`long double`	10 bytes	16 bytes

- **Enumerated type:** values are 0, 1, 2, … by default; values can be set differently if desired
- **Variable declaration:** declarations reserve memory space (for local variables in the runtime stack); there are no default values: any uninitialized variables have ***garbage values***
- **2 main scopes in C:** block scope and file scope; file scope variable are called ***global variables***
- **Linkage:** controls whether different declarations refer to the same thing or different things
    - **No linkage:** each declaration refers to a different entity
    - **Internal linkage:** all occurrences of it inside a given file refer to the same entity, but occurrences in different files refer to different entities (i.e., a different one in each source file)
    - **External linkage:** all occurrences of it refer to the same entity (i.e., there's only 1 in the whole program)
    - **By default:** identifies with *file scope* have external linkage; identifiers with *block scope* have no linkage
    - **Changing linkage:**
        - Putting `extern` before a block scope identifier changes its linkage from none to external
        - Putting `static` before a file scope identifier changes its linkage from external to internal

Lecture 3: *Chp 3, 4, 5, 8* ✓

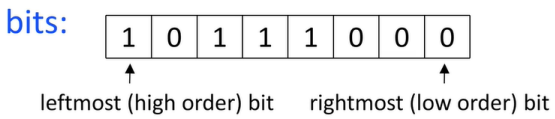
- **2 types of storage:**
    - **Automatic:** remains in memory only while it is in scope
    - **Static:** remains in memory the entire time the program is executing; global variables ALWAYS have static storage and cannot be changed
- **L-value and R-value:** left side of the equals sign is an L-value; right side of the equals sign is an R-value
- **Side effect:** an expression statement is only useful when the expression has a side effect
    - *Ex.* `y = 17;` or `x++;` or `z *= 3 + w++ - 4;`
    - *Not an example of a side effect:* `3 + w - 4;`
- **C has no boolean type:** 0 means "false" and any nonzero value means "true"
- **`assert()`:** if its argument is "false" when a call to assert is executed, the program aborts with a message; use these for testing your code!
- **Symbolic contents:**
    - **`#define`** is another preprocessor directive
    - **Syntax:** #`define` *name value*
    - **Example:** `#define NUM_STUDENTS 543`
- ***Covers array basics; see **Chapter 8** for more information.
- ***Cover pass by value and pass by reference; see **Chapter 7** ***(Function Arguments)*** for more information.
- ***Covers arithmetic operators; see **Chapter 5** ***(Types of Operators)*** for more information.

Lecture 4: *Chp 5* ✓

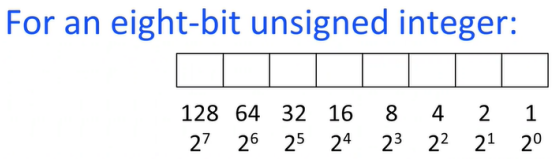
- **Conditional statements:**
  - *Ex.* `(x > y) ? x++ : y++;`
    If `(x > y)`, then `x++`. Else, `y++`
- **L-value and R-value:** left side of the equals sign is an L-value; right side of the equals sign is an R-value
- **Promotion of char and short:** if a number value is stored in a `char` or `short`, they are promoted to an `int` before the expression is evaluated
- **Arithmetic conversions:** the "smaller" type is promoted to the "higher type in this order

```
long double  ← highest type
double
float
unsigned long
long
unsigned int
int
```

- **Bit operations:**



  - **Representation of unsigned integers:** in an unsigned number, bit position i represents the value 2^i (i is 0 for the right most bit)



  - *Covers bit shift operators; see **Chapter 5 *(Bitwise & Bit Shifting)*** for more information.
- *Covers arithmetic operators; see **Chapter 5 *(Types of Operators*)** for more information.
- *Covers equality, relational, and logical operators; see **Chapter 5 *(Types of Operators*)** for more information.

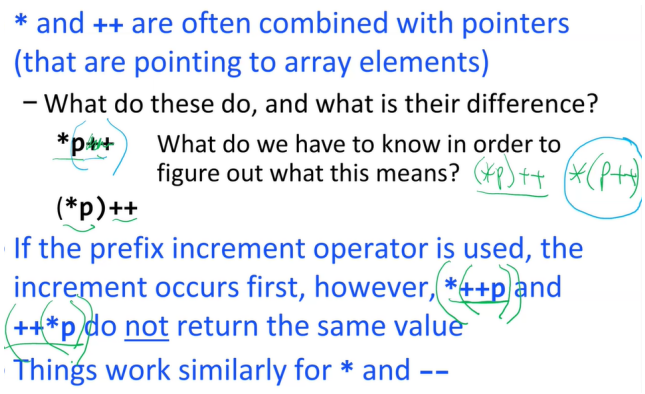
Lecture 5: *Chp 5* ✓

- **Bit masking:** using the bit operators to apply a value to an integer that will isolate some of its bits to be able to examine or modify only those bits
  - See **Chapter 5 *(Bitwise & Bit Shifting Chart*)**
- *Covers bit operators with binary numbers; see **Chapter 5 *(Bitwise & Bit Shifting*)** for more information.
- *Covers Worksheet 1.

Lecture 6: *Chp 5, 6* ✓

- **Mixed type assignments:** the value on the right side of an assignment (oritialization) is converted to the type of the L-value on the left before being stored (if their types are different)
  - For variables with values outside their type, the value wraps to fit
- **NULL pointer:** a pointer value that doesn't point anywhere
  - It's defined in a library header ***`stddef.h`***, but can be used if you include any of the library header files

- **Pointers and pointer arithmetic:**
  - A function's return type can be a *pointer*
  - Pointers can point to array values
  - Pointers of the same type can be compared with == and !=
  - The operators ++ and -- can be applied to them
  - An integer value can be added or subtracted from a pointer
  - If 2 pointers are the same type, they can be compared with <, <=, >, and >=



- **Casting example:**

```
int x= 10, *p= &x;
double *q;
q= (double *) p;
```

- **`const:`** indicates a variable or parameter that can't be changed
- **Typedef:** gives a new name to a type; Syntax: **`typedef`** `type-name new-name`
  - *Ex.* (note capitalization) `typedef float Interest_rate;`
- Most scalar types can be convert to other scalar types using ***casts***
  - *Ex.* `int *device = (int *) 100;`
- **Generic (void) pointers:** can point to any type; can be assigned to any "real" pointer type variable *(and vice versa)* without casting *Ex.* `void *p;`
- *Covers pointers and void pointers; see **Chapter 6** for more information.
- *Covers Worksheet 3.

Lecture 7: *Chp 7, 8* ✓

- *Covers functions, prototypes, arguments, and parameters; see **Chapter 7** for more information.
- *Covers arrays vs. pointers; see **Chapter 8** for more information.
- *Covers multidimensional arrays and passing arrays to functions; see **Chapter 8** for more information.

Lecture 8: *Chp 8, 10* ✗

- **(watch the video; this is incomplete)**
- *Covers arrays vs. pointers; see **Chapter 8** for more information.
- int pointer in grace is 8 bites chomp chomp chomp chomp chomp chomp chomp chomp

Lecture 9: *Chp 10, 11* ✓

- **Unions:** looks like a structure but all its fields share the same memory space; only one of them "exists" or is "active" at a time
  - A common paradigm— include a union inside a structure, along with an enum that can be used to keep track of which union variant is currently active
  - A structure can have common fields and varying fields by having a union member

```
typedef struct {

  int num_wheels;
  double cost;  /* how much $$$ */

  enum { BICYCLE, CAR, BUS } vehicle_type;

  union {
    unsigned short num_speeds;
    float miles_per_gallon;
    unsigned int num_passengers;
  } data;

} Vehicle;
```

- **Stack memory vs. heap memory:** allocation and deallocation order
  - Why use dynamically-allocated memory:
    - To avoid compile-time limits
    - Linked data structures
  - Common errors:
    - Not checking the return value of `malloc()` or `calloc()`
    - Not allocating enough memory
    - Using `malloc()` and not initializing the memory returned if it needs to be initialized
  - Memory errors to avoid:
    - Calling `free()` on something not dynamically allocated
    - Freeing the same thing more than once (also undefined)
    - Calling `free()` on a pointer pointing anywhere other than the beginning of a dynamically allocated memory region

Functions for allocating memory in **stdlib.h**:

`void *malloc(size_t size);`

- Allocates **size** bytes of memory from the heap (if enough is available) and returns a pointer to the beginning of it
- Does not initialize the memory at all!

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

- Allocates enough memory (if enough is available) to store **num** things, each of size **size** bytes
- Returns a pointer to the beginning of the allocated memory, after initializing it to all 0 bits

Both return **NULL** if the requested amount of memory can't be allocated

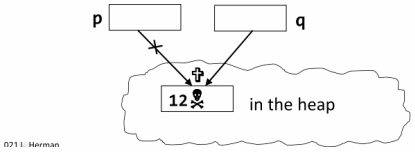
`void free(void *p);`

- Releases the memory pointed to by **p** and returns it to the free memory pool in the heap
- Always free memory when done with it!
- **`free()`** releases memory, but it does not:
  - change the values stored in the memory
  - change the value of the pointer it's called on
- Since **`free()`** does not change its parameter pointer, it's a good idea to set the argument to **NULL** right after the call (or point it somewhere else)

- **`free(NULL)`** is not an error (just does nothing)

- ***Dangling pointer**: when an object is destroyed but the pointer's value has not been changed
- ***Pointer aliasing**: See lecture video for in-depth explanation.

```
int *p= NULL, *q= NULL;
p= malloc(sizeof(*p));
q= p;
free(p);
p= NULL;
… 0
*q= 12;
```



Lecture 10: *EXAM WEEK*

Lecture 11: *Chp 9, 11* ✓

- **`realloc`:** Check **Chapter 11** ***(Calloc and Realloc)***
  - `void *realloc(void *prt, size_t new_size);`
- Recall that functions can return pointers: ---------->
- You do not need to cast the return values of the memory allocation functions.

Recall that functions can return pointers

```
int *new_arr(int num, int initial_value) {
  int *ptr= malloc(num * sizeof(int));
  int i;

  for (i= 0; i < num; i++)
    ptr[i]= initial_value;

  return ptr;
}
…
int *values= new_arr(10, 42);
```

BAD

```
int *p;
p= (int *) malloc(sizeof(*p));
```

GOOD

```
int *p;
p= malloc(sizeof(*p));
```

- Garbage collection: less work for programmer; slower process; does not happen in C
- **NEVER** use **`strtok`** — Larry Herman
- Writing your own string function:

Suppose you wanted to write a string copy function, with prototype

```
void my_strcpy(char *q, const char *p);
```

You want to be able to call it like this:

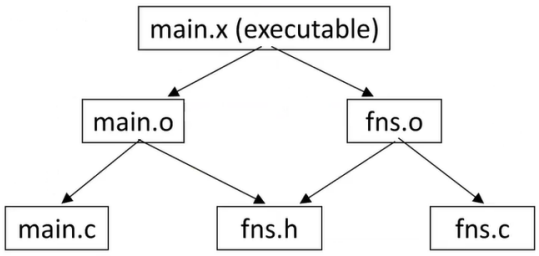
```
char str1[80]= "", str2[80]= "duck";
my_strcpy(str1, str2);  /* copy str2 to str1 */
```

You really want to use pointer arithmetic (because you are an overachiever)

- Goes over Worksheet 7 **(Chapter 9)**; Review **Chapter 9** and **Chapter 11**

Lecture 12: *Make and Makefiles* ✓

- **Compiling:**
  - You have to compile a source file every time you make a change, even if it's just changing 1 line. This is quicker to do for a small program; breaking a program up into smaller source files makes it more manageable.
- **Header file:** contains prototypes of functions so that they are defined before they are called
  - NEVER contains statements; includes enums, structure definitions, function prototypes, symbolic constants,typedefs, and sometimes extern global variable declarations
- **make:** utility that automates the process of building programs
  - Uses a ***makefile*** that specifies commands to create files and specifies which program files depend on which other ones
  - Anatomy of a ***makefile***:



Composed of *rules*

target — dependency list

```
main.o: main.c fns.h
        gcc -c main.c
```

← action

tab character

- A rule typically contains a *target*, a *dependency list*, and one or more *actions* (or directives)
- The target appears first (before a colon)
- The dependency list (after the colon) is the files that, if changed, should cause the target to be rebuilt
- Action lines follow a target, and must begin with a tab character

```
ss Makefile
main.x: main.o fns.o
        gcc main.o fns.o -o main.x

fns.o: fns.c fns.h
        gcc -c fns.c

main.o: main.c fns.h
        gcc -c main.c
grace10:~/<2>lectures/examples/lecture12/first-makefile-example:
```

```
main.o needs to be (re)created if main.c or fns.h changes
  the command that will (re)create it  gcc -c main.c

fns.o needs to be (re)created if fns.c or fns.h changes
  the command that will (re)create it  gcc -c fns.c

main.x needs to be (re)created if main.o or fns.o
  the command that will (re)create it  gcc main.o fns.o -o main.x
```

- **Running *make*:**

Running make

- See the complete makefile for the example on ELMS (just omitting some gcc options for now) in **`first-makefile-example`**
- If the command **`make`** is run it will read a makefile named **`makefile`** in the current directory, and try to build the first target in it
- If there isn't a makefile named **`makefile`** in the current directory but there's one named **`Makefile`** it'll be used instead
- The **`-f`** *makefile-name* make tells make to read a makefile with a different name

Running make, con't.

- "**`make`** *target*" will try to build the target *target* instead of the first target in the makefile
  - You can also run make with multiple targets, e.g., "**`make`** *target1 target2 target3*"
- A target can have multiple actions, which will be executed sequentially (unless one of them has an error, in which case make will stop)
- The **`-n`** option lets you test a makefile by printing the commands that would be executed to build a target, without actually performing them (**`make -n`** or **`make -n`** *target*)

- **Modification time:**

- Make decides what to do using files' modification times
- **`ls`** can display files' modification time:
  - **`ls -l`**
  - **`ls -lt`** modification time order
  - **`ls -lrt`** invert listing order
- What affects files' modification times?
  - Changing the contents of a file in any way
  - The **`touch`** command can be used (**`touch`** *file*) if you need to do it manually
    - Use only when really necessary

Make's operation

- Make constructs a project dependency tree based on the makefile
- A target is considered to be out of date either if it doesn't exist, or if any one of its dependencies is newer than itself
- If a target is out of date, make performs its associated action
  - This process is performed bottom to top in the dependency tree

Formal definition of make

- For any target z, note that z's dependencies may be empty
- **`make x`** (**`x`** is a target) can be defined as follows:

```
make x:
  for every d in x's dependencies where the makefile
    contains a target named d
      perform "make d"
  if ((x does not exist) or
      (x's dependencies has an element newer than x))
        perform x's action
```

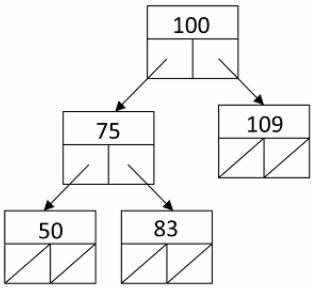
- **Macros in *makefiles*:** a macro (makefile variable) is defined as a variable = value, and is used later by enclosing its name in `$(...)`
- For projects, you can only use features that are explained in this lecture. There will be **no leniency** for grading problems caused by using other features that you are being told not to use.

Lecture 13: *Chp 12, 13; Linked lists & Graphs* ✓

- Watch the lecture:
  - **Linked data structures**
  - **Singly-linked list**
  - **Nodes**
  - **Binary search tree ---------->**

A typical declaration:

```
typedef struct node {
  int data;
  struct node *left;
  struct node *right;
} Node;
```



  - **Directed graph:**

```
struct vertex;  /* an incomplete type */

typedef struct edge {
  struct vertex *destination_vertex;
  struct edge *next_edge;
  /* perhaps other data for an edge here
     as well */
} Edge;

typedef struct vertex {
  Edge *edge_list;
  /* other data for a vertex here as well*/
} Vertex;
```

  - **Doubly linked list:**

Inserting into a doubly-linked list

Suppose we want to create an ordered, doubly-linked list and maintain a tail pointer pointing to the list's last node at all times

```
typedef struct node {
  int data;
  struct node *next;
  struct node *prev;
} Node;

Node *head, *tail;
```

Insertion must handle four cases:
- inserting into an empty list,
- inserting a new first element,
- inserting a new last element, and
- inserting into the middle of a list

Inserting into a doubly-linked list

The four cases in more detail (try drawing pictures of these yourself):
- An initially empty list:
  - Set the list's head and tail pointers (to a new node created)
  - Set the new node's next and previous pointers to **NULL**
- Beginning of the list (add a new first element):
  - Set the new node's previous pointer to **NULL**
  - Set the new node's next pointer to the old first node
  - Set the old first node's previous pointer to point to the new node
  - Set the list head pointer to the new node

  - **Advanced pointers:**

- How are declarations read in general in C?

```
int *p;
int *x[10];
int (*f)(void);
float (*z)(int *, char);
char *(*h[5])(int);

typedef void (*Ifp)(int);
Ifp p;
```

- See the examples `simple-fn-ptrs.c` and `fn-ptrs2.c`

- Review **Chapter 13**

Lecture 14: ***Chp 13, 14*** ✓

- **Command-line arguments:**

`main()` can also be written with "parameters", as

```
int main(int argc, char *argv[]) {
  …
}
```

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

`argc` will have the number of arguments that appeared on the command line when the program was run, including the program's name

`argv` will be an array of the arguments (as C strings) that appeared on the command line when the program was run

- `argv[0]` – the command that was invoked to start the program
- `argv[n]` – the n^{th} command line argument
- `argv[argc]` is always `NULL`

- **Environment variables:**
  - **`HOME`:** contains the location of a user's home directory
  - **`PATH`:** is used by the shell (and other programs) to find programs
  - **`envp`:** will be an array of null-terminated strings, each of the orm KEYWORD = VALUE, where KEYWORD is an environment variable name
    - *Ex.* `int main( int argc, char *argv[], char *envp[] )`
- Steps in compiling a C program:
  - Source files are converted to object files
    - Preprocessing
    - Compilation (or translation of modified files):
      - Scanner
      - Parser
      - Type checker
      - Code generator
      - Optimizer
    - Compilers often produce assembly code, then run the assembler on it
  - Object files are converted to an executable
    - This is done by the linker, which resolves symbols
      - Function use and definitions
      - Global variable declarations and external use

- **#define preprocessor directive**
  - **Syntax:** `#define` *name replacement*
  - **__FILE__**: the name of the source file being compiled
  - **__LINE__**: the line number of the current line in the file
  - **__DATE__**: the date the file was compiled
  - **__TIME__**: the time the file was compiled
  - **__STDC__**: 1 if the compiler conforms to ANSI C
  - Note: these start and end with 2 underscores
- **Macros:**
  - **Syntax:** `#define` *NAME* (parameters) *replacement*
  - **Invocation:** *NAME* (parameters)
  - **Ex.**

```
#define SUM(a, b) a + b
…
x= SUM(2, 5);
x= SUM(6, 4);
```

  - Macros are a single line, but you can define a "continuation" by ending a long line with a backslash:

```
#define COMPARE(s, t) s.x == t.x ? \
                      0 : \
                      (s.x < t.x ? -1 : 1)
```

  - Macros cannot be recursive
  - The preprocessor doesn't check the types of marco arguments
  - Macro substitution is not performed in string literals (or in comments)
- **Conditional Compilation:**
  - **#if *const-expr1*:** code here is included by the preprocessor if and only if *const-expr1* is true
  - **#elif *const-expr2 /* optional */*:** code here is included by the preprocessor if and only if *const-expr1* is false and *const-expr2* is true
  - **#else *const-expr2 /* optional */*:** code here is included by the preprocessor if and only if *const-expr1* is false and *const-expr2* is false
  - **#endif**

    - One type of *const-expr* is **defined(*NAME*)**, which has the value 1 if *NAME* is a defined preprocessor symbol, and 0 otherwise
    - The preprocessor directive **#error** *string* stops compilation at the point it appears, and prints *string*
    - Conditional compilation is useful if code must be different on different systems, but overuse makes code hard to read
- **Nested or multiple inclusion of header files:**

- Usually done with all header files, so they can be included anywhere in any order without problems
- Use names of this format for conditional compilation— don't make up your own

A header file like `term.h` is always written as

```
#if !defined(TERM_H)
#define TERM_H
(all the previous contents of term.h appear here)
#endif
```

- **File inclusion and makefiles:**

What would dependencies in a makefile look like when one header file includes another one?

term.h:

```
#if !defined(TERM_H)
#define TERM_H
definitions and prototypes here
#endif
```

main.c:

```
#include "polynomial.h"
int main(void) {
 code here
}
```

polynomial.h:

```
#if !defined(POLYNOMIAL_H)
#define POLYNOMIAL_H
#include "term.h"
definitions and prototypes here
#endif
```

- **Other preprocessor capabilities:**
  - **`#undef` *NAME***
    - Undefines the preprocessor symbol *NAME*
  - The compiler passes command-line definition of symbols to the preprocessor

```
gcc -D SIZE=100 file.c -o file.x
```

Lecture 15: *Chp 12* ✓

- **Designing linked list operations:**

Empty list case:

pre: `curr == NULL == head == tail`

post:
```
new_node->prev == NULL == head == tail == curr
new_node->next == NULL == head == tail == curr
head == new_node
tail == new_node
```

Beginning of list case:

pre: `curr == head`

post:
```
new_node->prev == NULL == curr->prev
new_node->next == head == curr
head->prev == curr->prev == new_node
head == new_node
```

End of list case:

pre: `curr == NULL`

post:
```
new_node->prev == tail
new_node->next == NULL == curr
tail->next == new_node
tail == new_node
```

Middle of list case:

pre: `curr` points to a node in the list
data at `curr` is greater than `value`
data at `curr->prev` is less than `value`

post:
```
new_node->prev == curr->prev
new_node->next == curr
curr->prev->next == new_node
curr->prev == new_node
```


280

- **Testing for errors:**

What do we try to test?

Normal cases
- Does the code work on valid inputs or parameters?

Error cases
- Does the code do what it's supposed to for invalid inputs or parameters, and what happens after an error (can we continue)?
- Often most bugs lie in this part of the code, since it has many infrequently-executed cases

Environmental errors
- Examples: out of memory, network is down, disk drive is full
- These situations can often be very hard to test

- **White box (or clear box) testing:**
  - Create test cases by examining te statements (source code) of the code being tested
  - Try to get maximum coverage of a set of tests

```
if (x)
  a;
if (y)
  b;
```

  - Line or statement coverage
    - **Example set of tests:**
      - x = 1, y = 1
  - Branch coverage (at least one true, at least one false)
    - **Example set of tests:**
      - x = 1, y = 1
      - x = 0, y = 0
    - **Another example set of tests:**
      - x = 1, y = 0
      - x = 0, y = 1
  - Path coverage:
    - **Example set of tests:**
      - x = 1, y = 1
      - x = 1, y = 0
      - x = 0, y = 1
      - x = 0, y = 0

White-box testing- how to test different types of statements
- If-then-else- are all cases executed by at least some test?
- Switch- are all branches executed in at least some test?
- Loops
  - Is the loop run different numbers of time?
  - Is every possible termination condition tested by some test?
- Assignment statements- are a range of possible values assigned in different tests?

- **Black box testing:**
  - Create test cases based on the specifications of the code to be tested, without looking at the code
  - Test for incorrect or missing functionality or behavior
  - The problem with black-box testing is that edge, boundary, and corner cases may not be tested if tests are created without looking at the code

Lecture 16: *Chp 15, 16* ✓

- **perror():**

In addition to **`perror()`**, the string library function **`strerror(int err_number)`** returns a (pointer to a) string describing an error, similar to what **`perror()`** prints (pass in **`errno`**)

- **Buffering:**

There are three types of buffering: *unbuffered*, *block buffering*, and *line buffering*

– Printing a newline also flushes a line-buffered stream

- **assert():**

- About **`assert()`**– if the preprocessor symbol **`NDEBUG`** is defined (including on the command line) when **`<assert.h>`** was last included, any assertions don't have any effect at all
- In UNIX, a program using math library functions in **`<math.h>`** must be linked with the **`-lm`** option
- You can use the **`man`** command to view the manpage of any standard library functions

- **Implementing dynamic allocation:**

Where does memory for a program's heap come from?

– The memory allocation functions (**`malloc()`**, **`calloc()`**, and possibly **`realloc()`**) make calls to the OS (*system calls*) that add memory to the heap (move the top of heap pointer)

- In Linux:

```
void *sbrk(int incr);
```

adds (at least) **`incr`** bytes to the program's data segment

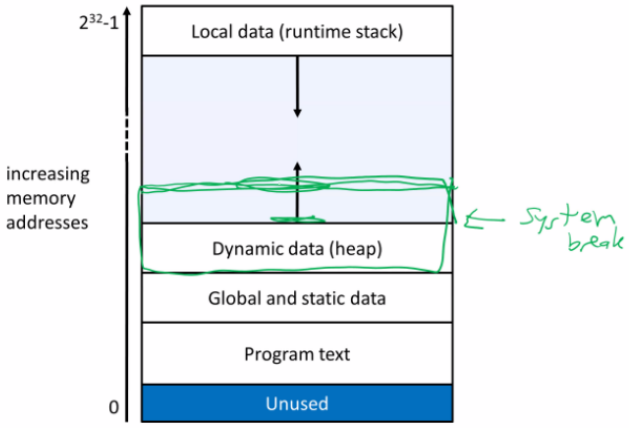
For efficiency, the memory allocation functions get big chunks of memory from the OS (via **`sbrk()`**) when the heap size is increased, then hand out small regions to the program when allocation requests occur

How is the memory in the heap managed?

– Data structures, such as the *free list*, need to be maintained to keep track of what memory is in use and where the free memory regions are

– Calls to the memory allocation functions update these structures

Virtual address space (from Lecture #1)



- **Heap manager requirements:**

  - The heap manager must be able to handle arbitrary sequences of requests to allocate and deallocate memory
  - It must respond immediately
  - All data it uses must be stored in the heap itself
  - The heap must be *aligned* to be able to hold any type of data
  - It can't modify already-allocated memory regions by increasing their size, changing their location, etc.- it will only modify free areas of memory

  Heap manager goals

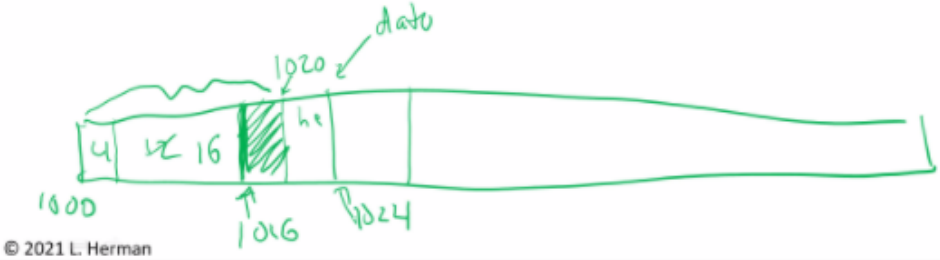
  - Maximize throughput (speed)
    - But it's often OK if the average memory allocation request is fast yet occasional calls are slow
  - Minimize *fragmentation* (wasted space)- memory that is not available to satisfy allocation requests, even though it's not in use
    - There are two types of fragmentation:
      - *Internal fragmentation*
      - *External fragmentation*

- *Covers Worksheet 12.

Lecture 17: *Dynamic Memory Allocation* ✓

- **Heap:**
  - The heap is divided up into blocks, which are formed of one or more 8-byte chunks
  - Blocks will always begin at a memory address that is a multiple of 8
  - Each block has an adjacent 32-bit header (just before it in the memory) that describes it

- **Block header format:**
  - **29 bits** define the size of the block
  - **1 bit** indicates whether the block is free or allocated
  - **2 bites** are unused (some heap managers use them, but not here)
  - The size stored in the header is the number of bytes allocated for data, plus the size of the header itself
  - **4 bytes** of the memory before a block may be wasted, since the header must be located so that the following block begins on an address divisible by 8
  - The end of the heap is indicated by a block with 0 in its size (its free/allocated bit is 1)



- **`malloc()/calloc()` using the simple heap- allocation; `free()` using the simple heap-allocation**

Using the simple heap- allocation

What the memory allocation functions (`malloc()`/`calloc()`/possibly `realloc()`) will do:

– They need to find a free block to satisfy the request:
  - They will start at the first block, and traverse the list of heap blocks, looking for a free one that's large enough to satisfy the request
  - They'll mark the header of the selected block as allocated, and return a pointer to it (its memory address)

Using the simple heap- deallocation

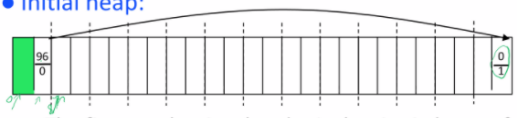
What `free()` will do:

– The block's header is four bytes before the pointer that `free()` is being called on– go to the header and mark the block as free

- **Initial heap:**
  - Dash line is every 8 bites
  - Solid line is every 4 bites

Example, assuming a small 104-byte heap

- Initial heap:



– The first number in a header is the size in bytes of the following block, including the header itself
– The second number in a header is the free/allocated bit– 0 means the space following that header is free and 1 means it's allocated (blue areas will be allocated memory)
– Green areas are wasted space, due to the necessity of aligning blocks on 8-byte boundaries

Review lecture 17 slides for the memory allocation example in detail.

- **Two possible ways to choose (allocate) a block:**
  - ***First fit:*** choose the first available block that's large enough
  - ***Best fit:*** choose the smallest block that's large enough
- **Coalescing free blocks:**
  - An adjacent sequence of free blocks can be merged into a larger free one
  - A *coalescing* head implementation will update the header of the first of the adjacent free blocks to the size of the full free space

What about splitting a large block into two blocks when allocating, one that's allocated, and one that's free?

– This gives the most efficient utilization of space at first, but it can waste space by leading to external fragmentation after repeated allocations and deallocations

How can the prior block's header be found?

– Search from the beginning of the heap, or
– Include a footer at the end of each block, which also stores the block's size, or
– Include an explicit back pointer in each block's header, which stores the address of the previous block's header

Some allocation patterns may cause repeated splitting and merging

– Consider `p= malloc(n); free(p); q= malloc(n); free(q);` etc. If the first `malloc()` involves a split, the rest will too

- **Explicit free lists:**

An alternative to an implicit free list: explicit free lists

- Maintain separate free lists of different sized chunks
  - For example, ones of 8, 16, 32, 64, 128, 256, 512, 1024, 4096 bytes, and anything larger than 4096
- For an allocation request, return space from whichever list has chunks of the same size or just larger, if available
- Never split a block

- If a free list of the needed size is empty:
  - Get one or more blocks from the next larger size free list, and if that fails, call `sbrk()` to get more storage from the operating system (some multiple of the size of the empty free list)
  - Create a new free list of the needed size from the memory obtained
- `free()`:
  - Add the memory being freed back to the free list for the target size
  - If all the chunks of a given size are free, release the space for that list to the common free pool

- **Automatically detecting heap problems:**

Automatically detecting heap problems

- Auxiliary header storage– duplicate copies of headers can be stored elsewhere in the heap
- To enable detecting when programs write beyond the boundaries of allocated memory, a preamble and trailer can be added to each allocated block containing specific bit patterns

- Preamble and trailer:
  - Have the memory allocation routines allocate 2 * Y bytes extra for each allocation request
  - Fill the first and last Y bytes with a known nonzero bit pattern
  - Return a pointer Y bytes into the allocated region
  - When a region is freed, or when the user requests it, verify the Y bytes at the start and end of the region haven't been changed

- **Virtual memory:**
  - A program thinks it has memory addresses beginning with 0 (its virtual address space)
  - Actual memory (the machine's *physical address space*) is divided up into pages (on Grace, 4K, or 4096 bytes)
  - A program's memory can be loaded into physical memory anywhere
  - When a program refers to a memory location, *address translation* is performed
- **Address translation:**
  - The operating system maintains a data structure called they ***page table*** that keeps track of which physical pages a program's virtual pages are stored in
    - When a program refers to a memory location, the hardware and operating system (working together) split the memory address into 2 parts:
      - The low order bits (12 bits on a machine with 4K pages, because 2^12 is 4096) is an offset into a page
      - The system used the high order 20 bits as an index into the page table, which says where in the physical memory that page is

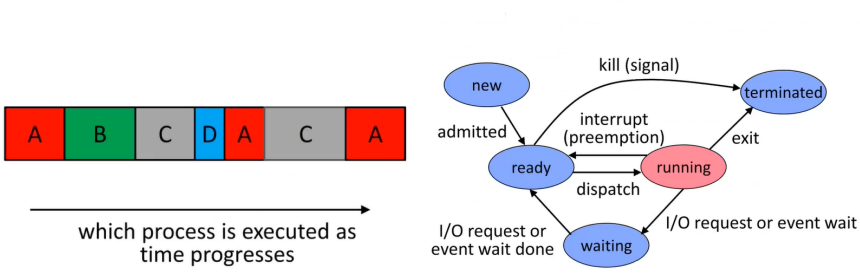
- Simplified example: an instruction in a program refers to memory location 0x00200068
  - The rightmost 12 bits are 068_{16} (104_{10})
  - The high-order 20 bits are 00200_{16} (512_{10})
  - Suppose the page table says that page 512 is stored in physical memory at actual starting address 4000_{10}
  - Then the hardware and OS use actual memory location 4104_{10} as the actual location that the instruction refers to
- This all happens invisibly to the program
- This process is performed by the *MMU* (*memory management unit*) of the hardware

Advantages of virtual memory

- A program's memory doesn't even have to be contiguous!
- A program can have a larger virtual address space than the machine's physical address space (i.e., a program can think it has more memory than the machine actually has)
- A program can be loaded anywhere into memory and run fine
- A program's virtual memory doesn't have to all be in physical memory at the same time
- Programs can easily share memory with the OS, and with each other, when needed

Lecture 18: *Process Control* ✓

- Read the textbook reserve.
- The ***kernel*** is the core part of an operating system that is always in memory
    - It runs directly on the hardware and provides a more convenient abstraction of the machine for programs to use
- Modern systems have certain *privileged instructions* that can only be executed in ***kernel mode***, not ordinary ***user mode***
    - To execute these instructions, a normal user program has to do something to enter kernel mode, like making a ***system call***
- **Multitasking:**
    - **Process:** writing the same code but with different data
        - **Program counter:** special register that always keeps the memory address of the next instruction to be fetched and executed
        - **Memory segment:**
            - Program code (often called the text segment)
            - Data (global variables and the heap)
            - Its runtime stack
        - **Contents of the processor registers:**
            - The program counter
            - General-purpose CPU registers
    - **Multitasking:** More than one process at the same time
        - **Data structures used to maintain the context of processes are:**
            - **The *page* table:** stores process' address spaces
            - **The *process* table:** stores information about each process
            - **The *file* table:** lists which files are opened by each process



- **Linux commands for viewing processes:**
    - **`ps:`** list current processes
        - **`ps`, `ps -e`,** and **`ps -f`**
    - **`pgrep`** *`name`***`:`** prints process number(s) of processes with *`name`* in their names
    - **`w:`** shows who's logged in and what they're doing
    - **`top`**
    - **Ctrl-Z, `bg`, `fg`,** and **`&:`** for running a process in the background
    - **`jobs:`** see the background processes currently running
    - **`pkill`** *`name`***`:`** kills processes with *`name`* in their names
    - **`kill`** *`process-ids`***`:`** kills processes given their process IDs

- **Signals:** a message to a process to notify it of an event

The kernel notifies processes of many events:

SIGSEGV: segmentation violation (a.k.a. segfault)

SIGFPE: floating point exception

SIGILL: illegal instruction

Users can trigger sending of signals:

SIGINT: interrupt (Ctrl-C)

SIGQUIT: quit and dump core (Ctrl-\)

SIGTSTP: terminal stop (Ctrl-Z)

- Processes have default actions when receiving signals (sometimes ignore, sometimes quit)
- There are mechanisms for processes to block signals, but two (**SIGKILL** and **SIGSTOP**) can't be blocked
- Processes can also send signals to each other (via the kernel)
- Signals are sent from the shell with the **kill** UNIX command, or by a program using the **kill()** system call
  - They do not always send the signal **SIGKILL**!

- **fork():**

The **fork()** system call

Creates a new copy of the calling process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- It returns once in each process:
  - Once from the initial call
  - And also in the new process
- Its return value indicates which process is which:

0	The child (new) process
> 0	The parent (original) process, and the value returned is the child's process ID
-1	The parent, but the child was not created (error case)

fork() semantics

Some things are inherited by a child from its parent process:

- A copy of the parent's memory contents, including program code, runtime stack, and heap
- The current working directory
- The environment
- The controlling terminal (the program that controls **stdin**, **stdout**, and **stderr** for the process, which is usually a shell), so the child reads input from and prints output to the same devices the parent does

More things that are inherited by a child from its parent process:

- Process credentials: user and group ID (UIDs and GIDs in UNIX terminology)
- Open files (copies of *file descriptors*, to be covered shortly)
- Resource limits (that can be set and viewed with the tcsh **limit** command)
- Signal handling settings
- Process priority for scheduling by the operating system (called "nice" value in UNIX)

Some things are unique to a child process:

- Its process ID
- Its parent process ID
- It has its own copies of its parent's file descriptors and directory streams (to be covered shortly)
- Its process times
- Its resource utilizations are initially set to 0
- Its pending signals are initialized to the empty set

- **Getting process IDs:**

Getting process IDs

```
pid_t getpid(void);
```

- Returns the process ID of the current process

```
pid_t getppid(void);
```

- Returns the process ID of the parent of the current process

```
#include <sys/types.h>
#include <unistd.h>
```

- See the example **fork-example.c**

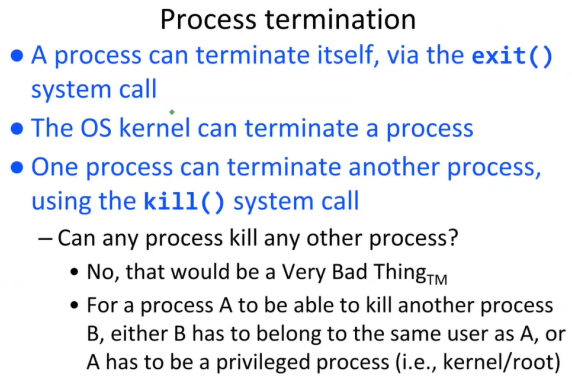
What if you overfork?

- The process table in the kernel can hold only a finite number of processes; what happens if it gets filled up?
- A *fork bomb*:

```
#include <unistd.h>

int main(void) {
  while (1)
    fork();
  return 0;
}
```

- A fork bomb often requires sysadmin intervention (e.g., reboot, killing all user processes)
- Be careful when using **fork()** in a loop!



Lecture 19: *Chp* ☒

- Text

Lecture 19: *Chp* ☒

- Text

Exam 1 - Last Minute Notes

Compilation Process

1. Pre-processing
2. Compilation
3. Assembly
4. Linkage

Scopes

- **File scope:** another way of saying “global variable”; always has static storage
  - **`static:`** putting this before a *file scope* identifier changes its linkage from internal to external
  - **Static** before a block variable changes its storage type but not its scope type
- **Block scope:** same as Java; coded in curly braces in a block
  - `Extern:` putting this before a *block scope* identifier changes its linkage from none to external

Static variables

- Can be used anywhere (and across functions) within the same source file

Scalar types: integer types, floating-point types, and pointer types; passed by value

Void pointers: cannot be dereferenced; can point to or be assigned to any type

Side effect: any modification to variables, files, etc. outside of a function’s local environment

Typedef: gives a new name to a type; Syntax: **`typedef`** `type-name new-name`

- *Ex.* (note capitalization) `typedef float Interest_rate;`

Also know:

- Pass by value vs. Pass by reference

Exam 2 - Last Minute Notes

- **String.h**
- `argc:` counts the number of arguments
- `argv:` the array that holds the arguments
- **Regression:**
    - **White box testing:**
    - **Black box testing:**
- **fgets():**
- fscanf():
-

Exam 3 - Last Minute Notes

Text