Spring 2013

CSE2421 Systems1

Introduction to Low-Level Programming and Computer Organization

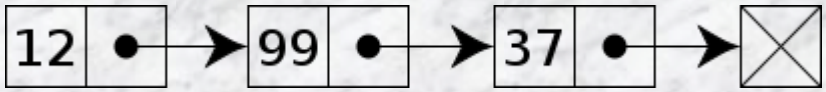
Kitty Reeves

TWRF 8:00-8:55am

Linked List Node structure

- A linked list is…
  - a dynamic data structure consisting of a group of nodes which together represent a sequence and whose length can be increased or decreased at run time
- Simply, each node is composed of a data and a reference (in other words, a *link*) to the next node in the sequence
- Allows for efficient insertion or removal of elements from any position in the sequence (vs an array).
- Data items need not be stored contiguously in memory
- Major Disadvantage:
  - does not allow random access to the data or any form of efficient indexing

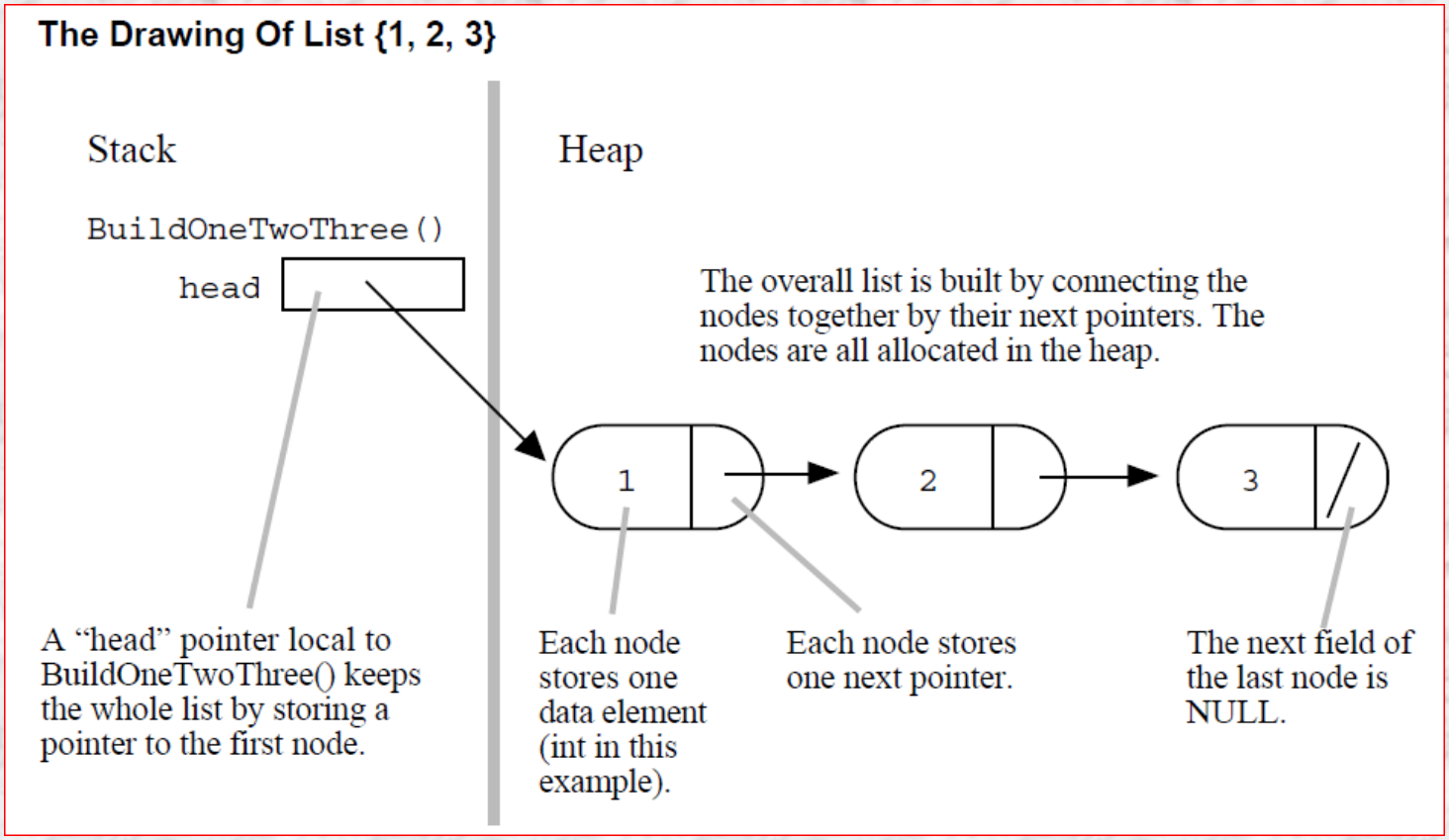
```
/* Node Structure */
struct node {
    int data;
    struct node *next; }
```



A linked list whose nodes contain two fields: an integer value and a link to the next node. The last node is linked to a terminator used to signify the end of the list.

Linked List stack and heap

- Each node is allocated in the heap with a call to malloc(), so the node memory continues to exist until it is explicitly deallocated with a call to free().



Build {1,2,3} linked list

```
/* Build the list {1, 2, 3} in the heap and store its head pointer in a local stack
variable. Returns the head pointer to the caller. */
struct node* BuildOneTwoThree() {
        struct node* head = NULL;
        struct node* second = NULL;
        struct node* third = NULL;
// allocate 3 nodes in the heap
        head = malloc(sizeof(struct node));
        second = malloc(sizeof(struct node));
        third = malloc(sizeof(struct node));
        head->data = 1; // setup first node
        head->next = second; // note: pointer assignment rule
        second->data = 2; // setup second node
        second->next = third;
        third->data = 3; // setup third link
        third->next = NULL;
// At this point, the linked list referenced by "head“ matches the list in the drawing.
        return head;  }
```

Linked List example

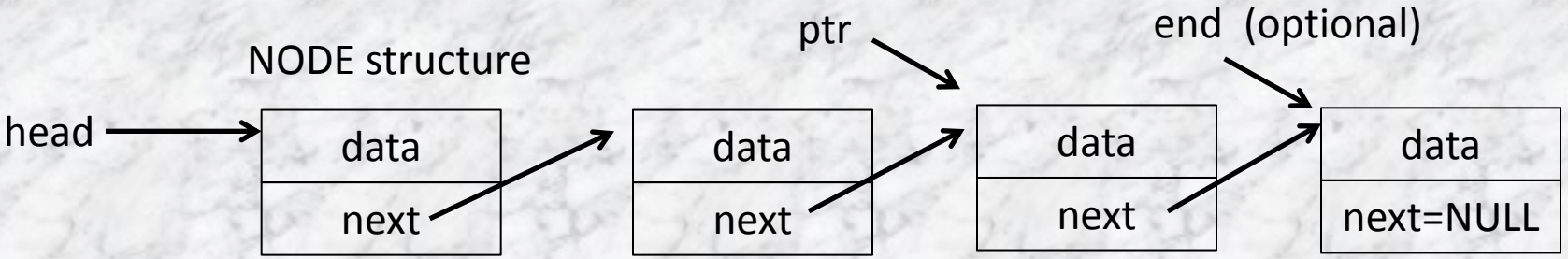
```
#include<stdlib.h>
#include<stdio.h>
struct list_el {
    int val;
    struct list_el * next; };
typedef struct list_el item;
void main() {
    item * curr, * head;
    int i;
    head = NULL;
    for(i=1;i<=10;i++) {
        curr = (item *) malloc(sizeof(item));
        curr->val = i;
        curr->next = head;
        head = curr; }
    curr = head;
    while(curr) {
        printf("%d\n", curr->val);
        curr = curr->next ; }
}
```

- What does this do?

Linked list basics

- The first node is always made accessible through a global ‘head’ pointer.
  - This pointer is adjusted when first node is deleted.
- Similarly there can be an ‘end’ pointer that contains the last node in the list.
  - This is also adjusted when last node is deleted.
  - The last node in a list always has a NULL value so you don’t *have* to keep track of the end of the list, just check for a NULL pointer.
- Whenever a node is added to linked list, it is always checked if the linked list is empty then add it as the first node.
- You can pass the list by passing the head pointer.

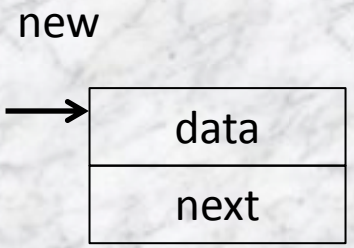
Linked List setup



NEED TO:

Allocate a new node structure with DMA
(dynamic memory allocation)

Add information to data section



Linked List ADD/DELETE node

OPERATION	FRONT	END	MIDDLE
INSERT	new->next = head head = new	ptr->next = new new->next = NULL	new->next = ptr->next ptr-> next= new

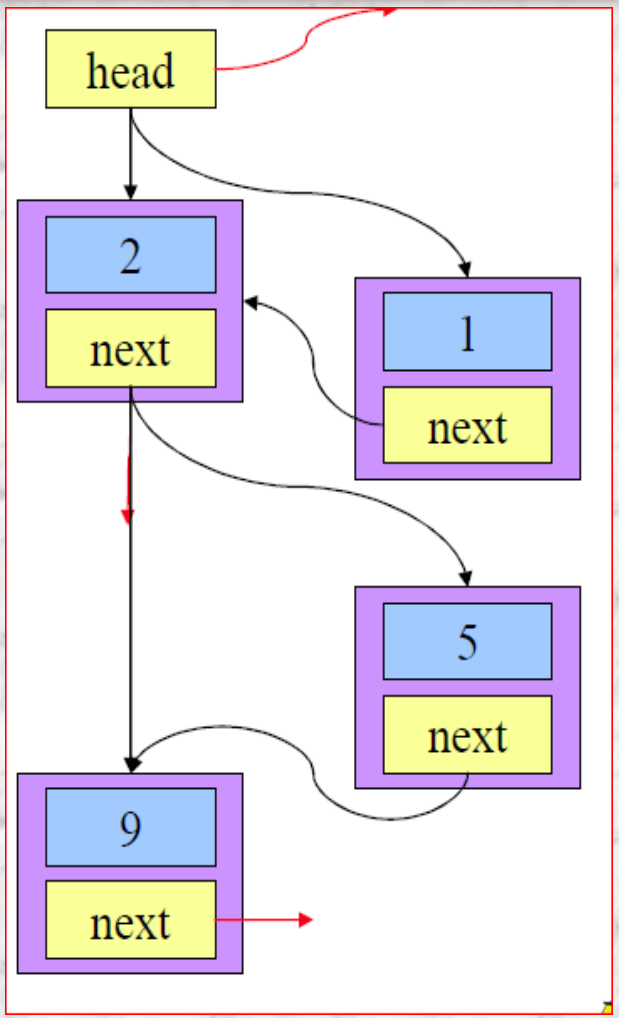
SEARCH

```
found = false;
ptr = head;
while(ptr != NULL)                  //what if nothing in list?
{    if(ptr->data = = val)          // found what searching for
     {    found = true;
          break; }
     else { ptr = ptr->next; }
}                                   // if found still false, didn't find
```

DELETE
(fyi: free ptr)
	head = ptr->next
	// what if only node?
		//if ptr->next=NULL
		prev = ptr
		prev->next = NULL
			prev->next = ptr->next

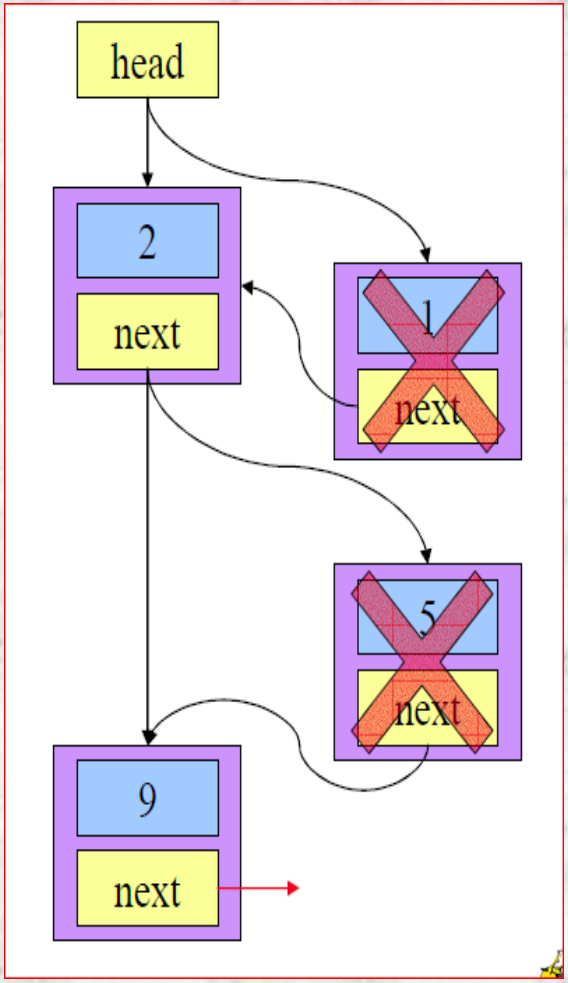
Link list pictorial view - INSERT

- Inserting into a link list has two cases
  - First in the list
  - Not first in the list
- If going at head, modify head reference (only)
- If going elsewhere, need reference to node before insertion point
  - New node.next = cur node.next
  - Cur node.next = ref to new node
  - Must be done in this order!



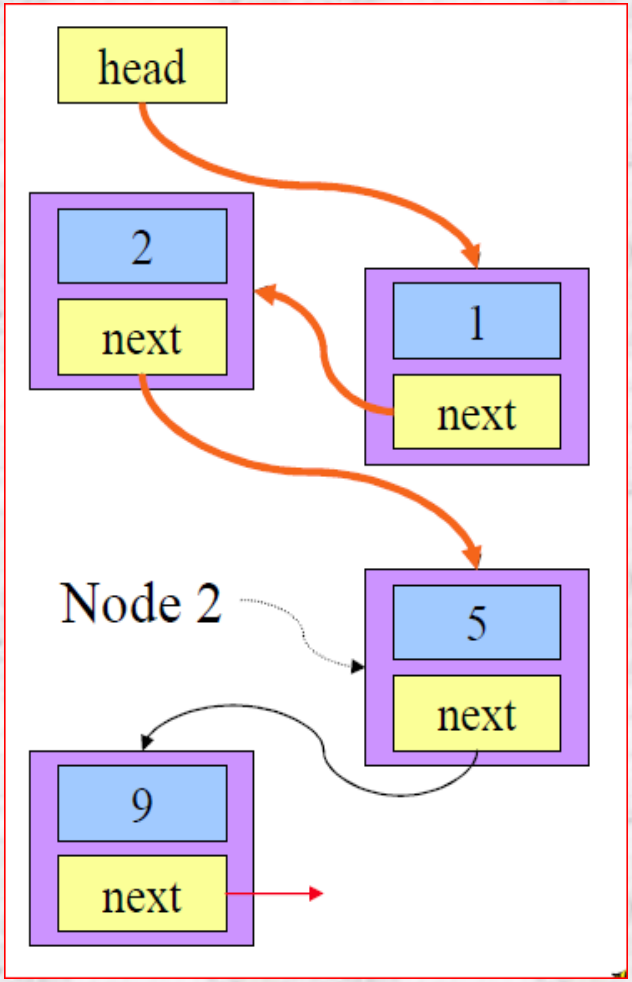
Link list pictorial view - DELETE

- Deleting a link list has two cases
  - First in the list
  - Not first in the list
- If deleting from head, modify head reference (only)
- If deleting elsewhere, simply “point around” the deleted node
- Be sure to free the deleted nodes



Link list pictorial view - TRAVERSE

- Start at the head
  - Use a "current" reference for the current node
- Use the "next" reference to find the next node in the list
- Repeat this to find the desired node
  - *N* times to find the *n*th node
  - Until the object matches if looking for a particular object
    - Caution: objects can "match" even if the references aren't the same…
- Don't forget to check to see if this is the last node



Linked List operations

- Initialize the list
- Push/Insert a value onto the list
- Search the list
- Pop/Remove a value off of the list
- Print the list

```
void InitList(struct list *sList);

/* Initializes the list structure */
void InitList(struct list *sList) {
    sList->start = NULL; }
```

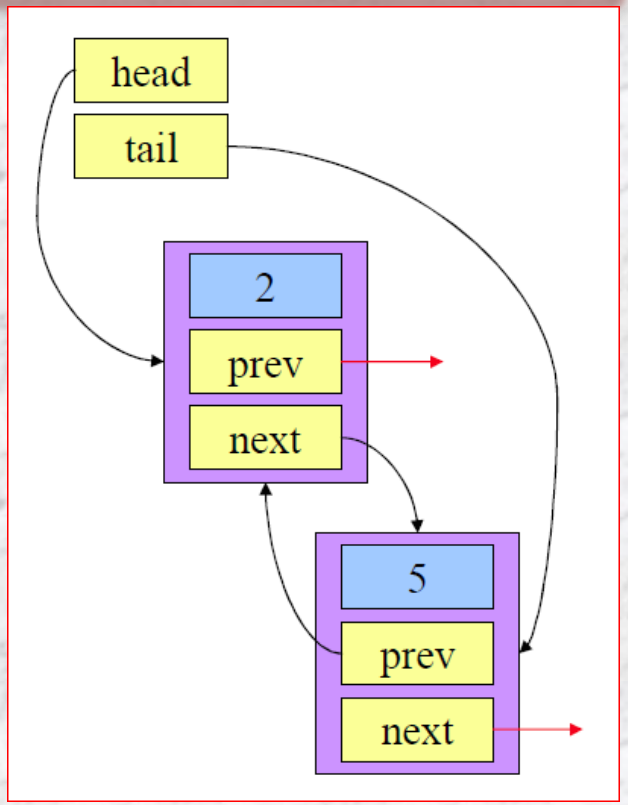
```
void push(struct list *sList, int data);

/* Adds a value to the front of the list */
void push(struct list *sList, int data) {
    struct node *p;
    p = malloc(sizeof(struct node));
     p->data = data;
    p->next = sList->start;
    sList->start = p; }
```

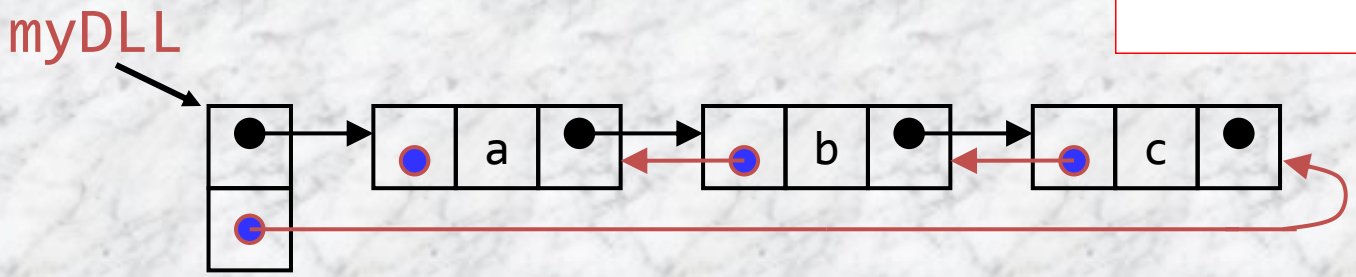
```
void pop(struct list *sList)

/* Removes the first value of the list */
void pop(struct list *sList) {
    if(sList->start != NULL) {
        struct node *p = sList->start;
        sList->start = sList->start->next;
        free(p); } }
```

(see linklst2.c)

Double Linked List (DLL)

- A more sophisticated form of a linked list data structure.
- Each node contains a value, a link to the next node (if any) and a link to the previous node (if any)
- The header points to the first node in the list and to the last node in the list (or contains null links if the list is empty



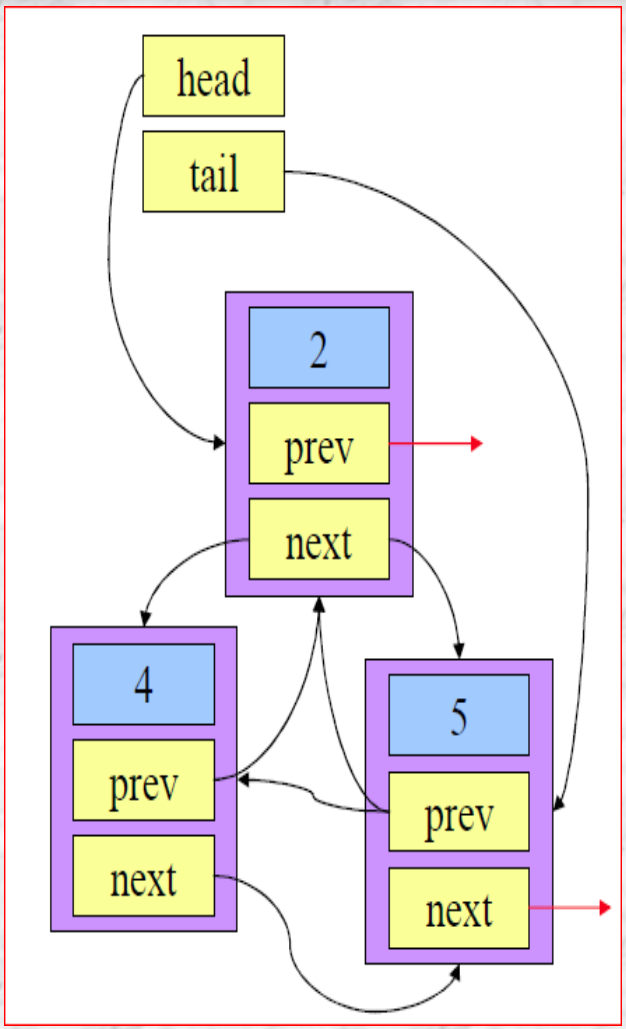


DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

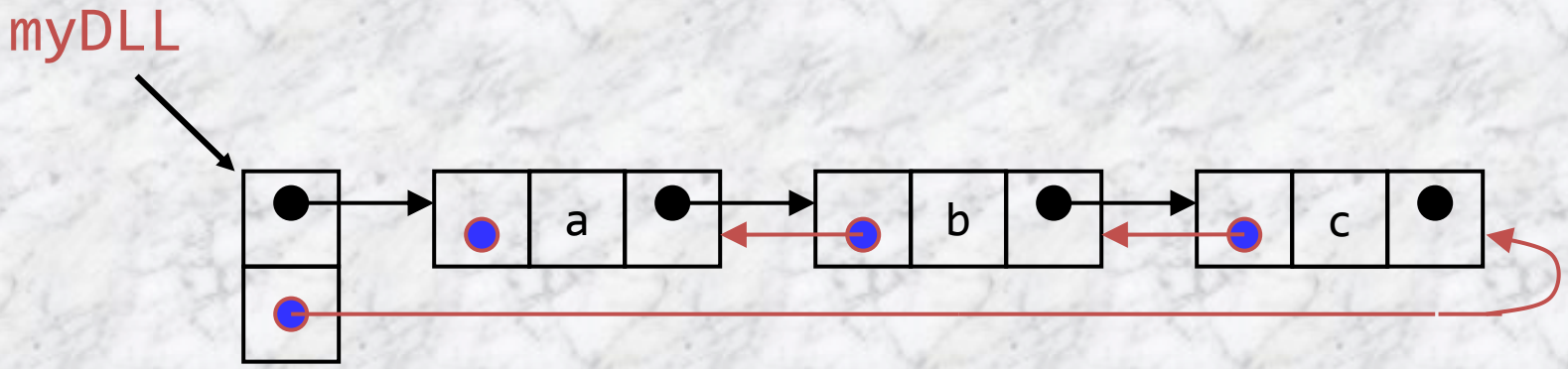
Double linked list – INSERT

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



Deleting a node from a DLL

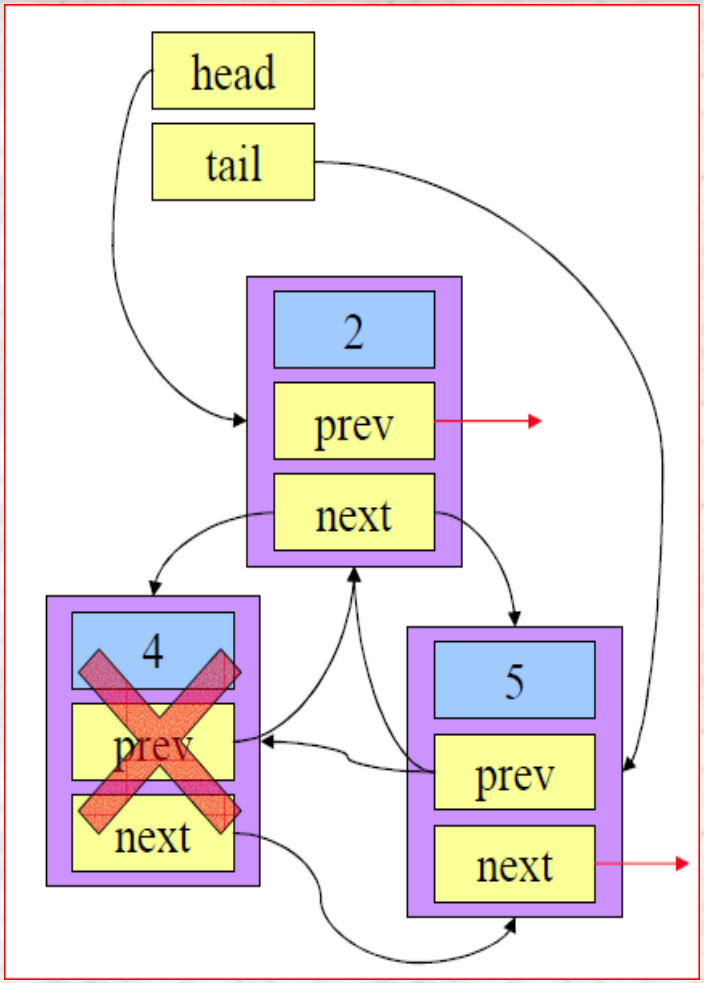
- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b



- Deletion of the first node or the last node is a special case

Double linked list - DELETE

- As with singly linked lists, special case for head
  - Also special case for tail
- Need to update two nodes
  - Node before new node
  - Node after new node
- Hook up new node *before* modifying other nodes
  - Don't overwrite necessary information before relocating it!
- Head & tail: if a link is null, update the head or tail as appropriate



Other operations on linked lists

- Most "algorithms" on linked lists—such as insertion, deletion, and searching—are pretty obvious; you just need to be careful
- Sorting a linked list is just messy, since you can't directly access the n^{th} element—you have to count your way through a lot of other elements

Double linked lists - SETUP

```
//DECLARATIONS

/* The type link_t needs to be forward-
declared in order that a  self-reference
can be made in "struct link" below. */
typedef struct link link_t;

/* A link_t contains one of the links of
the linked list. */
struct link                 {
    const void * data;
    link_t * prev;
    link_t * next;          };

/* linked_list_t contains a linked list. */
typedef struct linked_list  {
   link_t * first;
   link_t * last;   }  linked_list_t;
```

```
/* The following function initializes
the linked list by putting zeros into
the pointers  containing the first and
last links of the linked list. */
static void
linked_list_init (linked_list_t * list)
{     list->first = list->last = 0;     }
```

Double linked lists - ADD

```
/* The following function adds a new link to the end of the linked   list. It allocates
memory for it. The contents of the link are   copied from "data". */
static void
linked_list_add (linked_list_t * list, void * data){
      link_t * link;
      link = calloc (1, sizeof (link_t));       /* calloc sets the "next" field to zero. */
      if (! link) {
            fprintf (stderr, "calloc failed.\n");
            exit (EXIT_FAILURE);    }
      link->data = data;
      if (list->last) {                 /* Join the two final links together. */
            list->last->next = link;
            link->prev = list->last;
            list->last = link;    }
      else {
            list->first = link;
            list->last = link;    }}
```

Double linked lists - DELETE

```
static void
linked_list_delete (linked_list_t * list, link_t * link) {
        link_t * prev;
        link_t * next;

        prev = link->prev;
        next = link->next;
        if (prev) {
                if (next) {
                                prev->next = next;
                                next->prev = prev; }
                else {
                                prev->next = 0;
                                list->last = prev; } }
        else {
                if (next) {
                                next->prev = 0;
                                list->first = next; }
                else {
                                list->first = 0;
                                list->last = 0; } }
        free (link); }
```

Both the previous and next links are valid, so just bypass “link” without altering “list” at all

Only the previous link is valid, so "prev" is now the last link in "list".

Only the next link is valid, not th eprevious one, so “next” is now the first link in the “list”

Neither previous not next links are valid, so the list is now empty

Double linked lists - FREE

```
 /* Free the list's memory. */ s
tatic void
linked_list_free (linked_list_t * list)
{
      link_t * link;
      link_t * next;
      for (link = list->first; link; link = next)
      {
/* Store the next value so that we
don't  access freed  memory. */
            next = link->next;
            free (link);
       }
}
```

Don't FORGET!

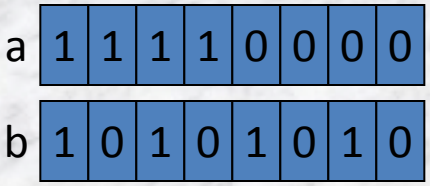
Bitwise Operations

- Many situations need to operate on the bits of a data word –
  - Register inputs or outputs
  - Controlling attached devices
  - Obtaining status
- Corresponding bits of both operands are combined by the usual *logic operations*.
- Apply to all kinds of *integer* types
  - Signed and unsigned
  - char, short, int, long, long long

Bitwise Operations (cont)

- **&** – AND
  - Result is **1** if both operand bits are **1**
- **|** – OR
  - Result is **1** if either operand bit is **1**
- **^** – Exclusive OR
  - Result is **1** if operand bits are different
- **~** – Complement
  - Each bit is reversed
- **<<** – Shift left
  - Multiply by 2
- **>>** – Shift right
  - Divide by 2

Examples



NOTE: when signed → all the same

FYI: integers are really 32 bits so what is the "real" value?

~a has preceding 1's and a<<2 is 0x 3c0

```
unsigned int c, a, b;
   c = a & b;              // 1010 0000
   c = a | b;              // 1111 1010
   c = a ^ b;              // 0101 1010
   c = ~a                  // 0000 1111
   c = a << 2;             // 1100 0000
   c = a >> 3;             // 0001 1110
```

Bitwise AND/OR

char x = ‘A’;
tolower(x) returns ‘a’… HOW?

char y = ‘a’;
toupper(y) returns ‘A’… HOW?

‘A’ = 0x41 = 0100 0001
‘a’ = 0x61 = 0110 0001

“mask” = 0010 0000
Use OR

‘A’ = 0100 0001
mask = 0010 0000 |
‘a’ 0110 0001

“mask” = 1101 1111
Use AND

‘a’ = 0110 0001
mask = 1101 1111 &
‘A’ 0100 0001

Notice the masks are complements of each other
TRY: char digit to a numeric digit

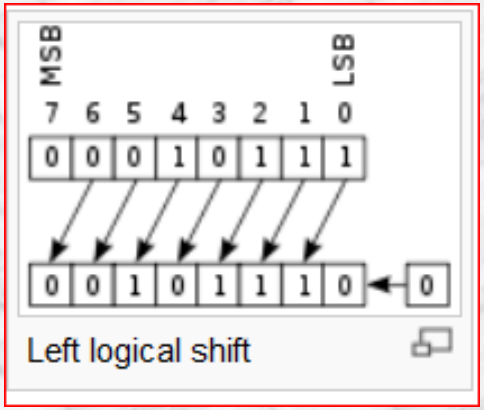
Bitwise XOR

- The bitwise XOR may be used to invert selected bits in a register (toggle)
- XOR as a short-cut to setting the value of a register to zero

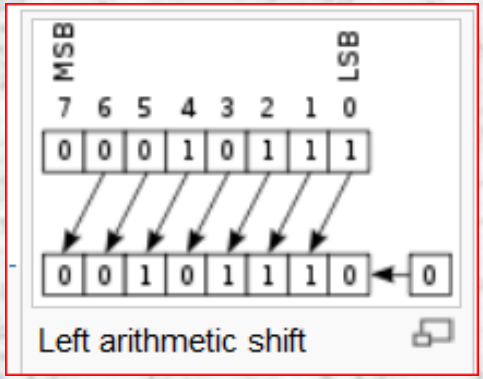
0100 0010
0000 1010 XOR (toggle)
0100 1000

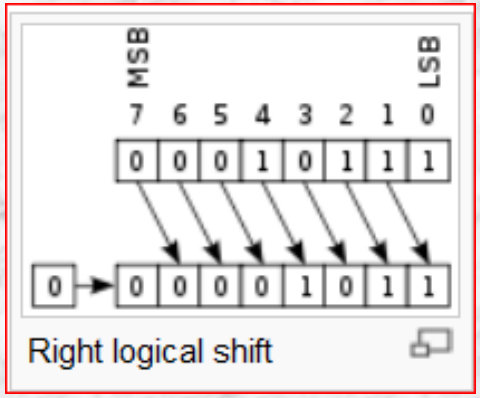
Bitwise left/right shifts

- Possible overflow issues
- Exact behavior is implementation dependent



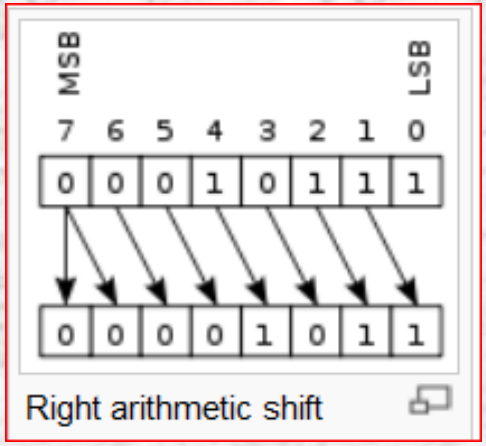
When you shift <u>left</u> by **k** bits ==
multiplying by 2^K





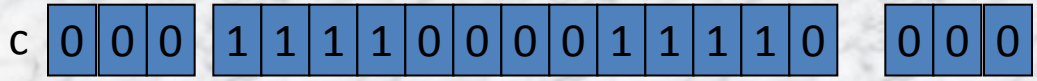
When you shift <u>right</u> by **k** bits ==
dividing by 2^K

***** If it's signed, then it's*****
implementation dependent.

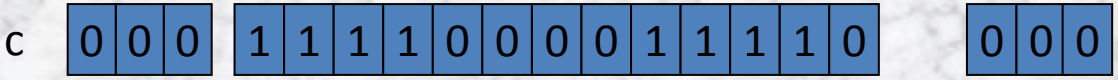


Bitwise right shifts


```
unsigned int c, a;
   c = a >> 3;
```




```
signed int c, a, b;
   c = b >> 3;
   c = a >> 3;
```



EXAMPLE: 8-bit instruction format
101 01000 // ADD 8 → ALU adds ACC reg to value at address 8
To get just the instruction i.e. 101… shift right by 5
To get just the address i.e. 01001… shift left by 3, then right by 3

C example…

```
#include <stdio.h>
void main()
{
  signed int c, d, a, b, e, f;
  a = 0xF0F0;
  b = 0x5555;
  e = 0b01000001;
  f = 'A';

  c = b >> 3;
  d = a >> 3;

  printf("b >> 3 is %x\n",c);
  printf("a >> 3 is %x\n",d);
  printf("binary = %x\n",e);
  printf("char a = %c",f);
}
```

Output is:

b >> 3 is aaa

a >> 3 is 1e1e

binary = 41

char a = A

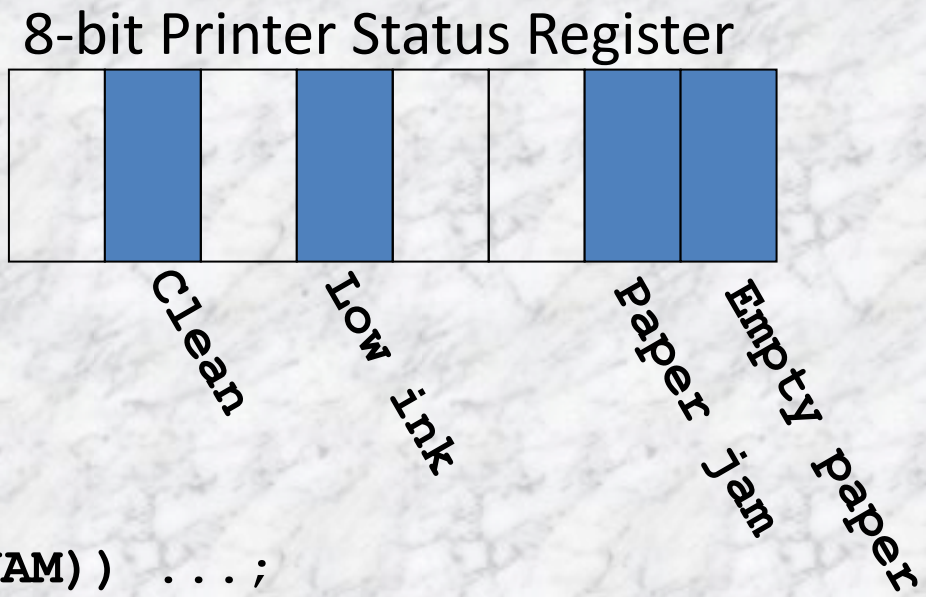
Bit example exceptions

```
#include <stdio.h>
void main()
{   int a, b, c, d, e, f;
  a = 0xF0F0F0F0;
  b = 0x55555555;
  c = a >> 3;        // repeats sign bit of 1
  d = b >> 3;        // repeats sign bit of 0
  e = a >> 35;       // 35(k) % 32(w) but technically undefined
  f = b >> -3;       // 32(w) - 3(k) but technically undefined
  printf("a >> 3 is %.8x\n",c);
  printf("b >> 3 is %.8x\n",d);
  printf("a >> 35 is %.8x\n",e);
  printf("b >> -3 is %.8x\n",f);
  printf("a << 3 is %.8x\n",a << 3);
  printf("b << 3 is %.8x\n",b << 3);
  printf("a << 35 is %.8x\n",a << 35);
  printf("b << -3 is %.8x\n",b << -3);
}
```

```
% gcc -o bitex bitex.c
bitex.c: In function ‘main’:
bitex.c:11: warning: right shift count >= width of type
bitex.c:12: warning: right shift count is negative
bitex.c:21: warning: left shift count >= width of type
bitex.c:22: warning: left shift count is negative
```

```
% bitex
a >> 3 is fe1e1e1e
b >> 3 is 0aaaaaaa
a >> 35 is fe1e1e1e
b >> -3 is 00000002
a << 3 is 87878780
b << 3 is aaaaaaa8
a << 35 is 87878780
b << -3 is a0000000
```

Traditional Bit Definition




```
#define EMPTY    01
#define JAM      02
#define LOW_INK  16
#define CLEAN    64

char status;
if (status == (EMPTY | JAM)) ...;
if (status == EMPTY || status == JAM) ...;
while (! status & LOW_INK) ...;

int flags |= CLEAN          /* turns on CLEAN bit */
int flags &= ~JAM           /* turns off JAM bit */
```

Traditional Bit Definitions

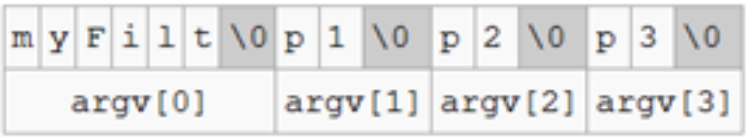
- Used very widely in *C*
  - Including a *lot* of existing code
- No checking
  - You are on your own to be sure the right bits are set
- Machine dependent
  - Need to know *bit order* in bytes, *byte order* in words
- Integer fields within a register
  - Need to **`AND`** and shift to extract
  - Need to shift and **`OR`** to insert

Int main(with arguments)

- Options:
  - int main(void);
  - int main();
  - int main(int argc, char **argv);
  - int main(int argc, char *argv[]);

myFilt p1 p2 p3

results in something like:

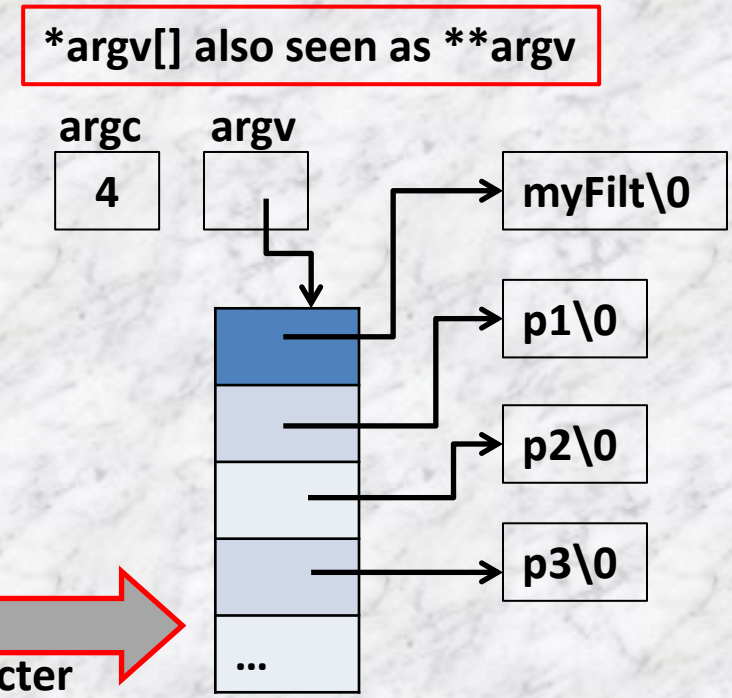


- The parameters given on a command line are passed to a C program with two predefined variables
  - The count of the command-line arguments in argc
  - The individual arguments as a character strings in the pointer array argv
  - The names of argc and argv may be any valid identifier in C, but it is common convention to use these names
- But wait… There is no guarantee that the strings are stored as a contiguous group (per normal arrays)

int main(argc, *arv[])

- The name of the program, argv[0], may be useful when printing diagnostic messages
- The individual values of the parameters can be accessed:

```
#include <stdio.h>

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

    printf ("argc\t= %d\n", argc);
    for (i = 0; i < argc; i++)
        printf ("argv[%i]\t= %s\n", i, argv[i]);
    return 0;
}
```



Command Line Arguments

- It is guaranteed that argc is non-negative and that argv[argc] is a null pointer.
- By convention, the command-line arguments specified by argc and argv include the name of the program as the first element if argc is greater than 0
- For example, if a user types a command of "rm file", the shell will initialize the rm process with argc = 2 and argv = ["rm", "file", NULL]
- The main() function is special; normally every C program must define it exactly once.

CLA - Example 1

```
#include <stdio.h>
int main(int argc, char *argv[]) {
   if ( argc != 3)
        printf("Usage:\n %s Integer1 Integer2\n",argv[0]);
   else
        // ascii to integer
        printf("%s + %s = %d\n",argv[1],argv[2], atoi(argv[1])+atoi(argv[2]));
  return 0;
}
```

CLA – Example 2

```
#include <stdio.h>
#include <stdlib.h>
main( int argc, char *argv[])
{ FILE *in_file, *out_file, *fopen();
int c;
if( argc != 3 ) {
    printf("Incorrect, format is FCOPY source dest\n");
    exit(2);     }
in_file = fopen( argv[1], "r");
if( in_file == NULL )
    printf("Cannot open %s for reading\n", argv[1]);
else { out_file = fopen( argv[2], "w");
    if ( out_file == NULL )
        printf("Cannot open %s for writing\n", argv[2]);
    else { printf("File copy program, copying %s to %s\n", argv[1], argv[2]);
        while ( (c=getc( in_file) ) != EOF )
            putc( c, out_file );
        putc( c, out_file); /* copy EOF */
        printf("File has been copied.\n"); fclose( out_file); } fclose( in_file); } }
```

Rewrite the program which copies files, ie, FCOPY.C to accept the source and destination filenames from the command line. Include a check on the number of arguments passed.

Redirection File I/O

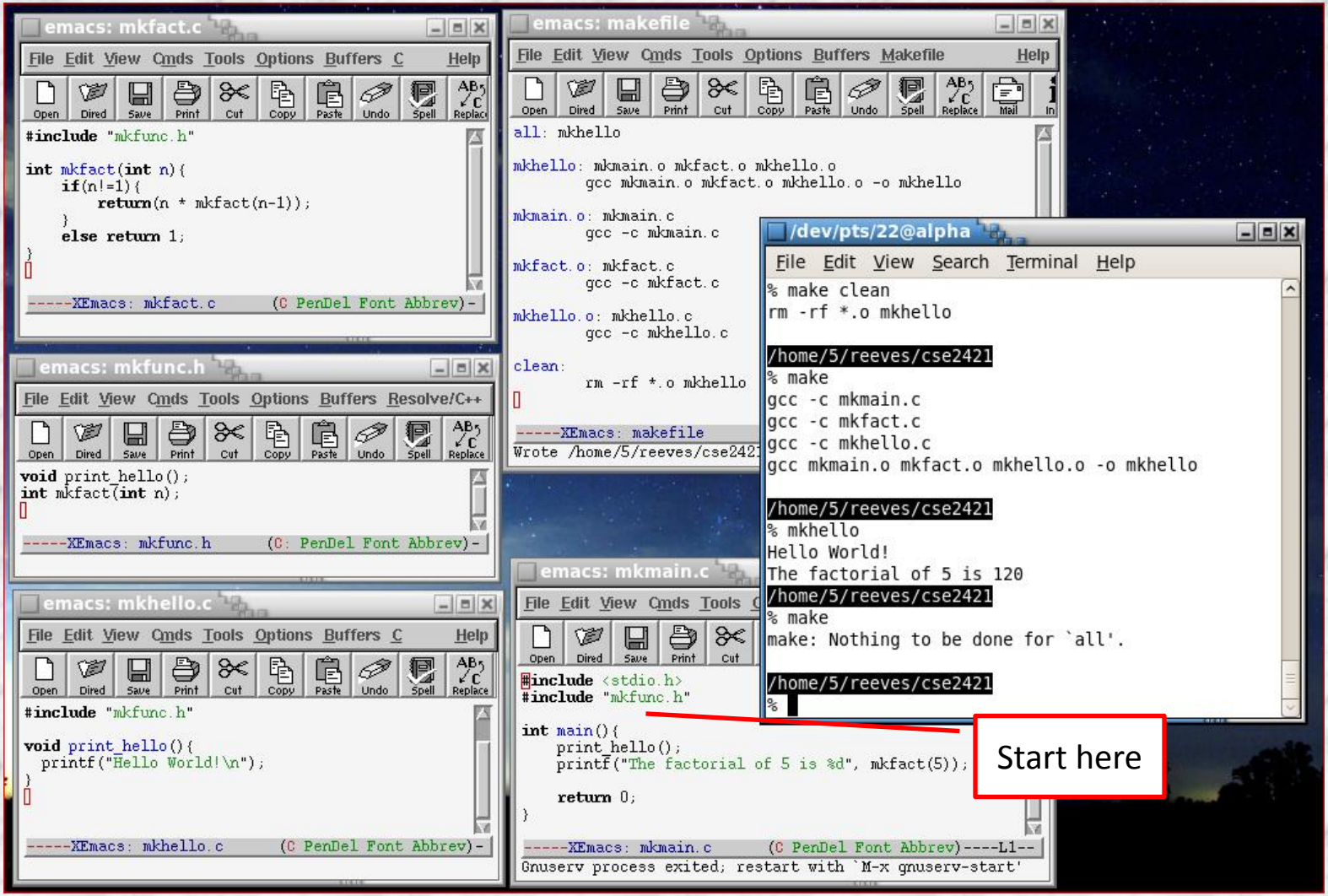
- Part of the operating system (linux)
- % lab2p2file < lab2p2in >! lab2p2out
- ! overwrites if the file already exists

```
input = 0;
scanf(…, input);
while (input != 0)
  {   loop stuff…
      input = 0;
      scanf(…, input);
   }
```

Input File:

```
War_Eagle!
How_many_WORDS_workhere?
i
$hake_u_r_booty:)_!
Og_sbuCk!!!
REALLy_really_really_really___really_long??!!_
Hi.01234_How_R_U_?
|
```

Header and Makefile example



What is happening?

- The -c option on the gcc command only *compiles* the files listed
- Once all 3 C files are correctly compiled, then using gcc with the -o option allows object files (notice the .o extensions) to be merged into one executable file.
- Notice where all “mkfunc.h” is included

Library includes

- The compiler supports two different types of #includes
  - Library files
  - Local files

- #include <filename>
- #include “filename”

- By convention, the names of the standard library header files end with a .h suffix
  - Where? ➔ /usr/include

Creating header files

- In our case, be sure to save your header file in a 'directory where you are going to save the program' (NOTE: This is important. Both the header file and the program must be in the same directory, if not the program will not be able to detect your header file).

- The header file cannot be included by
  - #include <headerfilename.h>

- The only way to include the header file is to treat the filename in the same way you treat a string.
  - #include "headerfilename.h"

Makefile Overview

- Makefiles are a UNIX thing, not a programming language thing
- Makefiles contain UNIX commands and will run them in a specified sequence.
- You name of your makefile has to be: makefile or Makefile
- The directory you put the makefile in matters!
- You can only have one makefile per directory.
- Anything that can be entered at the UNIX command prompt can be in the makefile.
- Each command must be preceded by a TAB and is immediately followed by hitting the enter key
- MAKEFILES ARE UNFORGIVING WHEN IT COMES TO WHITESPACE!
- To execute… must be in the directory where the makefile is:
  - % make tag-name (also called section name)

Makefile Details

- Compiling our example would look like:
  - gcc -o mkhello mkmain.c mkhello.c mkfact.c
  - OR
  - gcc mkmain.c mkhello.c mkfact.c -o mkhello

- The basic makefile is composed of lines:
  - *target: dependencies [tab] system command*
  - *"all"* is the default target for makefiles
    - all: gcc -o mkhello mkmain.c mkhello.c mkfact.c
  - The *make* utility will execute this target, "all", if no other one is specified.

Makefile dependencies

- Useful to use different targets
  - Because if you modify a single project, you don't have to recompile everything, only what you modified

- In the class example of the makefile:
  - All has only dependencies, no system commands
  - If order for *make* to execute correctly, it has to meet all the dependencies of the called target (i.e. in this case *all*)
  - Each of the dependencies are searched through all the targets available and executed if found.
  - make clean
    - Fast way to get rid of all the object and executable files (to free disk space)
    - -f do not prompt
    - -r remove directories and their contents recursively

Why use OOP in general?

- The concepts and rules used in object-oriented programming provide these important benefits:
  - The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself (typedef).
  - The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding.
  - Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding (i.e. encapsulation) provides greater system security and avoids unintended data corruption.

WHY use OOP in general (cont)?

- Facilitates utilizing and creating reusable software components
  - The definition of a class is reuseable not only by the program for which it is initially created but also by other object-oriented programs (and, for this reason, can be more easily distributed for use in networks).
- Better suited for team development
- Easier software maintenance

OOP simple review

- Class
  - A software construct that abstractly models something
  - Defines a structure to hold some sort of state
  - Defines operations that mutate or recall this state somehow
- Object
  - A specific instance of a Class
  - Holds the state representing a particular instance of the Class
- Examples:
  - Class – Person
  - Instance – William Gates

OOP simple review (cont)

- Define method (~function)
  - To emulate member functions, you can put function pointers in structs.
- Define inheritance
  - a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support
- Define polymorphism
  - the ability to create a variable, a function, or an object that has more than one form
  - allows values of different data types to be handled using a uniform interface (malloc returns void type)
- Encapsulation == information hiding
  - C has language support for private encapsulation of both variables and functions, through the static keyword

Porting object-oriented concepts to C

- It's possible to create object-oriented like code in C, which is very useful for mimicking standard libraries and objects found in OOPs
  - Ex. Stack and Queue classes with push/pop methods = functions - the basis for modular structured programming in C.
  - Header files = Use to define global constants and variables
- Take a class design from what would be standard OOP, retain strictly only member variables, and move them to a struct.
- Within global space, create functions that take a pointer to a related struct instance and manipulate accordingly. For every instance of the object, only use the related functions instead of directly accessing the data. This is to mimic the data hiding found in OOPs.

Features of OOC

- Encapsulation and data hiding (can be achieved using structs/opaque pointers)
- Inheritance and support for polymorphism (single inheritance can be achieved using structs - make sure abstract base is not instantiable)
- Constructor and destructor functionality (not easy to achieve)
- Type checking (at least for user defined types as C doesn't enforce any)
- Instead of passing pointers to structs, you end up passing pointers to pointers to structs. This makes the content opaque and facilitates polymorphism and inheritance. The real problem with OOP in C is what happens when variables exit scope. There's no compiler generated destructors and that can cause issues. MACROS can possibly help but it is always going to be ugly to look at.

Class

- The basic idea of implementing a class in C is to group the data for a C object into structs so that you can have any number of objects. The struct is declared in the .h file so that it is shared between the class and its clients. In addition, you usually need a function that initializes the objects in the class.

Methods

- If you think of methods called on objects as static methods that pass an implicit 'this' into the function it can make thinking OO in C easier.
- For example:
  - String s = "hi";
  - System.out.println(s.length());
- becomes:
  - string s = "hi";
  - printf(length(s)); // pass in s, as an implicit this

Objects

- At the most basic level, you just use plain structs as objects and pass them around by pointers.

```
struct monkey
{
   float age;
   bool is_male;
   int happiness;
};

void monkey_dance(struct monkey *monkey)
{
   /* do a little dance */
}
```

Opaque Pointers

- An **opaque pointer** is a special case of an opaque data type, a data type declared to be a pointer to a record or data structure of some unspecified type.
- Opaque pointers are a way to hide the implementation details of an interface from ordinary clients, so that the implementation may be changed without the need to recompile the modules using it.
- This benefits the programmer as well since a simple interface can be created, and most details can be hidden in another file.
- This example demonstrates a way to achieve the information hiding (encapsulation) aspect of Object-Oriented Programming using the C language. If someone wanted to change the declaration of struct obj, it would be unnecessary to recompile any other modules in the program that use the obj.h header file unless the API was also changed.

Opaque Pointer example

```
/* obj.h */

struct obj;

/* The compiler considers struct
obj an incomplete type.
Incomplete types can be used in
declarations.   */

size_t obj_size(void);
int obj_setid(struct obj *, int);
int obj_getid(struct obj *, int *);
```

```
/* obj.c */
#include "obj.h"
 struct obj {
    int id;    };

/* The caller will handle allocation.
Provide the required information only */

size_t  obj_size(void)
{ return sizeof(struct obj);  }

int obj_setid(struct obj *o, int i)
{  if (o == NULL) return -1;
       o->id = i;
       return 0;  }

int obj_getid(struct obj *o, int *i)
{ if (o == NULL) return -1;
       *i = o->id;
        return 0;     }
```

Opaque Pointer example

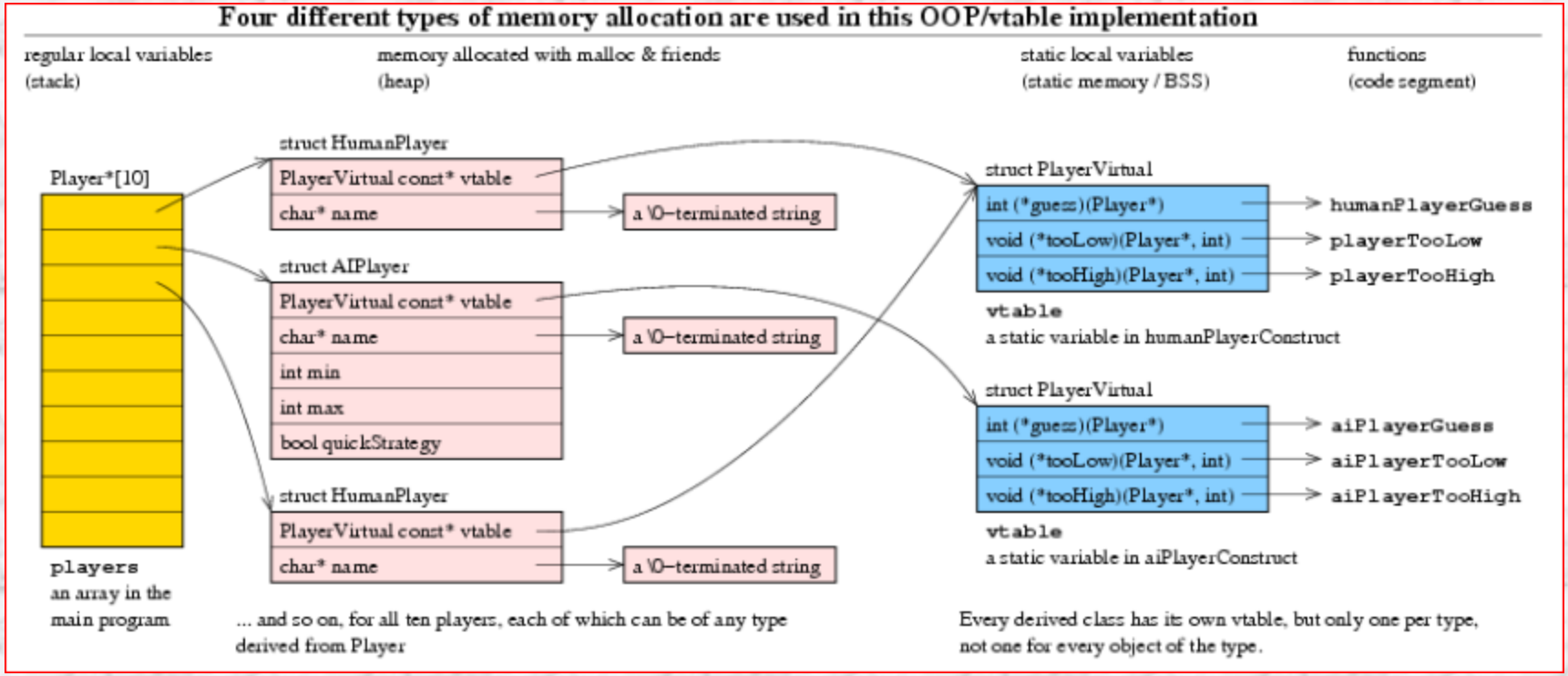
- The crux of the matter is to separate declaration from definition just like we were always told In the example of the fibheap, we want the user to have a handle to the fibheap but to otherwise not be able to affect the heap except through the API functions. So, we do this
- /* FILE: fibheap.h */
  - typedef struct FibHeap FibHeap;
  - FibHeapElement *fibHeapGetRoot(FibHeap*);
  - int fibHeapGetSize(FibHeap*);
- And that's it. There is **no** definition of the structure here, only a declaration of it. Then:
- /* FILE: fibheap.c */
  - struct FibHeap {
  - FibHeapElement *root;
  - unsigned int size; };
- Because only the declaration is in the header, whenever the user does #include "fibheap.h" he only gets the name of the struct, not the layout. Thus the user cannot access either of the fields without the accessor functions:
  - fibHeapGetRoot(FibHeap*)
  - fibHeapGetSize(FibHeap*)

Virtual Tables in C (vtables)

- A virtual table, or "vtable", is a mechanism used in Programming languages to support dynamic polymorphism, i.e., run-time method binding.
- **A more in-depth answer...**
  - Each function has an address in memory somewhere. Function names are just pretty ways of referring to a position in memory. When a program is linked (after the compiler is finished compiling) all those names are replaced with hardcoded memory addresses.

Vtables and polymorphism

- You can implement polymorphism with regular functions and virtual tables (vtables)



Inheritance

- To get things like inheritance and polymorphism, you have to work a little harder.

- You can do manual inheritance by having the first member of a structure be an instance of the superclass, and then you can cast around pointers to base and derive classes freely

```
struct base
{
   /* base class members */
};
struct derived
{
   struct base super;
   /* derived class members */
};
struct derived d;
struct base *base_ptr = (struct base *)&d;  // upcast
struct derived derived_ptr = (struct derived *)base_ptr;  // downcast
```

Define object then inherit

- Each object has its own file.
- Public functions and variables are defined in the .h file for an object.
- Private variables and functions were only located in the .c file.
- To "inherit“, a new struct is created with the first member of the struct being the object to inherit from.
- Inheriting is difficult to describe, but basically it was this:
  - struct vehicle { int power; int weight; }
- Then in another file:
  - struct van { struct vehicle base; int cubic_size; }
- Then you could have a van created in memory, and being used by code that only knew about vehicles:
  - struct van my_van;
  - struct vehicle *something = &my_van;
  - vehicle_function(something);
- It worked beautifully, and the .h files defined exactly what you should be able to do with each object.

OOC Example

```
#include "triangle.h"
#include "rectangle.h"
#include "polygon.h"
#include <stdio.h>

int main()                                  {
   Triangle tr1= CTriangle->new();
   Rectangle rc1= CRectangle->new();

   tr1->width= rc1->width= 3.2;
   tr1->height= rc1->height= 4.1;

   CPolygon->printArea((Polygon)tr1);

   printf("\n");

   CPolygon->printArea((Polygon)rc1);    }
```

Output:
6.56
13.12

This is real, pure C, no preprocessor macros. We have inheritance, polymorphism and data encapsulation (including data private to classes or objects). There is no chance for protected qualifier equivalent, that is, private data is private down the inheritance chain too; but not an inconvenience because not necessary.

CPolygon is not instantiated because we only use it to manipulate objects of down the innheritance chain that have common aspects but different implementation of them (Polymorphism)