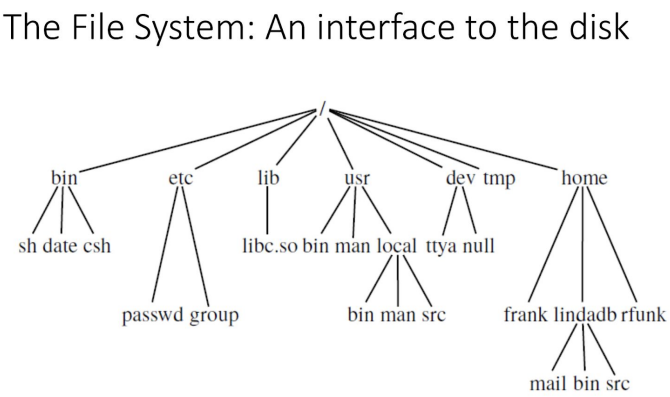
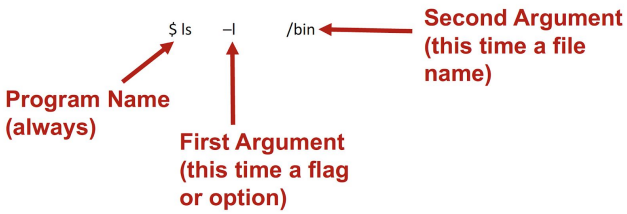
COMP 206 Midterm Review

An incomplete list of definitions and concepts:

- Filesystem

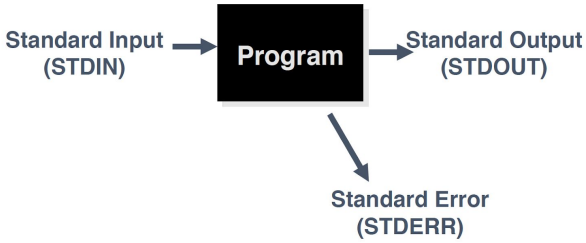


- Command-line arguments
  Type the program's name followed command-line arguments, the shell executes this program, feeding the options you specified



  The "--help" argument (e.g., "$ ls –help") can help understanding available flags and needed arguments

- Standard input/output



- Wildcards
- Interpreted vs compiled languages
  compiled language: source code cannot be run directly, but rather needs to be processed by a compiler which produces machine code.
- Pass-by-value
- C strings
  Arrays of characters (e.g. char name[100] = “David”;)

- Pointers and Arrays
- File IO
- 2D arrays
- Arrays of pointers
- Memory regions within a running process: stack, heap, data, text, BSS (not needed: kernel or mem map)

An incomplete list of programming elements in BASH:

- Frequently used commands
  ls - list files
  cd - change directory
  pwd - where am I now? (present working directory)
  mv - move files to directories
  find - search for files with given properties
  chmod - change permissions
  cp - copy files or directories
  cat - concatenate input files

  echo - copy input to output (why is this needed?)
  grep - filter input based on a pattern
  tr - translate inputs to outputs
  sort - sort inputs, then output
  ps - display running processes (once)
  top - display the running processes (continuous) and resource usage
  uname - print system information (which Linux version)
  ssh - remotely connect to another computer

  man – A linux command that shows the manual page for other commands!
  The "--help" argument (e.g., "$ ls –help")

  “/” is the root of the file system. Every other file falls below “/” in the directory tree (e.g., $ ls /)

  “~” is the current users home directory (e.g., $ ls ~/)

  “.” is means right here when it starts a path, and nothing if it occurs within a path (2nd case just a convenience for programming) (e.g., $ ls . or $ ls /usr/./bin)

  “..” means the parent directory (e.g. $ cd ..)

  Examples:
  $ ls

$ ls /usr
$ ls –l
$ du –h –-max-depth=1 /dev

$ echo hello world
$ cut –f3 –d" "
$ tr [a-z] [A-Z]
$ sort

- Variables

  Shell variable with special meanings:

  PWD current working directory
  PATH list of places to look for commands
  HOME home directory of user
  MAIL where your email is stored
  TERM what kind of terminal you have
  HISTFILE where your command history is saved
  PS1 the string to be used as the command prompt

  "set" command:

  The **set** command with no parameters will print out a list of all the shell variable.

  Fancy bash prompts:

  # PS1="Next command: "
  # PS1="# "

  \t is replace by the current time
  \w is replaced by the current directory
  \h is replaced by the hostname
  \u is replaced by the username
  \n is replaced by a newline

```
======= [foo.cs.rpi.edu] - 22:43:17 =======
/cs/hollingd/introunix echo $PS1
======= [\h] - \t =======\n\w
```

  Example:

  # my_var=Hello
  # echo $my_var

  # ls -al $HOME
  # var=ls
  # $var -al $HOME

```
# options=-al
# $var $options $HOME

# PS1="Next command: ”
# PS1=“# “
```

- For loops

```
for variable in wordlist
do
        stuff
done
```

Example:

Should we delete all the files?
(Example)

```
for fn in *
  do
  echo Should I delete file $fn
  read ok
  if test $ok != “no”

      then
      echo deleting $fn
      sleep 2
      rm $fn
      fi
  done
```

In this example, **!= “no”** needs to be changed to **== “yes”** (do not do this undiscoverable thing by default)

- While loops

```
while program
do
        list_of_commands
Done
```

Example:

```
x=1
while $x > 10
do
  print $x
done
```

```
x=1
while test $x -lt 10
    do
    echo $x
     x=`expr $x + 1`
    done
```

(the green one is correct)

- If conditionals

```
If program1
then
        commands
```

elif *program2*
then
    commands
else
    commands
fi

Example:

1) Command produces a nice output code that works as a “if” condition.

```
if date | grep Mon > /dev/null
then
    echo Another week starts.
fi
```

In this example:

1. In the “if” condition, we use the return code of “grep”: 0 (true) if some pattern found, 1 (false) otherwise
2. use pipe
3. use/dev/null, which is a special "file" specifically for the purpose of deleting whatever is put into it (the black hole of the file system!)

2) “Test” program for more general case

```
x=1
while test $x -lt 10
    do
    echo $x
     x=`expr $x + 1`
    done
```

- Test is so connected to shell conditionals that it has a special syntax:

```
if [[ b = a ]]
then
  echo hi
fi
```

- Note the spaces between each of the [[, each argument, and the]]
- Similar syntax for test patterns with one argument:

```
if [[ -r my_file.txt ]]
then
  echo I can do something with the file!
fi
```

with multiple conditions, we can use “&&” or “||” in the double square brackets

Syntax: test flags(s) arguments

– test -r file
  - is the file readable?
– test -w file
– test arg1 = arg 2
  - are the strings identical?
  - test arg1 != arg2: are the NOT equal?
– -gt, -le, -eq etc: numerical tests: greater than, less than or equal to, equal, ...

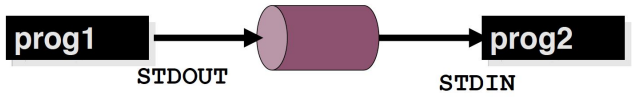
- Command substitution to variable $() or ``
  Re-use output through
  1) “>” storing the command’s output in a file
  2) skip the filesystem (efficiency of memory vs
  disk) and re-use the output within our BASH program

  Format: # variable=`command`

  Anything that # command alone would output to the terminal is now stored as the value of variable. Access it with $variable. (of course this name is just an example, we can pick any other, e.g. # daves_var=`comand`)

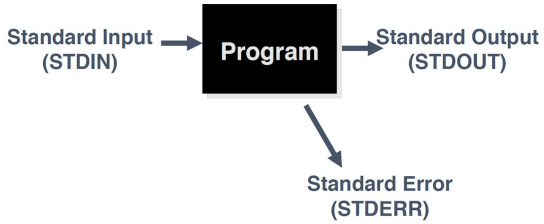
  A nearly equivalent syntax is # variable=$(command)

- Pipes
  A holder for a stream of data
  Can be used to hold the output of one program and feed it to the input of another



  Example:
  $ grep pattern < search_file.txt
  $ ls | grep
  $ ls > file_info.txt
  $ sort < nums.txt (The command above would sort the lines in the file nums and send the result to stdout)
  $ sort < nums > sortednums
  $ tr a-z A-Z < letter > rudeletter
  $ ls > foo.txt (create new file)
  $ ls /etc >> foo.txt (append to file, create new if not exist)
  $ ls | sort
  $ ls | sort > sortedls

- Input and output re-direction



Things you can attach to...
Standard Input:
keyboard
File, using “<” operator
The output of another command, using the “|” operator (pipe)

Standard Output (or stderr):
screen
file, using “>” operator (error stays on terminal, only standard output enters the file)
The input of another command, using the “|” operator (pipe)

Standard Error:
“1>” means to send standard output only (same as “>”)
”2>” means to send standard error only (std out might stay on terminal)
“&>” means to send both standard error and output
Note: Give different file names if both ”1>” and “2>” are used together. Otherwise, you will only get standard out.

- Performing math
  Enclose your computation in $((computation))
  Example:
  # a=$((3+5))
  # echo There are $((60*60)) seconds in an hour

- Wildards
  ? matches any single character
  ls Test?.doc
  [abc…] matches any of the enclosed characters
  ls T[eE][sS][tT].doc
  [a-z] matches any character in a range
  ls [a-zA-Z]*
  [!abc…] matches any character except those listed.
  ls [!0-9]*

- Quoting
  Double Quotes:
  To turn off special meaning - surround a string with double quotes (e.g. echo here is a star "*")

  With this “ “ solution:
  For $ and “, we have to use \ (e.g. \$ or \”)
  Math in $((..)) is still evaluated

Command-substitutions using $(…) or `..` are still evaluated

example:

```
echo here is a star "*"
```

Single Quotes: (Nothing is escaped)

$variables are not replaced by their value

Backslash is now no longer special

Math within $((…)) does not work

Command-substitution using $(…) does not work

example:

```
>echo 'This is a quote \" '
This is a quote \"
```

An incomplete list of programming elements in C:

- Variables

Some examples of valid (but not very descriptive) C variable names:

`foo`
`Bar`
`BAZ`
`foo_bar`
`_foo42`
`_`
`QuUx`

Some examples of invalid C variable names:

`2foo` (must not begin with a digit)
`my foo` (spaces not allowed in names)
`$foo` ($ not allowed -- only letters, and _)
`while` (language keywords cannot be used as names)

Standard C there are four basic data types. They are **int**, **char**, **float**, and **double**.

- Basic control constructs: while, for, if

```
for ( init; condition; increment ) {
   statement(s);
}

while(condition) {
   statement(s);
}
```

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
} else {
   /* statement(s) will execute if the boolean expression is false */
}
```

- Data types
  - Characters
    char (8 bits)
    - AASCI table
      A mapping between our printable letters and the 0's and 1's in memory
    - Equivalent integer/binary representations

• Single quotes for literals:	• char char_variable = 'w';
• Math allows moving alphabetically forward or backwards, finding relative positions	• char_variable++; (it now = 'x'); • char_variable - 'a' (tells you what position in alphabet, 23 here)
• Logic works via alphabetical order	• char_variable == 'x' (evals true) • char_variable > 'z' (evals false)

  - Integers
    - Binary values
      binary → decimal: Sum of 2^i for every "1", where i is the position in the number, with 0 on right
      e.g.1101 = 2^0 + 2^2 + 2^3 = 1 + 4 + 8 = 13
      Decimal → binary: while decimal_val is not zero, find the largest power of 2 that is less than or equal and add a "1" in the binary number in position i, where i is the position in the number, with 0 on right, decimal_val -= 2^i
      e.g. 48 gives 32=2^5, leaving 16, gives 16=2^4, leaving 0. So 48=110000
    - Various "lengths" and their memory implications
      Short (16 bits)
      int (32 bits)
      long (64 bits)
  - Floats
    float (32 bits)
    double (64 bits)
    long double (128 bits)
    - Conversion to/from integers

- Functions:
  - Return values
    A return type (can be void)

- Pass by value

```
int increment( int fnvar)
{
  fnvar++;
  return fnvar;
}

int main(){
  int a = 5;
  a = increment(a);
  printf( "The value of a is now %d.\n", a );
}
```

Note: you need **a =** increment(a) in order for the value of a to change. In order to change multiple values in a function, we need to instead pass the address to the variable, called a pointer.

- Variable scope
  Define a local scope for variables

- Text data
  - Printing to standard output or a file
  - Reading from standard input or a file
    Read from a file:
    use fopen(...):
    1) Returns a file pointer, always check if there was any problem NULL(=0). Otherwise, it is safe to use other file operators.
    2) Mode string indicates what we want to do with the file
    "r": read only. The file must exist previously with read permission.
    "w": write only. Create new file or overwrite previous contents.
    "a": append. Create new file or add to the end of previous contents.
    "b": can be added to any (e.g., "rb"), meaning to intpretthe file as binary. This is for next week.

    use fgetc(...):

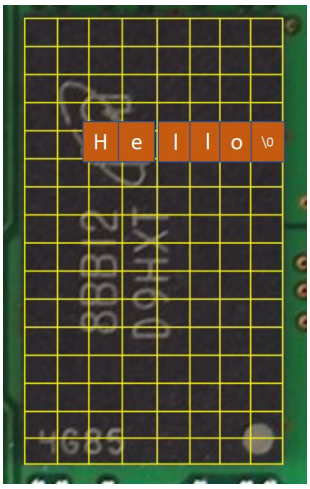
```
while( (input_char = fgetc( fp )) != EOF ){
    printf( "I read the character %c.\n", input_char );
}
```

Read line by line till the end-of-file is reached. EOF has the value –1. This is not a valid AASCI code, so we cannot mistake it for real data.

- Parsing text, lines into words, words into characters, based on delimiters such as comma, etc

- Creating text, building lines from words, words from characters, CSV, etc

- Concept of \0, use in various functions such as string length
  String in memory looks like the following...



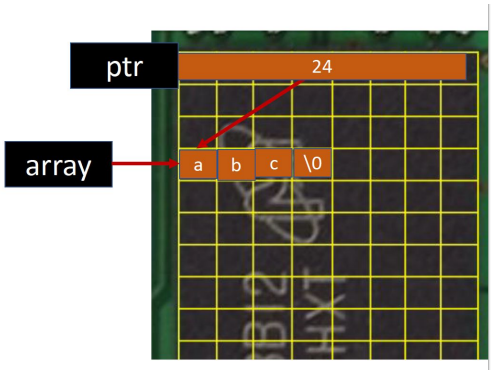

```
char str_var[100] = ”hello”;
for( int pos=0; pos<100; pos++ ){
   if( str_var[pos] == ‘\0’ ) break;
   printf( “%c”, str_var[pos] );
}
```

note: check for NUL (\0) for the end of string.

We get \0 at the end of our string for free when we create it with the “char str_var[5] = ”hello” syntax. This can lead us to start forgetting to add it when we create e.g.

```
char str[10];
str[0] = ‘h’;
str[1] = ‘i’;
str[2] = ‘\0’;
```

- Arrays and pointers:
  - How they are similar and different
    Similarity:
    1) An array in C is implemented as the address of its first entry. So, we “point to” the array

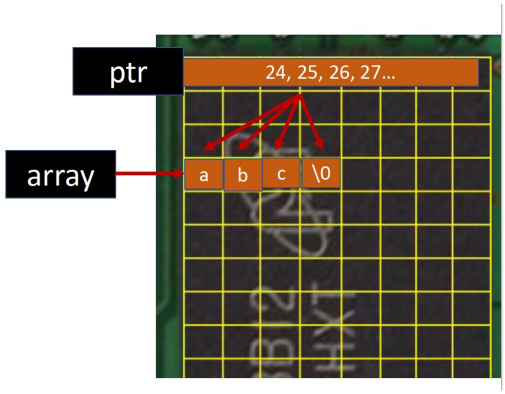


Example:

```
char array[4] = “abc”;
char *ptr = array;
```

Differences:

1) The array variable holds the address to the start of this memory always, while the pointer is more flexible

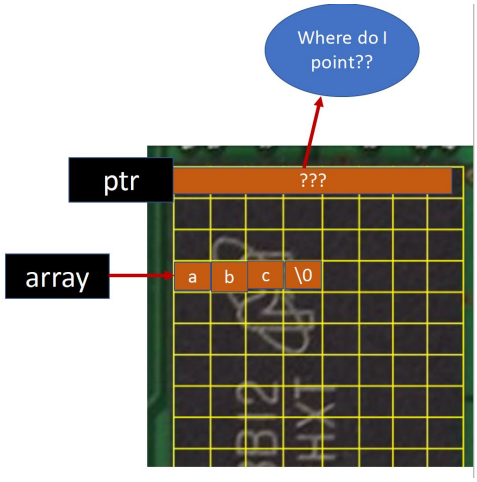


Example:

```
char array[4] = “abc”;
char *ptr = array;
ptr++; ptr++; ptr++; ptr++; // These work

array++; // This is an error!
```

2) An array says how much memory it requires at the beginning, whereas a ptr declaration alone does not request memory to store real data



Example:

```
char array[4];
array[0] = ‘a’; // This line is OK

char *ptr;
ptr[0] = ‘a’;    // This line may seg fault
```

Practice:

C stings using pointers

- Pointing to start of literal or array works
- Pointer math moves us around the string and computes distances
- Logic is based on the pointer position

- char *ptr = "hello";
- char str_array[100] = “hello”;
- char *ptr2 = str_array;

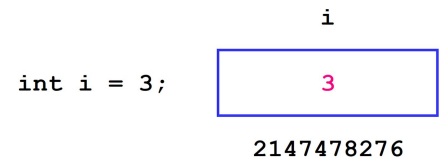
- ptr = ptr + 3; // Now points to lo
- ptr = ptr – 1; // Back to llo
- char *ptr2 = str_array;
- ptr – ptr2; // Gives position
  // difference, 2 here

- ptr == ptr2; // False, not same spots
- ptr > ptr2; // True, ptr is farther along

- Dereference operator

Variable identifier represents the value of the variable
Variable identifier proceeded with & represents the address of the variable

& means “address of” or “points to” variable
* means “dereference” or “get values at” pointer

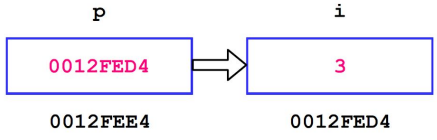



```
printf("The value of i = %d\n", i);
printf("The address of i = %p\n", &i);
```

Example:
1)

Example

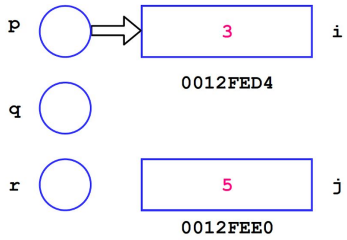
```
int i = 3;
int *p = &i;
printf("The value of i = %d\n", i);
printf("The value of i = %d\n", *p);
printf("The address of i = %p\n", &i);
printf("The address of i = %p\n", p);
printf("The address of p = %p\n", &p);
```



2)

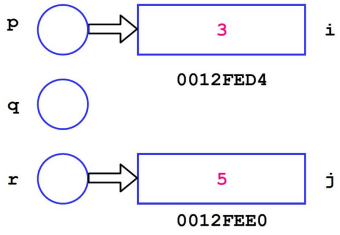
Example

```
int i = 3, j = 5;
int *p = &i;
int *q, *r;
```



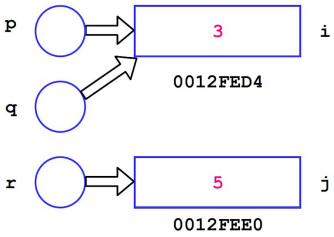
Example

```
int i = 3, j = 5;
int *p = &i;
int *q, *r;
 r = &j;
```



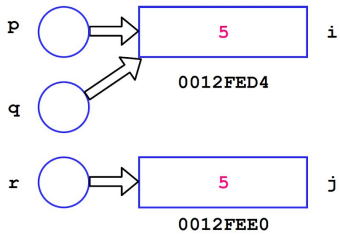
Example

```
int i = 3, j = 5;
int *p = &i;
int *q, *r;
r = &j;
 q = p;
```



Example

```
int i = 3, j = 5;
int *p = &i;
int *q, *r;
r = &j;
q = p;
 *p = *r;
```



void pointer:
It can point to any data type
Pointed data cannot be referenced directly
Type casting must be used to turn the void pointer to a concrete data type pointer

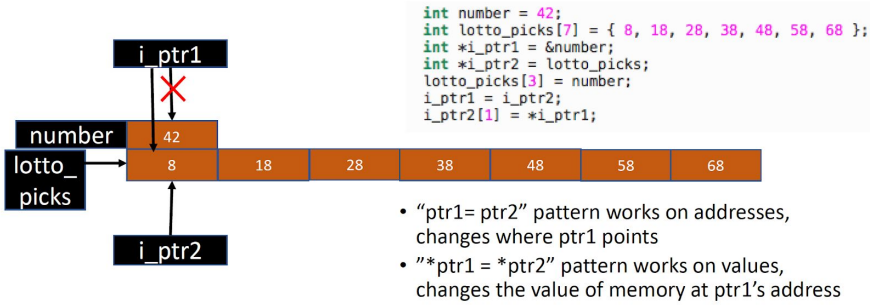
Example

```
int i = 3;
float j = 3.14;
void *p;
p = &i;
printf("The value of i = %d\n", *(int *)(p));
p = &j;
printf("The value of i = %f\n", *(float *)(p));
```

Practice from class:

```
char letter = 'b';
char sentence[100] = "2 words.";
char *c_ptr1 = &letter;
char *c_ptr2 = sentence;
sentence[3] = 'i';
c_ptr2[2] = *c_ptr1;
```

What will sentence and c_ptr2 print?



What will i_ptr1, i_ptr2 and lotto_picks print?

- Array indexing
  Array variables can be thought of as pointer always
  e.g. sentence[0], sentence[1]

- Relation between the two above
  Pointers can be indexed with [i] notation just like arrays
  *ptr is equivalent to ptr[0]
  More generally *(ptr+i) equiv ptr[i],

- Working properly with array lengths
- Using pointers to emulate "pass by reference"
  Call by address or pass by reference (return more than one values from a function):

Example – Pass by Reference

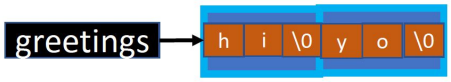
```
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
int main()
{
  int i=1, j=2;
  printf("%d %d\n", i, j);
  swap(&i,&j);
  printf("%d %d\n", i, j);
}
```

Example

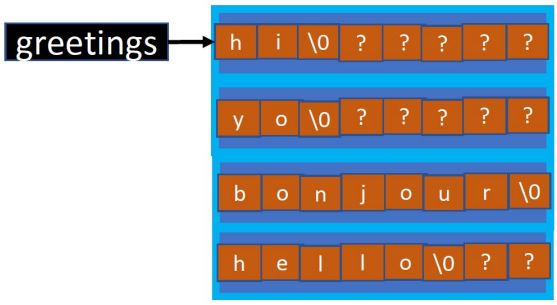
```
float compute(int r, float *p)
{
  float a;
  a = 3.14 * r * r;
  *p = 2 * 3.14 * r;
  return a;
}
int main()
{
  int r = 2;
  float area, perimeter;
  area  = compute(r, &perimeter);
  printf("%f %f\n", area, perimeter);
}
```

- 2D arrays

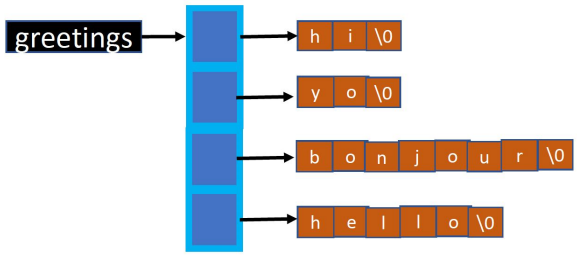
  char[2][3] greetings = { ”hi”, “yo” };



  char greetings[4][8] = {“hi”, “yo”, “bonjour”, “hello”}



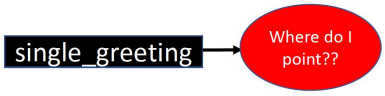
  char *greetings[4] = {“hi”, “yo”,”bonjour”, “hello”}



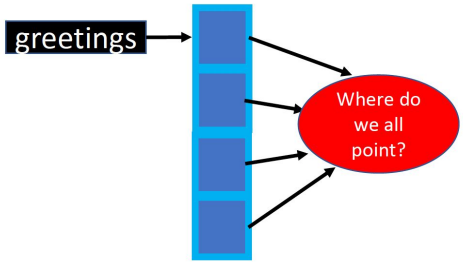
- Arrays of pointers

  uninitialized pointers:

  char* single_greeting;



char *greetings[4];



Assume we have char ttt[3][3], suppose we'd like a function that we can call like f(ttt): (lecture 11)

void f(char current_board[3][3]){ /* function code */ }

1. Match the types exactly: it must work!

void f(char current_board[][3]){ /* function code */ }

1. C needs to know the entry type to read the data correctly, so the char array of length 3 must be present
2. C does not need the length of the outer array: remember, it doesn't take care of this for us anyhow!

void f(char(*current_board)[3]){ /* function code */ }

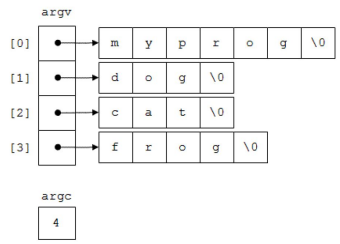
1. This says "pointer to data of type 3-length char array"
2. Same reasoning, as we can note the outer array with unknown size is equivalent to pointer
3. The brackets around (char*) are essential to distinguish this from an array of pointers (It is a good sanity check for you to be really sure you now know the difference yourself!)

Assume we have our char* greetings[4], and would like to be able to call f(greetings):

void f(char *greetings[4]){ /* function code */ }
void f(char *greetings[]) { /* function code */ }

1. Same reasoning as the above, we only have 1 array to track now and C can do that without the size

This is how "main" function argument works…
Type is char* argv[], an array of pointers, each to one of the argument



- Memory:
    - Parts of a running process
      Statically sized parts that come from gcc...
      Text space:
        1. Our runnable machine code
        2. Stored in the a.out
        3. The process loads these instructions and steps through line by line

      Data segment:
        1. Elements of fixed size that are known at compile time
        2. Stored in the a.out
        3. E.g. string literals
        4. Values cannot be changed

      BBS segment:
        1. Space to hold “static” variables without initial values
        2. Fill with zero at runtime
        3. Not actually stored in a.out, but only the size needed, since there is no initial data
        4. Changeable

      Elements that change in size...
      Stack:
        1. Space to store the local variables within each function
        2. Stack “frames” are created when the function is called, and removed when the function returns.
           This means stack variables, including arrays are temporary and we should not keep pointers to them

      Heap:
        1. Space for programmers to control dynamically
        2. Where we are allowed to request the most available resources
        3. Grow and shrink at our request
        4. In object oriented languages, used for “new” object

For later...
Kernel space:

1. we get to see a copy of a portion of the kernel in each process
2. This is a ”trick” of the operating system to give us low-level functionality

Memory mapping:

1. It’s for access to files and libraries that the Operating System connects us to

- Stack memory limitations

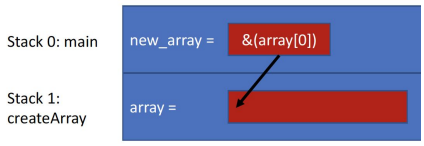
1) This array live on stack and there is a size limit on stack.

```
char array[100];
Array[99] = ‘a’
```

2) Variable on the stack are cleared after the function return. The common error “returning pointer to stack memory” (one may not exist without your awareness, stack for the variable was popped free)

When creating pointers, we must think about the stack push-pop behaviour. Similar example on Github: “stack_pointer_gotcha.c”

```
char* createArray( size ){
  char array[size];
  return array;
}
int main() {
  char *new_array = createArray( 10 );
  new_array[0] = ‘a’; // THIS LINE CAN SEGFAULT
}
```



- Malloc and the heap

  Provides flexible, persistent memory across function calls

  Request for N bytes of heap memory (not initialized):

  ```
  void *malloc(size_t numberOfBytes);
  ```

  Request for an array of N elements each with size bytes, and initializes the values all to 0:

  ```
  void *calloc(size_t num, size_t size_of_each);
  ```

  malloc and calloc return a void pointer (void *)

  It must be cast before it can be de-referenced:

  ```
  int *a = (int *) malloc( sizeof(int) * 40 ); // OR
  int *a = (int *) calloc(40, sizeof(int));
  ```

  The sizeof() function simplifies the allocation of memory by calculating the size of the provided data type.

  You have 3 TYPEs to fill in:

  - TYPE1 variable_name = (TYPE2)malloc(sizeof(TYPE3)*number);
  - int *pi = (int*)malloc(sizeof(int) * 1);

  RULE1: TYPE1 matches TYPE2, both are pointer types:

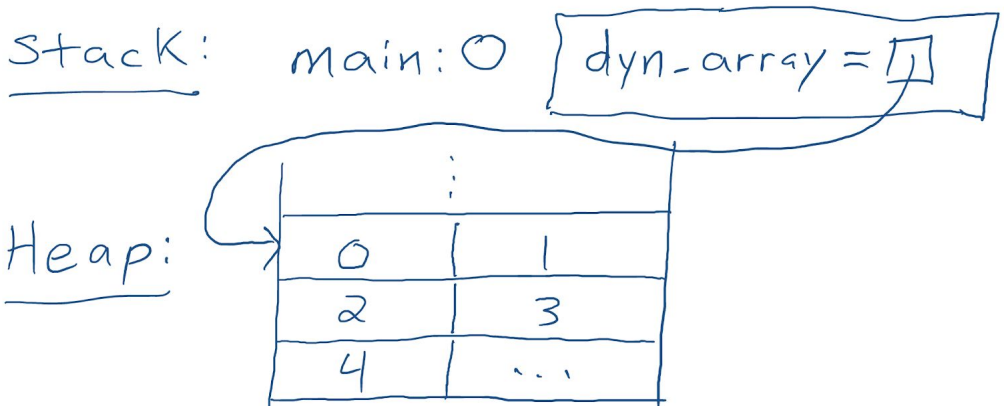
  - The point of casting is to tell C how we plan to interpret the heap memory allocated for us by malloc. Malloc returns void* so that it can handle any type. Since this is initially empty memory, casting is always safe.

  RULE2: TYPE3 is the de-referenced version of TYPE2

  - "One less star"
  - When we de-reference the variable with "*variable_name", C will assume the memory is of the pointer's underlying type

Malloc visual example

```
int *dyn_array = (int*)malloc(5*sizeof(int));
```



- Allocate a single integer on heap:
  - int *pi = (int*)malloc(sizeof(int));
- Allocate an array of 10 integers on heap:
  - int *my_numbers = (int*)malloc(10*sizeof(int));
- Allocate a single integer pointer on heap:
  - int **ppi = (int**)malloc(sizeof(int*));
  - This is our first time seeing a double pointer. It will be explained in due order, but note it's nothing scarier than single pointers. Just follow the arrow twice!