MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.1900 (6.0004): Introduction to Low-level Programming in C and Assembly

Fall 2022, Quarter 2

Name:	**Kerberos:**
	MIT ID #:

#1 (17)	**#2 (11)**	**#3 (8)**	**#4 (16)**	**#5 (20)**	**#6 (12)**	**#7 (16)**	**Total (100)**

Exam content is on **both sides** of the exam sheets.

Enter your answers in the spaces designated in each problem. Show your work for potential partial credit.

This page intentionally left blank

Problem 1. Binary Encoding and Arithmetic (17 points)

A. (2 points): What is `17` in 8-bit two's complement notation? What is `-17` in 8-bit two's complement notation? Please write your answers in binary.

17 in 8-bit 2's complement notation (0b):
-17 in 8-bit 2's complement notation (0b):

B. (2 points) The 2026 FIFA World Cup will have `48` participating teams. How many bits would be needed to represent the 48 unique values `0-47`? If you are declaring a C variable that needs to be able to represent the values `0-47`, what data type should you use to minimize the number of bits that go unused?

Number of bits needed:
C data type used:

C. (3 points): What is `(0xE0 & 0xA1) | 0xF5`? Provide your result in both unsigned 8-bit binary and unsigned 8-bit hexadecimal.

Result in unsigned 8-bit binary (0b):
Result in unsigned 8-bit hexadecimal (0x):

Problem continued on next page.

D. (3 points): Compute the 8-bit two's complement sum of `0x22` and `0xFA` using two's complement arithmetic. Provide your answer in 8-bit two's complement binary notation. **If the result cannot be expressed in 8-bit 2's complement, write "Not Possible". To receive credit, you must show your work using two's complement arithmetic.**

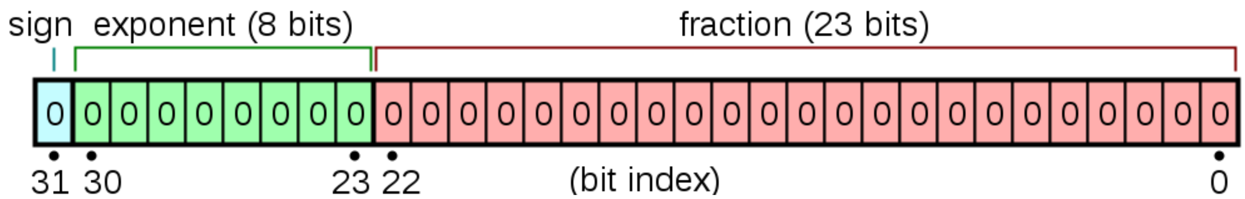
`0x22` + `0xFA` **in 8-bit two's complement binary (0b):**

E. (4 points): You have a 5-bit value with the binary encoding `xyz10` where `x`, `y`, and `z` can be either a 0 or a 1. Determine the two intermediate bitwise operations that should be performed on this number in order to end up with the result **`x1z01`.** In other words, toggle bits 0 and 1 (so that a 0 "flips" to a 1 or a 1 flips to a 0) and set bit 3 to be 1. Bits 2 and 4 should not be modified. For each intermediate operation, specify both the operator and the value of the second operand (ex. and 01010).

First bitwise operation to perform on `xyz10`
Second bitwise operation (to be performed on the result of the first bitwise operation):

Problem continued on next page.

F. (3 Points) What is the decimal equivalent of the 32-bit floating point number `0x414c0000`? The format of 32-bit floating point encoding is shown below. Show your work for full credit. *Note that the number shown in the figure is NOT* `0x414c0000`*.*



$$\texttt{Value} = (-1)^{\text{sign}} \cdot 2^{\text{exp}-127} \cdot \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

Decimal equivalent of 32-bit floating point number 0x414c0000:

Problem 2. The Incredible Bulk (11 points)

We build a struct that enables us to modify multiple bits stored in a `uint32_t` somewhere in memory.

```
#include <stdint.h>
struct bulkReadOp {
  uint32_t *valAddr; // address of value to modify
  uint8_t start;     // first (less significant) bit to read/write
  uint8_t end;       // last (more significant) bit to read/write
};
```

The struct member `valAddr` holds the address of a 32-bit integer. Struct members `start` and `end` hold the indices of which bits to read from the value located at address `valAddr.` So, a bulk operation will read bits `start` through `end`. An example of a 3-bit wide `bulkReadOp`, located between bits 5-7 at address `UNDISCLOSED_ADDRESS`, is shown below:

```
struct bulkReadOp b;
b.valAddr = UNDISCLOSED_ADDRESS; // address of value to read
b.start = 5;                     // start at bit 5 (inclusive)
b.end = 7;                       // end at bit 7 (inclusive)
```

A. (5 points) Write a function `bulkMask` that returns a `uint32_t` value in which bits at positions `start` through `end` (inclusive) are 1's and all other bits are 0's.

E.g., if `start=5` and `end=7`, `bulkMask` should return `0b00000000000000000000000011100000`.

For full credit, your solution should not use a loop or recursion.

```
uint32_t bulkMask(struct bulkReadOp op){









}
```

B. (6 points) Create a function, **`bulkRead`**, a function that receives **a pointer to a `bulkReadOp` instance** and returns the value of consecutive bits `start` through `end` in `valAddr`. The least significant bit of the result should correspond to the value of the `start` bit.

```
uint32_t x = 0b00010000001100001100000010110001;

struct bulkReadOp b;
b.valAddr = &x;
b.start = 5;                      // start at bit 5 (inclusive)
b.end = 7;                        // end at bit 7 (inclusive)

uint32_t result = readBulk(&b);   // result == 5 (0b101)
```

For full credit, your solution should *not* use a loop or recursion. You may assume your `bulkMask` implementation from the previous part is correct for use here.

```
uint32_t readBulk(struct bulkReadOp *op){

}
```

This page intentionally left blank

Problem 3. Arrays (8 points)

Consider the function below.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
      int x[] = {1, 3, 5};
      int y[] = {10, 30};
      int z[] = {100, 300};
      int* arr[] = {x, y, z};
      // CHECKPOINT
      return 0;
}
```

What would the following expressions evaluate to if inserted at `CHECKPOINT`? If the behavior is undefined, write UNDEFINED.

Expression	Evaluation
`*y`	
`*(y+1)`	
`**arr`	
`*(arr[0])`	
`*arr[0] + 2`	
`*(arr[0] + 2)`	
`arr[1][3]`	
`(arr + 2)[0][1]`	

Problem 4. Mystery Function (16 points)

Study the functions and determine what they do. An ASCII table is provided to you for reference at the end of the exam.

```
#include <stdio.h>

int len(char* str) {
    int count = 0;
    while(*str != 0) {
        count++;
        str++;
    }
    count++;        // Pay attention to this line
    return count;
}

void mystery1(char* s1, char* s2, char* s3) {
    char* s4 = s3 + len(s1);
    char* s5 = s4 + len(s2);

    while(s3 < s4) {
        *(s3++) = *(s1++);
    }
    while(s3 < s5) {
        *(s3++) = *(s2++);
    }
}

void mystery2(char* s, int strLen) {
    int i;
    for (i=0; i<strLen - 1; i++) {
        if(s[i] < s[i + 1]) {
            s[i] += 32;
        }
    }
    s[i] = 0;
}
```

Problem continued on next page.

A. (4 points) Consider the test code below:

```
char s1[] = "MIT";
char s2[] = "FUN";
char s3[100];

mystery1(s1, s2, s3);

printf("%s\n", s3);   // PRINT A
```

What will be printed by the line labeled `PRINT A`?

What will be printed by the line labeled `PRINT A`:

B. (4 points) Consider the test code below:

```
char s1[] = "MIT";
char s2[] = "FUN";
char s3[100];

mystery2(s1, len(s1));
mystery2(s2, len(s2));
mystery1(s1, s2, s3);

printf("%s\n", s3);   // PRINT B
```

What will be printed by the line labeled `PRINT B`?

What will be printed by the line labeled `PRINT B`:

Problem continued on next page.

C. (4 points) Consider the test code below:

```
char s1[] = "MIT";
char s2[] = "FUN";
char s3[100];

mystery1(s1, s2, s3);
mystery2(s3, len(s3));

printf("%s\n", s3);   // PRINT C
```

What will be printed by the line labeled `PRINT C`?

What will be printed by the line labeled `PRINT C`:

D. (4 points) Consider the test code below:

```
char s1[] = "MIT";
char s2[] = "FUN";
char s3[100];

mystery1(s1, s2, s3);
mystery2(s3, len(s1) + len(s2));

printf("%s\n", s3);   // PRINT D
```

What will be printed by the line labeled `PRINT D`?

What will be printed by the line labeled `PRINT D`:

This page intentionally left blank

Problem 5. Assembly Language (20 points)

(A) (2 points) Convert the 32-bit encoding **`0xFF52A393`** to its corresponding RISC-V assembly instruction. Make sure to include all operands of the instruction.

RISC-V instruction:________________________

For the RISC-V instruction sequences below, provide the hexadecimal values of the specified registers after each sequence has been executed. Assume that execution of each sequence ends when it reaches the `end` label. Also assume that all registers are initialized to 0 before execution of each sequence begins.

(B) (12 points)

The first instruction executed is located at address `0x100`.

`. = 0x100`
`    lui a1, 0x73`
`    addi a2, a1, 0x300`
`    li a3, 0x42`
`    slli a4, a3, 8`
`    ori a5, zero, 0x510`
`    andi a6, a5, 0x374`
`    lw t0, -8(a5)`
`    xori t1, zero, 0xFFF`
`end:`
 	
`. = 0x500`
`    .word 0x11111111`
`    .word 0x22222222`
`    .word 0x33333333`
`    .word 0x44444444`
`    .word 0x55555555`
	Value left in a1: 0x____________________
	Value left in a2: 0x____________________
	Value left in a3: 0x____________________
	Value left in a4: 0x____________________
	Value left in a5: 0x____________________
	Value left in a6: 0x____________________
	Value left in t0: 0x____________________
	Value left in t1: 0x____________________

Problem continued on next page.

(C) (6 points)
The first instruction executed is located at address `0x100`.

<pre>. = 0x100
      li a0, 0x234
      li a1, 6
      jal ra, mystery
      li a1, 5
end:
 	
mystery:
      mv t0, zero
loop: andi t1, a0, 1
      add t0, t0, t1
      srli a0, a0, 1
      addi a1, a1, -1
      bnez a1, loop
      mv a0, t0
      ret</pre>
	Value left in ra: 0x____________________
	
	Value left in a0: 0x____________________
	
	Value left in a1: 0x____________________

Problem 6. Call Me (12 points)

Ben Bitdiddle wants to translate the following C functions into RISC-V Assembly procedures.

```
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
int less_than(int a, int b) {
  return a < b;
}
void insert(int *A, int i) {
  while (i > 0 && less_than(A[i], A[i-1])) {
    swap(&A[i], &A[i-1]);
    i--;
  }
}
```

A. (2 points). The following is Ben's implementation for **swap** and **less_than**. For each procedure, determine whether the implementation follows the calling convention.

1. Does **swap** follow the calling convention? If not, why?

`swap:`
`  lw s0, 0(a0)`
`  lw s1, 0(a1)`
`  sw s0, 0(a1)`
`  sw s1, 0(a0)`
`  ret`
	Circle one: YES NO
	One-sentence explanation if NO is circled:

2. Does **less_than** follow the calling convention? If not, why?

`less_than:`
`  slt a2, a0, a1`
`  mv a0, a2`
`  ret`
	Circle one: YES NO
	One-sentence explanation if NO is circled:

Problem continued on next page.

B. (10 points). The following is Ben's implementation for **`insert`**. Unfortunately, the program does not adhere to the calling convention. Assuming that Ben's `swap` and `less_than` implementations follow the calling convention, **add appropriate instructions into the blank spaces <u>on the next two pages</u>** to make `insert` follow the calling convention. You may only:

- increment/decrement stack pointer
- load word from stack
- save word to stack.

You may assume that the implementation will work as expected once it follows the calling convention. You may not assume any further details about `swap` and `less_than` (e.g. the implementations may not be the same as part A and may override caller-saved registers). You may not need all the blank lines.

```
insert:  # parameters: a0 = A, a1 = i
  mv s0, a0
  mv s1, a1

insert_loop:
  ble s1, zero, insert_end

  # calculate the address
  slli t0, s1, 2
  addi t0, s0, t0    # A+4*i

  # get the values
  lw t1, 0(t0)       # A[i]
  lw t2, -4(t0)      # A[i-1]

  # set up arguments
  mv a0, t1
  mv a1, t2
  call less_than  # returns a0 = (A[i]<A[i-1])

  # check the returned value / break out of the loop
  beq a0, zero, insert_end
  addi t3, t0, -4  # A+4*(i-1)

  # set up argument
  mv a0, t0  # argument 0: A+4*i
  mv a1, t3  # argument 1: A+4*(i-1)
  call swap

  addi s1, s1, -1  # i--
  j insert_loop

insert_end:
  ret                             (Write your answers on the next page.)
```

Write your answers in the given blank lines:

```
insert:  # parameters a0 = A, a1 = i

  ________________________________

  ________________________________

  ________________________________

  ________________________________

  ________________________________

  mv s0, a0
  mv s1, a1

insert_loop:
  ble s1, zero, insert_end

  # calculate the addresses
  slli t0, s1, 2
  addi t0, s0, t0    # A+4*i

  # get the values
  lw t1, 0(t0)       # A[i]
  lw t2, -4(t0)      # A[i-1]

  ________________________________

  ________________________________

  # set up arguments
  mv a0, t1
  mv a1, t2

  ________________________________

  ________________________________

  call less_than  # returns a0 = (A[i]<A[i-1])

  ________________________________
```

(Continued on the next page.)

```
  ______________________________

  # check the returned value / break out of the loop
  beq a0, zero, insert_end
  addi t3, t0, -4  # A+4*(i-1)

  ______________________________

  ______________________________

  # set up argument
  mv a0, t0  # argument 0: A+4*i
  mv a1, t3  # argument 1: A+4*(i-1)

  ______________________________

  ______________________________

  call swap

  ______________________________

  ______________________________

  addi s1, s1, -1  # i--
  j insert_loop

insert_end:

  ______________________________

  ______________________________

  ______________________________

  ______________________________

  ______________________________

  ret
```

Problem 7. I Got Your Stack (16 points)

Consider the following C function which takes in an `int` array of length `length` and returns a pointer to the first element in the array that is cleanly divisible by `factor`. If no value is ever found, the function returns a `NULL` pointer.

```
int* find_clean_factor(int* arr, int factor, int length){
 if(length==0){
   return 0;
 } else{
   if ((*arr)%factor==0){
     return arr;
   else{
     return find_clean_factor(arr+1, factor, length-1);
   }
 }
}
```

An equivalent assembly procedure is shown on the next page.

Problem continued on next page

```
find_clean_factor: #find_clean_factor procedure
   addi sp, sp, -16
   sw ra, 0(sp)
   sw a0, 4(sp)
   sw a1, 8(sp)
   sw a2, 12(sp)
   beq a2, zero, found_none
   lw a0, 0(a0)
   call rem
   beq a0, zero, found_one
   lw a0, 4(sp)
   addi a0, a0, 4
   lw a1, 8(sp)
   lw a2, 12(sp)
   addi a2, a2, -1
   call find_clean_factor
   j found_done
found_none:
   addi a0, zero, 0
   j found_done
found_one:
   lw a0, 4(sp)
found_done:
   lw ra, 0(sp)
   addi sp, sp, 16
   ret

#.....further down the file

rem: #remainder procedure
   addi sp, sp, -4
   sw ra, 0(sp)
   blt a0, a1, r_done
   sub a0, a0, a1
   call rem
r_done:
   lw ra, 0(sp)
   addi sp, sp, 4
   ret
```

Problem continued on next page.

An array is created and the procedure `find_clean_factor` is called. The contents of the array, the value of `factor`, and the length of the array are unknown. During the run of the code, snapshots of registers `a0`, `a1`, and `a2` as well as a consistent portion of the stack are grabbed at two spots in the code:

- Just **before** the `call find_clean_factor` at line 16.
- Just **before** the `call rem` at line 63.

The nine resulting snapshots are provided on the following pages **in chronological order**. For snapshots 2 through 9, the memory locations that have been written since the previous snapshot are bolded and italicized. For each snapshot the location of the stack pointer at that point in time is indicated with the arrow. Analyze them and answer the questions found on page 24.

Snapshot 1:	Snapshot 2:	Snapshot 3:
a0 =0x00000016	a0 =0x00000011	a0 =0x0000000c
a1 =0x00000005	a1 =0x00000005	a1 =0x00000005
a2 =0x00000004	a2 =0x00000004	a2 =0x00000004
Address: Data:	Address: Data:	Address: Data:
0x80280: 0x000001f3	0x80280: 0x000001f3	0x80280: 0x000001f3
0x80284: 0x0000022a	0x80284: 0x0000022a	0x80284: 0x0000022a
0x80288: 0x0000b0b0	0x80288: 0x0000b0b0	0x80288: 0x0000b0b0
0x8028c: 0x00004000	0x8028c: 0x00004000	0x8028c: 0x00004000
0x80290: 0x00000005	0x80290: 0x00000005	0x80290: 0x00000005
0x80294: 0xFFFFFFFF	0x80294: 0xFFFFFFFF	0x80294: 0xFFFFFFFF
0x80298: 0x00000001	0x80298: 0x00000001	0x80298: 0x00000001
0x8029c: 0x00000000	0x8029c: 0x00000000	0x8029c: 0x00000000
0x802a0: 0x00000003	0x802a0: 0x00000003	0x802a0: 0x00000003
0x802a4: 0x00000111	0x802a4: 0x00000111	***0x802a4: 0x000002ac*** ←sp
0x802a8: 0x00000000	***0x802a8: 0x000002ac*** ←sp	0x802a8: 0x000002ac
0x802ac: 0x00000230 ←sp	0x802ac: 0x00000230	0x802ac: 0x00000230
0x802b0: 0x00000208	0x802b0: 0x00000208	0x802b0: 0x00000208
0x802b4: 0x00004000	0x802b4: 0x00004000	0x802b4: 0x00004000
0x802b8: 0x00000005	0x802b8: 0x00000005	0x802b8: 0x00000005
0x802bc: 0x00000004	0x802bc: 0x00000004	0x802bc: 0x00000004

Problem continued on next page.

Snapshot 4:	**Snapshot 5:**	**Snapshot 6:**
a0 =0x00000007	a0 =0x00000002	a0 =0x00004004
a1 =0x00000005	a1 =0x00000005	a1 =0x00000005
a2 =0x00000004	a2 =0x00000004	a2 =0x00000003
Address: Data:	Address: Data:	Address: Data:
0x80280: 0x000001f3	0x80280: 0x000001f3	0x80280: 0x000001f3
0x80284: 0x0000022a	0x80284: 0x0000022a	0x80284: 0x0000022a
0x80288: 0x0000b0b0	0x80288: 0x0000b0b0	0x80288: 0x0000b0b0
0x8028c: 0x00004000	0x8028c: 0x00004000	0x8028c: 0x00004000
0x80290: 0x00000005	0x80290: 0x00000005	0x80290: 0x00000005
0x80294: 0xFFFFFFFF	0x80294: 0xFFFFFFFF	0x80294: 0xFFFFFFFF
0x80298: 0x00000001	0x80298: 0x00000001	***0x80298: 0x000002ac***
0x8029c: 0x00000000	***0x8029c: 0x000002ac*** ←sp	0x8029c: 0x000002ac
0x802a0: 0x000002ac ←sp	0x802a0: 0x000002ac	0x802a0: 0x000002ac
0x802a4: 0x000002ac	0x802a4: 0x000002ac	0x802a4: 0x000002ac
0x802a8: 0x000002ac	0x802a8: 0x000002ac	0x802a8: 0x000002ac
0x802ac: 0x00000230	0x802ac: 0x00000230	0x802ac: 0x00000230
0x802b0: 0x00000208	0x802b0: 0x00000208	0x802b0: 0x00000208 ←sp
0x802b4: 0x00004000	0x802b4: 0x00004000	0x802b4: 0x00004000
0x802b8: 0x00000005	0x802b8: 0x00000005	0x802b8: 0x00000005
0x802bc: 0x00000004	0x802bc: 0x00000004	0x802bc: 0x00000004

Snapshot 7:	**Snapshot 8:**	**Snapshot 9:**
a0 =0x0000000a	a0 =0x00000005	a0 =0x00000000
a1 =0x00000005	a1 =0x00000005	a1 =0x00000005
a2 =0x00000003	a2 =0x00000003	a2 =0x00000003
Address: Data:	Address: Data:	Address: Data:
0x80280: 0x000001f3	0x80280: 0x000001f3	0x80280: 0x000001f3
0x80284: 0x0000022a	0x80284: 0x0000022a	0x80284: 0x0000022a
0x80288: 0x0000b0b0	0x80288: 0x0000b0b0	0x80288: 0x0000b0b0
0x8028c: 0x00004000	0x8028c: 0x00004000	0x8028c: 0x00004000
0x80290: 0x00000005	0x80290: 0x00000005	0x80290: 0x00000005
0x80294: 0xFFFFFFFF	0x80294: 0xFFFFFFFF	***0x80294: 0x000002ac*** ←sp
0x80298: 0x000002ac	***0x80298: 0x000002ac*** ←sp	0x80298: 0x000002ac
0x8029c: 0x00000230 ←sp	0x8029c: 0x00000230	0x8029c: 0x00000230
0x802a0: 0x0000024c	0x802a0: 0x0000024c	0x802a0: 0x0000024c
0x802a4: 0x00004004	0x802a4: 0x00004004	0x802a4: 0x00004004
0x802a8: 0x00000005	0x802a8: 0x00000005	0x802a8: 0x00000005
0x802ac: 0x00000003	0x802ac: 0x00000003	0x802ac: 0x00000003
0x802b0: 0x00000208	0x802b0: 0x00000208	0x802b0: 0x00000208
0x802b4: 0x00004000	0x802b4: 0x00004000	0x802b4: 0x00004000
0x802b8: 0x00000005	0x802b8: 0x00000005	0x802b8: 0x00000005
0x802bc: 0x00000004	0x802bc: 0x00000004	0x802bc: 0x00000004

Problem continued on next page.

Answer the following questions:

A. (1 points) What is the length of the array being analyzed?

B. (1 points) What is the address of the array being analyzed?

Address: (0x)

C. (2 points) What is the `factor` being analyzed?

Factor: (integer)

D. (2 points) What is the address of the instruction that initially calls `find_clean_factor`?

Address: (0x)

E. (2 points) What is the address of the instruction that recursively calls `find_clean_factor`?

Address: (0x)

F. (2 points) What is the address of the instruction that initially calls `rem`?

Address: (0x)

G. (2 points) What is the address of the instruction that recursively calls `rem`?

Address: (0x)

H. (4 points) Specify a C array below that is as identical as can be determined to the one the user must have handed into `find_clean_factor`.

```
uint32_t a[    ] =
```

This page intentionally left blank

This page intentionally left blank

Appendix 1: String functions

`char *strcat(char *dest, const char *src)` - appends the string pointed to by `src` to the end of the string pointed to by `dest`. This function returns a pointer to the resulting string `dest`.

`char *strncat(char *dest, const char *src, size_t n)` - appends the string pointed to by `src` to the end of the string pointed to by dest up to `n` characters long. This function returns a pointer to the resulting string `dest`.

`char *strcpy(char *dest, const char *src)` - copies the string pointed to, by `src` to `dest`. This returns a pointer to the destination string `dest`.

`char *strncpy(char *dest, const char *src, size_t n)` - copies up to `n` characters from the string pointed to, by `src` to `dest`. In a case where the length of `src` is less than that of `n`, the remainder of `dest` will be padded with null bytes. This function returns the pointer to the copied string.

`int strcmp(const char *str1, const char *str2)` - compares the string pointed to, by `str1` to the string pointed to by `str2`. This function return values that are as follows −

- if Return value < 0 then it indicates `str1` is less than `str2`.
- if Return value > 0 then it indicates `str2` is less than `str1`.
- if Return value = 0 then it indicates `str1` is equal to `str2`.

`int strncmp(const char *str1, const char *str2, size_t n)` - compares at most the first `n` bytes of `str1` and `str2`. This function return values that are as follows −

- if Return value < 0 then it indicates `str1` is less than `str2`.
- if Return value > 0 then it indicates `str2` is less than `str1`.
- if Return value = 0 then it indicates `str1` is equal to `str2`.

`char *strchr(const char *str, int c)` - searches for the first occurrence of the character `c` (an `unsigned char`) in the string pointed to by the argument `str`. This returns a pointer to the first occurrence of the character `c` in the string `str`, or `NULL` if the character is not found.

`char *strrchr(const char *str, int c)` - searches for the last occurrence of the character `c` (an `unsigned char`) in the string pointed to, by the argument `str`. This function returns a pointer to the last occurrence of character in `str`. If the value is not found, the function returns a `null` pointer.

`char *strstr(const char *haystack, const char *needle)` - function finds the first occurrence of the substring `needle` in the string `haystack`. The terminating `'\0'` characters are not compared. This function returns a pointer to the first occurrence in `haystack` of any of the entire sequence of characters specified in `needle`, or a null pointer if the sequence is not present in `haystack`.

`char *strtok(char *str, const char *delim)` - breaks string `str` into a series of tokens using the delimiter `delim`. This function returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve.

Appendix 2: ASCII Table

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Appendix 3: C Operator Precedence

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td rowspan="5">1</td><td>++ --</td><td>Suffix/postfix increment and decrement</td><td rowspan="5">Left-to-right</td></tr>
<tr><td>()</td><td>Function call</td></tr>
<tr><td>[]</td><td>Array subscripting</td></tr>
<tr><td>.</td><td>Structure and union member access</td></tr>
<tr><td>-></td><td>Structure and union member access through pointer</td></tr>
<tr><td rowspan="6">2</td><td>++ --</td><td>Prefix increment and decrement</td><td rowspan="6">Right-to-left</td></tr>
<tr><td>+ -</td><td>Unary plus and minus</td></tr>
<tr><td>! ~</td><td>Logical NOT and bitwise NOT</td></tr>
<tr><td>(*type*)</td><td>Cast</td></tr>
<tr><td>*</td><td>Indirection (dereference)</td></tr>
<tr><td>&</td><td>Address-of</td></tr>
<tr><td>3</td><td>* / %</td><td>Multiplication, division, and remainder</td><td rowspan="11">Left-to-right</td></tr>
<tr><td>4</td><td>+ -</td><td>Addition and subtraction</td></tr>
<tr><td>5</td><td><< >></td><td>Bitwise left shift and right shift</td></tr>
<tr><td rowspan="2">6</td><td>< <=</td><td>For relational operators < and ≤ respectively</td></tr>
<tr><td>> >=</td><td>For relational operators > and ≥ respectively</td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
<tr><td>13</td><td>?:</td><td>Ternary conditional</td><td rowspan="6">Right-to-left</td></tr>
<tr><td rowspan="5">14</td><td>=</td><td>Simple assignment</td></tr>
<tr><td>+= -=</td><td>Assignment by sum and difference</td></tr>
<tr><td>*= /= %=</td><td>Assignment by product, quotient, and remainder</td></tr>
<tr><td><<= >>=</td><td>Assignment by bitwise left shift and right shift</td></tr>
<tr><td>&= ^= |=</td><td>Assignment by bitwise AND, XOR, and OR</td></tr>
</table>