**GRADES ARE OUT!**

**ITS OVER :) :((((((**

**since everyone’s anonymous post your grade below so we can get an idea of what the average is**

**ANONYMOUS GRADES**

**93**

**60**

**73**

**70**

**83**

**75**

**95**

**88 = 88/110 = 80%**

**102/110 → ~92%**

**86/110**

**95/110**

**….damn everyone did better than me… and i thought i did well coming out of the exam**

**Is this out of the 120, or the 110?**

**\*\*\*anyone remember what proff said yesterday about lseek filling with 0s in the gap?**

**i looked it up on the linux man and it says that if you have a gap, subsequent reads of the fill in the gap will return null bytes**

**anyone find the same thing <- found the same- M.S.**

**Regarding L-Seek:**

**with L-seek, you can go past the “end” of a file when writing. When you do this, the OS actually lets you and just fills in the gap between the end of the file and where you write beyond it with 0s.**

**did the professor say something about “he is still coming up with a question regarding makefile”?**

**I don’t think he’s going to put it on this exam, but yes he did say that -- thanks!**

**Information hiding: opaque types, generic pointers, separation of internal implementation details from external functional interface**

**anyone have any ideas?**

* **encapsulation:**
* **abstraction:**

isolating design decisions to certain parts of code:

* info that user doesn’t need to know
* dealing with modification protection

consistent with principle of least knowledge

isolate design decisions to single modules

each module knows or assumes as little as possible about the other modules

Advantages of info hiding:

1)flexible code- modify code and know that nobody knows what you’re doing but you

2)intellectual manageability

**JCaverly - There really isnt much on this. It’s just a great way to hide data so that the user never knows anything inside the struct/data. Just keep passing pointers to functions that know how to internally deal with said given data.**

**directories: structure, I-nodes - NOT IMPORTANT/NOT ON EXAM**

The directory structure is the organization of files into a hierarchy of folders. It should be stable and scalable; it should not fundamentally change, only be added to. Computers have used the folder metaphor for decades as a way to help users keep track of where something can be found.

l-nodes: whuuuut is this

**Did he ever mention function type macros in class? or just object type? and macro expansion?**

**synchronous events inside process, asynchronous events outside process - what?**

**Debuggers : does not actually debug your code, it's just a viewer. They allow you to step through your program as its executing, in order to find out what is wrong with it.**

**self explanatory?**

**Not drowning yourself in debugging output.**

**also self explanatory? or is there more he mentioned in class?**

**Debuggers:** It’s a viewer in the sense that it allows you to see the “symbols” you load with the object file. that’s what the -g flag is for when you compile and build your object file from your .c and .h files.

**Not Drowning yourself in debugging output:** you can use the conditional compilation approach or use a debugger.

Conditional Compilation:

#IF DEBUG == 3

printf(“var x is %d FILE %s LINE %d.\n, x, \_\_FILE\_\_, \_\_LINE\_\_);

#ENDIF

gcc -DDEBUG=3 myprog.c

this defines DEBUG to have value 3 and that is the only time it would include the printf line

**Binary search approach to isolating bugs.**

**-The idea behind this is you keep manually “halving” your code into sections to test in order to pinpoint where the bug is coming from, instead of sequentially checking every single possible point/line starting from the beginning of your code. ji9p**

**Multiprogramming:**

**Multiprogramming**

**One time, I was at the post office standing in line waiting my turn to be served. My turn came but I was not fully prepared because my mail was not prepackaged. They gave me an empty box to do that on the side. I started packaging my mail while another customer is occupying my spot. It does not make sense to block the whole line while packaging my mail however it is a better idea to allow other customers proceed and get served in the meantime. I think this example (to some extent) is very similar in concept to multiprogramming model where programs are like customers and CPU is like the post office assistant. Assuming one assistant (single processor system) then only one customer can be served at a time. While a customer is being served he or she continues until he or she finishes or waits on the side. As long as the assistant is helping a customer he does not switch to serve other customers.**

**In a multiprogramming system there are one or more programs (processes or customers) resident in computer’s main memory ready to execute. Only one program at a time gets the CPU for execution while the others are waiting their turn. The whole idea of having a multi-programmed system is to optimize system utilization (more specifically CPU time). The currently executing program gets interrupted by the operating system between tasks (for example waiting for IO, recall the mail packaging example) and transfer control to another program in line (another customer). Running program keeps executing until it voluntarily gives the CPU back or when it blocks for IO. As you can see, the design goal is very clear: processes waiting for IO should not block other processes which in turn wastes CPU time. The idea is to keep the CPU busy as long as there are processes ready to execute.**

**Note that in order for such a system to function properly, the operating system must be able to load multiple programs into separate partitions of the main memory and provide the required protection because the chance of one process being modified by another process is likely to happen. Other problems that need to be addressed when having multiple programs in memory is fragmentation as programs enter or leave (swapping) the main memory. Another issue that needs to be handled as well is that large programs may not fit at once in memory which can be solved by using virtual memory. In modern operating systems programs are split into equally sized chunks called pages but this is beyond the scope of this article.**

**In summary, Multiprogramming system allows multiple processes to reside in main memory where only one program is running. The running program keeps executing until it blocks for IO and the next program in line takes the turn for execution. The goal is to optimize CPU utilization by reducing CPU idle time. Finally, please note that the term multiprogramming is an old term because in modern operating systems the whole program is not loaded completely into the main memory.**

**VS**

**y**

**Multi Threading**

**Before we proceed, let us recap for a minute. Multiprogramming refers to multiple programs resident in main memory and (apparently but not exactly) running at the same time. Multitasking refers to multiple processes running simultaneously by sharing the CPU time. Multiprocessing refers to multiple CPUs so where does multi threading fit in the picture.**

**Multi threading is an execution model that allows a single process to have multiple code segments (threads) run concurrently within the context of that process. You can think of threads as child processes that share the parent process resources but execute independently. Multiple threads of a single process can share the CPU in a single CPU system or (purely) run in parallel in a multiprocessing system. A multitasking system can have multi threaded processes where different processes share the CPU and at the same time each has its own threads.**

**The question is why we need to have multiple threads of execution within a single process context. Let me give an example where it is more convenient to have a multi threaded application. Suppose that you have a GUI application where you want to issue a command that require long time to finish for example a complex mathematical computation. Unless you run this command in a separate execution thread you will not be able to interact with the main application GUI (for example updating a progress bar) because it is going to be frozen (not responding) while the calculation is taking place.**

**Multi threading is a smart way to write concurrent software but it also comes with a price because the programmer has to be aware of race conditions when two or more threads try to access a shared resource and leave the system in an inconsistent state or a deadlock. Thread synchronization (for example using locks or semaphores) is used to solve this problem which is beyond the scope of this article.**

**printing stuff in color**

**set color with different leading and trailing hex values?**

**does anyone know how he did this in class because there are a thousand different ways to print something in color online**

[**http://linux.byexamples.com/archives/184/print-text-in-colors-with-a-simple-command-line/**](http://linux.byexamples.com/archives/184/print-text-in-colors-with-a-simple-command-line/)

**NOT ON THE EXAM**

**What the -g compilation option does : generate source-level debugging info.**

**kill, detach debugged process.**

**differences between executable file and executing process/things process have that executable files do not have**

* **Program:** Program is a set of instructions which is in human readable format.(HelloWorld.c)
* **Executable:** Executable is a compiled form of a Program (HelloWorld.exe file)
* **Process:** Process is the executable being run by OS. The one you see in Task Manager or Task List (HelloWorld.exe Process when we double click it.)

compilation system/executable process? - \*from textbook\*

1st have source program (text) in form of a .c file (ex. hello.c)

step 1: Preprocessing Phase - .c file goes through the preprocessor (cpp)

here, (cpp) modifies original C program according to directives that begin with #

ie all of the #include parts

the #include <stdio.h> line of a prgm would tell preprocessor to read

contents of the system header file stdio.h and insert it into the prgm text

result is new program with the .i suffix

after this step runs, have modified source program file that is still a text file (ex. hello.i)

step 2: Compilation phase - .i file goes through the compiler (cc1) that translates it to a .s file

text modified source code becomes the assembly text file

that describes one low-level machine lang instruction in a standard text form

after this runs have hello.s that is the assembly program text

step 3: Assembly phase - .s becomes .o through the assembler(as)

translates assembly into relocatable object program in a .o file

first occurrence of a binary file in the process

this binary encodes the assembly instructions rather than characters so if tried to view

it in text editor, just looks like gibberish

after this phase runs, have hello.o - relocatable object program (binary)

step 4: linking phase - links/merges .o file with any other .o file the program calls (ld)

ie in hello world program that calls printf from standard C library, need to link

hello.o with printf.o

after this phase runs, the .o file becomes the executable object file (binary)

executable \*object\* file

running a program in C involves translating original source code to executable object

program that gets stored as a binary disk file

executable object file comes from the last step (linking) of the compilation system

readily loads to memory and executes by system

**what is an address space**

virtual address space (from textbook) -

set of all possible addresses;

conceptual image presented to machine-level program - organized as array of

N contiguous byte-sized calls stored on disk

each byte has unique virtual address that serves as index to array

contents of array on disk cached in main memory

actual implementation uses combo of RAM, disk storage, special hardware, and operating

system software to provide the program with what appears to be a monolithic byte array

size of Virtual address space determined by the word size of the computer

eg. machine with w--bit word size has a virtual address space that can range from

0 to 2^w - 1 giving a program access to at most 2^w bytes

most personal computers limited to 4 GB with 32 bit word size (says the book)

every program only has access to a few megabytes or so

the operating system manages the virtual address space translating the addresses into

the physical addresses of values in the actual processor memory

in linux, topmost region of the address space reserved for code and data in operating system

that is common to all processes

lower holds code and data defined by users processes

increase from bottom to top

consists of number of well-defined areas each with a specific purpose

areas : (\*\*bottom to top\*\*)

program and code data

begins at same fixed address for all processes

followed by data locations that correspond to global C variables

initialized directly from contents of executable object file

fixed size once the process begins running

heap

expands and contracts dynamically at run time as result of calls to

standard C library routines like malloc and free.

shared libraries

eg, standard C library and math library

stack

compiler uses this to implement function calls

expands and contracts like heap during execution of program

each time call a function the stack grows

each time return, contracts

kernel virtual memory

kernel is part of OS that is always resident in memory

top part always reserved for kernel

application programs not allowed to read or write the contents of this

area or to directly call functions defined in kernel code

address space

ordered set of nonnegative integer addresses

if integers are consecutive can say its a linear address space

(\*\*book always assuming linear\*\*)

size of address space characterized by number of bits needed to rep

the largest address

ie virt add space with N = 2^n addresses called n-bit address space

CPU generates virtual addresses from an address space of N=2^n address

(virt add space)

A system also has a physical address space that corresponds to M bytes of

physical memory in the system

address space is important concept because makes distinction between

data objects (bytes) and their attributes (addresses)

generalize and allow each data object to have multiple independent

addresses, each from different address space

**C data types, variables, constants, structs**

data types/variables - same as java

ints can be signed or unsigned

type and description size

char - 1 byte, holds 1 character from ASCII table

short int - 16 bit int = 2 bytes

int/long/long int - 32 bits = 4 bytes

long long = 64 bit = 8 bytes

float - (single precision floating point) 4-byte

double - (double precision floating point) 8-byte

structs

We first need to define a new type for the compiler and tell it what our struct looks like.

struct flightType {

char flightNum[7]; /\* max 6 characters (‘\0’ null terminator at end)\*/

int altitude; /\* in meters \*/

int longitude; /\* in tenths of degrees \*/

int latitude; /\* in tenths of degrees \*/

int heading; /\* in tenths of degrees \*/

double airSpeed; /\* in km/hr \*/

};

This tells the compiler how big our struct is and how the different data items are laid out in memory

But it does not allocate any memory -> Memory is only allocated when a variable is declared

Can declare an array of struct items:

struct flightType planes[100];

Each array element is a struct item of type “struct flightType”

To access member of a particular element:

planes[34].altitude = 10000;

Because the [] and . operators are at the same precedence,and both associate left-to-right, this is the same as:

(planes[34]).altitude = 10000;

We can declare and create a pointer to a struct:

struct flightType \*planePtr;

planePtr = &planes[34];

To access a member of the struct addressed by dayPtr:

(\*planePtr).altitude = 10000;

Because the . operator has higher precedence than \*, this is NOT the same as:

\*planePtr.altitude = 10000;

C provides special syntax for accessing a struct member through a pointer:

planePtr->altitude = 10000;

Unlike an array, a struct item is passed by value

Most of the time, you’ll want to pass a pointer to a struct.

int Collide(struct flightType \*planeA, struct flightType \*planeB){

if (planeA->altitude == planeB->altitude) {

…

}else

return 0;

}

constants

-Constants are not supposed to change; if you try to change a const variable then you’ll get a compiler error.

**C arrays, pointers**

Arrays are contiguous sequences of data items

All data items are of the same type

Declaration of an array of integers: “int a[20];”

Access of an array item: “a[15]”

Array index always start at 0

The C compiler and runtime system do not check array boundaries

The compiler will happily let you do the following:

int a[10]; a[11] = 5;

Elements of an array are stored sequentially in memory

char grid[10];

First element (grid[0]) is at lowest address of sequence

Knowing the location of the first element is enough to access any element

An array name is essentially a pointer to the first element in the array

1. char word[10]; array of 10 with address to first

2. char \*cptr; just character pointer

3. cptr = word; /\* points to word[0] \*/ points to first elements of word

Difference:

Line 1 allocates space for 10 char items

Line 2 allocates space for 1 pointer

Can change value of cptr whereas cannot change value of word

Can only change value of word[i]

Arrays are passed by reference (Makes sense because array name ~ pointer)

Array items are passed by value (No need to declare size of array for function parameters)

**C unions - (source: http://www.tutorialspoint.com/cprogramming/c\_unions.htm):**

stores different data types in the same memory location

can define union with many members but only 1 can have a value at a given time

example:

union [union tag]{

member definition;

member definition;

….

} [one or more union variables];

union tag is optional

member definitions are normal variable definitions

specific example:

union Data{

int i;

float f;

char str[20];

} data;

memory occupied by union will be large enough to hold the largest member of the

union

ie. here it is 20 bytes because the largest is the 20 char array of 1 byte chars

example explaining a little more in depth

#include <stdio.h>

#include <string.h>

union Data{

int i;

float f;

char str[20];

};

int main(){

union Data data;

data.i = 20;

data.f = 220.5;

strcpy ( data.str, “C Programming”);

printf( “data.i : %d\n”, data.i );

printf( “data.f: %f\n”, data.f );

printf( “data.str : %s\n”, data.str );

return 0;

}

output looks like this:

data.i : 1917853763

data.f : 3543213434344653.00000

data.str : C Programming

other values get corrupted because last assignment takes up the space

would work if you used each aspect of it one at a time

#include <stdio.h>

#include <string.h>

union Data{

int i;

float f;

char str[20];

}data;

int main(){

data.i = 10;

printf( “%d”, data.i);

data.f = 220.5;

printf( “%f”, data.f);

strcpy (data.str, “C Programming”);

printf( “%s”, data.str);

return 0;

}

here everything would print out normally because nothing gets over written

**C enumeration types**

way to create a type that is just a list of keywords

ie symbolic names for a list of related ideas

enum Boolean{

false,

true

};

or something like

enum Security\_Levels{

black\_ops,

top\_secret,

secret,

non\_secret

};

so when using it in a program its not like strings, its a type so the checks would

look like

Security\_Levels myLevel = top\_secret;

if(myLevel == non\_secret){

...

} //and so on

Enum types provide a symbolic name to rep one state out of a list of states

while they aren't string values, compiler treats them like integers

ie for Security\_Levels

black\_ops = 0

top\_secret = 1

secret = 2

non\_secret = 3

\*\*\*\*easy trick to print out string representations is to just use an array with

indexes corresponding to the level of the member in the enum

**C typedefs**

give a type a new name that can be used as abbreviation for the type

OR could use with a struct to directly declare struct variables

ie

typedef struct Books{

char \*title;

char \*author;

char \*subject;

int book\_id;

}Book;

int main(){

Book book;

...

}

typedef vs. define

typedef is limited to giving symbolic names to types

#define can be used to define alias for values as well

typedef done by compiler

#define done by preprocessor

ie define

#define TRUE 1

#define FALSE 0

...

**C pointer arithmetic**

Given

char word[10];

char \*cptr;

cptr = word;

Each row in following table gives equivalent forms

cptr word &word[0] //address

cptr + n word + n &word[n] //address

\*cptr \*word word[0] //value

\*(cptr + n) \*(word + n) word[n] //value

Be careful when you are computing addresses

Address calculations with pointers are dependent on the size of the data the

pointers are pointing to

Examples:

int \*i; …; i++; /\* i = i + 4 \*/

char \*c; …; c++; /\* c = c + 1 \*/

double \*d; …; d++; /\* d = d + 8 \*/

Another example:

double x[10]; // double = 8 bytes

double \*y = x; // y becomes pointer to beginning of array

\*(y + 3) = 13; /\* x[3] = 13 \*/

**C functions**

return\_type name (parameters){ … }

call by value

method copies values of argument to parameter

changes made in method have no effect on the original value

call by reference

method copies ADDRESS of arg

edit from Jason: “The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.” - source (http://www.tutorialspoint.com/cprogramming/c\_function\_call\_by\_reference.htm)

inside function, address used to access argument and therefore actually

changes the original value

**C pointer to functions**

/\*basically, pointers to a function can be used as another way to access the function,

if i understand correctly

ie

start with a basic function that we will point to

int addInt(int n, int m){

return n+m;

}

so a pointer we could use to point to this function needs to receive 2 ints and return 1

int (\*functionPointer)(int,int);

then to set the pointer equal to the function we need the ~address of the function?

functionPointer = &addInt;  
**(EDIT: I think it should be functionPointer = addInt according to my notes)**

**(RESPOND TO EDIT:**

Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.

2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function’s name can also be used to get functions’ address. For example, in the below program, we have removed address operator ‘&’ in assignment. We have also changed function call by removing \*, the program still works.

|  |
| --- |
| #include <stdio.h>  // A normal function with an int parameter  // and void return type  void fun(int a)  {  printf("Value of a is %d\n", a);  }    int main()  {  void (\*fun\_ptr)(int) = fun; // & removed    fun\_ptr(10); // \* removed    return 0;  } |

then to use it

int sum = (\*functionPointer)(2,3); //so 2 and 3 get sent to addInt and the return value

//gets put back in sum so its 5

/\*\*\*\*pass the pointer to another function is basically the same\*\*\*\*\*/ ???

int add2to3( int (\*functionPointer)(int,int) ) {

return (\*functionPointer)(2,3);

}

seems really over complicated but sure

we can use function pointers in return values as well….

int (\*functionFactory(int n)) (int, int){ //function called functionFactory that takes in n

printf(“got parameter %d”, n);

int (\*functionPointer)(int,int) = &addInt; //returns a pointer to another function that takes 2 ints

return functionPtr; //then functionFactory returns 1 int in the end

}

but then the page i took notes from says it its better to use type deff anyway

typedef int (\*myFuncDef)(int,int);

//typedef name is myFuncDef

myFuncDef functionFactory(int n){

printf(“got parameter &d”,n);

myFuncDef functionPointer = &addInt;

return functionPointer;

}

\*\*little examples\*\*

int f (int) ; // defined but not implemented function

int (\*pf) (int); //pointer to a function, need parentheses, returns an int

int g(int);

int f(int); // 2 defined but not implemented functions

int x; //regular int x

x = f(3); //x holds the integer that gets returned from when 3 is passed to the function f

pf = f; //pf now points to the FUNCTION f itself

x = (\*pf)(3); // i think this does the same thing as what happened before

// because pf is a pointer to f, x then holds the int that is returned

//after f runs with parameter 3

pf = g; // then the pointer is overridden and pf now points to the function g

**printf(), scanf()**

printf("format", arguments) - format and print data

need #include <stdio.h>

use to print different types to command line

%c - character

%s - string

%f - float

%d - integer

%e - scientific notation

scanf("format", &whereInputWillBeStored) -

need #include <stdio.h>

scans input from standard input stream according to the format specified

works like a pointer\*\*\*\*\*

eg scanf("%d", &input); //need to use address

for more substantial input where don't know format, full files, or

strings with white space need fgets

uses same %\_ things as printf does

**C library string functions: strcpy(), strcat(), strcmp(), etc**

Useful string related functions in standard C libraries

#include <string.h>

char \*strcpy(char \*d, char \*s) Copy string s to d

int strcmp(s1, s2) Compare string s1 to s2

if return val < 0, s1 < s2

> 0, s1 > s2

= 0, s1 = s2

size\_t strlen(s) Returns length of cs

char \*strcat(char \*s1, const char \*s2); adds s2 to the end of s1

**ctype.h: isalpha(), isdigit(), etc**

int isalpha(int c)

returns non-zero if c is alpha, 0 if not

int isdigit(int c)

non-zero if digit, 0 if not

**Dynamic memory: malloc(), free(), realloc(), calloc()**

What if we want to write a program to handle a variable amount of data?

E.g., sort an arbitrary set of numbers

Can’t allocate an array because don’t know how many numbers we will get

Could allocate a very large array

Inflexible and inefficient

Answer: dynamic memory allocation

Similar to “new” in Java

When a function call is performed in a program, the run-time system must allocate resources to execute it

Memory for any local variables, arguments, and result

The same function can be called many times (Example: recursion)

Each instance will require some resources

The state associated with a function is called an activation record

Activation records are allocated on a call stack

Function calls leads to a new activation record pushed on top of the stack

Activation record is popped off the stack when the function returns

What if we want

Memory area whose lifetime does not match any particular function?

Memory area whose size is not known at compile time?

Two ways to “get memory”

Declare a variable Placed in global area or stack

Either “lives” forever or “live-and-die” with containing function

Size must be known at compile time

Ask the run-time system for a “chunk” of memory dynamically

After function returns, memory is still allocated

Request for dynamic chunks of memory performed using a call to the underlying

runtime system (a system call).

Commands: malloc and free

Another area region of memory exists, it is called the heap

Dynamic request for memory are allocated from this region

Managed by the run-time system(actually, just a fancy name for a library that’s linked with all C code)

The Standard C Library provides a function for dynamic memory allocation

void \*malloc(int numBytes);

malloc() (and free()) manages a region of memory called the heap

malloc() allocates a contiguous region of memory of size numBytes if there is enough free memory and

returns a pointer to the beginning of this region

Returns NULL if insufficient free memory

How do we know how many bytes to allocate?

Function

sizeof(type)

sizeof(variable)

Allocate right number of bytes, then cast to the right type

int \*numbers = (int \*)malloc(sizeof(int) \* n);

Once a dynamically allocated piece of memory is no longer needed, need to release it

Have finite amount of memory - If don’t release, will eventually run out of heap space

Function:

void free(void\*);

calloc

void \*calloc (size\_t numElementsToAllocate, size\_t sizeOfElements)

only diff from malloc is this sets the memory to zero

realloc

resize memory block previously referred to by malloc or calloc

void \*realloc (void \*ptr, size\_t size)

ptr is pointer to mem block - if NULL a new block gets allocated and a ptr is returned by the funct

size is new size for membloc

**C program structure: headers files, source files -- what to put in them**

source file - c code for program

header file -

what’s in it

method declarations

macro definitions

global variables

request to use in program with #include statement

using header file same thing as copying content into source code

**static variables, static functions**

(1)static variable in a function keeps the value between invocations

ie.

**#include <stdio.h>**

**void foo() {**

**int a = 10;**

**static int sa = 10;**

**//only sets to 10 upon initialization of var**

**a += 5; sa += 5;**

**printf("a = %d, sa = %d\n", a, sa); }**

**int main() { int i; for (i = 0; i < 10; ++i) foo(); }**

this prints

**a = 15, sa = 15**

**a = 15, sa = 20**

**a = 15, sa = 25**

**a = 15, sa = 30**

**a = 15, sa = 35s**

**a = 15, sa = 40**

**a = 15, sa = 45**

**a = 15, sa = 50**

**a = 15, sa = 55**

**a = 15, sa = 60**

use when you don’t want global vars but want to keep some state constant

through every use

(2)static global variable or function is “seen” only in the file where it’s declared

“access control” feature

public functions exposed to users while static can’t be accessed

encapsulation

sets scope of global var or function to containing file

**building linked data structures -- linked lists, etc.**

use struct node

ie

struct node{

struct \*node next;

type data;

};

**memcpy(),memcmp(),memset()**

void \*memcpy(void \*str1, const void \*str2, size\_t n)

copies n characters from memory area str2 to memory area str1

ie

**#include <stdio.h>  
 #include <string.h>  
  
 int main (){  
 const char src[50] = "http://www.tutorialspoint.com";  
 char dest[50];  
  
 printf("Before memcpy dest = %s\n", dest);  
 memcpy(dest, src, strlen(src)+1);  
 printf("After memcpy dest = %s\n", dest);  
   
 return(0);}**

**/\* output is**

**Before memcpy dest =**

**After memcpy dest =** [**http://www.tutorialspoint.com**](http://www.tutorialspoint.com)

**\*/**

int memcmp (const void \*str1, const void \*str2, size\_t n)

compares first n bytes of memory area str1 and str2

if return value <0 then str1<str2

>0 then str1>str2

=0 then they are equal

ie.

**#include <stdio.h>  
 #include <string.h>  
  
 int main (){  
 char str1[15];  
 char str2[15];  
 int ret;  
  
 memcpy(str1, "abcdef", 6);  
 memcpy(str2, "ABCDEF", 6);  
  
 ret = memcmp(str1, str2, 5);//compare first 5 bytes of str1 and str2**

**if(ret > 0) {  
 printf("str2 is less than str1");  
 }else if(ret < 0){  
 printf("str1 is less than str2");  
 }else{  
 printf("str1 is equal to str2");  
 }  
   
 return(0);  
}//the output for this looks like**

**// str2 is less then str1**

**//\*\*\*EDIT: actually, this should output “str2 is less than str1”**

**//This is because the ascii value of str1 is greater than that of str2**

**//Lowercase letters have higher ascii values than uppercase so a>A and by that effect, str1>str2**

**//im not sure, i copied that right off of the web and it does a byte compare not like as if they were strings?**

**//**[**http://www.tutorialspoint.com/c\_standard\_library/c\_function\_memcmp.htm**](http://www.tutorialspoint.com/c_standard_library/c_function_memcmp.htm)

**//??**

**//i just copied wrong sorry you right friend**

void \*memset (void \*str, int c, size\_t n)

copies char c (unsigned) to first n chars of string pointed to by str

returns pointer to memory area str

ie.

**#include <stdio.h>  
 #include <string.h>  
  
 int main (){  
 char str[50];**

**strcpy(str,"This is string.h library function");  
 puts(str);  
  
 memset(str,'$',7); //replaces/sets first 7 bytes (chars) with ‘$’  
 puts(str);  
   
 return(0);  
 } // output**

**//This is string.h library function**

**//$$$$$$$ string.h library function**

**const and pointers and const qualifier**

**The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed ( Which depends upon where const variables are stored, we may change value of const variable by using pointer ). The result is implementation-defined if an attempt is made to change a const (See** [**this**](http://geeksforgeeks.org/forum/topic/const-in-c) **forum topic).**

**1) Pointer to variable.**

|  |
| --- |
| **int \*ptr;** |

**We can change the value of ptr and we can also change the value of object ptr pointing to. Pointer and value pointed by pointer both are stored in read-write area. See the following code fragment.**

|  |
| --- |
| **#include <stdio.h>**  **int main(void)**  **{**  **int i = 10;**  **int j = 20;**  **int \*ptr = &i; /\* pointer to integer \*/**  **printf("\*ptr: %d\n", \*ptr);**    **/\* pointer is pointing to another variable \*/**  **ptr = &j;**  **printf("\*ptr: %d\n", \*ptr);**    **/\* we can change value stored by pointer \*/**  **\*ptr = 100;**  **printf("\*ptr: %d\n", \*ptr);**    **return 0;**  **}** |

**Output:**

**\*ptr: 10  
 \*ptr: 20  
 \*ptr: 100  
2) Pointer to constant.**

**Pointer to constant can be declared in following two ways.**

|  |
| --- |
| **const int \*ptr;** |

**or**

|  |
| --- |
| **int const \*ptr;** |

**We can change pointer to point to any other integer variable, but cannot change value of object (entity) pointed using pointer ptr. Pointer is stored in read-write area (stack in present case). Object pointed may be in read only or read write area. Let us see following examples.**

|  |
| --- |
| **#include <stdio.h>**  **int main(void)**  **{**  **int i = 10;**  **int j = 20;**  **const int \*ptr = &i; /\* ptr is pointer to constant \*/**    **printf("ptr: %d\n", \*ptr);**  **\*ptr = 100; /\* error: object pointed cannot be modified**  **using the pointer ptr \*/**    **ptr = &j; /\* valid \*/**  **printf("ptr: %d\n", \*ptr);**    **return 0;**  **}** |

**Output:**

**error: assignment of read-only location ‘\*ptr’**

**Following is another example where variable i itself is constant.**

|  |
| --- |
| **#include <stdio.h>**    **int main(void)**  **{**  **int const i = 10; /\* i is stored in read only area\*/**  **int j = 20;**    **int const \*ptr = &i; /\* pointer to integer constant. Here i**  **is of type "const int", and &i is of**  **type "const int \*". And p is of type**  **"const int", types are matching no issue \*/**    **printf("ptr: %d\n", \*ptr);**    **\*ptr = 100; /\* error \*/**    **ptr = &j; /\* valid. We call it as up qualification. In**  **C/C++, the type of "int \*" is allowed to up**  **qualify to the type "const int \*". The type of**  **&j is "int \*" and is implicitly up qualified by**  **the compiler to "const int \*" \*/**    **printf("ptr: %d\n", \*ptr);**    **return 0;**  **}** |

**Output:**

**error: assignment of read-only location ‘\*ptr’**

**Down qualification is not allowed in C++ and may cause warnings in C. Following is another example with down qualification.**

|  |
| --- |
| **#include <stdio.h>**    **int main(void)**  **{**  **int i = 10;**  **int const j = 20;**    **/\* ptr is pointing an integer object \*/**  **int \*ptr = &i;**    **printf("\*ptr: %d\n", \*ptr);**    **/\* The below assignment is invalid in C++, results in error**  **In C, the compiler \*may\* throw a warning, but casting is**  **implicitly allowed \*/**  **ptr = &j;**    **/\* In C++, it is called 'down qualification'. The type of expression**  **&j is "const int \*" and the type of ptr is "int \*". The**  **assignment "ptr = &j" causes to implicitly remove const-ness**  **from the expression &j. C++ being more type restrictive, will not**  **allow implicit down qualification. However, C++ allows implicit**  **up qualification. The reason being, const qualified identifiers**  **are bound to be placed in read-only memory (but not always). If**  **C++ allows above kind of assignment (ptr = &j), we can use 'ptr'**  **to modify value of j which is in read-only memory. The**  **consequences are implementation dependent, the program may fail**  **at runtime. So strict type checking helps clean code. \*/**    **printf("\*ptr: %d\n", \*ptr);**    **return 0;**  **}**    **// Reference** [**http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf**](http://www.dansaks.com/articles/1999-02%20const%20T%20vs%20T%20const.pdf)    **// More interesting stuff on C/C++ @** [**http://www.dansaks.com/articles.htm**](http://www.dansaks.com/articles.htm) |

**3) Constant pointer to variable.**

|  |
| --- |
| **int \*const ptr;** |

**Above declaration is constant pointer to integer variable, means we can change value of object pointed by pointer, but cannot change the pointer to point another variable.**

|  |
| --- |
| **#include <stdio.h>**    **int main(void)**  **{**  **int i = 10;**  **int j = 20;**  **int \*const ptr = &i; /\* constant pointer to integer \*/**    **printf("ptr: %d\n", \*ptr);**    **\*ptr = 100; /\* valid \*/**  **printf("ptr: %d\n", \*ptr);**    **ptr = &j; /\* error \*/**  **return 0;**  **}** |

**Output:**

**error: assignment of read-only variable ‘ptr’  
4) constant pointer to constant**

|  |
| --- |
| **const int \*const ptr;** |

**Above declaration is constant pointer to constant variable which means we cannot change value pointed by pointer as well as we cannot point the pointer to other variable. Let us see with example.**

|  |
| --- |
| **#include <stdio.h>**    **int main(void)**  **{**  **int i = 10;**  **int j = 20;**  **const int \*const ptr = &i; /\* constant pointer to constant integer \*/**    **printf("ptr: %d\n", \*ptr);**    **ptr = &j; /\* error \*/**  **\*ptr = 100; /\* error \*/**    **return 0;**  **}** |

**Output:**

**error: assignment of read-only variable ‘ptr’  
 error: assignment of read-only location ‘\*ptr’**

**makefiles: macros, dependencies, de facto conventions**

**makefiles: - NO MAKEFILES**

projects makefiles and directory I/O

ar key libfile obj1.o obj2.o …

r - add

t

d

v - verbose

c

ar rv libsl.a sorted\_list.o

basename [directory path]/[basename].[extension]

[basename].[extension]

drname [directory path]/ [basename].[extension]

[directory path]

this literally means nothing to me but this is all he has in his notes for makes directory I/O and projects

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ so this thing tells me

makefiles are a simple way to organize code compilation

the three example files here are

hellomake.c

#include <hellomake.h>

int main(){

//call a function in another file

myPrintHelloMake();

return (0);

}

hellofunc.c

#include <stdio.h>

#include <hellomake.h>

void myPrintHelloMake(void){

printf(“Hello makefiles!\n”);

return; //not 1 or 0 because not main DUH. i got confused

}

hellomake.h

/\*

example include file

\*/

void myPrintHelloMake(void);

so to compile this mound of shit need to run

gcc -o hellomake hellomake.c hellofunc.c -I.

this compiles the two .c files and names the executable hellomake,

the -I used so that gcc will look in the current directory (.) for the include file hellomake.h

without makefile typical approach to the test/modify/debug cycle is to use up arrow in a terminal to

go back to your last compile command so you don’t have to type it each time, especially once

you’ve added more.c files to the mix.

downfalls to this

switching computers means you have to retype it or if you lose it you have to retype it

changing only one of the many .c files, compiling everything over and over again inefficient

so basically just supposed to make compiling easier and faster so you don’t have to type

the long ass commands every time

simplest makefile will look something like this

hellomake: hellomake.c hellofunc.c

gcc -o hellomake hellomake.c hellofunc.c -I.

if put this into a file named Makefile or makefile and type make into command line will execute

the command as you have it in the makefile so the whole gcc command.

NOTE; make with no arguments executes the first rule in the file

by putting the list of files on which the command depends on the first line after the :, make knows that

the rule hellomake needs to be executed if any of those files change

so solves the first problem but doesn’t solve the problem of executing everything again if you

only changed one small thing

solve that with

CC=gcc

CFLAGS=-I.

hellomake: hellomake.o hellofunc.o

$(CC) -o hellomake hellomake.o hellofunc.o -I.

this defines constraints (CC) and CFLAGS that tell make how we want to compile hellomake.c and

hellofunc.c

wtf okay he says

the macro CC is the C compiler to use, and CFLAGS is the list of flags to pass to the compilation

command. by putting the object files—hellomake.o and hellofunc.o — in the dependency list

and in the rule, make knows it must first compile the .c versions individually, and then build

the executable hellomake

but then that isn’t the best because if you make a small change to the .h file you have to re-compile all

of the .c files, even the ones that don’t involve the .h file that you changed SO you need to do this

CC=gcc

CFLAGS =-I.

DEPS = hellomake.h

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o

gcc -o hellomake.o hellofunc.o \_I.

this addition creates the macro DEPS, that is the set of .h files on which the .c files depend.

then define a rule that applies to all files ending in the .o suffix.

rule we have in this file says .o depends on the .c version of the file and the .hs included in the DEPS

then to generate the .o make needs to compile the .c files using the

compiler defined in the CC macro.

-c flag says to generate the object file

-o $@ says to put the output of compilation in the file to the left of :

the $< is first item in dependencies list

CFLAGS macro defined at top

in order to make the compilation rule more general all include files should

be listed as part of the macro DEPS and all object files listed as

part of the macro OBJ

CC = gcc

CFLAGS = -I.

DEPS = hellomake.h

OBJ = hellomake.o hellofunc.o

%.o: %.c %(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)

gcc -o $@ $^ $(CFLAGS)

okay so then can use a make to put different files in different directories

ie. .h in include

source in src

local libraries in lib

\*\*\*\*\*\*\*\* HIDIING .o files - in obj sub direct of src directory

has macro defined for any libraries you want to include ie math

makefile should be in the src directory

type make clean to clean up source and object directories

.PHONY rule keeps make from doing something with a file named clean?

last example make file

IDIR = ../include

CC = gcc

CFLAGS = -I$(IDIR)

ODIR = obj

LDIR = ../lib

LIBS = -lm

\_DEPS = hellomake.h

DEPS = $(patsubst %, $(IDIR)/%, $(DEPS))

\_OBJ = hellomake.o hellofunc.o

OBJ = $(patsubst %, $(ODIR)/%,$(\_OBJ))

$(ODIR)/%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CLFAGS)

hellomake: $(OBJ)

gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY clean

clean:

rm -f $(ODIR)/\*.o \*~ core $(INCDIR)/\*~

GNU MANUEL MORE INFO ON MAKE FILES NEED TO GO THROUGH

http://www.gnu.org/software/make/manual/make.html

all those notes were from:

http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

dependencies:

defined in a makefile, the files the executable will be dependent upon

ie

hello: main.o factorial.o hello.o

$(CC) main.o factorial.o hello.o -o hello

tell make that hello is dependent on main.o, factorial.o and hello.o

but before that need to tell make to create the .o files so need to define dependencies as

ie

main.o: main.c functions.h

$(CC) -c main.c

factorial.o: factorial.c functions.h

$(CC) -c factorial.c

hello.o: hello.c functions.h

$(CC) -c hello.c

macros:

fragment of code which has been given a name

name gets replaced by contents of macro when compiled

2 types

object like - resemble data when used

give symbolic names to numeric constants

create with #define NAME

ie.

#define BUFFER\_SIZE 1024

//so can use BUFFER\_SIZE in place of 1024

foo = (char\*) malloc (1024);

foo = (char\*)malloc(BUFFER\_SIZE);

//both of these are the same

can define them on multiple lines

ie.

#define NUMBERS 1, \

2, \

3

int x[ ] = {NUMBERS};

//gives int x[] values 1 2 3

C preprocessor scans program sequentially so this

foo = X;

#define X 4

bar = X;

gives you this

foo = X; // literally

bar = 4;

when the preprocessor expands a macro name, macro’s expansion

replaces macro invocation, then expansion is examined for more

macros to expand

function like - resemble function calls

use like a function call

ie

#define lang\_init() c\_init()

Defining rules in makefile

general syntax of a makefile target rule

target [target…] : [dependent…]

[command...]

//arguments in brackets optional

// … means more then one

// tab before each command is required

simple example how to define rule to make target from 3 other files

hello: main.o factorial.o hello.o

$(CC) main.o factorial.o hello.o -o hello

//here would need to give rules to make all object files from source files

when say “make target”

make finds target rule that applies and

if any dependents are newer than the target

make executes commands one at a time (after macro substitution)

if any dependents have to be made, that happens first (so recursion)

make will terminate if any command returns a failure status

make ignores return statues on command line that starts with dash so if you don’t care if there

is a core file then do

clean:

-rm \*.o \*~ core paper

make echoes the commands, after macro substitution to show you what is happening as it

happens

ie.

install:

@echo You must reboot to install

always browse first

reasonable to expect that the targets all (or just make), install and clean are there

make all - compiles everything so that you can do local testing before installing apps

make install - installs apps at right places

make clean - it cleans apps up, gets rid of executables, any temp file, object files

command that should make executable X out of the source code x.cpp stated as implicit rule

.cpp:

$(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@

// this is implicit rule and says how to make X out of x.c -- run cc on x.c and call the output x.

//implicit because no particular target is mentioned. can be used in all cases

another commmon implicit rule is for construction of .o (object) files out of .cpp (source files)

.cpp.o:

$(CC) $(CFLAGS) -c $<

alternatively

.cpp.o:

$(CC) $(CFLAGS) -c $\*.cpp

Basic Makefile: (in terms of target and dependencies)

*target: dependencies  
 [tab] system command*

**opendir(),readdir(),seekdir(),telldir(),closedir() -** all need either both #include <dirent.h> #include <sys/types.h> or just #include <dirent.h>

DIR \*opendir(const char \*name); - opens a directory stream corresponding to

the directory name and returns a pointer to it. Stream is positioned at the

first entry in the directory

struct dirent \*readdir(DIR \*dirp);

int readdir \_r(DIR \*dirp, struct dirent \*entry, struct dirent \*\*result);

readdir() returns pointer to dirent struct reping the next directory entry in

directory stream pointed to by dirp

returns NULL on reaching end of directory stream or if error

int closedir(DIR \*dirp); - function that closes directory stream associated with dirp

successful call closes the underlying file descriptor associated with dirp

dirp not available after this call

//\*\*\*\*linux.die describes an lseek not a seek ?

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

off\_t lseek (int fd, off\_t offset, int whence);

repositions the offset of the open file associated with the file descriptor fd to argument

offset according to the directive whence as follows

SEEK\_SET - offset is set to offset bytes

SEEK\_CUR - offset set to current location plus offset bytes

SEEK\_END - offset is set to size of file plus offset bytes

lseek allows the file offset to go beyond the end of the file but doesn’t change the size of the file

if data is written beyond EOF, later reads of the data include NULL bytes in the gap

long telldir(DIR \*dirp); - returns current location in directory stream

on error returns -1

needs #include <dirent.h>

**C Preprocessor: Macros, file inclusion, conditional compilation**

**c preprocessor =** cpp, a macro processor used automatically by the C compiler

to transform program before compilation. called macro processor cuz

allows user to define macros, which are brief abbreviations for longer

constructs

**macros -** abbreviations for arbitrary fragments of C code which the preprocessor will replace with their definitions throughout the program

#define

2 kinds

object-like

function-like

**conditional compilation / wrapping debugging statements in #ifdef DEBUG ... #endif**

**In computer programming, conditional compilation is compilation implementing methods which allow the compiler to produce differences in the executable produced controlled by parameters that are provided during compilation.**

**Conditional compilation**[[edit](https://en.wikipedia.org/w/index.php?title=C_preprocessor&action=edit&section=3)]

**The** [**if-else**](https://en.wikipedia.org/wiki/If_else#If.E2.80.93then.28.E2.80.93else.29) **directives #if, #ifdef, #ifndef, #else, #elif and #endif can be used for** [**conditional compilation**](https://en.wikipedia.org/wiki/Conditional_compilation)**.**

**#if VERBOSE >= 2  
 print("trace message");  
#endif**

**Most compilers targeting** [**Microsoft Windows**](https://en.wikipedia.org/wiki/Microsoft_Windows) **implicitly define \_WIN32.**[**[1]**](https://en.wikipedia.org/wiki/C_preprocessor#cite_note-1) **This allows code, including preprocessor commands, to compile only when targeting Windows systems. A few compilers define WIN32 instead. For such compilers that do not implicitly define the \_WIN32 macro, it can be specified on the compiler's command line, using -D\_WIN32.**

**#ifdef \_\_unix\_\_ */\* \_\_unix\_\_ is usually defined by compilers targeting Unix systems \*/*  
# include <unistd.h>  
#elif defined \_WIN32 */\* \_Win32 is usually defined by compilers targeting 32 or 64 bit Windows systems \*/*  
# include <windows.h>  
#endif**

**The example code tests if a macro \_\_unix\_\_ is defined. If it is, the file <unistd.h> is then included. Otherwise, it tests if a macro \_WIN32 is defined instead. If it is, the file <windows.h> is then included.**

**A more complex #if example can use operators, for example something like:**

**#if !(defined \_\_LP64\_\_ || defined \_\_LLP64\_\_) || defined \_WIN32 && !defined \_WIN64  
 *// we are compiling for a 32-bit system*  
#else  
 *// we are compiling for a 64-bit system*  
#endif**

**Translation can also be caused to fail by using the #error directive:**

**#if RUBY\_VERSION == 190  
# error 1.9.0 not supported  
#endif**

**Reference counts applied to second assignment**

you increment a ref counter for every time a new pointer points to a node

decrement ref counter for every time you remove pointer from node

free memory for node when decrement to 0

iterator pointing to node that is printing or up to some operation can make the

ref counter not zero or 1, ...leading to interesting scenarios

**ar command**

archive tool to combine objects to create an archive file with .a extension (library)

how to create user defined static library

compile .c files with gcc to get .o files

create .a library with example command

ar cr libTest.a add.o multiply.o

view object files in archive with ar command

ar t libTest.a

//prints out 2 example .o files that were used to create it

extract object files from archive using ar command, option x

mkdir object // make a directory named object

cp libTest.a object

cd object // enter into object directory

ar x libTest.a

ls \*.o // should list all files that were in lib, here add.o and multiply.o

add object file to existing archive using ar, option r

for example, assume have object file named sub.o

use following example to extend libTest.a by inserting sub.o

ar r libTest.a sub.o

ar t libTest.a

//should list add.o multiply.o and sub.o

using this if something already exists in the library with same name, would get replaced

could just add it to the end of the archive by using -q option

delete specific archive member using ar, option d

ar d libTest.a add.o

ar t libTest.a

//lists everything left multiply.o and sub.o

**System calls: (errno is an integer set by syscalls in the event an error occurs**

**a syscall returns either 1 or an errno)**

**open(2),**

int open (const char \*pathname, int flags);

given a pathname for a file, returns a file descriptor, a small

nonneg integer for use in subsequent system calls. The file descriptor returned by successful call will be lowest-numbered file descriptor not currently open for the process.

**read(2),**

ssize\_t read (int fd, void \*buf, size\_t count);

attempts to read up to count bytes from file descriptor fd into the

buffer starting at buf

**write(2),**

ssize\_t write(int fd, const void \*buf, size\_t count);

write up to count bytes from the buffer pointer buf to file referred

to by the file descriptor fd

**lseek(2),**

off\_t lseek(int fd, off\_t offset, int whence);

repositions the offset of the open file associated with the file descriptor fd to the argument offset according to the directive whence as follows:

SEEK\_SET - offset is set to offset bytes

SEEK\_CUR - offset is set to its current location plus offset bytes

SEEK\_END - offset to size of file plus offset bytes

this function allows file offset to go beyond EOF (but does not change the size of the file) if data is written passed EOF subsequent reads of the data in the gap return null bytes (‘\0’) until data is actually written into the gap

**close(2)**

int close (int fd); - returns 0 on success, -1 on error

closes file descriptor so that it no longer refers to any file and may be reused.

any record locks held on file and owned by the process are removed

if fd is the last copy of particular file descriptor the resources with it are freed

if it was the last reference to a file which has been removed using unlink the file is deleted.

**errno and return values from system calls, examples like EEXISTS, etc.**

errno.h is a header file in the standard library of the C programming language. It defines macros for reporting and retrieving error conditions through error codes stored in a static memory location called errno(short for "error number").

only significant when return value of the call indicated an error

E2BIG Argument list too long (POSIX.1)  
  
 \*\* EACCES Permission denied (POSIX.1)  
  
 EADDRINUSE Address already in use (POSIX.1)  
  
 EADDRNOTAVAIL Address not available (POSIX.1)  
  
 EAFNOSUPPORT Address family not supported (POSIX.1)  
  
 EAGAIN Resource temporarily unavailable (may be the same  
 value as EWOULDBLOCK) (POSIX.1)  
  
 EALREADY Connection already in progress (POSIX.1)  
  
 EBADE Invalid exchange  
  
 EBADF Bad file descriptor (POSIX.1)  
  
 EBADFD File descriptor in bad state  
  
 EBADMSG Bad message (POSIX.1)  
  
 EBADR Invalid request descriptor  
  
 EBADRQC Invalid request code  
  
 EBADSLT Invalid slot  
  
 EBUSY Device or resource busy (POSIX.1)  
  
 ECANCELED Operation canceled (POSIX.1)  
  
 ECHILD No child processes (POSIX.1)  
  
 ECHRNG Channel number out of range  
  
 ECOMM Communication error on send  
  
 ECONNABORTED Connection aborted (POSIX.1)  
  
 ECONNREFUSED Connection refused (POSIX.1)  
  
 ECONNRESET Connection reset (POSIX.1)  
  
 EDEADLK Resource deadlock avoided (POSIX.1)  
  
 EDEADLOCK Synonym for EDEADLK  
  
 EDESTADDRREQ Destination address required (POSIX.1)  
  
 EDOM Mathematics argument out of domain of function  
 (POSIX.1, C99)  
  
 EDQUOT Disk quota exceeded (POSIX.1)  
  
 \*\*EEXIST File exists (POSIX.1) [\*Pretty important]  
  
 EFAULT Bad address (POSIX.1)  
  
 EFBIG File too large (POSIX.1)  
  
 EHOSTDOWN Host is down  
  
 EHOSTUNREACH Host is unreachable (POSIX.1)  
  
 EIDRM Identifier removed (POSIX.1)  
  
 EILSEQ Illegal byte sequence (POSIX.1, C99)  
  
 EINPROGRESS Operation in progress (POSIX.1)  
  
 EINTR Interrupted function call (POSIX.1); see [signal(7)](http://man7.org/linux/man-pages/man7/signal.7.html).  
  
 \*\*EINVAL Invalid argument (POSIX.1) [\*Pretty important]  
  
 EIO Input/output error (POSIX.1)  
  
 EISCONN Socket is connected (POSIX.1)  
  
 \*\*EISDIR Is a directory (POSIX.1) [\*Pretty important]  
  
 EISNAM Is a named type file  
  
 EKEYEXPIRED Key has expired  
  
 EKEYREJECTED Key was rejected by service  
  
 EKEYREVOKED Key has been revoked  
  
 EL2HLT Level 2 halted  
  
 EL2NSYNC Level 2 not synchronized  
  
 EL3HLT Level 3 halted  
  
 EL3RST Level 3 halted  
  
 ELIBACC Cannot access a needed shared library  
  
 ELIBBAD Accessing a corrupted shared library  
  
 ELIBMAX Attempting to link in too many shared libraries  
  
 ELIBSCN lib section in a.out corrupted  
  
 ELIBEXEC Cannot exec a shared library directly  
  
 ELOOP Too many levels of symbolic links (POSIX.1)  
  
 EMEDIUMTYPE Wrong medium type  
  
 EMFILE Too many open files (POSIX.1); commonly caused by  
 exceeding the RLIMIT\_NOFILE resource limit described  
 in [getrlimit(2)](http://man7.org/linux/man-pages/man2/getrlimit.2.html)  
  
 EMLINK Too many links (POSIX.1)  
  
 EMSGSIZE Message too long (POSIX.1)  
  
 EMULTIHOP Multihop attempted (POSIX.1)  
  
 ENAMETOOLONG Filename too long (POSIX.1)  
  
 ENETDOWN Network is down (POSIX.1)  
  
 ENETRESET Connection aborted by network (POSIX.1)  
  
 ENETUNREACH Network unreachable (POSIX.1)  
  
 ENFILE Too many open files in system (POSIX.1)  
  
 ENOBUFS No buffer space available (POSIX.1 (XSI STREAMS  
 option))  
  
 ENODATA No message is available on the STREAM head read queue  
 (POSIX.1)  
  
 ENODEV No such device (POSIX.1)  
  
 \*\* ENOENT No such file or directory (POSIX.1) [\*Pretty important]  
  
 ENOEXEC Exec format error (POSIX.1)  
  
 ENOKEY Required key not available  
  
 ENOLCK No locks available (POSIX.1)  
  
 ENOLINK Link has been severed (POSIX.1)  
  
 ENOMEDIUM No medium found  
  
 ENOMEM Not enough space (POSIX.1)  
  
 ENOMSG No message of the desired type (POSIX.1)  
  
 ENONET Machine is not on the network  
  
 ENOPKG Package not installed  
  
 ENOPROTOOPT Protocol not available (POSIX.1)  
  
 ENOSPC No space left on device (POSIX.1)  
  
 ENOSR No STREAM resources (POSIX.1 (XSI STREAMS option))  
  
 ENOSTR Not a STREAM (POSIX.1 (XSI STREAMS option))  
  
 ENOSYS Function not implemented (POSIX.1)  
  
 ENOTBLK Block device required  
  
 ENOTCONN The socket is not connected (POSIX.1)  
  
 \*\*ENOTDIR Not a directory (POSIX.1) [\*Pretty important]  
  
 ENOTEMPTY Directory not empty (POSIX.1)  
  
 ENOTSOCK Not a socket (POSIX.1)  
  
 ENOTSUP Operation not supported (POSIX.1)  
  
 ENOTTY Inappropriate I/O control operation (POSIX.1)  
  
 ENOTUNIQ Name not unique on network  
  
 ENXIO No such device or address (POSIX.1)  
  
 EOPNOTSUPP Operation not supported on socket (POSIX.1)  
  
 (ENOTSUP and EOPNOTSUPP have the same value on Linux,  
 but according to POSIX.1 these error values should be  
 distinct.)  
  
 EOVERFLOW Value too large to be stored in data type (POSIX.1)  
  
 EPERM Operation not permitted (POSIX.1)  
  
 EPFNOSUPPORT Protocol family not supported  
  
 EPIPE Broken pipe (POSIX.1)  
  
 EPROTO Protocol error (POSIX.1)  
  
 EPROTONOSUPPORT Protocol not supported (POSIX.1)  
  
 EPROTOTYPE Protocol wrong type for socket (POSIX.1)  
  
 ERANGE Result too large (POSIX.1, C99)  
  
 EREMCHG Remote address changed  
  
 EREMOTE Object is remote  
  
 EREMOTEIO Remote I/O error  
  
 ERESTART Interrupted system call should be restarted  
  
 EROFS Read-only filesystem (POSIX.1)  
  
 ESHUTDOWN Cannot send after transport endpoint shutdown  
  
 ESPIPE Invalid seek (POSIX.1)  
  
 ESOCKTNOSUPPORT Socket type not supported  
  
 ESRCH No such process (POSIX.1)  
  
 ESTALE Stale file handle (POSIX.1)  
  
 This error can occur for NFS and for other  
 filesystems  
  
 ESTRPIPE Streams pipe error  
  
 ETIME Timer expired (POSIX.1 (XSI STREAMS op  
 (POSIX.1 says "STREAM [ioctl(2)](http://man7.org/linux/man-pages/man2/ioctl.2.html) timeout")tion))  
  
  
 ETIMEDOUT Connection timed out (POSIX.1)  
  
 ETXTBSY Text file busy (POSIX.1)  
  
 EUCLEAN Structure needs cleaning  
  
 EUNATCH Protocol driver not attached  
  
 EUSERS Too many users  
  
 EWOULDBLOCK Operation would block (may be same value as EAGAIN)  
 (POSIX.1)  
  
 EXDEV Improper link (POSIX.1)  
  
 EXFULL Exchange full

**file holes with lseek(2)**

if you seek PAST the end of a file and write, C will create a gap between the EOF and the data you write and that gap will be filled with NUL characters/null bytes

## **NAME** [**top**](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

lseek - reposition read/write file offset

## **SYNOPSIS** [**top**](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

**#include <sys/types.h>**  
 **#include <unistd.h>**  
  
 **off\_t lseek(int** *fd***, off\_t** *offset***, int** *whence***);**

## **DESCRIPTION** [**top**](http://man7.org/linux/man-pages/man2/lseek.2.html#top_of_page)

The **lseek**() function repositions the offset of the open file  
 associated with the file descriptor *fd* to the argument *offset*  
 according to the directive *whence* as follows:  
  
 **SEEK\_SET**  
 The offset is set to *offset* bytes.  
  
 **SEEK\_CUR**  
 The offset is set to its current location plus *offset* bytes.  
  
 **SEEK\_END**  
 The offset is set to the size of the file plus *offset* bytes.  
  
 The **lseek**() function allows the file offset to be set beyond the end  
 of the file (but this does not change the size of the file). If data  
 is later written at this point, subsequent reads of the data in the  
 gap (a "hole") return null bytes ('\0') until data is actually  
 written into the gap.

**GDB : stepping into/over calls, setting/clearing breakpoints**

https://sourceware.org/gdb/onlinedocs/gdb/Continuing-and-Stepping.html

https://sourceware.org/gdb/onlinedocs/gdb/Breakpoints.html#Breakpoints

(gdb) step

(gdb) break

(gdb) clear

**continue execution, return from function call**

(gdb) continue

(gdb) return

**print command to see variables**

(gdb) print

**set variable command to change values**

(gdb) set

**stack trace.**

(gdb) backtrace

**Debugging without GDB : printf()s, \_\_FILE\_\_, \_\_LINE\_\_**

printf()s - equivalent of checking variable values in java with System.out.print()

\_\_FILE\_\_

\_\_FILE\_\_ is a preprocessor macro that expands to full path to the current file. \_\_FILE\_\_ is useful when generating log statements, error messages intended for programmers, when throwing [exceptions](http://www.cprogramming.com/tutorial/exceptions.html), or when writing [debugging](http://www.cprogramming.com/debuggers.html) code.

int logError (const char\* file, const std::string& message)  
{  
 cerr << "[" << file << "]" << message << endl;  
}  
#define LOG( message ) logError( \_\_FILE\_\_, message )  
  
// LOG can now be used to produce a log message that includes the file in which the log occurrred

The \_\_FILE\_\_ macro is often combined with the [\_\_LINE\_\_](http://www.cprogramming.com/reference/preprocessor/__LINE__.html) macro, which expands to the current line number.

[**http://www.gnu.org/software/libc/manual/html\_node/Signal-Handling.html**](http://www.gnu.org/software/libc/manual/html_node/Signal-Handling.html)

**gcc -DDEBUG to define DEBUG macro for compilation**

**Signals - notification to process of events**

**Signal life cycle : generated -> pending -> blocked/unblocked -> delivered**

**Signal disposition : default action (usually kill process), ignore or catch**

**sigmask - each process has set of blocked signals, initially empty**

**Operations on signal mask sigset\_t : sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember()**

**sigset\_t as opaque type**

**getting and affecting process signal mask : siprocmask() SIG\_BLOCK, SIG\_UNBLOCK, SIG\_SETMASK**

**Signal handler functions -- invoked, but not called. Returns to next instruction without call instruction.**

**setting up signal handlers with sigaction() system call.**

**Signal Handler mechanics**

**sigaction struct**

**Timers :- NOPE**

**- timeout on responses from external sources**

**- controlling shifts in program internal state**

**- reporting/saving/logging program state information**

**- checking for loss of contact/communication with other processes, other machines**

**Timer activation, deactivation, expiration**

**setitimer() system call : activating, deactivating timers**

**kill(2)**

## **Name**

kill - send signal to a process

## **Synopsis**

**#include <**[**sys/types.h**](http://linux.die.net/include/sys/types.h)**>  
#include <**[**signal.h**](http://linux.die.net/include/signal.h)**>   
  
int kill(pid\_t** *pid***, int** *sig***);**

Feature Test Macro Requirements for glibc (see [**feature\_test\_macros**](http://linux.die.net/man/7/feature_test_macros)(7)):

**kill**(): \_POSIX\_C\_SOURCE >= 1 || \_XOPEN\_SOURCE || \_POSIX\_SOURCE

## **Description**

The **kill**() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the calling process.

If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.

If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is *-pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID.

For a process to have permission to send a signal it must either be privileged (under Linux: have the **CAP\_KILL** capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of **SIGCONT** it suffices when the sending and receiving processes belong to the same session.

**process stuff**

**fork() - creates child processes. General structure of programs using fork().**

**wait(2), waitpid(2) system calls, exit status**

**\_exit(2) and exit(3) calls**

**exec functions : execcl(), execcv(), execle(), execve(), execlp(), execvp().**

what is a process?

instance of a program in execution

as soon as computer starts - kernel activates

has complete control of everything computer does

we never directly interact with

commandline interacts with it, processors (CPU memory disk drive), hardware

computer always running many diff things

2 processes can't be happening at same time so need contact switch

ie break switch

happens so fast back and forth that we can't recognize it

all code for processors is in user mode

below that is kernel mode

switching between is mode switch - user to kernel

process switching is contact swtich

processes can be in 3 diff states of execution

running state

1 in running state per CPU

ready state

once system swtiches to run another process, no CPU available but could continue

blocked

even if CPU available, couldn't run because waiting for something like resource

running can become blocked, or ready

blocked can only become ready because has to do this before ready

ready can become running - cuz scheduler says that there is a CPU ready

how to create a process in code?

2 ways

linux/unix

call fork ()

clones process into child process and runs it there

windows

call exec()

spawns new process with new program

How to create a process in C

fork - creates a child process

need #include <unistd.h>

creates new process by duplicating calling process and the new process (child)

is an exact duplicate of the calling process, referred to as the parent

except for the following points

child has unique process ID that doesn't match any existing process group

child's parent process ID is the same as parent's process ID

child doesn’t inherit parents memory locks

process resource utilizations and CPU time counters reset to 0 in child

child's set of pending signals initially empty

child does not inherit timers from parent

**how to call fork**

in main eg

int main (){

//create var later want to print out

int x = 1;

int returnValue = fork();

- if (returnValue == 0) //gets weird here because used to writing code that will //run sequentially but...

//only child process executes code here

printf("child , x = %d\n", ++x);

}else{

//only parent executes this happens because fork returns 2xs

//first time is either 0 or non zero

printf("Parent,x = %d\n", --x);

}

printf("Exiting with x = %d\n", x);

return 0;

}

so how many times is everything going to run?

code in parent runs first:: both prints give you x = 0

after that child executed and child prints out 2!

even if changed variable in parent doesn't change in child

when call fork, all code above gets dupicated for itself

so its as if the first run of the if statement didn't happen and it

started from scratch

**working more with fork()**

int main(){

//special data types for fork

//kind of just an int but special limits for fork

pid\_t childProcessIDorZero = fork();

if(childProcessIDorZero <0){

perror("fork() error");

exit(-1);

}

if(childProcessIDorZero != 0) { //know we are in parent process

printf("im in the parent %d, my child is %d\n", getpid(), childProcessIDorZero);

//childProcessIDorZero becomes PID of child

//just in case child never finishes or parent finishes first need it to wait

wait(NULL); // wait for child to join parent

}else{

printf("im the child %d and my parent %d\n", getpid(), getppid());

//getpid here gets child id from in child and need getppid to get parent

execl("/bin/echo", "echo", "Hello, World", NULL);

//call executable in this process

//most of the code is going to be in child dealing with logic so want parent to WAIT

}

return 0;

}

//shell uses fork