Learning C

Table of Contents

Topic	Page
Introduction to C programming language	1
How C works - compile linker	2-4
Constants, variables, data types, keywords	5-8
C standard library	9-10
Comments	10 (bottom)
Basic Hello world program	11
printf function	12
Arithmetic Operators	13
logical and comparison operators	14
Conditional statements (if if else, else, ternary)	15
while and do while loops	16
for loops	17
arrays	18
functions	19-21
global variable scopes	22
control flow (return, break, continue)	22-23
switch statements	24-25
address operator &	26
pointers	27-28
pass an array to a function - pointers	29
relationship between arrays and pointers	30
const keyword	31
array ranges in functions	32
structures (struct)	33-36
sizeof operator	37
unions (union)	38
memory allocation (malloc, calloc, realloc, free)	39-44
File handling - many functions	45-50
strings and string functions	51
preprocessor directives (#include, #define, and many more)	52-56

Table of Contents

Topic	Page

Table of Contents

Topic	Page

Table of Contents

Topic	Page
typedef keyword	57
storage classes (auto, register, static, extern)	58-60
time and date	61
character functions <ctype.h>	62
string functions <string.h>	63
additional standard library functions <stdlib.h>	64
math functions <math.h>	65

Introduction to C Programming language

About C

- created in 1972 in Bell Lab by Dennis Ritchie
- very popular language over the years
- simple and easy to learn language
- fast and great to use in high control such as microcontroller programming
- High performance and efficient
- mother of all languages
- low level access to memory, simple set of keywords and clean style

References

- Youtube series - C programming language - learning lad
- Geeks for Geeks site C programming
- Stack Overflow - good refrence for questions
- tutorialspoint -

Documentation

- Text - The C programming language - second edition - Brian W. Kernighan

How C Programming language works / executes

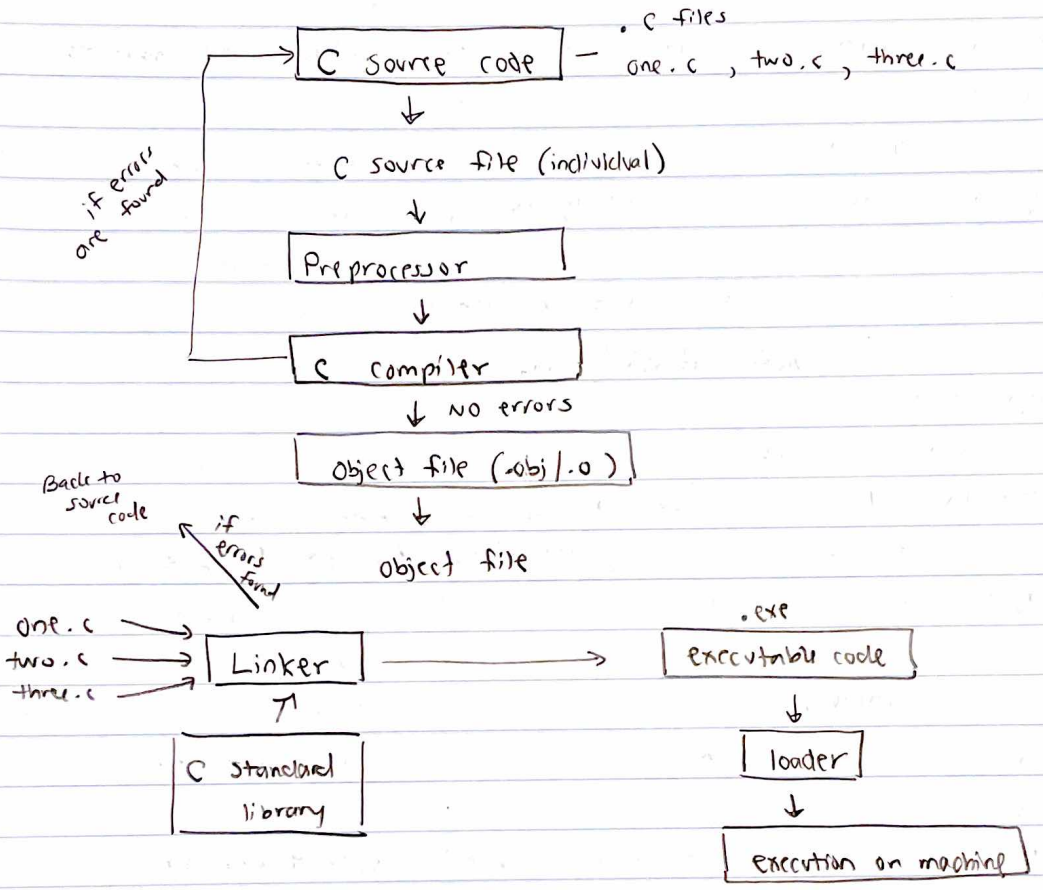
Basics

- Computer can only understand machine langvage code which is a combination of 0s and 1s
- three basic tools to build C applications : compiler, linker and archiver
- a collection of the above programs and possibly others tools are called a toolset
- First, the code is preprocessed, where all preprocessor directives are carried out such as #include, #define, ...
- The compiler then takes the source code after preprocessing and produces a object file (.obj) which can contain a mixture of machine-executable code and symbolic references to functions and data
- The linker basically takes all the object files and the C standard library and resolves everything, producing one final executable file
- errors will be shown for both the linker and compiler, separately
  - ↳ visual studio example :
    - forget to include int main () { } and you will get a linker error starting with LNK...
    - forget a semicolon (;) and you will get a compiler syntax error starting with C...
- The linker can produce either a exe file or dynamic library

Executable vs dynamic library

- an executable or application is any program that can be executed by the operating system
- dynamic library (shared library) is like an exe, but can not be run on its own (dll) - dynamic link libraries

How C works



- All these processes will be handled by a toolset, there are many toolsets
  - ↳ examples: GCC, turbo C, microsoft visual C, GNU C
  - → GNU toolchain / toolset – Richard Stallman
    - ↳ GCC compiler which means GNU C compiler
  - → To set GNU on windows → mingw or cygwin
    - ↳ single command to build, no editor or interface
- IDE – Integrated Development Environment
  - ↳ text editor, file management, compile / linker, see all errors and debug
  - → code blocks, eclipse, turbo C, microsoft visual studio, and much more!

How C works

Constants, Variables, Data types, keywords

- alphabets, numbers, and special symbols are combined to form constants, variables and keywords
- A <u>constant</u> is an entity that doesnt change
  - ↳ example: integer constants - 28, 102, 857, etc
    - float / real constants - 25.04, 100.581, 99.999, etc
    - character constants - 'a', 'A', 'Z', '=', '?', etc
- A <u>variable</u> is an entity that does change
  - In programming, we have to store the result of calculations, and it is stored in the computer memory
  - variable names are the names given to the memory locations
- Keywords - by using keywords, we can construct the meaningfull statements
  - predefined words the compiler knows / understands
    - int, char, if, for, else, ... etc
  - keywords can not be used as variable names
- data types - kind of data stored, ie: integer, float, characters

<u>There are 32 keywords in C</u> - (There are 60 in C++)

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void
volatile	while				

Data types in C

- each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it.
- Very common data types in C
  - char – most basic, requires one byte of memory
  - int – stores an integer
  - float – stores decimal point values, single precision
  - double – double precision decimal point values
- int – 4 bytes typically
  - ↳ signed – range −2B to 2B about

    4 bytes = 4 * 8 bits = 32 bits

    max = 2^{32-1} = 2.1B ← one bit reserved for sign (+ or −)
  - ↳ unsigned will have 2^{32} = 4.29B

Data type	memory (bytes)	Range
short int	2	−32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	−2,147,483,648 to 2,147,483,647
long int	4	−2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-2^{63} to $(2^{63})-1$
unsigned long long int	8	0 to 18.446×10^{18}
char	1	−128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	

- modifiers – short, long, long long can be used to modify data type size and unsigned can be used to use only positive values
- At the end of the day, characters are just numbers in ascii 'A' = 65

Data types in C

- Size of operator can be used to check the size in bytes of any data type

example:

```
#include <stdio.h>   // input and output
int main()
{
    int a = 1;
    char b = 'A';
    double c = 3.14;
    printf("Hello world!\n");      // \n for new line
    printf("%d\n", sizeof(a));
    printf("%d\n", sizeof(b));
    printf("%d\n", sizeof(c));
    printf("%d\n", sizeof(float));
    return 0;
}
```

- Boolean / bool data type - basic C doesnt support bool, but there are a few ways to implement it

- #include <stdbool.h> // C99

OR

- #define bool int
  #define true 1
  #define false 0

OR

- typedef enum {false = 0, true = 1} bool;

- Strings - also needs to be implemented, but rather easy - declare as a one dimensional array

→ char string_name [size]; → ie. char eric [4] = "eric";

```
#include <stdio.h>
int main()
{
    char str[] = "Geeks";
    printf("%s", str);   // output Geeks
    return 0;
}
```

Data types in C

- Type casting in C
- String.h

C standard library

- the C standard library - tutorial point - orange!
- <assert.h>
  - ↳ basically a macro to help debug code
    defined as: #define assert (ignore) ((void) 0)
- <Ctype.h>
  - ↳ declares several functions that are useful for testing and mapping characters
    All non-zero (true) return if satisfied otherwise zero
    examples: int isalnum (int c), int isalpha (int c), int iscntrl (int c), ...
- <errno.h>
  - ↳ can be set up to indicate what went wrong when an error occurs
- <float.h>
  - ↳ a set of various platform-dependent constants related to floating point values
    examples: FLT_MAX, FLT_MIN, FLT_MANT_DIG
- <limits.h>
  - ↳ Determines various properties of the various variable types. The macros defined in this header, limits the values of variably types
    examples: CHAR_BIT, SCHAR_MIN, SCHAR_MAX
- <locale.h>
  - ↳ defines the location specific settings, such as date formats and currency symbols
- <math.h>
  - ↳ defines various mathmatical functions - take double and return double
    examples: double acos (double x), double log (double x), double tan (double x), double cos (double x), double exp (double x), double sqrt (double x) and many more
- <setjmp.h>
  - ↳ Defines macro setjmp(), function longjmp() and variable jmp_buf
- <signal.h>
  - ↳ Defines variables, macros and functions associated with signals reported during a programs execution
- <stdarg.h>
  - ↳ used to get the number of arguments in a function when the number of arguments are not known

C Standard Library

- <stddef.h>
  - ↳ defines various variable types and macros, many also appear in other headers
  - examples: size_t, NULL, and a few more
- <stdio.h>
  - ↳ handles input and output - File handling, printing
  - function examples int fclose (FILE *stream), int ferror (FILE *stream), int printf (...)
  - 40+ other useful functions
- <stdlib.h>
  - ↳ various functions for general things
  - examples: void *calloc (..), void free (...), void exit (...)
- <string.h>
  - ↳ defines a string which is an array of chars and manipulation of them
  - examples: void *memchr (...), void *memcpy (...),
- <time.h>
  - ↳ manipulating time and date
  - examples: clock_t clock(void), struct tm
- <stdbool.h>
  - ↳ defines boolean to be true or false - See page 7 for examples

Comments

- leaving notes for your furture self or other ppl
- Preprocessor - just deletes them all out before compiling
  - ↳ // Single line comment
  - ↳ /* multi-line comment */

Basic Hello World Program

```
#include <stdio.h>
#include <conio.h> // console input/output
int main ()
{
  printf ("Hello world");
  getch(); // waits for a keystroke , visual wanted _getch() instead
  return 0;
}
```

```
#include <stdio.h>
#include <conio.h>
int main ()
{
 int age;
 float weight = 150;
 char sex;
 age = 26;
 sex = 'm';
 printf("%d %of %c", age, weight, sex);
 getch();
 return 0;
}
```

printf function

- function within C standard library <stdio.h>

/* printf("format strings", list of variables); */

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int age = 26;
  printf("Hi this is eric \n my Age is %d", age);
  getch();
  return 0;
}
```

New line

/* print int - use %d
print float - use %f
print char - use %c
*/

print in hex - %X
print in hex with 0X start %#X
print memory address - %p
↳ printf("%p", &ptr);

Scanf function

- function in <stdio.h> to take input from the keyboard

/* scanf("format string", list of address variables); */

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int age;
  printf("Enter your age\n");
  scanf("%d", &age);
  printf("Your age is %d", age);
  getch();
  return 0;
}
```

Arithmetic Operators

Add - x+y

subtract - x-y

multiply - x*y

devide - x/y ← make sure $y \neq 0$

Increment and decrement operators (++, --)

x++ // Add 1 to x, evaluates to original x

y-- // subtract 1 from y, evaluates to original y

} Postfix

++x // Add 1 to x, evaluates new value

--y // subtract 1 from y, evaluates new value

} Prefix

modulus and short_hand operators (%, +=, -=, ...)

modulus (%) - remainder operator ie. num1 % num2

```
num2 ) num1
       ----
       remainder
```

short hands

num = num1 + num2 instead use num1 += num2;

can also use -=, *=, /=, %=

statements

x=y; // every expression is a statement

int x; // declarations are statements

; // empty statement

{ // A block is a statement

Logical and comparison operator

Comparison

```
x < y     // less than
x > y     // greater than
x <= y    // less than or equal to
x >= y    // greater than or equal to
x == y    // equal to
x != y    // not equal to
```

Logical

```
x & y     // bitwise and
x ^ y     // bitwise not
x | y     // bitwise or
x && y    // x and y (evaluates if both not 0)
x || y    // x or y (evaluates if x or y is true
```

logical - comparison example

```
if (date >= 1)
    if (date <= 31)
        printf("date is valid");
```

instead use ⟹

```
if( date >= 1 && date <= 31)
    printf("date is valid");
else
    printf("date not valid");
```

Bitwise logical operator truth tables

& -(AND)		
0	0	0
0	1	0
1	0	0
1	1	1

| -(OR)		
0	0	0
0	1	1
1	0	1
1	1	1

^ - (XOR)		
0	0	0
0	1	1
1	0	1
1	1	0 (Not both)

~ - (complement)

```
010 -> 101   // flips bits
011 -> 100
```

extra

logical and &&

```
if( condition1 && condition2)
```

logical or ||

```
if( condition1 || condition2)
```

shifting << (left) and >> (right)

```
0110 >> 1 = 0011
0110 << 1 = 1100
```

☆ Can also use the short-hands

&=, |=, ^=, ~=, >>=, <<=

Conditional Statements if, else if, else

```
if (condition)  // will either be true or false
{
    execute if condition is satisfied  // brackets not needed for one line statement
}
else
{
    execute if condition not met
}
```

Use comparison operators for conditions (==, !=, <=, >=)

Ternary Operator (?:) — one line if statement

```
(value == 5) ? (true) : (false);
```

condition — if true execute (?) — else execute (:)

→ useful to use in preprocessor macros

```
(value == 5) && (than expr);  // only need than expression
(value == 5) || (else expr);  // only need else expression
```

if (condition) { } , else if (condition){}, else { }

Check for null pointer

```
const char* ptr = "Hello";
if (ptr)
    printf("Exists");
```

☆ you can use debugging to check if conditional statements are executed or not

logical not can be used for error statements

```
if (!condition)
{
    if condition is false, it will evaluate
}
```

While and do while loops

<u>while loop</u>

```
while (x) { expression }    // while x is true or = 1, execute the expression,
                                x can also be conditional

int main ( ) {
  int counter = 1;
  while ( counter <= 30 ) {   // while loop prints statement 30 times
      printf ("Hello");
      counter ++;
  }
  return 0; // must return because main is a function with int return ( 0 means no errors)
}
```

- be careful of infinite loops, program can get stuck i.e. while ( )

<u>do while loop</u>

```
do { statement } while ( expression );

int main ( ) {
  char input;
  do {
   printf ("Hello");
   scanf ("%c", &input );   // doesnt quite work, need mod
  } while (input != 'x');
  return 0;
}
```

* If a condition already exists, use a while loop
* Do while have a lot less uses than while and for loops

for loops

for (x; y; z) { execute }

x - initialization

y - condition

z - updation

for (int i=0; i< 20; i++) { execute } // executes 20 times then kicks out

- make sure they system can't get stuck in for loop

for (variable declaration; conditional; code called) { }

```
for ( int i=0; i<5; i++ )   //runs through 5 times
{
  printf ("Hello");
}
```

* you can even move things around

```
int i = 0;
for (  ; i<5;   )
{  //print
   i++;
}
```

- very useful for traversing arrays and executing a statement x many times
- searching for things

Arrays

- container to store many values in one place

<u>single dimensional array</u>

- int array [6];

→ collection of similar elements of same data type

syntax: type name [element number];

- int array [6] = { 10, 20, 30, 40, 50, 60 } // filling array of size 6

  possitions 0 1 2 3 4 5 // note they start at 0

- accessing array - start at 0 - Get

  array [2]; // will equal 30

- individual setting - Set

  array [0] = 15; // change position 0 value to 15

- when filling an array when declaring, size not needed

  int array [] = { 10, 20, 30 };

<u>Multidimensional array</u>

- creating an array build of an array

  array [x] [y]; // x rows and y columns

  int array [2] [6] = {

	0	1	2	3	4	5
0	10	20	30	40	50	60
1	12	22	33	44	55	66

  };

- accessing

  getarray = array [0] [2]; // sets to 30

  x y

- you can create as many dimensions you need

Functions

A set of statements that takes inputs, executes some specific task / computation, and then produces an output

Syntax: returnType functionname (parameters-inputs) { expressions }

- function prototyping – if you declare a function / create it after main function then you need to say it exists

```
void display ()
{
  printf(" Hello ");
}
int main () {
   display();  // function call
   return 0;
}
```

OR

```
void display () //prototype
int main () {
   display();
   return 0; }
void display ()
{
  printf ("Hello");
```

example:

```
int addTwo (int , int);
int main () {
  int result;
  result = addTwo (20, 30);
  printf ("%d", result);
  return 0;
}
int addTwo (int a, int b) {
   return a+b;
}
```

- if a parameter is initialized in the function then you don't need to put it in the call
  ↳ int addTwo (int a , int b= 5) → call addTwo (5) //adds 5+5
- default return type if not listed is int

Functions

inline function - C99 - otimized for speed

- used for very short / simple functions to speed up process
- automatically places code in place of where it is called

```
inline void display() {
   printf("Hello");
}
```

- then call it like a regular function

```
//function takes two values and returns the max value
int max (int x, int y) {
  if (x>y)
    return x;
  else
    return y;
}
```

→ can return only one value unless pointers are used

- when program reaches a function, it reserves variable and result space on the stack then jumps to the definition of the function then jumps back, it takes time. unless compiler decides to make it an inline on its own.

Functions

Global variable scopes

scope - region of a program

variable scope - region of a program where a variable is accessable

can be declared in three places

↳ declared inside a function

→ local scope

↳ declared outside a function

→ global scope

global scope - can be accessed anywhere

Control flow - return

```
int multiply (int a , int b) {
   result = a*b;
   return result;
}
```

- must return same type as outlined in function declaration
- you can place returns anywhere in function
- exits once return is come across

Control flow - break

- break statement will stop a loop and go back to the regular program
- example: looking for things and switch statements

```
int main() {
  int counter;
  for (counter = 1; counter <=10; counter++) {
    print("%d", counter);
    if (counter==5)
        break; //stop looping
  } return 0; }
//output: 1, 2, 3, 4, 5
```

Control flow - continue

- skips the rest of the current loop and forces it to continue again

```
int main() {
  int counter;
  for (counter = 1; counter <=10; counter++) {
    if (counter ==5) {
        continue;   //skips remainder and loops again
    }
    printf("%d", counter);
  }
//output: 1, 2, 3, 4, 6, 7, 8, 9, 10
```

Switch statement

Basically many if statements based on a condition with a default case if none of the cases are entered

```
switch (expression)
{
  case constant expression :
      statements;
      break;
  case constant expression 1
      statements;
      break;
// any number of cases
  default:
      statements;
}
int main() {
  char grade = 'A';
  switch (input) {
    case 'A' :
        printf("excellent");
        break;
    case 'B' :
        printf("very good");
        break;
    default :
        printf("invalid grade");
  }
```

switch statement

Address operator - & operator

The address of operator - & is used to get an address location of a variable in memory

```
int main() {
  int age = 100;
  int weight = 60;
  printf("%d", age); //outputs 100
  printf("%p", &age); // outputs 007DFBD4 ← address location in memory
```

Pointers

- Variable that contains an address, location for another variable

```
int main() {
  int age = 26;
  bool human = true;  //need <stdbool.h>
  int *ageptr;   //make empty pointers
  bool *humanptr;
  ageptr = &age;  // fill pointer with address
  humanptr = &human;
  return 0;
}
```

- used to more easily complete tasks, used for dynamic memory allocation

- A pointer is an integer that stores a memory address
- Pointers are used to manipulate memory

```
int main() {                                  * NULL is just #define NULL 0
  //void *ptr = NULL or = 0        // C++ has nullptr keyword
  int var = 8;
  void *ptr = &var;
```

pointer to start address — mem address of var

- Double pointer

```
int main() {
  int var = 8;
  int* ptr = &var;
  int** dptr = &ptr;
}
```

- types really don't matter much, just an address

Pointers

- dereferencing a pointer *ptr - read or write to it

```
int main() {
  int var = 8;
  int* ptr = &var;
  *ptr = 10; // changes the value stored at this memory location to 10,
             // var is this memory data, var will now equal 10
```

Passing an Array to a function

example:

```
void show (int[] ,int);
int main() {
  int numbers [] = { 5,10,15,20};
  int length = 4;
  show (numbers, length);
  return 0;
}
void show (int numbers[] , int length ) {    //print entire array
  for (int counter = 0; counter < length; counter ++ ) {
      printf( "%d", numbers [counter]);
}
```

Pass by address - Pass by reference (pointers)

```
void display (int *ptr);
int main () {
  int age = 26;
  display (&age ); //pass in address of age
  return 0;
}
void display (int *ptr) {
   printf ("%p", ptr );
   *ptr =100; // change value to 100
```

Relationships between arrays and pointers

```
int main () {
  int number [5] = { 29, 55, 44, 88, 89 };
  printf ("%d", number); //prints memory address of number [0] value 29
  printf ("%d ", *number); //prints value at number [0] spot - 29
  return 0;
}
```

→ we can access an array location two ways: index and pointer

number [2] → value of 44

*(number + 2) → value of 44

Const Keyword

- makes an element read-only, can not change it
- many different uses
- Safer code / protection

```
int main() {
  const int pi = 3.1415;  // Need to set variable when you declare it
```

function example:

```
void display ( const int num[] , int limit )
{    }
```

// makes the array read only and the function can not modify the values
// at all //

Array Ranges in functions

example:

```
int numbers [] = { 10, 12, 13, 44, 11, 12, 17, 18, 19 };
display (numbers , numbers +9); //passing by address

void display ( const int *start , const int *end ) {
  const int *ptr;
  for (ptr = start; ptr != end; ptr ++ ) {
    printf ("%d", *ptr ); //prints all array elements
```

Structures - struct

- A collection of elements of different types, basically an array that excepts different data types
- Basically a simplified class from c++ without data protection, overloading, functions

```
struct student {
    int idnum;
    char sex;
} eric, mike; // creates a struct for eric and mike globally

int main() {
    student evan, kyle; //local decler for evan and kyle

// How to add contents to a struct
    student kyle = { 1111, 'm' }; //set during init
    eric.idnum = 2222; // added after init with dot operator
    eric.sex = 'm';

    printf("%d %c", eric.idnum, eric.sex );

    return 0;
}
```

Structures - struct

Struct - Arrow operator with pointers

- can be used to access struct variables through a pointer

ex)

```
struct student {
  int idnum;
  char sex;
};
int main () {
   student eric; // create eric named student struct
   student *ericptr; // create ericptr named struct pointer
   ericptr = &eric; // fill
   eric.idnum = 1234; //dot operator
   ericptr -> sex = 'm'; // arrow operator with pointer
   printf("%d", ericptr -> idnum); //print with arrow
   printf("%d", eric.sex); //print with dot
```

Passing structure to function by value, pointer

```
struct student {
  int idnum;
  char sex;
}
void display (student s) {
   printf("%d", s.idnum);
   s.idnum = 0000; // will not change value in main() because passed by value
void show (student *s) {
   printf("%d", s->idnum);
   s.idnum = 0000; // will change because passed by pointer
// call with
student eric = {1234, 'm'};
display (eric);
show (&eric);
- // define structures before function prototypes
```

Struct – nested and dot operator

```
struct address {
   int houseNo;
   char street [ ]; }
struct student {
   char name [ ];
   int rollno;
   address addr; } //nested structure
int main() {
   student eric;
   eric.name = "eric";
   eric.rollno = 1234;
   eric.addr.houseno = 68;
   eric.addr.street = "sumac";
   //printf ("%d", eric.addr.houseno);
   return 0;
}
```

Struct – nested and -> operator

```
   student eric;
   student *ericptr = &eric;
   ericptr -> name = "eric";
   ericptr -> addr.houseno = 68;
                 ↓not pointer, so use dot
   //printf("%d", ericptr -> addr.houseno
```

Should set up to pointer *addr to use arrow operator

```
address addr;
address *addrptr = &addr;
```

Size of operator - sizeof( )

Keyword - sizeof is a compile time operator that determines the size in bytes of a variable or data type

```
int main() {
  printf("%d", sizeof(int));
        "        sizeof(short int)
        "        sizeof(char)
        "        sizeof(float)
                 :

  int age;
  printf("%d", sizeof(age));

// Arrays
  int array[10];
  sizeof(int); // equals 4
  sizeof(array); // equals 40 = 4*10
  sizeof(array[0]); // equals 4
  sizeof(array / sizeof(array[0]); // number of elements in an array

  for (int i=0
```

Unions

Unions are a user defined data types similar to a struct but in unions, you can only store the value in only one member at a time because the memory is shared between the members of the union. Union memory is shared between all members.

syntax:

```
union unionName {
    datatype data;
    datatype data2;
}
```

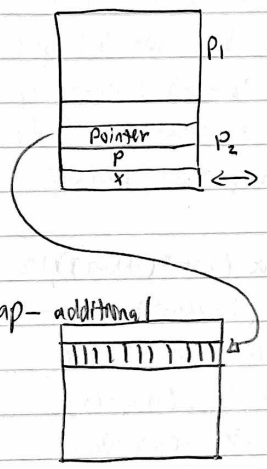
ex1

```
union employee {
    int id;
    float salary;
} eric, eran;   // can declare right away like structs
int main() {
    employee mike, todd;
    todd.id = 25;
    printf("%d", todd.id);  //25
    eric.salary = 10;
    printf("%d", eric.salary);  // 10
    printf("%d", todd.id);  // 10
      ↳ // the value was overwritten because only one storage slot in memory
```

Memory Allocation

<u>dynmaic memory allocation</u> is the process of allocating memory while the program is running. The system will allocate space to store variables and data called the stack.

Stack - normal storage



int x = 10;
float p = 6.5;

<-> comes in and out but if you have too much memory to store, it may over write things

heap is additional memory separate from the stack memory to store large things to keep the stack safe

→ create a pointer in stack that points to memory location in the heap

Memory allocation - malloc and free

- need #include <stdlib.h> to use malloc and free

Syntax:

```
void *malloc (no_of_bytes); // allocate
void free (current_storage_pointer); // deallocate memory
```

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

int main() {
  int *p;
  p = (int *) malloc (sizeof (int));
  if(p == NULL) {
     puts ("failed to allocate memory");
  }
  else {
  *p = 20;
  printf ("%d", *p);
  free (p);
  }
  getch();
  return 0;
}
```

```
// for a float
float *p;
p= (float *) malloc (sizeof (float));
  : and so on
```

memory allocation - malloc and free

ex) user input

```
int main() {
  char *p;
  int length;                    *//puts is a simple print function
  puts ("Enter length of string");
  scanf ("%d", &length);
  p= malloc (sizeof (char)* length+1);   plus one because of extra null char in strings
  if ( p == NULL) {
    puts ("memory not avalible");
  }
  else {
    fflush (stdin); //flush screen buffer
    puts ("enter value");
    gets (p); // stores input
    puts (p); // print what you entered
    free (p); // free memory
  }
  return 0;
}
```

memory allocation — calloc and free

→ need #include < stdlib.h>

syntax:

```
void *calloc (no_of_items, size_of_each_item);
void free (current_storage_pointer);
```

```
int main() {
  int *marks, length, counter;
  puts ("number of subjects");
  scanf("%d", &length);

  marks = (int *) calloc (length, sizeof (int));
  if (marks = NULL) {
    puts ("unable to allocate memory");
  }
  else {
   for (counter = 0; counter < length; counter ++) {
      printf ("enter the mark of %d subject", counter + 1);
      scanf ("%d", &marks [counter]);
   }
  for (counter = 0; counter < length; counter ++) {
    printf ("%d\n", marks [counter]);

   }
  }
 free (marks);
 getch();
 return 0;
}
```

☆ will set initial values to 0 in the heap
↳ malloc will have garbage values

☆ It is better to use malloc over calloc because malloc is faster!
Unless you need the zero initialization

☆ You can achieve calloc with malloc and memset

```
ptr = malloc (size);
memset (ptr, 0, size);
```

memory allocation - realloc

→ lets you reallocate already allocated memory to increase or decrease the allocated size

syntax:

```
void *realloc (current_storage_pointer, new_size);
```

```
int main(){
  char *str;
  str = (char*) malloc (13);
  if (str == NULL) {
     puts ("unable to allocate memory");
  }
  else {
    strcpy (str, "learning lad"); // copy 12 char max string to str
    puts (str); //print

    str= (char*) realloc (str, 19); //reallocate memory
    strcat (str, " rocks"); // concatinates to the end
    puts (str); // prints learning lad rocks - 18 chars and one cant use =19
    free (str);
  }
  getchar();
  return 0;
}
```

memory allocation

File Handling — opening / closing

fopen ("file_name", "mode");

↳ modes — "r" - reading - must be double quotes
- if file is not present fopen will return NULL

"w" - finds, deletes, creates new, empty to fill up - write to
doesnt find, creates new, empty to fill up

"a" - append to, finds file and you can add to it
OR doesnt find it and it will make a new one to write to

"r+" - read and write mode, makes new if not found

"w+" - read and write, whips out contents if found

"a+" - read and write and appends

↳ modes for binary files - add a "b"

"rb", "wb", "ab", "r+b", "w+b", "a+b"

#include <stdio.h> has a structure called FILE

<u>example:</u> Opening a file and close it

```
#include <stdio.h>
#include <conio.h>
int main() {
  FILE *Fpointer;
  Fpointer = fopen ("exit.txt", "w");
  if ( Fpointer == NULL ) {
    printf ("unable to create file ");
  }
  else {
    printf( " file opened successfully ");
    fclose ( Fpointer ); //must close files after you are done
  }
  getch();
  return 0;
}
```

File Handling - writing to a file

```
#include <stdio.h>
#include <conio.h>
int main() {
  FILE *Fpointer;
  char data [25] = "Eric Rocks";
  int length = strlen (data); // sets to length
  int counter;
  Fpointer = Fopen ("eric.txt", "w");
  if ( Fpointer == NULL) {
     printf ("unable to create or open file");
  }
  else {
    for ( counter = 0, counter < length; counter ++); {
      printf ("writing the character %c \n", data [counter] );
      fputc (data [counter], Fpointer);
    }
    printf( "data written successfully");
    fclose (Fpointer);
  }
  getch();
  return 0;
}
```

File Handling – Additional functions

- gets(name) – string input function from keyboard

```
int main(){
  char name [20];
  gets (name);
  printf("%s", name);
  getch();
  return 0;
}
```

- fputs( ) – writing to files

```
int main(){
  FILE *pf;
  char input[25];
  pf =fopen ("exe.txt", "w");
  if (pf = NULL){
    printf("unable to open");
  }
  else {
    printf("enter string");
    gets (input);
    fputs (input, pf);
    printf("data entered successfully");
    fclose(pf);
  }
  getch();
  return 0;
}
```

File Handling – Additional functions

- fprintf() – print to file and formatable like the printf statement

syntax:

fprintf (file-pointer, "format specifier", list of variables);

example:

```
fprintf (pf, "%s \t %d", name, age);
// learningtad C programming 46 example
```

- fgetc() – Reading from files – reads individual chars

```
int main() {
  FILE *pf;
  char ch;
  pf = fopen ("eric.txt", "r");
  if (pf == NULL) {
    printf("unable to open");
  }
  else {                          // did we reach the end of the file – true if yes
    while (!feof(pf)) {           //read all contents
      ch = fgetc(pf);             // get each character individually
      printf ("%c", ch);          // print that character
    }
    fclose (pf);
}
```

File handling – additional functions

- fscanf ( ) – reads from files like scanf does with input

Syntax:

```
fscanf ( file-pointer, "formatstrings", list_of_address_variables );
```

example:

```
fscanf ( pf, "%s", data );
```

↳ data is a char array of 10

↳ pf is a fopen file pointer like before

- fgets( ) – read strings from file

Syntax:

```
fgets ( variable_to_store , length_to_read , file-pointer );
```

example:

```
fgets (data , 15 , pf); //stores 15 characters from file into data
// use a while (!feof(pf)){} loop to get all the data
```

- rename ( ) – allows you to rename a file name

syntax:

```
int rename ( old_file_name , new_file_name );
```

→ returns 0 if successful , otherwise returns nonzero if error occured

```
if( rename ("errs.txt", "miles.txt") == 0 ) {
    printf (" file renamed successfully");
}
else {
    printf( "Error while renaming file" );
}
```

File Handling – additional

- remove() – deletes a file in C

Syntax:

```
int remove (file_name);
```

if return is 0 → file deleted successfully

non zero → Error while deleting

→ if(remove("err.txt") == 0) { print success } else print failure

- ferror and clearerr functions

syntax:

```
int ferror (FILE *fp);    // detect read/write errors
void clearerr (FILE *fp); // clear errors
```

- rewind() – reset file position to the beginning

syntax:

```
void rewind (file *fp);
```

- tmpfile() – creates a temporary file

Syntax:

```
FILE *tmpfile();
```

↳ creates a temporary binary file and opens in w+b mode

↳ used to handle processing of a lot of data when you dont want to use variables

- tmpnam() – creates a temporary file name

Syntax:

```
char *tmpnam (char *name);
```

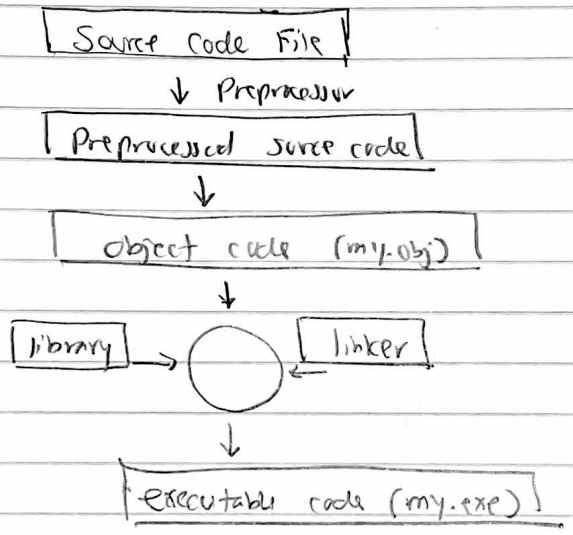
Strings and string functions

- There is no std::string like in C++ so you must define a string as an array of char type values

```
int main ( ) {
    char source[] = "Eric Moser";  // create a string
    char target [25];

    // strcpy (target , source)   // string copy function

    strcpy (target , source );
    printf ("%s \n", source );
    printf ( "%s \n", target);

    // strcat (target, source ) // concatinate to string
    char extra [6] = "Rocks";
    strcat ( target , extra );
    printf ("%s \n", target); // prints Eric Moser Rocks

    // strlen ( string)  // length of string
    char name = { 'E','r','i','c', '\0' };   null char
    char sirName [] = " Moser";
    int length , length2;
    length = strlen (name);
    length2 = strlen (sirName);
    // printfs
```

Preprocessor Directive #define

pre-processor : The C preprocessor modifies source code before handing it off to the compiler, allowing conditional compilation with #ifdef, defining constants with #define, including header files with #include



→ # define

```
#define PI 3.14159      // will replace all "PI" in code with 3.14159 before compiling
                        // notice no " = " sign or " ; " signs used

// function for area of a circle
float circleArea (float rad) {
   float result;
   result = (r * r) * PI ;    // will replace PI with 3.14159 before compile
   return result;
}

// additional
#define ONE 1
#define TWO ONE + ONE
#define THREE TWO + ONE
```

★ Make sure to Always use all caps to define a #define!

Preprocessor directives #include

- include a header file or other source files with #include
- file named eric.c

```
void display ( ) {
    printf ("This printed from a function in another file");
}
```

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

#include "eric.c"

int main ( ) {
    display ( );

    getch();
    return 0;
}
```

- very simple → finds file and pastes it into this file when compiling obj file

< > - relitive refrence

" " - everything else, can even use in place of < >

```
#include <stdlib.h>  // C standard library
```

Preprocessor directives — function like macros

// macro that takes number and makes sure it is always positive

```
#define ABS(number) (number) < 0 ? -(number) : (number)   //ternary operator

int main () {
    printf("%d", ABS(-2));
    printf("%d", ABS(4));
```

Conditional Compilation Macros: #if, #endif, #elif, #else

```
#define WINDOWS 1
#define MAC 2
#define LINUX 3
#define OS LINUX

int main () {
#if OS == WINDOWS          // will only compile the true statement
    printf("windows");
#elif OS = MAC
    printf("mac");
#else
    printf("linux");
#endif                     // will then compile the rest of the code
    printf("Hello world");
    return 0;
}
```

Conditional compilation macros - ifdef and ifndef

ifdef - if defined

ifndef - if not defined

```
#include <stdio.h>
#define ERIC 1
int main () {
 #ifdef ERIC
     printf("Hey Eric");
 #else
    printf( "Hey");
 #endif

#ifndef SAM
    printf("No sam here");
```

Preprocessor directive #undef

- Used to undefine a #define

```
#define ERIC 1
int main () {
  #undef ERIC
  #ifndef ERIC
     printf("not defined");
  # endif
  return 0;
}
```

Preprocessor Directive

#pragma once

↳ make sure that this is only included once in a file

↳ used for header files

↳ #ifndef is used also for the exact same thing

typedef

- typedef is used to rename data types, just giving a new name to an existing data type. Can be used to make code easier to read.

Syntax:

```
typedef type new_name
```

```
int main() {
  typedef int eric_int;
  eric_int age = 26;
  printf("%d", age);
  typedef eric_int test_int; // can make new also
```

typedef structures

- Used to give a name to a structure

Syntax:

```
typedef struct_or_union {
    type member_1;
    type member_2;
    :
} type_id;
```

```
int main() {
  typedef struct {
    int age;
    char name[10];
  } person;

  person var;
  var.age = 26;
  strcpy(var.name, "eric muser");
```

Storage classes in C - Automatic

- when the variable will be stored, scope, life
- default storage class is automatic
- Keyword : auto

* stored — stack
* initial value — garbage
* scope — local to block
* life — entire program

```
int main() {
  auto int x;  //variable stored in memory, initial value = garbage, scope is local
               // to the block it was defined in (block = { })
```

→ Not really any practical use as it is just the default anyways, C++ can use auto for automatic type detection

Storage class — Register

- Used to store variables that are very frequently used, stored in cpu register — fast
-
- Keyword : register

* stored — cpu
* initial value — garbage
* scope — local to block
* life -- exist until control leaves block defined

```
int main() {
  register int x; //default value = garbage, scope = local to block defined
                  // life = exist until control comes out of the block
                  // limited storage for registers on the cpu
                  // if you exceed the cpu storage, the stack will be used
```

Storage classes - static

- most commonly stored in the heap, default initial value = 0, the scope of the static variable is local to the block defined and the life time will be the entire program life.

- when a variable is declared as static, space for it gets allocated for the life time of the program on the heap. Even if the function is called multiple times, space is allocated only once. The variable in previous calls is carried through. Useful for implementing co-routines in C or any application where the previous state of a function/variable needs to be stored.

```
int main() {
    display(); // prints 0
    display(); // prints 1
    display(); // prints 2
}
void display() {
    static int counter = 0; //sets to zero on first call
    counter = counter++;
    printf("%d", counter);
}
```

- ☆ stored - heap
- ☆ initial value - 0
- ☆ scope - local to block
- ☆ life - entire program

☆ Java doesn't allow static local variables in functions

Storage classes in C - external

- Keyword: extern
- will be stored in the heap, initialized to a value of 0, the scope of the extern variables will be global, life will be entire program life
- Used if you need to use a variable in many functions but dont want to pass it

* stored - heap
* initial value - 0
* scope - global
* life - entire program

```
int main() {
  extern int x;
  printf("%d", x);
  display();
  return 0;
}
int x = 10;
void display() {
  extern int x;
  printf("%d", x);
}
```

time and date functions

- must use the #include <time.h>
- take a look at the time.h file, many values, structure, functions

- functions

time()

localtime()

gmtime()

asctime()

ctime()

strftime()

localeconv() - locational

mktime()

difftime()

clock()

All have examples on learningtad ep 80-86

Character and string functions

- #include <ctype.h> // character functions → needed for these functions
- #include <string.h> // string functions

Syntax:

```
int isalnum (int ch); // return non-zero if character is a alphabet or digit
int isalpha (int ch); // return non-zero if character is an alphabet character
int isdigit (int ch); // return non-zero if character is a number
```

example:

```
char ch;
if (isalpha (ch)) {
   printf("%c is a alphabet\n", ch);
}
else {
   printf("%c is not a alphabet\n", ch);
}
```

Syntax:

```
int isprint (int ch); // if character is printable or not, ASCII are defined from 0x20 to 0x7E

int iscntrl (int ch); // if a character is control character or not
                      // ASCII values 0 to 0x1F or equal to 0x7F (DEL)
```

```
int islower (int ch); // checks if a character is lower case
int isupper (int ch); // checks if a character is an upper case
```

Convert a character to or from lower case to uppercase

```
int tolower (int ch);
int toupper (int ch);

int ispunct (int ch); // if punctuation character or not
int isspace (int ch); // if character is a space or not
```

String functions

Syntax: #include <strings.h>

- void *memch (const void *buffer, int ch, size_t count);

- int memcmp (const void *str1, const void *str2, size_t count);

if return value is

< 0 str1 is less than str2

= 0 str1 is equal to str2

> 0 str1 is greater than str2

- void *memcpy (void *to, const void *from, size_t count);

- void *memmove (void *to, const void *from, size_t count);

- void *memset (void *buf, int ch, size_t count);

Additional functions - stdlib.h

#include <stdlib.h>

- int atoi (const char *str);
  ↳ converts a character to its integer/decimal value

```
char num1[] = "12";
char num2[] = "34";
int number1, number2;
number1 = atoi (num1);
number2 = atoi (num2);
```

- div_t div (int numerator, int denominator);

// gets the quotient and remainder of a division operation

The structure type div_t has 2 elements

→ int quot;

→ int rem;

```
div_t n;
n = div(10, 3);
printf("the quotient is %d\n The remainder is %d", n.quot, n.rem);
```

Additional functions – math.h

– #include <math.h>

double sqrt (double num);
float sqrtf (float num);
long double sqrtl (long double num);

squared root $\sqrt{x} = x^{1/2}$

double cbrt (double num);
float cbrtf (float num);
long double cbrtl (long double num);

cubed root $\sqrt[3]{x} = x^{1/3}$

→ they all return the result of the operation

double ceil (double num); // return smallest integer, not less than value passed
// ceil (9.9) → returns 10

double floor (double num); // return largest integer, not greater than value passed
// floor (9.9) → returns 9

double round (double arg); // return nearest integer value
// round (9.9) → returns 10

double pow (double base, double exp); // used to calculate the value of a number raised to a certain power

// pow (2.0, 8.0) → returns 256, or $2.0^{8.0}$