcard searches your filesystem for matching filenames. (usually used in a wildcard function)

```
thing_wrong := *.o # Don't do this! '*' will not get expanded
thing_right := $(wildcard *.o)

all: one two three four

# Fails, because $(thing_wrong) is the string "*.o"
one: $(thing_wrong)

# Stays as *.o if there are no files that match this pattern :(
two: *.o

# Works as you would expect! In this case, it does nothing.
three: $(thing_right)

# Same as rule three
four: $(wildcard *.o)
```

- The % wildcard can be used for
  1. matching: matches one or more characters in a string (the match is called the stem)
  2. replacing: it takes the stem that was matched and replaces that in a string.
  3. % is most often used in rule definitions and in some specific functions.

- Automatic Variables:
  - "$@" → target
  - "$?" → all prerequisites newer than target
  - "$^" → all prequisites
  - "$<" → first prerequisite

- Important Variables (can be used by implicit rules)
  - "CC": program for compiling C programs, default is cc
  - "CFLAGS": Extra flags to give to the C compiler
  - "LDFLAGS": Extra flags to give to compilers when they are supposed to involve the linker.

- Adding a "@" before a command will stop it from being printed

↳ running with "-s" will add an "@" before each line.