+

Dynamic Allocation *con't*

Heap memory review

- Things allocated in heap memory…
  - Structures whose size varies dynamically
    - e.g. length of arrays or strings decided/modified at runtime.
  - Structures that are allocated dynamically
    - e.g. records in a linked list.
  - Structures created by a function that must *survive after the call returns*.
- Is not managed for you the way the stack and static regions are.

malloc(), sizeof() & free()

- There are often used together to size an allocation

```
int* pointer = malloc(10 * sizeof(int));
// do pointer things...
free(pointer);
```

- See memory_management/malloc_sizeof_free.c

free() problems

- Once pointer to memory is lost, no way to free it.
- Failure to call free() on heap allocated memory causes a **memory leak**.
  - Details on the consequences of memory leaks can be found here https://en.wikipedia.org/wiki/Memory_leak#Consequences
  - Easy to understand.. worth a quick read.
- See memory_mngmt/memory_leak.c

free() problems *con't*

- Variables pointing at already free'd areas are know as "dangling pointers" (sometimes called ("dangling references")
- Attempts to read from the pointer may still return the correct value for a while after calling free, but allocations thereafter may overwrite the storage allocated with other values and the pointer would no longer work correctly.
- See memory_mngmt/dangling_pointers.c

free() problems *con't*

- Trying to free the same area twice will generate an error aka "double delete" problem
- Take care not to release pointer to middle of allocated region.

```
int* middle = &some_int_array[3];
free(middle);
```

- Take care not free automatic (local) variables
- See memory_mngmt/free_problems.c

calloc()

- **void* calloc(int count, int size_in_bytes);**
- Similar to malloc with two differences…
  - **malloc**() does not initialize the memory allocated, while **calloc**() guarantees that *all bytes of the allocated memory block have been initialized to zero*.
  - **malloc**() takes a single argument, while **calloc**() needs two arguments..

```
int* p1 = malloc(10 * sizeof (int));
// is logically equivalent to ...
int* p2 = calloc(10, sizeof (int));
```

  - See memory_management/malloc_vs_calloc.c

realloc()

- **void* realloc(void* ptr, int new_size_in_bytes);**
  - attempts to resize the region of memory to the ‘new_size’
  - attempts to do this “in place”…
    - for reductions this is no problem
    - for allocations, there may not be contiguous available memory
  - for the second case, the pointer returned will be different from the ‘ptr’ argument, and ‘ptr’ will be a ‘wild pointer’
  - if there program is OOM, the NULL pointer is returned.
- See memory_management/realloc.c

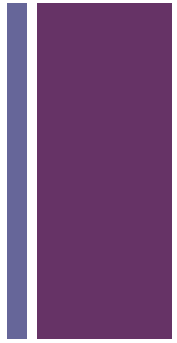
WARNING!

- The C dynamic memory allocation functions are defined in stdlib.h header
- You **must** include stdlib.h in order to use these functions

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

- If you do not you may get a version of malloc that will not work properly for 64-bit systems.
  - *"warning: incompatible implicit declaration of built-in function 'malloc' [enabled by default]"*
- See memory_management/forgotten_stdlib.c

Linked List



- We can demonstrate a number of the concepts we've covered today by looking at and example of a simple linked list implementation with only two operations.
- See memory_management/list.c

I/O

Library support

- In C, a library of functions and structs it aid in I/O operations.
  - The I/O library capabilities are listed the header file **<stdio.h>**
- These functions provide the abilities to do the obvious things, such as..
  - Reading from…
    - Standard input
    - Files
  - Writing to…
    - Standard output (via **printf**())
    - Files

Format specifiers review

- As you may recall, with **printf()** the type of the value to be printed must be matched with the correct *format specifier*.
  - This is ***also true when taking input from the console.***
- Format conversion specifiers
  - %d — displays a decimal integer
  - %p — displays a pointer address value
  - %f — displays a floating point value
  - %x — displays a number hexadecimal format
  - %c — displays a single character
  - %s — displays a string of characters
  - ……

Reading from stdin

- **void scanf("format", &var1, &var2, …, &varN);**
  - The "format" is a listing of the data types of the variables to be input
  - Each variable is a place to store the values read from stdin
  - Note the use of the addressOf operator before each variable name.
    - Another use of pointers effectively, telling **scanf()** *where* to store the value.
- See io/scanf.c

Files

- In C, each file is simply a sequential stream of bytes. C imposes no structure on a file.
- A file must first be **opened properly** before it can be accessed for reading or writing.
  - Successfully opening a file returns a pointer to a FILE struct.
  - You must check for success manually.
- When you are done with the file, you must **close the file.**

Declaring files

- Declaring a file in your C program looks something like this..

```
FILE *fptr1, *fptr2;
```

- The above *declares* that **fptr1** and **fptr2** are pointers to type FILE.
  - FILE is a struct in stdio.h.
- At this point they are uninitialized.
- What might happen if I tried to read from the file at this point?

Opening Files

- After declaration, to read from or write to a file, you must open the file and assign the address of its file descriptor to the file pointer variable.
  - A file descriptor is an OS abstraction used by a program to access a file or other i/o resource (such as a network connection).
- The following statement would open the file *mydata* for input.

```
fptr1 = fopen ( "mydata", "r" ) ;
```

- Note that second parameter “r”. This means this FILE is “read-only”.

Opening Files *con't*

- The following statement would open the file *results* for output.

```
fptr2 = fopen ("results", "w" ) ;
```

- Note that second parameter "w". This means this FILE is "writable".
- The following statement would open the file results for output, also.

```
fptr2 = fopen ("results", "a" ) ;
```

- Note that second parameter "a". This means this FILE will be "appended" to.

Opening file *con't*

- There are more 'modes'
- Note that once the files are open, they stay open until you close them or end the program (which will close all files.)

File access mode string	Meaning	Explanation	Action if file already exists	Action if file does not exist
"r"	read	Open a file for reading	read from start	failure to open
"w"	write	Create a file for writing	destroy contents	create new
"a"	append	Append to a file	write to end	create new
"r+"	read extended	Open a file for read/write	read from start	error
"w+"	write extended	Create a file for read/write	destroy contents	create new
"a+"	append extended	Open a file for read/write	write to end	create new

Testing for successful opening

- If the file was not able to be opened, then the value returned by the **fopen()** function is NULL.
- This could be because the file does not exists, or is in a different location than you are specifying.
- You must check for this yourself
- For example, let's assume that the file mydata does not exist..

```
FILE *fptr1 = fopen ( "myfile", "r") ;
if (fptr1 == NULL)
{
   printf ("File 'mydata' did not open.\n") ;
}
```

Reading From Files

- The **fscanf**() function is one way to read from a file.
- Example..

```
int a, b;
FILE* fptr1 = fopen ( "mydata", "r" );
fscanf (fptr1, "%d%d", &a, &b);
```

- The above code would read values from the file "pointed" to by **fptr1** and assign the values in it to **a** and **b**.

- **Note:** There are lots of ways to read and write from files in C, take a look… **https://en.wikipedia.org/wiki/C_file_input/output**

Reaching the end of file on read

- The end-of-file indicator informs the program when there are no more data (no more bytes) to be processed.
- There are a number of ways to test for the end-of-file condition.
- One way:

```
int current;
while(fscanf(fptr1, "%d", &current) == 1)  {
  printf("%d", current);
}
```

- Once we reach the end of the file, we must close the file by calling the file close function
  - **fclose(fptr1);**
- See io/read.c

Writing to a file

- There are a number of ways to write to files as well.
- An example:

```
int a = 5, b = 20 ;
FILE* fptr = fopen ( "results", "w" );
fprintf(fptr, "%d %d\n", a, b ) ;
```

- The **fprintf()** function write the values stored in **a** and b to the file "pointed" to by **fptr**.
- Note that we are using the “w” mode, which means that every time we right to the file, we destroy its existing contents first.
- Also, note that if the file does not exist, it will be created.
- See io/write.c