

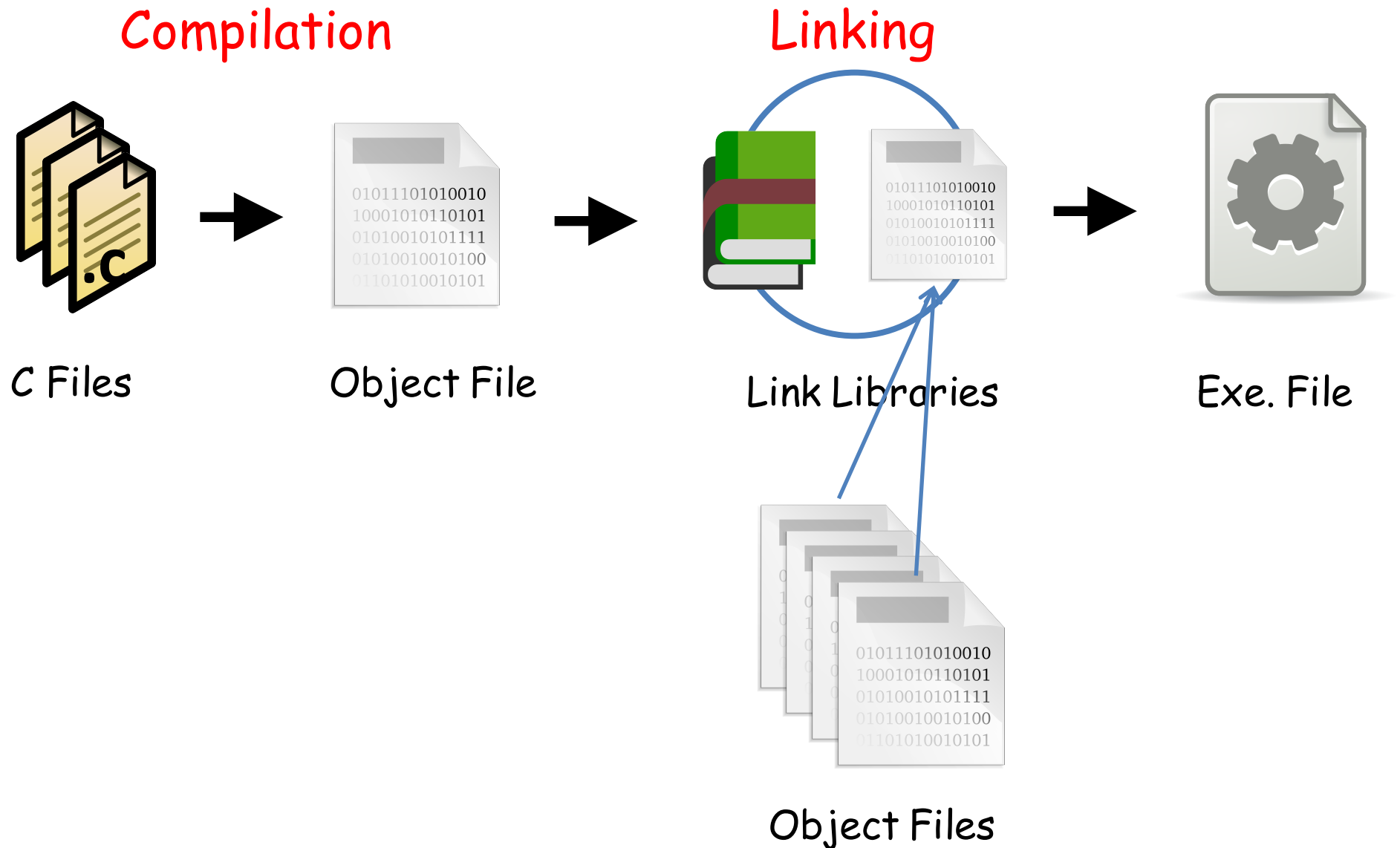
# Computer Systems Principles

Library functions and API design

# Learning Objectives

- Learn to compile static libraries in C
- Learn to write header files
- Learn to define Macros
- Learn the uses of different library functions in C
- Understand error codes and check functions
- Learn scoping rules to design better C programs and the use of static and external variables and static functions.
- Learn the use of pointers to pointers (double pointers)

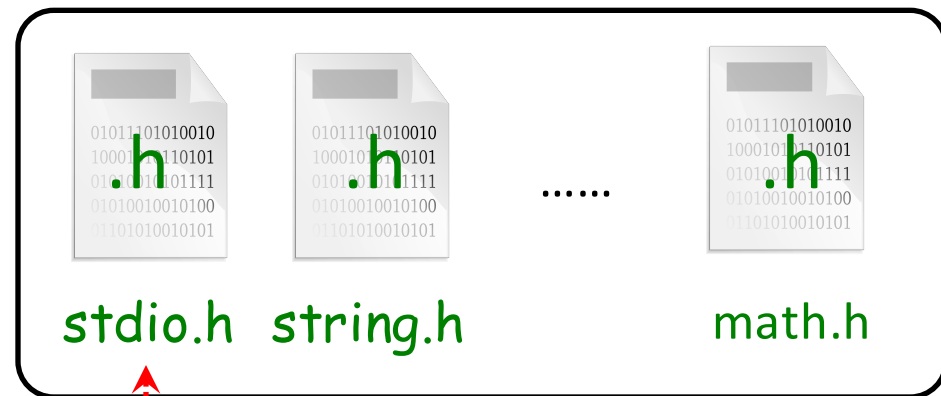
# Compiling a C program



# Library Functions

- **stdio.h:**
  - `printf()`
- **string.h:**
  - `strncpy()`, `strncmp()`
- **stdlib.h**
  - `malloc()`, `free()`
- **math.h:**
  - `pow()`, `sqrt()`.
- **time.h**
- **assert.h**
- **errno.h**
- **check.h**

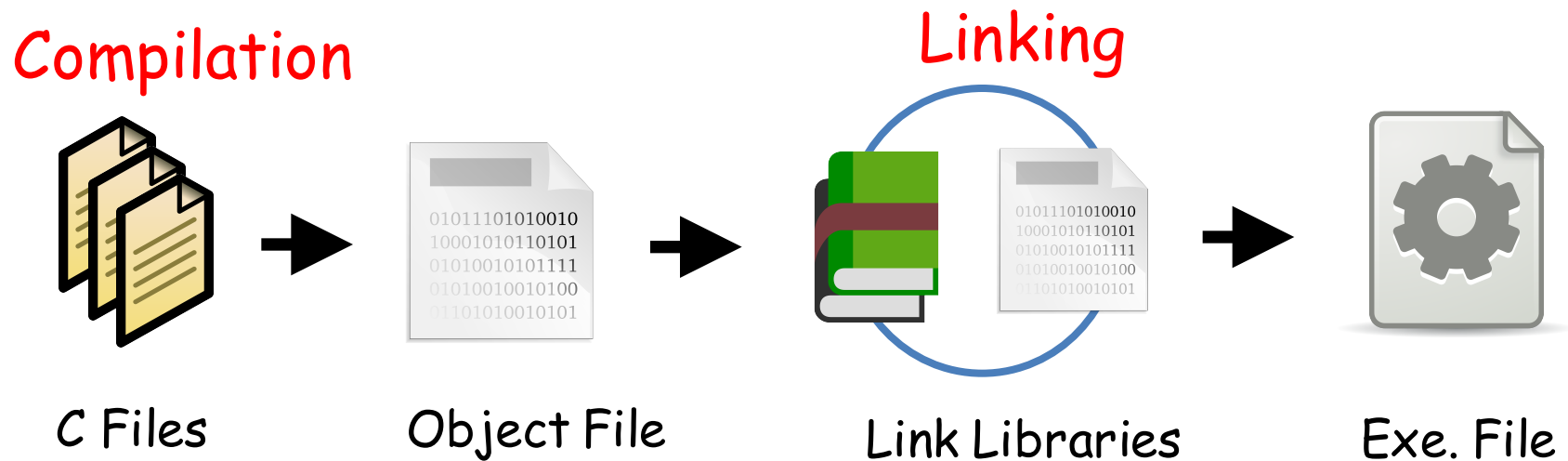
## Standard C Library



```
#include <stdio.h>
int main(){
....printf()
```

# What is a library?

- A library is a collection of precompiled object files which can be linked into programs.
- C Libraries provide system functions
- Today we will look at **Static Libraries**



# What is a library?

- Libraries that stored in special archive files with the extension “.a” are referred to as **static libraries**.
- They are stored usually at /usr/lib or /lib
  - The C math library: /usr/lib/libm.a (Unix)
  - The C standard library at /usr/lib/libc.a (Unix)
  - Note how many are named libxxx.a

# C Math library <math.h>

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

```
#include <math.h>
```

```
int main (void)
```

```
{
```

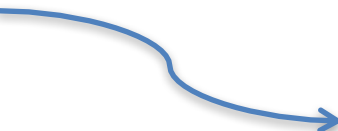
```
    double x = pow (4, 0.5);
```

```
    printf ("The square root of 2.0 is %f\n", x);
```

```
    return 0;
```

```
}
```

will not compile without  
#include. Remember to  
use -Wall to see all  
warnings generated!



Let's try creating an executable from this source file.

- \$ gcc -Wall calc.c -lm -o calc

# C Math library <math.h>

## Compiling

- Reference to 'pow' function cannot be resolved without external math library 'libm.a'.
- 'pow' not defined in default library 'libc.a'
- compiler provides a shortcut '-l' for linking libraries
  - `$ gcc -Wall calc.c -lm -o calc`
  - equivalent to selecting '/usr/lib/libm.a'
- In general, the compiler option **-lNAME** will attempt to link object files with a library file 'libNAME.a' in the standard library directories.
- **-L{path to file containing library} -l{library name}**



# iClicker question

What library is referred to in this compilation?

```
gcc foo -liberty foo.c
```

- A. One containing the *free* function
- B. One from the Free Software Foundation
- C. One called libiberty.a
- D. One called liberty.a
- E. More than one of the above

# iClicker question

What library is referred to in this compilation?

```
gcc foo -liberty foo.c
```

- A. One containing the *free* function
- B. One from the Free Software Foundation
- C. One called libiberty.a**
- D. One called liberty.a
- E. More than one of the above

# Link order of libraries

- Linkers search for external functions **from left to right** in the libraries specified on the command line.
- A library containing the definition of a function should appear **after** any source file or object files which use it.
- `$ gcc -Wall calc.c -lm -o calc` (**correct order!**)
- `$ gcc -Wall -lm calc.c -o calc` (incorrect order)
  - There is no library or object file containing `pow()` after ``calc.c'!`
- If we had a graphics library which in turn uses the math library:
  - `$gcc -Wall data.c -lgraph -lm` since the object files in ``libgraph.a'` use functions defined in ``libm.a'`.

# Using library header files

- The `#include <filename.h>` or `#include "filename.h"`
  - contents of filename.h to be **read, parsed and compiled** at that point.
- Essential to include appropriate header files in order **to declare function arguments** and return values with **correct types**.
- Without declarations: arguments of a function can be passed with the wrong type causing corrupted results.

# C Preprocessor

- The “preprocessor” is a translation phase that is applied to your source code before the compiler gets hands on it.
- The preprocessor performs textual substitutions on your source code in three ways:
  - **file inclusion**: inserting the contents of another file into your source file (as if you had typed it all in there).
  - **macro substitution**: replacing instances of one piece of text with another
  - **conditional compilation**: depending on various circumstances, certain parts of your source code are seen or not seen by the compiler at all.

# C Preprocessor: Syntax

The syntax of the preprocessor is **different** from the syntax of C:

- the preprocessor **is line-based**: each directive must:
  - begin- at the beginning **and**
  - end - at the end of a line.
- the preprocessor **does not know** about the structure of C!
  - functions, statements or expressions.

# C Preprocessor: File inclusion

- The `#include <filename.h>` or `#include "filename.h"`
  - contents of filename.h to be **read, parsed and compiled** at that point.
- After "filename.h"/<filename.h> is processed, compilation continues on the line following the `#include` line.
- Example: look at "wset.h" and "wset.c" from the wordfreq homework. What do you observe?

# C Preprocessor: File Inclusion

- The extension “.h” simply stands for “header”
- `#include` directives usually sit at the **top (head)** of your source file.
  - `<>`: standard directories
  - `“”`: current directory (files you’ve written)
  - Both are actually *paths* (lists of directories to search), `<>` uses the *system* library path and `“”` uses the *user* library path (then the system one)



# C Preprocessor: File inclusion

What should a header file do?

- Allow pulling that information into several different source files **consistently**.
- Be a place for common declarations and definitions.
  - more workable system
- The header file may provide:
  - declarations for functions, **but does not give the compiler the function bodies themselves!**
- Using a library is a two step process:
  - must **#include** header (where the function is called)
  - must tell the **linker to read in the functions** from the library

# What should you put in header files?

- External **declarations of** global variables and functions – ones a library client may want to use
- A global variable must have **exactly one** defining instance but it can have external declarations in many places.
- Check that the declaration and definition **match**.
- Preprocessor **macro** definitions.
- **Structure** definitions.
- **Typedef** declarations.

# Conditional compilation

```
>> #ifdef name //stands for if defined
```

```
    – program text
```

```
>> #else //optional
```

```
    – more program text
```

```
>> #endif
```

- #ifdef controls which parts of your program actually **get compiled**.
  - looks like an if statement: but behaves differently
- write a **portable program**, (but operations vary based on compiler, OS, or the computer you are using).
- compile several **different versions** with different features present in different versions.

# Conditional compilation

```
>> #ifndef name //(stands for if not defined)
    – program text //compiles code if the macro is not defined
>> #else
    – more program text
>> #endif
```

- Useful when we call the same header file in multiple places
- take a look at “hw3.h”, “wset.h” and “wset.c” from the wordfreq homework.
- take a look at check.h source at [check.sourceforge.net](http://check.sourceforge.net)
  - notice the `ifndef–endif` and `ifdef–endif` sections for various parameters

# Macro definition and substitution

- **#define** <name> <text> (no semicolon!)
  - defines a macro with name and its value is the replacement in text.
  - replacement text can be anything (not restricted to numbers, simple strings)
- Common use: propagate constants
- No evaluation is performed when the macro is defined!
  - If replacement text is an expression: text is substituted and evaluated later

# Macro definition and substitution

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

```
#define A 2
```

```
#define B 3
```

```
#define C A+B
```

```
int main(void){
```

```
    int x,y;
```

```
    x = C*2;
```

```
    printf("Value of C*2 is %d\n", x);
```

```
    return 0;
```

```
}
```

Output:?

a)10

b)5

c)8

d)Error

# Macro definition and substitution

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

```
#define A 2
```

```
#define B 3
```

```
#define C A+B
```

```
int main(void){
```

```
    int x,y;
```

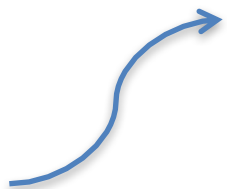
```
    x = C*2;
```

```
    printf("Value of C*2 is %d\n", x);
```

```
    return 0;
```

```
}
```

# define C (A+B) would  
give the desired output  
of 10. Always enclose  
#define replacement in  
( )!



Output:?

a) 10

b) 5

c) 8

d) Error

# Things not to put in header files

- Defining instances of global variables.
  - variable multiply defined!
- Function bodies (also defining instances)
  - function multiply defined!



# Lets look at some library functions..

Library	Function
<code>math.h</code>	define mathematical operations and constants
<code>string.h</code>	perform string manipulations
<code>ctype.h</code>	convert and test single characters.
<code>time.h</code>	read and convert time and date values.
<code>stdio.h</code>	standard input/output operations
<code>stdlib.h</code>	integer math, text conversions, storage allocation, environmental interactions *
<code>assert.h</code>	writes diagnostic information to the std. error
<code>errno.h</code>	general error handler with error code Macros
<code>check.h</code>	unit testing framework for C

\* The standard C library refers to “stdlib” header as `a hodgepodge of functions with no sensible home’!

# C Math Library

C Math library contains several math functions and several mathematical constants.

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    printf("HUGE_VALF = %f\n", HUGE_VALF);
    printf("HUGE_VAL = %f\n", HUGE_VAL);
    printf("HUGE_VALL = %Lf\n", HUGE_VALL);
    printf("M_E = %f\n", M_E);
    printf("M_PI = %f\n", M_PI);
}
```

# C Math Library

```
#include <stdio.h>
#include <math.h>

int main(void)
{
    int a = 1;
    float b = 0.5;
    long double ld = 1;
    double complex dc = 1 + 0.5*I; // -> capital (I) to indicate imaginary number
    double complex result = csqrt(dc);

    printf("cos(1) = %f\n", cos(i)); // -> cosine of I in radians
    printf("sin(0.5) = %f\n", sin(f)); // -> sinf(f)
    printf("acos(1) = %Lf\n", acos(ld)); // -> principle value of the arc cosine of x
    printf("Conjugate of (1 + 4i) = %f+%fi\n", creal(conj(dc)), cimag(conj(dc)));
    printf("Sqrt of (1 + 4i) = %f+%fi\n", creal(result), cimag(result));
}
```

# C String Library <string.h>

- The string comparison functions return a nonzero value if the strings are not equivalent rather than if they are (Unlike most comparison operations in C!).
  - a negative value indicates that the first string is "less" than the second, while a positive value indicates that the first string is "greater".
- `strcmp` compares two strings. `strcat()` concatenates strings. It **does not** give a third string,
  - extra space to append or overflow!
- `strlen` returns the length of the string not including `\0`.
- string functions **return type pointer to character**.

# C String Library

Overview of some **basic memory operations**: They are not strictly string functions , but are prototyped in <string.h>

```
int memcmp( const void* left, const void*right, size_t count );
```

- Compares the *count* bytes of memory beginning at *left* against the *count* bytes of memory beginning at *right*.
- Be careful using to compare objects **containing "holes"**:
  - structures (padding to enforce alignment), extra space at end of unions, and extra characters at the ends of strings whose length is less than their allocated size.
- The contents of these "holes" are indeterminate - may cause strange behavior when performing byte-wise comparisons.
- For more predictable results, perform explicit component-wise comparison.

# C String Library

Overview of some basic memory operations: They are not strictly string functions , but are prototyped in <string.h>

```
void* memcpy( void *dest, const void *src, size_t count );
```

- memcpy copies *count* bytes from the object beginning at *dest* into the object beginning at *dest*.
- The behavior of this function is undefined if the two arrays *dest* and *src* overlap (just as for strcpy); use memmove instead if overlapping is possible.

```
void* memchr( const void* ptr, int ch, size_t count );
```

- The memchr function locates the first occurrence of *ch* (converted to an unsigned char) in the initial *count* characters (each interpreted as unsigned char) of the object pointed to by *ptr*.

# C Library <ctype.h>

- Related library: `#include <ctype.h>` which contains many useful functions to convert and test single characters. (ctype = character type, not C type)
- The use of some of these functions are :
  - `int toascii(int a)` – Convert a to ASCII (zero 0x80 bit).
  - `int tolower(int a)` – Convert a to lowercase.
  - `int toupper(int a)` – Convert a to uppercase.
  - `int isalnum(int a)` -- True if a is alphanumeric.
  - `int isdigit(int a)` -- True if a is a decimal digit .
  - `int isprint(int a)` -- True if a is a printable character.

# C Library <ctype.h>

```
#include <stdio.h> // -> for printf() and scanf() functions
#include <ctype.h> // -> for isalnum()
int main(void) {
    char c;
    printf("Enter a character: ");
    scanf("%c",&c); —————> &(variable) when using scanf
    if (isalnum(c)==0) —————> returns "True" if
        printf("%c is not an alphanumeric character.\n", c);
    else
        printf("%c is an alphanumeric character.\n", c);
    return 0;
}
```



# C Library <ctype.h>

```
#include <stdio.h>
#include <ctype.h>
int main(void){
    char c;
    c='C';
    printf("Upper case: isupper returns %d\n",isupper(c));
    c='c';
    printf("Lower case: isupper returns %d\n",isupper(c));
    return 0;
}
```

# C Library <time.h>

- C library #include <time.h> contains functions to read and convert time and date values.
- A time value is stored in a time\_t struct
- The use of some of these functions are :
  - double difftime(time\_t t1, time\_t t2) – difference between two times, in seconds, returned as a double
  - char \*asctime(const struct tm \*t) – converts a (different) representation of time to a string
  - size\_t strftime(char \*s, size\_t max, const char \*format, const struct tm \*timep) – formats a time, the way you want it, into a bounded buffer

# C Library <time.h>

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

```
int main(void)
```

```
{
```

```
    srand(time(0));
```

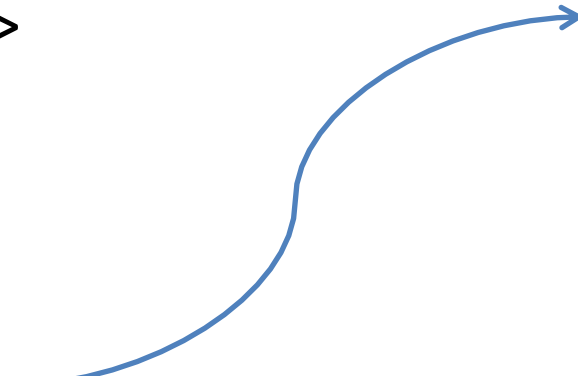
```
    int rv = rand();
```

```
    printf("Random value on [0,%d]: %d\n", RAND_MAX,
           rv);
```

```
    return 0;
```

```
}
```

use current time as  
seed for pseudo  
random numbers to  
subsequent calls to  
rand().



Constant value  
defined in stdlib.h



# C Library <stdio.h>

## Basic I/O

- There are a couple of function that provide basic I/O facilities.
- Probably the most common (after printf) are: `getchar()` and `putchar()`. They are defined and used as follows:
- `int getchar(void)` -- reads a char from stdin; note: EOF is -1, hence the use of an int return value
- `int putchar(char ch)` -- writes a char to stdout, returns character written.
  - `int ch; ch = getchar(); (void) putchar((char) ch);`
- Related Functions:
  - `int getc(FILE *stream),`
  - `int putc(char ch, FILE *stream)`

# C Library <stdlib.h>

`malloc()`, `realloc()` (return type `void*`), `free()`

- no dynamic storage classes in C – only way to refer to dynamically allocated space is through pointers.
- If no more space is available, `malloc` returns a **null pointer**.
  - always check the value of every call to `malloc`!
  - try writing a subroutine that calls `malloc` and reports an error if the value is a null pointer. Return only if the value is nonzero.
- liable to destroy the data if we go past the allocated memory space: to assign additional space use `realloc`.
- `EXIT_SUCCESS`, `EXIT_FAILURE`
  - expand to integral expressions - may be used as arguments for successful or unsuccessful termination for the `exit` function and also from the `main` function.

# C Library <assert.h>

## Diagnostic Information

- The C library macro `void assert(int expression)` writes diagnostic information to the standard error.
- `void assert(int expression);`
- Assertion failed (argument 0): expression, filename, line-number.
- Assertions help document the assumptions behind the code and debug code.
- Can be turned off for speed (avoids evaluating 'expression')

# C Library <errno.h>

- `errno` is a general error code facility.
- The error code macros expand into integer constant values.
- The functions `strerror` and `perror` give you the standard error message for a given error code; the variable `program_invocation_short_name` gives you convenient access to the name of the program that encountered the error.
- The error code values are all positive integers and are all distinct. Should not make any other assumptions about the specific values of these symbolic constants!

# C Library <errno.h>

Common mistake..

```
if (somecall() == -1) {  
    printf("somecall() failed\n");  
    if (errno == ...) { ... }  
}
```

where `errno` no longer needs to have the value it had upon return from `somecall()` (i.e., **it may have been changed by the `printf(3)`!**). If the value of `errno` should be preserved across a library call, it must be saved:

```
if (somecall() == -1) {  
    int errsv = errno;  
    printf("somecall() failed\n");  
    if (errsv == ...) { ... }  
}
```

- **Do not declare `errno` manually!**



# C Library <errno.h>

`char * strerror (int errnum)` (declared in `string.h`)

- maps the error code from `errnum` (from `errno`) argument to a descriptive error message string.
- returns pointer to the string.
- subsequent calls to `strerror`, the string might be overwritten. (But no library function ever calls `strerror` behind your back!)

`void perror (const char *message)`

- prints an error message to the stream `stderr`.
- call to `perror` with null pointer or an empty string prints message corresponding to `errno`.
- with non-null message argument, `perror` prefixes output with this string.

# C Library <check.h>

- unit testing framework for C.
- The basic unit test framework:

```
START_TEST (test_name)
```

```
{
```

```
    // unit tests
```

```
}
```

```
END_TEST //macros to setup testing structure
```

- `ck_assert`: takes Boolean argument (True = Pass)
- will print: file, line number, and the message
- `ck_assert_int_eq`; `ck_assert_str_eq` (int/string equality)
- `ck_assert_msg`: same as `ck_assert` with user defined message.

# C Library <check.h>

wordfreq homework public tests:

- Each start-end test pair specifies the function it tests.
  - eg: test\_wset\_add: tests the size of wset as words are incrementally added.
- Once all the tests are defined a tester\_suite() is created. We add each of the tests we have built.
- In main() create a suite (\*s) and SRunner sr (holds the state of running suite). Print the results of the tests using CK\_Normal flag to view all failed tests.

# Static Variables

- **Automatic** allocation - when you declare a function argument or **a local variable**.
- **Static** allocation - when you declare **a static variable**.
- Static variables are for the **private use** of the functions in their respective source files and are not meant to be accessed by anywhere else.
  - Internal static variables are local to a particular function but remain in existence rather than coming and going each time the function is activated.
  - Internal static variables provide private, permanent storage within a single function.
  - a static variable is initialized only the first time the block is entered.

# Static Variables

- External static variables provides a way to hide variable names which must be external:
  - they can be shared
  - but are not visible to users of those functions.
- If an external static variable is compiled in one file then:
  - other routines will not be able to access those variables
  - those names will not conflict with the same names in other files of the same program.

# External variables

- A C program consists of **a set of external objects** that are either variables or functions.
- External variables are defined outside of any function and potentially available to many functions. They may be defined in a separately compiled file (such as a library)
- **Functions are always external**
  - (C does not allow nesting of functions).
- *By default external variables and functions have the property that all references to them by the same name, even from functions compiled separately are references to the same thing. This property is called **external linkage**.*

# External Variables

- External variables are **permanent**, so they retain values from one function invocation to the next.
- A variable is external if it is defined outside of any function.

# Static Functions

- Functions are in C are normally global and visible to the entire program! External static declarations can be applied to functions as well.
- If a function is declared static its name is invisible outside of the file in which it is declared.
- The static declaration on an external variable or function limits the scope of that object to the rest of the source file being compiled.



# Constant Variables

- `const` qualifier can be applied to the declaration of any variable to specify that its **value will not be changed**.
  - integer, floating point, enumeration constants
  - `const double e = 2.71828...`
  - `const char msg[] = "warning"`
- `const` declaration can also be used with array arguments to indicate that the function does not change that array.
  - `func(const char[]); func(const *char);`
  - `int strlen(const char[])`

# Side note on data types

- Data types `char` and `int` of all sizes with and without sign, `enumerations` are all enumeration types (collectively called `integral types`).
- `Float`, `double` and `long double` are called `floating types`.
- `Void` type specifies an `empty set of values`. Used as the type returned by functions that generate no value.
- Arrays of objects, functions, pointers, structures, unions are `derived data types`.
  - these methods of constructing objects can be applied recursively

# Scope Rules

- Local variables of the same name in different functions are **unrelated**.
- The scope of an external variable or function lasts from the point at which it is declared to the end of the file being compiled.

# Scope Rules: External variables

- Important distinction between the **declaration** of an external variable and **its definition**.
- A declaration announces the properties of a variable (its type).
- A definition causes storage to be set aside.
- **Only one definition** of an external variable among all the files that make up the source program.
- Initialization of an external variable goes only with the definition.

# Scope: Block Structure

- C does not allow nesting of functions (functions within functions).
- But, variables can be defined in a block-structure within a function.
- Variables declared inside a block structure **hide any identically named variables in outer blocks** and remain in existence until the block is closed.
  - Generally a bad idea to use similarly named variables in nested blocks – leads to confusion!

# Scope: Block structure (bad practice)

```
int x;  
int y;  
f(double x)  
{  
    double y;  
}
```

- within f() occurrences of x
  - refer to the parameter `double x`
- outside of f occurrences of x
  - refer to the external `int x`.
- same is true of y.
- avoid variable names that conceal names in outer scope!

# Additional Notes on Pointers

## Pointers of type `void*`

- Any pointer to an object may be converted to type `void*` without loss of information.
- That is: if the result is converted back to the original pointer type: the original pointer is recovered.

# Pointers of type `void*` `malloc()`

- `void *malloc (size_t, n)`
- `malloc` obtain blocks of memory dynamically
  - returns pointer to `n` bytes of uninitialized storage
  - returns `NULL` if no space is available
- error to use something after it has been freed!

WRONG!

```
for ( p = head; p != NULL; p = p->next)
    free(p);
```

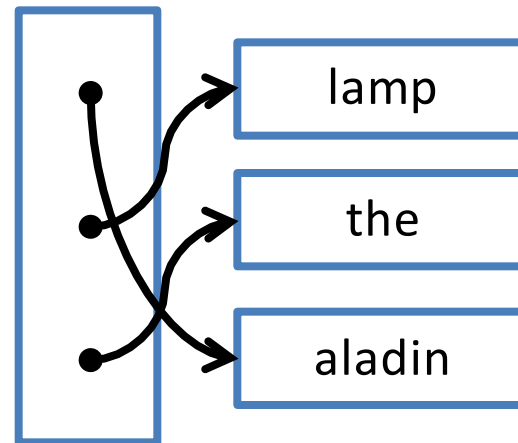
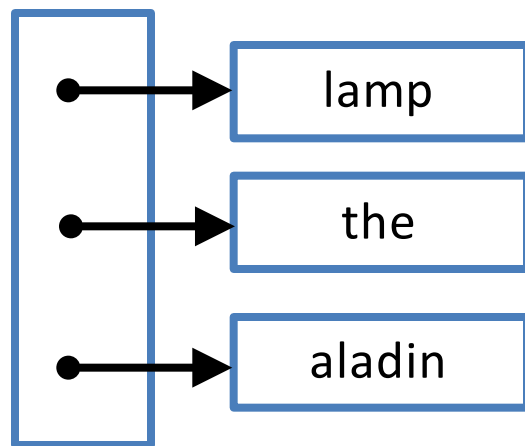
CORRECT!

```
for ( p = head; p != NULL; p = q)
{
    q = p->next;
    free(p);
}
```



# Pointer to a pointer (Double pointer)

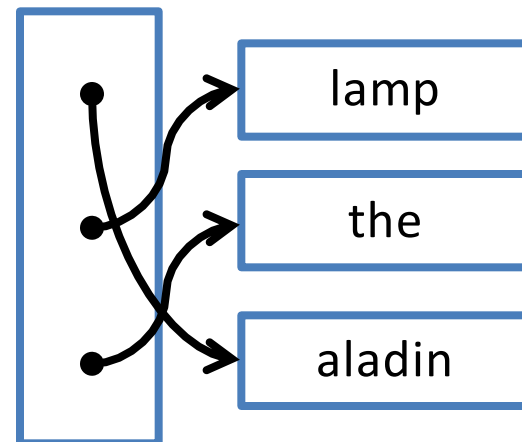
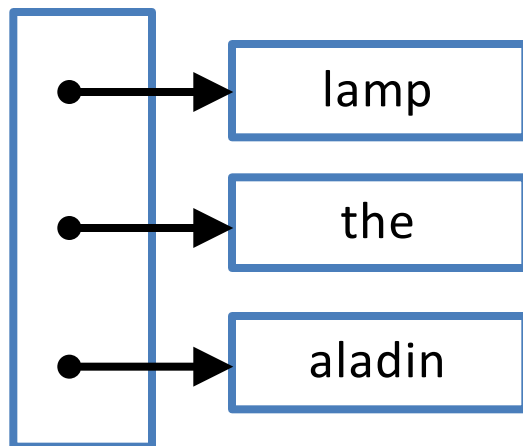
- Use case: we want to sort lines of text.
- each line can be accessed by a pointer to its first character.
- the pointers themselves can be stored in an array
- Two lines can be compared by passing their pointers to `strcmp()`



# Pointer to a pointer

## (Double pointer)

- Exchanging two lines: the pointers in the pointer array are exchanged not the text lines themselves!
- Eliminates problems of:
  - complicated storage management
  - high overhead of moving the lines themselves.



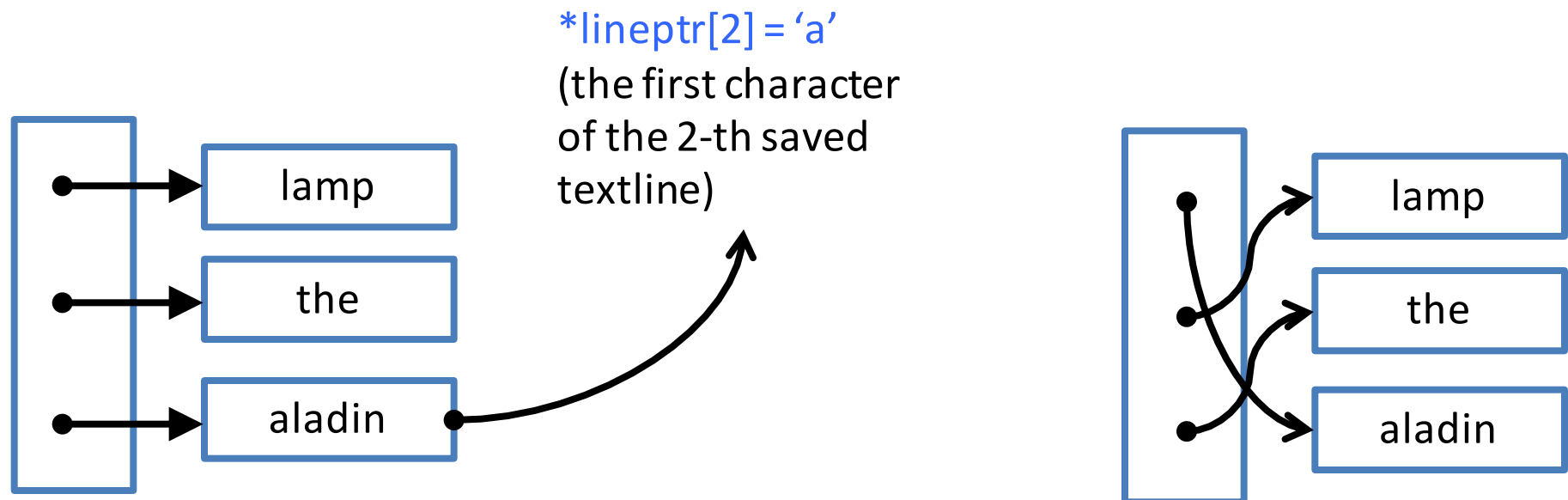
# Pointer to a pointer (Double pointer)

```
char *lineptr[MAXLINES] //pointers to text lines
```

↓

lineptr is an array of MAXLINES elements, each element of which is a pointer to a char.

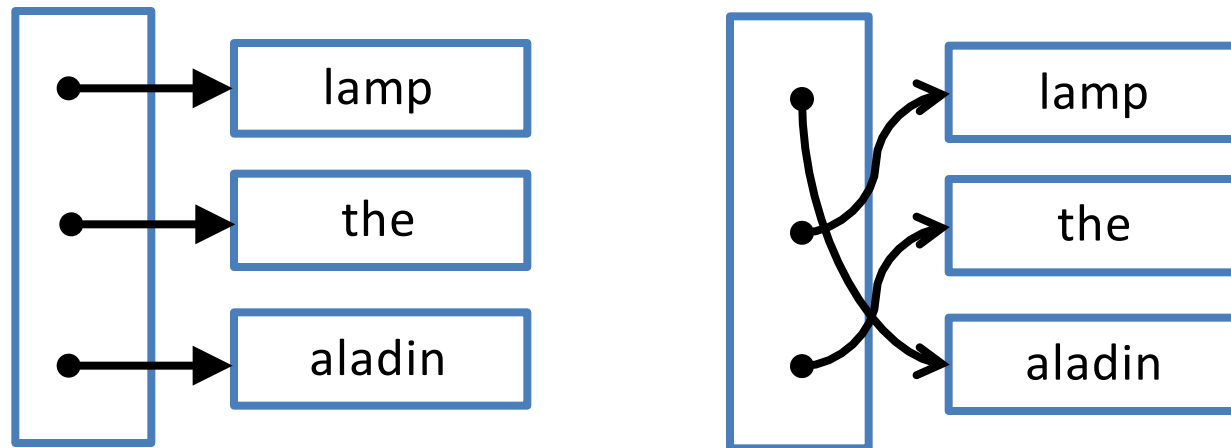
lineptr[i] is a character pointer



# Pointer to a pointer (Double pointer)

```
char *lineptr[MAXLINES] //pointers to text lines
```

- `lineptr` itself is the name of an array it can be treated as a pointer in the same manner.
- Initially `*lineptr` points to the first line;
- Each increment advances it to the next line pointer while number of lines is counted down.



# Pointer to a pointer (Double pointer)

- Swapping function - since each element of `lineptr` is a **character pointer**, `temp` must be also, so one can be copied to the other.

```
void swap(char *w[], int i, int j)
```

```
{
```

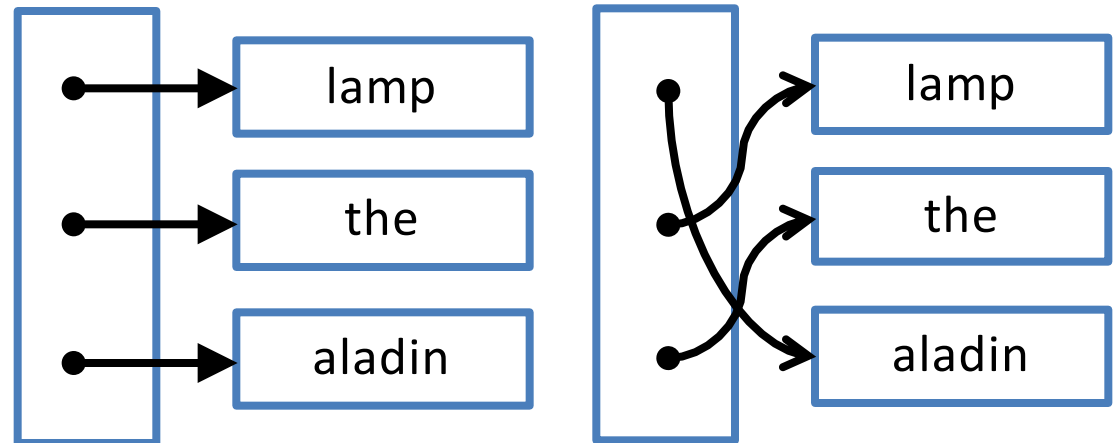
```
    char *temp;
```

```
    temp = w[i];
```

```
    w[i] = w[j];
```

```
    w[j] = temp;
```

```
}
```



# Next Class

- Dynamic Libraries!