Lecture 1

Mon. Aug. 26, 2019

Classic Book (to learn C)
- kernighan & Ritchie, The "K&R" of C

C is still evolving
- Different standard version
  - K&R C
  - ANSI C
    - C89, C90 (ISO 9899:1990)
  - C99 1999
  - C11 2011
  - C18 2018

C should be...
- Simple
- easy to compile
- typed [weakly]
- support low-level memory access
- ideal for embedded controller, OS (operating system), …

Yet…
- C is powerful
- C is fast

C is a purely procedural language
- No object-orientation whatsoever

Central Dogma
- Object of interest: Computations
- Main abstraction tool: Procedures/ functions
  - Caller / Callee
  - Abstracts away "How things are done"
- Programming means
  - Organizing processes as procedures
  - composing processes thru procedure calls

Procedural Programming in C

- Adheres to the philosophy
- Generates very efficient code
- Exposes as many low=level details you wish to see
- Provides full control over memory management (no Garbage Collection)
- The Programmer is fully in charge

CSE 3100 Master Notes

Resources

What do you need

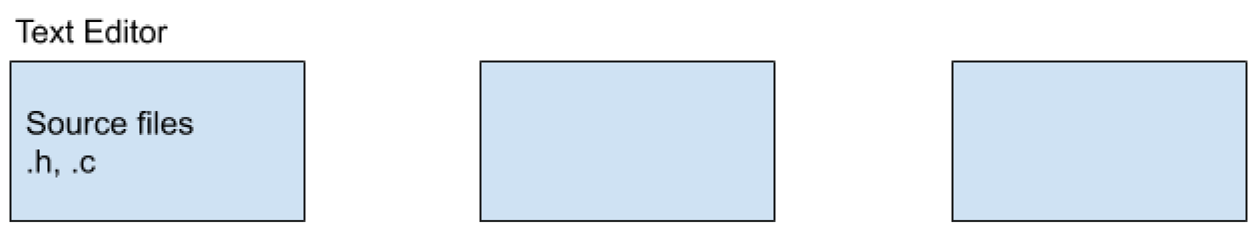
- C compiler
- linker
- text editor

We will use the GNU toolchain

- Compiler: GCC (or CC)
- Linker: ld (invoked by GCC)
- Editor: vim, nano, emacs

Workflow

- Use text editor to edit source files
- Use compiler to generate ".o" files
- Use linker to link multiple '.o" files into executables




```
#include <stdio.h>

/* comments */
// single linek comments

int main(void)
{
	printf("Hello, world! \n");
	return 0;
}
```

Comments

Compiler ignores everything between "/* */"

The 'main' function

A special function that defines the entry point for the program

- This is where the OS transfers control once the program starts

printf

- C library function to print on the standard output for the process
- takes at least a string as argument
  - Between double quotations like **"This is a string"**
- **'\n'** is a new line character

Including header files

#include

- imports a header files with the specification of functions that exists in a library to be linked w/ the program

Compile

What is CC really doing?

- 3 steps
  - *preprocesses* hello.c
  - *compile* hello.c to hello.o
  - *links* hello.o /w **libc**
  - *writes* executable file **a.out**
- **You can and often will separate the 3 steps!**
  - No necessarily that often unless working with big files
  -

A second program

Purpose

- Read an integer from the standard input: **n**
- Compute the sum of all integers between **1 & n**
- Print the result on the standard output

New concepts

- Standard input and output

Summary

C program consists of functions

- main() is the entrance of a program

function consists of a sequence of statements

Variables can be declared in a function (like main)

- These are local variables
- Variables must be declared

Statements are terminated with ';'

- Empty statements are allowed
  ;;; empty statements

The 'main function' taking arguments

'main' function can take 2 arguments

- argc: the number of arguments
- argv: an array of arguments

If-else statements

Lecture 3

Mon. Sept. 4, 2019

Operators

- Conventional arithmetic, bitwise, and logical operators
  - + - * / %
  - & | ~ ^ << >>
  - && || !
- Pre/post increment/decrement (as in Java, C++< etc.)
  - i++ ++i j++ --j
  - c = i++; // c will (i - 1)
  - c = ++i; // c will be the same as "i"
- Simple & Compound assignment operators
  - more to come!

Precedence & Associativity

- Precedence determines which operation is done first
  - If operators have the same precedence, use associativity
  - use parentheses

i + j * 10 - k / 20
(i + (j * 10)) - (k / 20)

Assignment Operators

- Assignment Operator
  - LHS (Left Hand Side) is something that can be written to (e.g. to a variable)

    LHS = Expression
  - LHS and Expression have "compatible" types

- The value of Expression is assigned to LHGS & becomes the value of the assignment operation
- Compound Assignment operators (+=, *=, …)
  - var op= expr ⇔ var = var op expr
- ex.):
  - a = x + y; b = c = d = 0;
  - i += 10; // i = i + 10
  - j *= 5; // j = j * 5

Assignments **ARE NOT** statements

- assignments are **expressions** and **"=" is the operator**
  - you can chain them!
  - you can use them inside larger expressions

Integer Data Types

- char
- short int → short
- int
- long int → long
- long long int → long long

*****C code can be confusing, people flex their skills in C, but it can be vary unreadable "Don't be that guy flexing skills, just because you can do it in C" - Wei We**

- Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 … 127	unsigned char 0 … 255
16	short -32786 … 32767	unsigned short
32	int	unsigned int
32	long $- 2^{31}$ .. 2^{32} - 1	unsigned long
64	long long	unsigned long long

- Consider x86_64 (64-bit architecture)

size (in bits)	signed	unsigned
8	char -128 … 127	unsigned char 0 … 255

16	short -32786 … 32767	unsigned short
32	int	unsigned int
32	long $- 2^{31}$ .. 2^{32} - 1	unsigned long
64	long long	unsigned long long

How much space?

- How to determine the amount of space for some type?
- Operator size of gives the # of bytes needed for a type of variable
    - You will need this later to dynamically allocate Space

Character (char) Data Type

- char has 8 bits (a byte)
- ASCII Code (American Standard Code for Information Interchange)
    - Characters are mapped to an integer in 0 … 127
    - An ASCII character can be stored in char
- Classes in ASCII
    - 0 … 31: "Control" character (a.k.a non printable)
    - 48 … 57: Digits
    - 64… 90: Upper case letters
    - 97… 122: Lower case letters

So…

- The character "h" is none other than … 72
    - char h1 = "H', h2 = 72; // h1 & h2 have the same value
- Observe how…
    - '0' through '9 are consecutive!
    - 'A' through 'Z' are consecutive!
    - 'a' through 'z' are consecutive!

  char ch = '8';
  int x = ch - '0' // what is the value of x?

Non-Printable Characters?

- These are sometimes useful
    - Showing the constant (literal)

'\n'	newline

‘\r’	carriage-return
‘\f’	form-feed
‘\t’	tabulation
‘\b’	backspace
‘\x7’	audible bell (x indicates hexadecimal)
‘\07’	audible bell (0 indicates octal)

Basic Data Types: Floating Point

- A few floating point types
  - Consider x86_64 again

size (in bits)		

Automatic Type Conversion

- When an operator has operands of different types, the operands are **automatically** converted to a common type by the compiler
  - In general, a lower rank operand is converted into the type of the higher rank one, where
    - *char < short < Int < long < long long < float < double < long double*
    - **long double is of highest rank**
  - E.g. “1” gets converted to double before performing the addition in the expression “1 + 2.5”
- Automatic conversion can also occur across assignments
  - The value of the expression on the right hand side may be widened to the type of the LHS, e.g., “double d = 1”
  - Or **narrowed** (possibly with information loss), e.g., “int i = 2.5”;
    - Read book for more details!

Type Casting: Explicit Type Conversion

- Useful to convert an operand to another type before doing arithmetic
  - *(<Type>)<expression>*

- Ex.): integer or double?

int x = 10; **int y = 3;** **double z = x / y**	

What about Booleans?

- K&R and C89/C90 do not have a Boolean data type
    - 0 "means" FALSE and anything else "means" TRUE
    - Common to use int or char to store Boolean values and define convenience macros
- C99 introduced Boolean

Be Mindful…

- Sometimes the results may not be as expected!
    - What is the size (in bits) of each operand?
    - Are your operands signed or unsigned?
- Ex.):

unsigned int x = 3;
unsigned int y = 7;
unsigned int z = x - y;
	z holds the binary representation of -4 but reading it as an unsigned int yields a very different value!

_Bool b1;
char b2, b3;
int i = 256; // 0x100
 	
b1 = i;
b2 = i;
b3 = i != 0;
	b1 is 1 because i is not 0
	b2 is - because the lowest 8 bits in "i" are 0
	b3 is 1 because i is not 0
	
	Do you want b2 or b3?

Lecture 4

Mon. Sept. 9, 2019

Flow of Control

- Statements are normally executed sequentiallu
- For selective or repeated execution we have all the suual suspects from Java / C++ / Python
    - blocks
    - if & if-else
    - while

- for
- switch
- break
- contine

Blocks (compound statements)

- List of statements enclosed by { and }
    - considered as a single statement
    - No semicolon after closing }
    - can be empty
    - can be nested (block in block)
    - useful for branching/ loop statements
    - can define variables at beginning of blocks
        - can mix declarations and code in c99

Comparison & Logical Operators

- Comparison operators that compare two expressions
    - Pay attention to types

== != > < >= <=

- Logical Operators

&& || !

- The result is either 0 or 1 (of int type)
    - 0 = false, 1 = true

Branching: if & else

- "exp" is typically a comparison or logical expression, but can be ANT expression (float/double, pointer, …)
- The statements can be compound statements (blocks)
- Or other if statements!
    - Beware of the dangling else ("else" matches the nearest preceding "if", use blocks to disambiguate)

***Lot of things are "passable" in C and can compile and run fine, but may not be readable whatsoever by other programmers. Beware of this when using branching - Wei Wei

Ex.):

```
int i, j, min;
if (i < j)
        min = i;
```

Ternary Operator

- Takes **3** expression as operands

  exp ? exp2 : exp3

  - exp1 is evaluated first
  - If exp1 is non-zero (true), exp2 is evaluated and its value is used as the value of the ternary expression
  - If exp1 is zero (false), exp3 is evaluated and its value is used as the value of the ternary expressions
- **ex.):**

  min = i < j ? i : j

While Loop

- Very similar to python,
- ex.): computing sum of 0 .. 99

```
int i = 0, sum = 0
while (i < 100){
        sum = sum + i
        i++;
}
// Same as
while (i < 100) sum += i++;
```

Do-While Loop

- Checks condition after executing loop body
  - the statement is executed at least once
- ex.): computing sum of 0 .. 99

```
int i = 0, sum - 0;
do{
        sum = sum + 1;
        i++;
} while (i < 100);
```

For Loop

- Sometimes called “counting” loop
  - more like swiss-army knife!
- Three expression:
  - initialization, condition, increment
- Equivalent to

```
exp1;
while (exp2){
        <stmt>;
        exp3;
}
```

- 4 ways to use for-loops to computing the sum 0..99

```
sum = 0;
for (i = 0; i < 100; i++) sum = sum +i;
```

2nd way - w/ all initializations inside

```
for (sum = i = 0; i < 100; i++) sum += i;
```

3rd Way - Empty Body

```
for (sum = i = 0; i < 100: sum += i++);
```

4th Way - Comma Operator

```
for (sum = 0, i = 0; i < 100; sum += i, i++);
```

Comma Operator

- Takes 2 expressions

  exp1, exp2

  - exp1 is evaluated first, then exp2 is evaluated
  - exp2 is the result of the whole expression
- Has the lowest precedence
- Associated from left to right

  exp1, exp2, exp3 <-> *(exp1, exp2), exp3)*

  - Order can make a difference, e.g.,

    for (sum = 0, i 0; i < 100; sum += i, i++);
  - is not the same as

    for (sum = 0, i = 0; i < 100; i++, sum += i);

Multiway branching using “else if”

- if statement can contain multiple else statements

Switch example

```
// Assume all variables are defined as int
```

```
switch(i){
        case 0;
                n0 ++; break; // Note the break statement
        case 1;
        case 2;
```

Break Statement

- Most commonly used in switch statements
  - Prevents “fall-through” to the next case
- Also works in loops (for, while, do-while)
  - Loop execution terminated immediately, control resumes at statement immediately following the loop

Continue Statement

- Skip the rest of the current loop iteration and continue to the next one
- Can be used within for, while, and do-while loops
  - Can appear in a nested if / else
  - If used in nested loops, it applies to the “innermost” enclosing loop
  - For “for” loops, go to the evaluation of the “increment” expression

```
{ // begin loop body
        ...
continue;
        …
} // end loop body
```

Lecture 8

Weds. Sept. 11, 2019

Function Definitions

- No nesting (cannot define functions in a function)
- Return type can be “void” (no return value expected)
  - if missing, compiler assumes int
- return statements

```
return;         // terminates execution and return control to caller
return expr;    // terminate and pass value of expr back to caller
```

- Execution also terminates if end of function body reached
  - return value undefined

Function Declarations (Prototypes)

- Functions can be defined in any order
    - declare a function before first use if definition comes later
    - function prototypes often placed in header files (and reused)

```
#include <stdio.h>
int fahrToCelsius(int);

int fahrToCelsius(int degF) {
        return 5 * (degF - 32) / 9;
}
```

Example: computing b^n

- Power function
    - returns an int
    - 2 parameters: base “b” and exponent “n”, both int
- Functions can declare local variables
    - parameters & local variables can only be accessed inside the function
    - Storage class “auto” by default i.e., discarded when function returns
- Parameters are passed by value
    - “n” is changed in the power function but “i” DOES NOT change

Static & Global Variables

- Static local variables
    - Not visible outside function but retain value across function calls
- Global variables
    - declared outside functions; retained for entire duration of program
    - Storage class “extern” by default
      → can be accessed from functions in other files
    - Static global variables
      → visible only in functions defined in the same file following variable declaration

Be Cautious with static and global variables

- “Nice” functions only depend on their inputs
- Static and global variables have “side-effects”
    - retain values across function calls
    - change the meaning of the function at each call
    - You can understand the function without holding all the code in your head
- unless you have really good reasons and really know what you are doing, DO NOT USE STATIC OR GLOBAL VARIABLES

Function call context

- Function call context includes
    - copies of function arguments (call-by-value_
    - auto local variables
    - return address
- Call contexts managed automatically using the **execution stack**
    - a stack frame is created automatically for each function call
    - the frame lasts for the duration of the call
    - discarded automatically when the function terminates
    - NOTHING in the frame survives the call

Lecture 9

Mon. Sept. 16, 2019

A C Primer (5): Arrays and Pointer Basics

Arrays

- New Data types
- Arrays represent a linear, contiguous collection of “things”
- Each “thing” in the array has the same fixed type
- ex.):
    - array of characters
    - array of integers
    - array of doubles…

Array example

```
int x[5]; // define an array of 5 int's
 // accessing array elements is similar to accessing list in Python
// the index starts from 0
x[0] = 1; x[1] = 2; x[2] = 3; x[3] = 4; x[4] = x[3] + 1;
// initialize array with a list
int y[5] = {1, 2, 3, 4, 5};
// Number of elements is optional if all elements are listed
int z[] = {1, 2, 3, 4, 5};
// Specify the value of first 2 elements. The rest are set to 0
```

```
int a[5] = {1, 2}; // C99. b will have 1, 2, 0, 0, 5.
int b[5] = {1, 2, [4] = 5};
```

Array In Memory

- Think about how array elementts are stored in memory
- Index starts from 0, the last one is 4 = (5 - 1)

String Initialization

- A string is a char array that ends with a 0 (null character)
    - Memory that stores 0 is part of the string
- It can be initialized with a list of characters
- or a string (double-quoted literal)

Arrays as Automatic Variables

- • You can declare arrays inside any function or block
    - Destroyed when exiting from the function or block
- Variable length arrays(VLA, C99) The size of your array can depend on function arguments or other known value

```
int foo(int n,int k) {
      int x[n]; // The value of n is known at this time
// Like other auto variables, x is kept on
// the stack and is NOT initialized

for (int i = 0; i < n; i++)
            x[i] = 0;
      ……
       return -1;
}
```

Array Assignment

- You CANNOT assign a whole array at once to another array
    - Even when the types match

Arrays and Functions

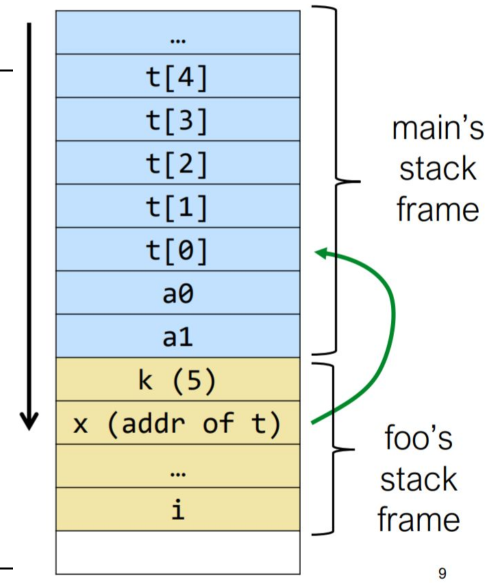
- Arrays can be passed to functions!
    - 1 BIG CAVEAT
- Calling convention in C
    - by value for everything
    - except arrays
- Arrays are always passed BY REFERENCE

  - passed as “pointers” - we’ll look at pointers soon
- Functions cannot return arrays
  - No easy assignments

Array argument Example

- Address of “t” is passed to foo()
- Modifications to “x” are visible in main

```
int foo(int x[], int k) {
      for (int i = 0; i < k; i++)
            x[i] = i;
      return x[0];
}

int main() {
      int t[5];
      int a0 = foo(t, 5);
      int a1 = t[4];
      printf("%d %d\n" , a0, a1);
      // more code
}
```



Multidimensional arrays

```
// declaration and initialization
int h[2][3] = { {0, 1, 2}, {10, 11, 12} };
```

	0	1	2
0	0	1	2
1	10	11	12

Pointers

- Perhaps the scariest part of C
- Yet…
  - The most useful part of C!
- Pointer is simply
  - a value
  - denoting address of a memory cell

Variables and Memory

- The memory is an array of bytes
- Every byte in memory is numbere: the address!
  - An address is just an unsigned integer
- Every variable is kept in memory, and is associated with 2 number
  - The address
  - The value stored at that address

Implicit Address Use

- address are used implicitly by the compiler all the time

```
int foo (int v)
{
      int a, b;  // allocate storage space for a and b
      a = 1;           // store 1 to memory, at a's address
      b = a;           // load value from a's address, write to b's
address
      v = v + b;       // load value from v's address, add to value at
                 // b's address, and write result to v's address
      return v;
}
```

	Address	Value
	100012	
b	100008	
a	100004	1

Explicit Use: Pointers

- A pointer is a variable that holds an address of something
- Declaration

// declare p to be a pointer to an int

- the value of p is an address of an int, p itself has an address

Referencing and dereferencing

- Two new operators
  - & Reference - "get" the address of something
  - * Dereference - "use" the address

Ex.):

```
int x = 10, y;
// px is a pointer to int, i.e., the address of an integer
int *px;         // px itself has an address

px = &x;         // &x is the address of x. Save it to px

// *px: use px as an address to get the value at that location
y = *px;         // and save the value in y

// save 20 to location pointed to by px (use px as an address)
*px = 20;        // px has x's address, so x becomes 20
```

Picture It

```
// assume 32 bits in and address
int x = 10, y;
int *px;
px = &x;
y = *px;
*px = 20;
```

?	Address	Value
	1007	
	1006	
x	1020	10
y	1016	?
px	1012	1020
	1008	
	1004	
	1000	

Revisit the Example

```
int foo(int x[], int k) {
    for (int i = 0; i < k; i++)
        x[i] = i;
    return x[0];
}
```

x is the starting address af an array, which is the same as the address of element 0.

```
// x is the address of an int
int foo(int * x, int k) {
    for (int i = 0; i < k; i++)
        x[i] = i; // use the pointer as an array
    return x[0];
}
```

Array Question 1

```
// how many integers are in a, and in b?
int a[] = {1, 2, 3, 4};
int b[4] = {1, 2, 3};
```

Both have 4 integers

Array Question 2

```
// how many bytes (characters) in c? and in d?
char c[] = {'a', 'b', 'c', 'd'};
char d[] = "abcd";
```

in “c” is 4, in “d” is 1

Array Question 3

`int a[10];`

`If a[0]'s address is 1000, what is a[4]'s address?`

1016

Array Question 4

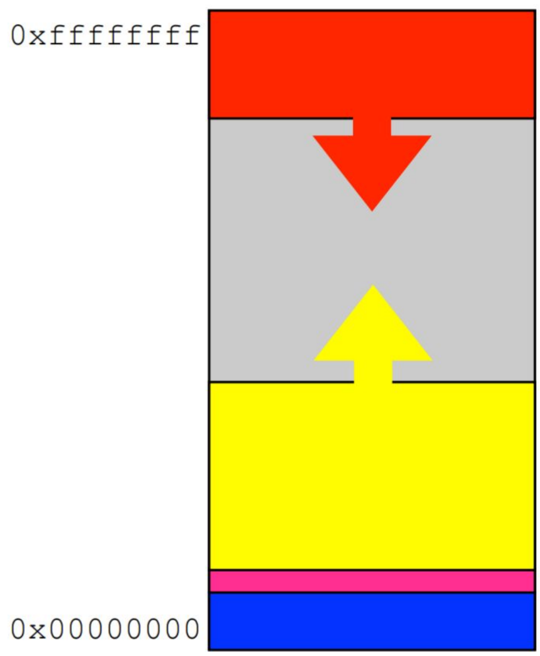
`char t[10][20];`

`If t[0][0]'s address is 1000, what is t[1][1]'s address?`

1041

Lecture 10 - C6: Dynamic Memory Allocation

Weds. Sept. 18, 2019



Recall the memory model…

- Three pools of memory
    - Static/global
    - Strack
    - Heap
- Each pool features
    - Different lifetime
    - Different allocation/deallocation policy

Static/global memory pool

- This is where…
    - All constant (including string literals) are held
    - Global variables
    - All variables declared
- Allocated when - the program starts
- Deallocated when - the program terminates
- FIXED size - compiler needs to know the size & make reservations

Stack

- This is where…
    - Memory comes from local variables in functions!
- Easy to manage because it is automatic!

- Allocated automatically when entering the function
- De-allocated automatically when you leave the func.
  - Scope is that of func.
  - Should not be used after the fuc. returns
    - For ex): indirectly used via a pointer ← DO NOT DO

default stack size is 2 MBs
need to increase stack size for large arrays and deep recursion

Heap

- This is where…
  - Memory comes from **manual** “on-the-fly” allocations
- Who is in charge?
  - **The programmer** for both allocation / deallocation
- Lifetime of memory blocks?
  - As long as they are not freed

Requesting memory on the heap

- `#include <stdlib.h>`
- `void* malloc(size_t size);`
  - size_t is an unsigned integer data type defined in <stdlib.h>
  - used to represent sizes of objects in bytes
- If successful, a call to malloc(n) returns a **generic pointer (void *)**
  - It points to a memory block of **“n”** bytes on the heap
- If not successful, NULL is returned

```
char* p = malloc(100); // request 100 bytes
```

Generic pointers: void*

- Pointer to a memory block whose content is “untyped”
- use for raw memory operations or in generic funcs.
- Automatic casting when assigned to other pointer types
  - `int * pox = malloc(6 * sizeof(int));`
- Requires casting before dereferencing for read / write
  - `*(int *)pv; // use pv as an int *`
- NULL, a special pointer value useful for initializations, error handling
  - `#define NULL ((void*) 0)`

Can’t always get what you want

- A call to malloc() may fail
  - for ex. when out of memory
- In this case you get back a NULL value

    - Not much to do except report the error (and terminate nicely)
- Idiom

```
char*p = malloc(100); // request 100 bytes
if (p == NULL) {
      // report error and finihs
      perror("Not enough memory");
      exit(1);
}
```

How much Space?

- You need to tell malloc() how many bytes you need
- To request space for an array, need
    - # of elements
    - amount of space for each array element
        - sizeof(t) returns # of bytes needed for a value for type T
- ex.):

```
int* pox = malloc(6 * sizeof(int)); // request space for 6 ints
if (pox == NULL)
        report error and finish;
```

Another Way

- `#include <stdlib.h>`
- `void* calloc(size_t, nmemb, size_t size);`
- calloc() is implemented in terms of malloc()
    - calloc() also initializes the content to 0

```
int* pox = calloc(6, sizeof(int)); // request space for 6 ints
if (pox == NULL)
      report error and finish;
```

Adjusting the Size

- `#include <stdlib.h>`
- `void * realloc(void* ptr, size_t size);`
- What if you change your mind?
    - you requested 100 bytes, but now need 200

```
char* p = malloc(100); // request 100 bytes
…
p = realloc(p, 200); // p may change!
```

- Before a call to realloc(p, size), p must be
    - A pointer returned by a previous malloc/calloc/realloc

- or NULL, in which case the call is equivalent to malloc(size)

Deallocation

- `#include <stdlib.h>`
- `void free(void *ptr);`
- Straightforward
  - simply call the lib. func. “free”
  - takes a pointer to the block to free

```
free(ptr);
```

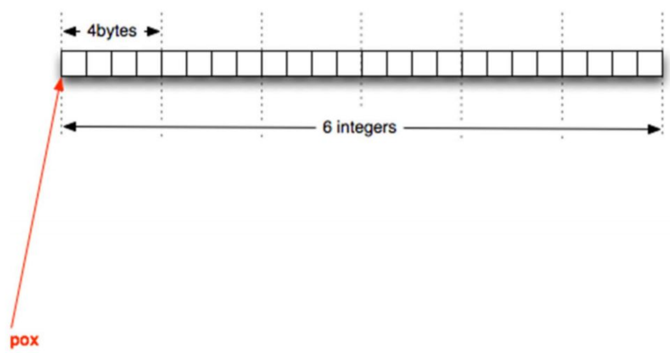
- Do not free a pointer twice!
  - After freeing a pointer, set it to NULL!

Two key rules

- Rule 1: Everything you request should be freed, eventually
- Rule 2: Only free what is allocated via malloc/calloc/realloc/
- Consequences of not following the rules
  - Memory “leaks”
    - Your program will eventually run out of memory
  - Undefined behavior and horrible crashes
    - Freeing unallocated mem. or already freed mem.
    - May cause a memory error and program crash
    - Wrose, may corrupt heap and cause crash later
    - Even worse, program may keep running, totally corrupting your data, without you knowing

Pointers and arrays

- `after pox = malloc(6 * sizeof(int))`
- That looks like an array
  - and you can use pox as if it is



Example: Use pointer as array

```
// programmers have to manage memory themselves before C99
void doSomething(int n){
      int * pox;
      pox = malloc(sizeof(int)*);      // request mem from heap…
      *pox = 0;        // set the int at the address pox to 0
      pox[0] = 0;      // same thing
```

```
    pox[1];          // the int after pox[0]
    // more lines...
    free(pox);       // remember to free!
}
```

Array of pointers

- Pointers, like integers, can be placed in an array

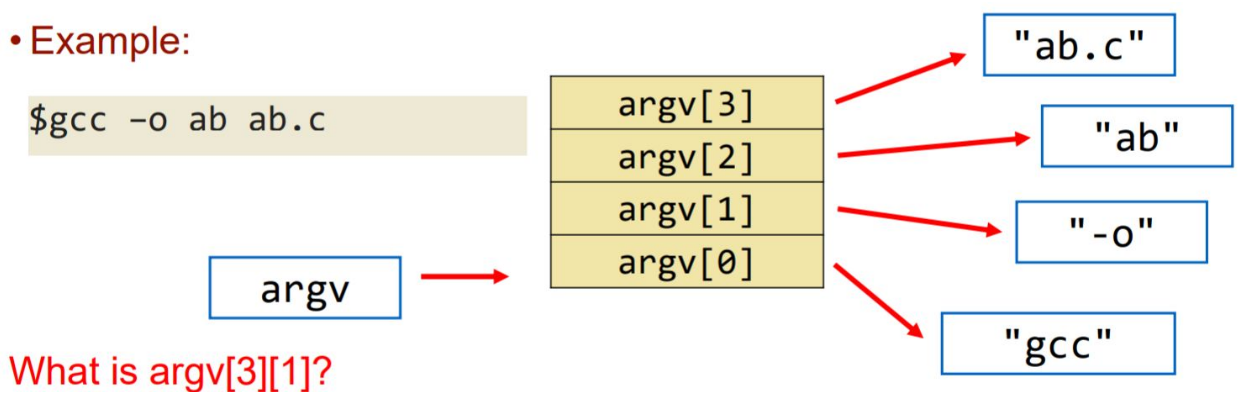
```
int  a0;
int a1;
int a2;


char *p0 = malloc(10);
char *p1 = malloc(10);
char *p2 = malloc(10);
```

```
// array of int's
int  a[3];

// array of pointers
char * p[3];

// Can also do with a loop
p[0] = malloc(10);
p[1] = malloc(10);
p[2] = malloc(10);
```

Example: Command line argument

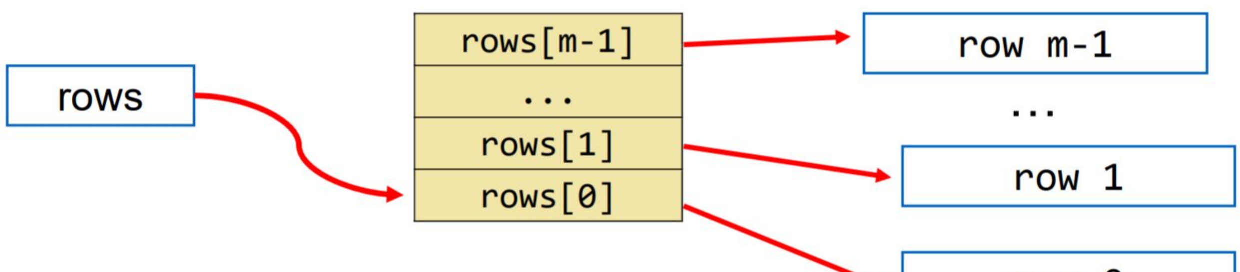
`int main (int argc, char *argv[]);`

- argc: the # of arguments on the command line
- argv: array of pointers to characters
  - the # of elements is argc
  - each element in an array points to null-terminated strings



Example: allocating 2d dynamical array

```
void doSomething(int m, int n)
{ int **rows;
  rows = malloc(sizeof(int *) * m);  // array of pointers
  for (int i = 0; i < m; i++)
    rows[i] = malloc(sizeof(int)*n); // one int array for each row
  ...
  for (int i = 0; i < m; i++)
    free(rows[i]);
  free(rows);
}
```

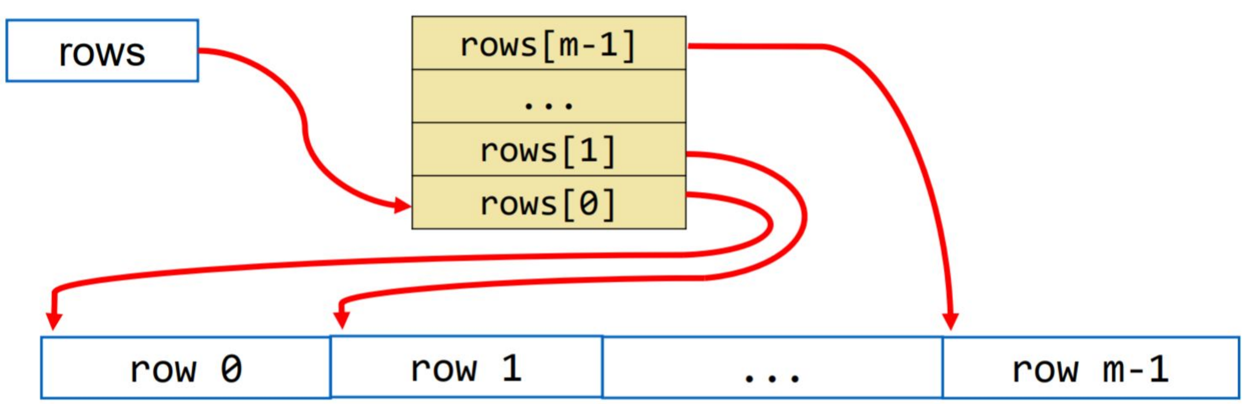


Looks similar to argv?

18

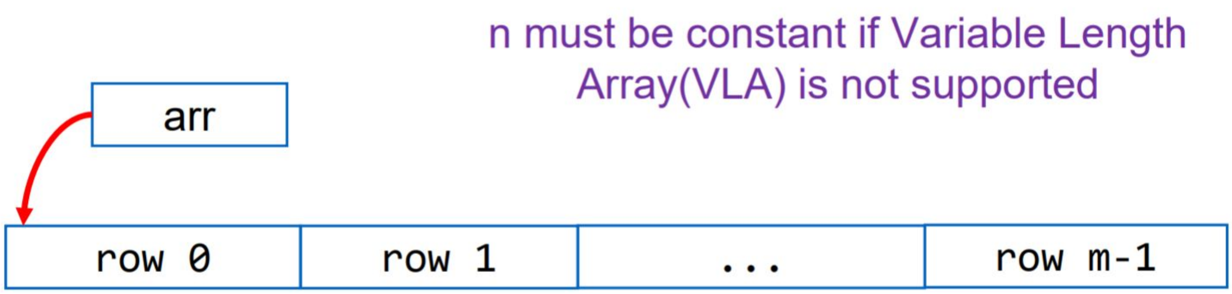
Example: allocating 2d dynamical array (take 2)

- Requesting all the memory space needed by data with one malloc() all
  - Instead of call malloc() m times, for each row
- Calculate rows[1], rows[2], and so on (pointer arithmetic! next lecture)
  - You can even calculate the address of each element (e.g. rows[10][5])



Example: allocating 2d dynamical array (take 3)

```
void doSomething(int m, int n)
{     int(* arr)[n];  // arr is a pointer to an array of n int's
      arr = malloc(m * n * sizeof(int));    // one malloc() for data
to access
      arr[1][2] = 1;
      arr[2][3] = arr[1][2] + 10;
      …
      free(arr);  // free memory
}
```



Pointers taking the address of…

- A static?
  - The address is never going to go "bad"
  - The static lives as long as the program!
- A stack [automatic] variable?
  - The address is valid as long as the variable!
  - When the function returns… The address is bogus
- A heap variable?
  - The address is valid as long as the variable is!
  - The variable disappears when explicitly de-allocated (freed)

Lecture 11 - C7: Pointer Arithmetic & Structures

Mon. Sept. 23, 2019

Pointers are addresses

- The value of a pointer is a byte address
    - Unsigned integer used to number bytes in memory
    - Range is between
        - 0x00000000 and 0xFFFFFFFF [32-bit]
        - 0x0000000000000000 and 0xFFFFFFFFFFFFFFFF [64-bit]
- Corollary
    - If a pointer is an integer, you can do arithmetic
    - To computer addresses

Pointer Addition Example

- Suppose p is a pointer to an int, and its value is 10000
- p + 1 is not the next byte address
- It is the address of next item (of type int)

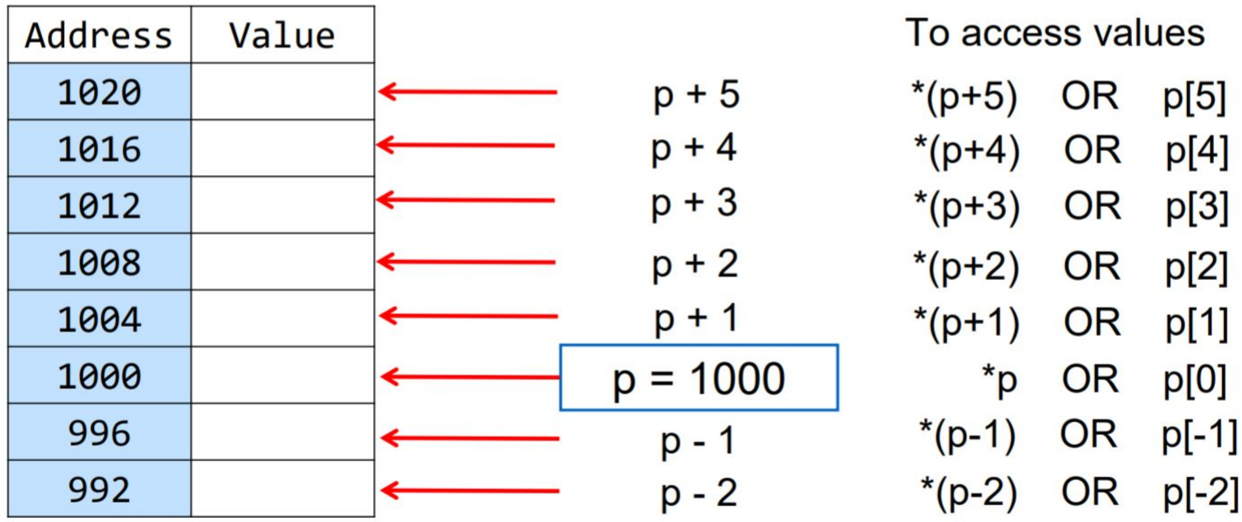
Adding a pointer and an integer

- Add an integer to a pointer, the result is a pointer of the same type
    - It is dif. from reg. integer addition
    - The integer is **automatically scaled** by the size of the type pointed to
- Suppose p is a pointer to type T, and k is an integer
    - Then both “p + k” and “k + p”
    - Are valid expressions that evaluate to a pointer to type T
    - Have a byte address equal to

```
(unsigned long)(address stored in p) + k * sizeof(T)
```

C standard does not allow arithmetic on void *

gcc has an extension, treating sizeof(void) as 1



Pointers Subtraction

- Subtract one pointer from another: both must have the same type
- The result is the number of data items between the two pointers!
    - Not the number of bytes

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

int main() {
   int *p = malloc(sizeof(int)*10);
   int *last = p + 9;
   int  dist = last - p; // both are int *
   printf("Distance is %d\n",dist);
  free(p);
  return 0;
}
```

Output

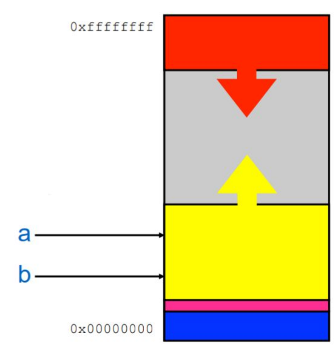
```
$ gcc ptrsub.c
$ ./a.out
Distance is 9
$
```

Pointer comparison

- You can also compare two pointers
- Purpose
    - Check boundary conditions in arrays
    - Manually manage memory blocks
- Semantics
    - Simply based on memory layout!
    - Compare bits in pointers as unsigned integers!

Check if you are done:

```c
while (b < a) {
    // do something
}
```



Effects of casting types?

- If you cast a pointer type…
    - Any subsequent pointer arithmetic will use the type you choose
- Do not wanting scaling? Casting a pointer to (char *)
    - Because sizeof(char) is 1

```
int * t;
char * p = (char *) t + 8;   // 8 is not scaled
char * q = (char *) (t + 8); // 8 is scaled
```

Arrays and pointers

- Arrays and pointers can often be used interchangeably

```
int  a[10];
int *p = a;

// all of the following evaluate to the value of a[0]
*p              p[0]        *a            a[0]

// all of the following evaluate to the value of a[1]
*(p+1)          p[1]        *(a+1)        a[1]

// all of the following evaluate to the address of a[0]
// type is int *
p               &p[0]       a             &a[0]
```

"array of int" becomes "pointer to int" (array decay)

Example: arrays and pointers

- Equivalent ways of initializing an array

```c
int a[10], *p = a;  // not *p = a; it is int *p; p = a;

for(int i=0; i<10; i++) a[i] = i;  // array indexing

for(int i=0; i<10; i++) p[i] = i;  // indexing via pointer

for(int i=0; i<10; i++) *(p+i) = i; // explicit pointer arithmetic

for(i=0; i<10; i++) i[p] = i;   // obfuscated but valid C!

for(i=0; i<10; i++) *p++ = i;   // common pointer use idiom
```

```c
int * tp = p;
p ++;
*tp = i;
```

Arrays and pointers are NOT the same

```c
int   a[10];
int   *p = a; // a is converted to int *

// a is still an array after &
&a   // pointer to array of 10 int's  int (*)[10]
&p   // pointer to a pointer to int   int **

// a is still an array after sizeof
sizeof(a) // 40 because a is an array of 10 int's
sizeof(p) // 8  because p is a pointer

p++;   // can increment p
a++;   // cannot increment a; this will not compile
       // Similar to n++ vs 2++
```

Example: Arrays and pointers are NOT the same

```
#include <stdio.h>

void foo(int *x)
{
   printf("%lu\n", sizeof(x));
}

int main()
{
   int a[10], *p;

   p = a;  // a is converted to int *
   // a is still an array in sizeof
   printf("%lu %lu\n", sizeof(a), sizeof(p));
   foo(a); // a is converted to int *
}
```

Output

```
% gcc array.c
% ./a.out
40 8
8
```

Structures

- Mechanism to define new types
  - Also known as “compound types”
  - Use to aggregate related variables of different types
- Structures
  - Structures can have a type name
  - Can have “members” of any types
    - Basic types
    - Pointer
    - Arrays
    - Other structures
- Structure variable definition
  - Specifies variable name
  -

```
struct student_grade {
    char *name;
    int  id;
    char grade[3];
};
…
struct student_grade student1;
struct student_grade
    student2, student3;
struct student_grade all[200];
```

Structure example

```c
struct Person {
   int      age;
   char gender;
};

int main(){
   struct Person p;

   p.age = 44;
   p.gender = 'M';

   struct Person q = {44, 'M'};

   return 0;
}
```

Structure *type declaration*

Structure *variable definition*

Syntax for field access similar to Java and Python

Structure *variable definition* and *initialization*

Example: Array of Structures

- Member name is a char array
- Cabeats
    - Names cannot be more than 31 characters long
    - Four person in family
    - Indexed 0..3
- Array of structures for the whole family
- Nested initializers

```c
#include <stdlib.h>

struct Person {
   char   name[32];
   int    age;
   char   gender;
};

int main()
{
   struct Person family[4] = {
      {"Alice",34,'F'},
      {"Bob",40,'M'},
      {"Charles",15,'M'},
      {"David",13,'M'}
   };
   int juniorAge = family[3].age;
   return 0;
}
```

typedef

- Struct. names can be long
- C provide the ability to define type abbreviations
    - typdef declaration
        - Give existing type new type name
- Make code more readable
- Structure and typedef declarations often combined

```c
struct Person {
   char   name[32];
   int    age;
   char   gender;
};
typedef struct Person TPerson;
int main()
{
    TPerson family[4];
    ...
    return 0;
}
```

```c
typedef struct Person {
   char   name[32];
   int    age;
   char   gender;
} TPerson;
```

Operations on struct

- Assignment
    - All struct members copied
- Can be passed to funcs.
- If pass by value, cannot change members in funcs.
- Passing or returning large structures can be costly

***use pointers to structures!

```
TPerson a, b, c;

…
a = b;
c = searchPerson("name");
```

Pass Structure by reference

```
typedef struct Person{
      char  name[32];
      int   age;
      char  gender;
}

TPerson * init_Person(TPerson *p, char * name, int age, char
gender){
      strcpy(p->name, name);      // (*p).name
      p->age = age;               // (*p).age
      p->gender = gender;         // (*p).gender
      return p;
}
// Study the demo code is in the demo repo
// especially the lines that copy name
```

Structure Alignment

- Structure members are aligned for the natural types

Alignment requirements on x64 architecture	
char	1
short	2
int	4
long	8
float	4
double	8

```
struct struct_random {
    char    x[5]; // bytes 0-4
    int     y;    // bytes 8-11
    double  z;    // bytes 16-23
    char    c;    // byte 24
};      // Total size 32
```

```
struct struct_sorted {
    double  z;    // bytes 0 - 7
    int     y;    // bytes 8 - 11
    char    x[5]; // bytes 12 - 16
    char    c;    // byte  17
};       // Total size 24
```

18

Arrays and pointers

- Copying arrays

```
// using array indexing
void copy_array0(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    target[i] = source[i];
}

// using pointers
void copy_array1(int source[], int target[], int n)
{
  for(int i = 0; i < n; i++)
    *target++ = *source++;
}
```

```
int * tp = source;
source ++;
*target = *tp;
target ++;
```

Typecasting Pointers

- C lets you cast pointers in any way you like
- You can “forge” pointers to point wherever you wish…

```
float f;
char p = * (char *) &f;          // 1st byte representing f
char ch = * (char *) 1000; // access any byte in memory
```

- THAT’S WHAT MAKES C VERY ATTRACTIVE FOR LOW-LEVEL PROGRAMMING
- THAT IS ALSO VERY POWERFUL AND THUS DANGEROUS

Returning more than one value from functions

- Use references (caller prepares the storage)

```
long int strtol (const char* str, char** endptr, int base);
```

- Use arrays (caller prepares the storage)

```
int pipe(int pipefd[2]);
```

- Return a structure (costly if structure is large)
- Return a pointer (to array or structure)
  - Must be dynamically allocated or static. RTM (read the manual)!

```
char strdup(const char *str1);
```

```
struct tm *localtime( const time_t *time );
```

- Use global variables (DON'T)!

```
errno
```

Typedef

```
// Example of typedef. Think about how you would define a variable
typedef    int   BOOL;
typedef    char  name_t[100];
typedef    char  *Pointer;
```

Self-referential structures

```
struct Person{
     int   age;
     char  gender;
     char  name[32];
     struct     Person * parents;     // A pointer to this type of
struct
}    person1, person2;           // Can define variables here
```

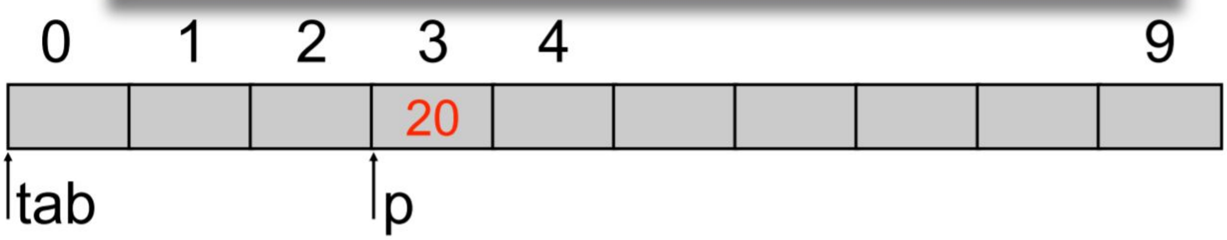
Self-referential structures - 2

```
struct student{
     char  name[128];
     // Can have a pointer to a struct defined later.
     // However, you cannot define an array of book here (e.g.
books[8])
     struct book * books;
};
struct book {
     char  title[128];
     struct student * owner;l
     struct book * next;   // A pointer to this type of struct
};
```

Example of Pointer Arithmetic

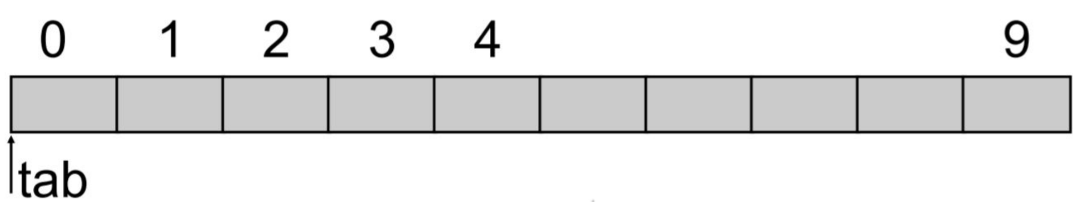
- Simple illustration

```c
#include <stdlib.h>

int main()
{
   int *tab = (int*)malloc(sizeof(int)*10);
   tab[3]    = 10;
   int *p    = tab + 3;
   printf(“What is at tab+3? = %d\n”,*p);
   *p = 20;
   printf(“What is at tab[3]? = %d\n”,tab[3]);
   return 0;
}
```



But what about memory addresses?

- Same story…!



0x11110000

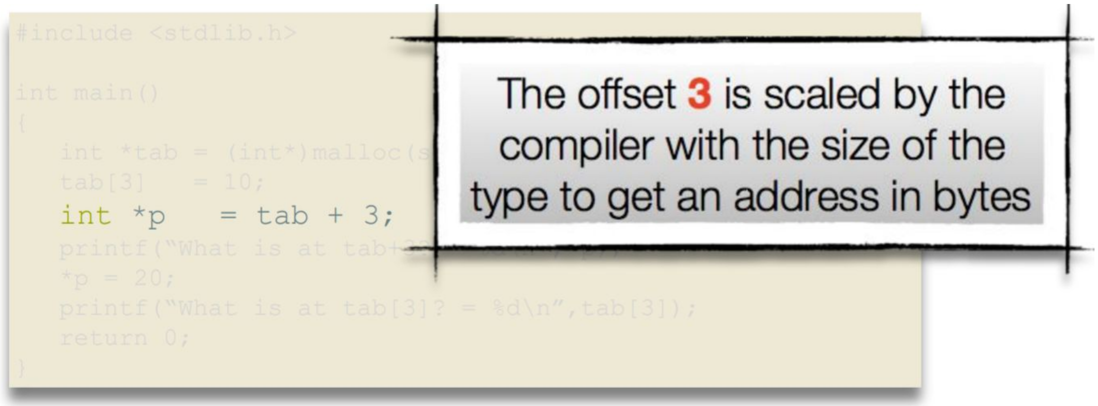
tab+1 = ?

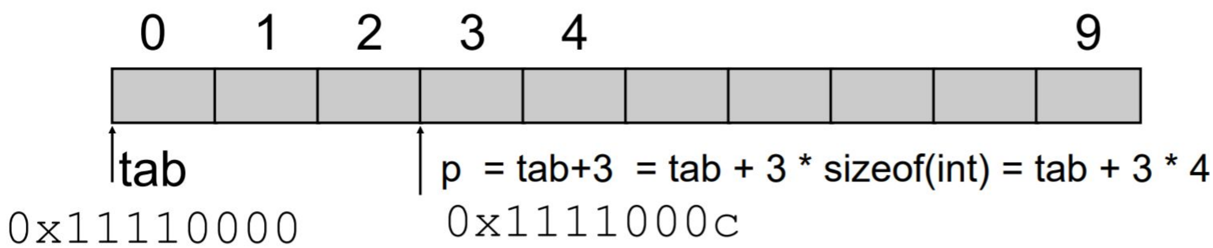
~~0x11110001 ?~~

0x11110004 ?

WHY?
Simply because tab is a pointer to an int and an int is 4-bytes wide!

Bottom Line





Memory Alignment Requirements

- Memory used to store a value of type X MUST
    - be lined-up on a multiple of natural alignment for X
- Why?
    - Performance!
- If you do not respect alignment requirements…
    - BUS ERROR (sigbus)
    - The O.S. will kill your program

Good News and Bad News

- The C compiler handles alignment 99% of the time
- Programmers have to handle the rest - when you do pointer arithmetic of course!
    - Do not assume the location of the fields
- When you call sys. routines w/ specific alignment needs
    - Your arguments must comply
    - Use compiler annotations to force specific alignment (beyond our scope, simply remember that this exists!)

Example

```
#include <stdio.h>
struct Person {
    char   name[32];
    int    age;
    char   gender;
};
typedef struct Person TPerson;

TPerson init(char name[], int age, char gender) {
  TPerson p;
  int i;
  for(i = 0; name[i]>0; i++)
     p.name[i] = name[i];
  p.name[i] = '\0';
  p.age = age;
  p.gender = gender;
  return p;
}

void print_info(TPerson p) {
  printf("name: %s, age: %d, gender: %c\n",
         p.name, p.age, p.gender);
}
```

```
int main() {
    TPerson family[4];
    family[0] = init("Alice",34,'F');
    family[1] = init("Bob",40,'M');
    family[2] = init("Charles",15,'M');
    family[3] = init("David",13,'M');
    print_info(family[0]);
    print_info(family[1]);
    print_info(family[2]);
    print_info(family[3]);
    family[1] = family[0];
    print_info(family[1]);
    return 0;
}
```

Output

```
./a.out
name: Alice, age: 34, gender: F
name: Bob, age: 40, gender: M
name: Charles, age: 15, gender: M
name: David, age: 13, gender: M
name: Alice, age: 34, gender: F
```

35

Lecture 12 - C8: Linked Lists, Enums, Func. Ptrs.

Weds. Sept. 25, 2019

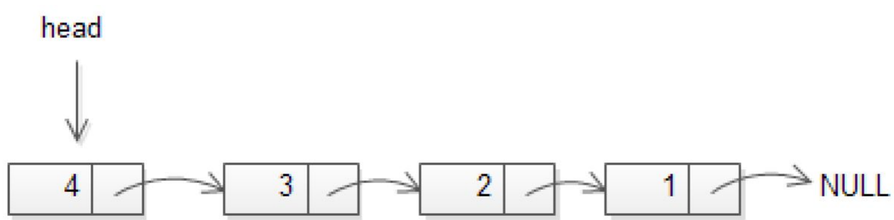
Example: Linked List

```
// a data structure that consists of a chain of nodes
// Starting from head, a node has a reference to the next node
typedef struct node_tag{
      int   v;                  // data
      struct node_tag * next;     // A pointer to this type of struct
} node;                           // Define a type. Easier to use.
```

Head

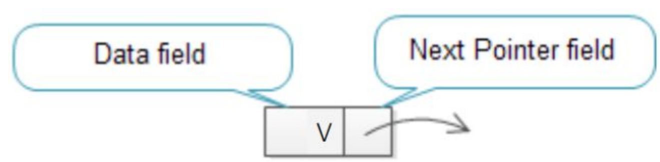
```
node * head;            // head is a pointer, not a node!
head = NULL;            // at beginning, it is empty
```

- After adding nodes into the list,



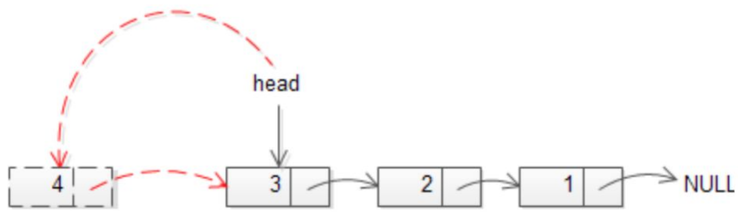
Create a node

```
node* new_node(int v) // create a node for value v
{
      node * p = malloc(sizeof(node)); // allocate memory
      assert(p != NULL);     // you can be nicer

      // Set the value in the node.
      p->v = v;  // you could do (*p).v
      p->next = NULL;
      return p;  // return
}
// is it similar to creating objects using "new"?
```



Prepend

```
node * prepend(node * head, node * newnode)
{
      // how?
}
```



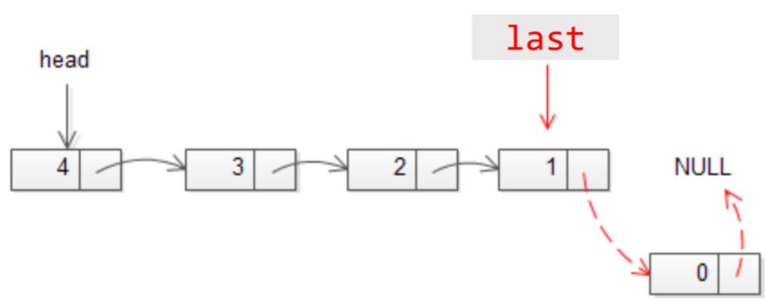
- cannot perform the following or else the head will point to the 1st node, and the 1st node points back to the head

Find the last one

```
node * find_last(node * head)
{
      if (head != NULL);
}
```

Append

```
node * append(node * head, node * newnode)
{
      node * last - find_last(head);    // find the last one
      if(last == NULL)        // if the list is empty, new node is the
head
            return newnode;
      last -> next = newnode;
      newnode -> next = NULL;
      return head;                 // return the (unchanged) head
}
```



Enumeration types (ABC 7.5)

- User-defined integer-like types:
- Names look like C identifiers
    - are listed (enumerated) in definition
    - treated as integer

Enumeration types

```
// enum start from 0 by default
enum week {Sun, Mon, Tue, Wed, Thur, Fri, Sat};
enum week dow = Mon;

// But can be initialized; Warning is 2, Error is 3, etc.
enum status {OK = 1, Warning, Error, Fatal};
```

Type qualifier: const

```
// constant int
const int  a = 10;          // cannot change a
// a pointer to constant int
const int  *pa = &a;  // can change pa, but not *pa
// a constant pointer to an int
int * const pb = &b         // can change *pb, but not pb
// constant pointer to a constant int
const int * cont pc = &a;   // cannot change *pc or pc

// cannot change the source string
char * strcpy(char * dest, const char* src);
```

Function pointers

```
/* function returning */
int func();
/* function returning to integer */
int * func();
/* pointer to function returning integer */
int (*func)();
/* pointer to function returning pointer to int */
int * (*func)();
```

Pointer to function example

```
int   mymax(int a,int b)
{
      return (a > b) ? a : b
}
// a pointer to function
int (*pf)(int a, int b);

// assign a value to the pointer
pf = mymax;           // C99 style. Note that it is not mymax()
pf(3, 5);
pf = & mymax;
(*pf)(3,5);
```

Use of function pointers

- Call-back mechanism
  - Generic functions (ex. coming next)
  - pthread_create()
  - Dynamic signal handlers, ….
- You can store function pointers in arrays
  - And arrays stored in structures!
  - And you can simulate objects in Object Oriented Languages!

Example: quicksort in C library

- The prototype (in <stdlib.h>)

```
void qsort(void * base
           size_t nel,
           size_t width,
           int (*compare)(const void *, const void *));
```

- qsort takes…
  - base:
  - nel:
  - width:
  - compare: a pointer to a function that compares two values

****sort only knows*

Why passing a function to qsort?

- Need to tell qsort() how to compare items in the array

  - we have a generic quickSort implementation
    - Do not want to implement one for each type of data
- The qsort() implementation calls the comparator to rank elements

```
int (*comprae)(const void *a, const voi *b);
```

  - the function takes the address of 2 items to be compared,
  - and returns:
    - 0 if *a EQUALS *b
    - a positive value if *a is GREATER THAN *b
    - A negative value if *a is LESS THAN *b

Example of compare() function

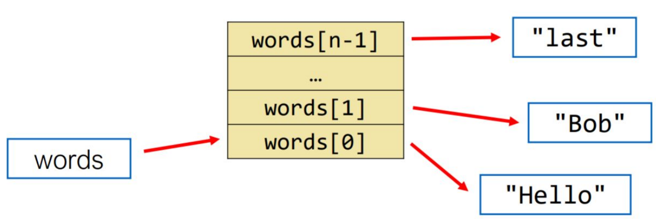
- when qsort() compares items, it provides the address of two elements to be compared.

```
int compare_int(const void *a, const void *b)
{     // qsort( does not know the type, but you know
      return *(int * a) a - *(int *)b;
}
int compare_double(const void (a, constn void *b)
{     // qsort( does not know the type, but you know
      return
}
```

Example: sort array of strings

- Example are pointers to strings
  - Need to compare string, instead of pointer

```
int compare_string(const void *a, const void *b){
// how to compare *a and *b?
// for example, a is &words[0] and b is &word[1]
```



Compare string pointers

- An element in array words is (char *).
- a is the address of an element of type (char *). So, a's type is (char*)*

```
int compare_string(const void * a,const void * b)
{
   char *s1, *s2;
   s1 = *(char **)a;     s2 = *(char **)b;

   return strcmp(s1, s2);  // use library function to compare
}

// or on one line
int compare_string(const void * a,const void * b)
{
   return strcmp(*(char**)a,*(char**)b);
}
```

Calling quicksort()

- See complete demo code in the demo repo

```
int compare_string(const void* a,const void* b)
{
   return strcmp(*(char**)a,*(char**)b);
}

int some_function(void)
{
   …
   char** words = malloc(sizeof(char*)*n);
   …
   qsort(words,n,sizeof(char*),compare_string);
   …
}
```

Type casting to char ** before dereferencing

Lecture 13 C8: I/O and Files

Fri. Sept. 27, 2019

CSE 3100 Master Notes

errno

- Most C library functions can “fail”
  - When they do, they return a flag reporting failure… (-1)
  - Some set of global variable to report the exact error code

errno

// to use errno, include <errno.h>

- Check manual page to interpret the error code
- Print a more descriptive message with perror()
- Avoid functions that set errno in multithreaded code
  - Prefer thread-safe version when available

```
void perror(const char *str);
```

Files and directories

- A file is an object that stores information, data, etc.
-
- Example: files you create with an editor (.c, .h, Makefile, readme, etc.) executable generated by the compiler, and gcc itself other devices, like screen, keyboard, …
- In Linux, files are organized in directories
  - A directory can have subdirectories and files
  - The top directory is /
- A path specifies the location of file/directory in the file system

/home/john

- In Unix/Linux, everything is a file

The stdio library

#include <stdio.h>

- Declares FILE type and function prototypes
  - FILE is an opaque type (system dependent) for operating on files
    - It is a structure, but do not try to change it directly!
  - Use library functions to access FILE objects, via pointers (FILE *)
- Defines “standard” streams stdin, stdout, stderr
  - Created automatically when program starts
  - They are files!
- The library is linked automatically by the compiler

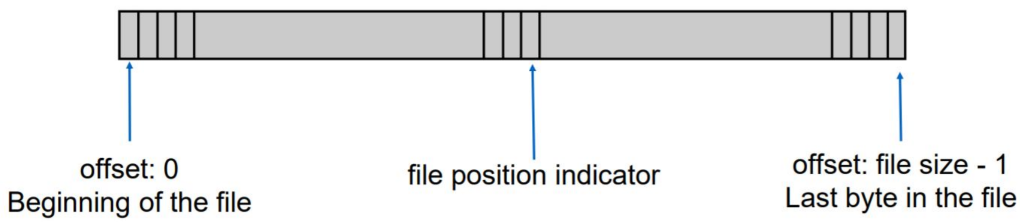
Files and I/O API

- In C, a file is simply a sequential stream of bytes
- The “f” family of functions (fopen, fclose, fread, fgetc, fscanf, fprintf,…) are C library functions to operate on files

- All these use a FILE* abstraction to represent a file
- The C library provides buffering
    - That's why sometimes you do not see output of printf immediately We will learn another set of functions provided by OS

File as stream of bytes

- Before use a file must be “open”
    - This sets a position indicator for reading and/or writing
- Each read/write starts from current position, and moves the indicator
    - Writing after last byte increases the file size
- Position indicator can also be changed with fseek
- All open files are closed when program ends
    - Good practice to close explicitly when no longer needed



Opening Streams

FILE *fopen(const char *filename, const char *mode)

- Open the file filename in mode as a stream of bytes
- Returns a pointer to FILE (FILE *) or NULL (and errno is set)
- Mode
    - “r” : Reading mode
    - “r+” : Read and write
    - “w” : Writing mode, file is created or truncated to zero length
    - “w+” : Read and write, but the file is created or truncated
    - “a” : Append mode, the file is created if it does not exist
    - “a+” : Read and append, the files is created if ti does not exist. Reading starts at the beginning, but writing done at the end

Closing Streams

int fclose (FILE *stream)

- Close a stream
- Returns
    - 0 if it worked
    - EOF if there was a problem (and errno is set)

fgetc / fputc (one byte at a time

```
int fgetc( FILE *stream);
int fputc(int c, FILE *stream):
```

- Read or write one (ASCII_ character (8-bits) at a time
  - Can be slow for large files
- fgetc reads a character from the stream and returns the character just read in (as unsigned char extended to int)
  - Returns EOF when at the end of file or on error
- fputc writes the character received as argument to the stream and returns the character that was just written out
  - Returns EOF on error

getc / putc and ungetc

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
```

- Same as fgetc/fputc except they may be implemented as macros
- Use fgetc/fputc unless you have strong reasons not to

```
int ungetc(int c, FILE *stream);
```

- Pushes last read char back to stream, where it is available for subsequent read operations
- Only one pushback guaranteed

getchar / putchar

```
int getchar(void)
// same as fgetc(stdin)
```

- Reads a character from stdin
- Returns the character just read in, or EOF on end-of-file errors

```
int purchat(int c)
// same as fputc(c, stdout)
```

- Writes the character received as argument on stdout
- Returns the character that was just written out, or EOF on errors

More than one byte: get a line

```
char *fgets(char *buf, in size, FILE *in)
```

- Reads the next line from in into buffer buf
- Halts at '\n' or after size-1 characters have been read
  - NUL is placed at the end
- Returns pointer to buf if ok, NULL otherwise
- Do not use gets(char *)! – buffer overflow

```
int fputs(const char *str, FILE *out)
```
- Writes the string str to out, stopping at '\0'
- Returns number of characters written or EOF

Formatted output

```
int fscanf(FILE *stream, const char *format, …);
int fprintf(FILE *stream, const char *format, …):
```
- Formatted input from file and output to file
- Like scanf() / printf(), but not from stdin or to stdout

For binary data

```
size_t fread (void *ptr, size_t sz, size_t n, FILE *stream);
size_t fwrite(void *ptr, size_t sz, size_t n, FILE *stream);
```
- Read / write a sequence of byte from / to a stream
- Return the number of items read or written
    - If smaller than n, EOF or error

```
Example:
int A[10][20];
size_t n = 10 * 20;
if (fwrite(A, sizeof(int), n, fp) != n) {
      // error
```

Moving file position indicator

```
long ftell(FILE *stream);
```
- Read file position indicator
- Return -1 on error

```
int fseek(FILE * stream, off_t offset, int whence);
```
- Set the file position indicator
- Return 0 on success and -1 on error

```
Example:
fseek(fp, 0, SEEK_SET);          // move to the beginning
fseek(fp, 200, SEEK_CUR);        // move forward 200 bytes
fseek(fp, -1, SEEK_END);          // move to the last byte
```

More useful stdio functions

```
//Check if end-of-file is set (after a read attempt!)
int feof(FILE * stream);

//Force write of buffered data
```

```
int fflush(FILE * stream);
```

- Read the manual pages!
- Check the return values!

Lecture 14

Mon. Sept. 30, 2019

Lecture 15

Weds. Oct. 2, 2019

Lecture 16

Fri. Oct. 4, 2019