Agenda

• Pointers
• Structures
• Bit-fields
• You can use an online C programming environment, such as
  • https://www.onlinegdb.com/online_c_compiler
  • https://www.programiz.com/c-programming/online-compiler/
Pointers and Memory

• A pointer is a variable that stores the address of a memory location.
• A C program works with three types of memory: Static/Global, Automatic, and Dynamic
  • Statically declared variables (within a function) and global variables use Static/Global memory. They are allocated when the program starts and remain in existence until the program terminates.
  • Global variables are accessible by all functions, but static variables are only accessible to their defining function
  • Automatic variables are declared within a function and are created (i.e., memory allocated) when the function is called. Their scope is restricted to the function, and their lifetime is limited to the time the function is executing.
  • Dynamic memory is allocated from the heap on demand and can be released as necessary. A pointer references the allocated memory. The scope is limited to the pointer(s) that reference the memory. Dynamic memory exists until it is released.
  • If a piece of dynamically allocated memory loses all reference to it (i.e., no pointer points to it) before it is released, a memory leak occurs.
  • Dynamic allocation of memory may fail. Not recommended in mission-critical programs.

```c
#include <stdio.h>
int main()
    int a, *p;
    p = &a;
    printf("%p %p\n", &a, p);
}
```

The following declarations are equivalent:
Pointers

int x = 1, y = 2, z[10];
int *ip;   /* ip is a pointer to int */
ip = &x;   /* ip now points to x */
y = *ip;   /* y is now 1 */
*ip = 0;   /* x is now 0 */
ip = &z[0]; /* ip now points to z[0] */

*p = &c; assigns the address of variable c to the variable p, and p is said to “point to” c.
The & (address of) operator only applies to objects in memory: variables and array elements; it cannot be applied to expressions, constants and register variables.
The unary operator * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to.
If ip points to the integer variable x, then *ip can appear in any context where x could, e.g., *ip = *ip + 10; increments *ip by 10.
The declaration int *ip; says that the expression *ip is an int.
For example, double *dp, atof(char *); says that in an expression *dp and atof(s) have values of type double, and that the argument of atof is a pointer to char.

*ip = *ip + 10;   /* increments *ip by 10 */
*ip += 1;        /* increments what ip points to */
++*ip;     /* same as above */
(*ip)++;     /* if you have ip = &c; somewhere earlier, think of this as c++; if you initialized ip by
char* ip = “string”; somewhere earlier, then this expression is invalid because it attempts to modify read-only
memory */
(*ip)+1;       /* the value of *ip plus 1 */
*ip++;        /* increment ip, the value of *ip++ is *ip, because ++ is a postfix here */
*(ip++)       /* same as above, the value is *ip, then ip is incremented */
 iq = ip;      /* copies the content of ip (an address value) into iq, so both ip and iq points to the same
place */
# Exercises

```c
#include <stdio.h>
int main(){
    char c = 'g';
    char *p = &c;
    printf("%c\n", *p+1);
    printf("%c\n", (*p)+1);
    printf("%c\n", *p);
}
```

- What do you expect the outcome to be?
- Run the program and see if the result matches what you expected.

```c
#include <stdio.h>
#define MAXLEN 1000 /* max length of any input line */
int readline(char s[], int lim);
int main(){
    char line[MAXLEN];
    readline(line, MAXLEN);
    printf("The line read is: %s\n", line);
}
```

- Write a function of prototype:
  ```c
  int readline(char s[], int lim);
  ```
  to read a line from the standard input into s, return the line length read, which cannot exceed lim
- Below is the test program.
# HW1 review (print binary)

```c
#include <stdio.h>

void bin(unsigned);
void main()
{
    bin(42);
}

void bin(unsigned n)
{
    unsigned i = 0;
    for(i = 1<<31; i > 0; i >>= 1){
        if((i & n) > 0){
            printf("1");
        } else {
            printf("0");
        }
    }
}
```

```c
#include <stdio.h>

void bin(unsigned n)
{
    if (n > 1)
        bin(n / 2);
    printf("%d", n % 2);
}

int main(void)
{
    bin(7);
}
```
# Exercises of pointer usage

```c
#include <stdio.h>
void main(){
    char s[10] = "1034567890";
    char *string = "Iamastring";
    char *p = s;
    char *ps = string;

    /* dereference p, return the value, then increment the value (not the pointer) */
    printf("%c\n", (*p)++);
    /* p is still at the original location but the value it points to have incremented */
    printf("%c\n", *p);

    /* wrong expression below, trying to modify string literal which is read-only memory*/
    //printf("%c\n", (*ps)++);

    /* this is correct, dereference ps, return the value, then increment ps */
    printf("%c\n", *(ps++));
    /* simply dereference ps */
    printf("%c\n", *ps);

    /* this is correct. inside the parenthesis: evaluate ps return as the value of the parenthesized
    * expression, then increment ps; then dereference the value of the parenthesized expression which is ps
    */
    printf("%c\n", *(ps++));
    printf("%c\n", *ps);
    /*ps = 'b'; // trying to change read-only memory, wrong
    // printf("%c\n", ++*ps); // Segmentation fault
    printf("%c\n", ++*ps);
}"
```

Output:
Pointers and Function Arguments

- C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.
- Pointer arguments can enable a function to access and change objects in the function that called it.
- If data needs to be modified in a function, it needs to be passed by a pointer.

```c
#include <stdio.h>

void swap(int *px, int *py) /* CORRECT */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main()
{
    int a = 2;
    int b = 3;
    printf("%d %d\n", a, b);
    swap(&a, &b);
    printf("%d %d\n", a, b);
}
```

```c
#include <stdio.h>

void swap(int x, int y) /* WRONG */
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 2;
    int b = 3;
    printf("%d %d\n", a, b);
    swap(a, b);
    printf("%d %d\n", a, b);
}
```

Write a function to swap the values of two integer variables.
Pointers and Arrays

In C, any operation that can be achieved by array subscripting can also be done with pointers, and the pointer version will in general be faster.

If pa points to a particular element of an array, then pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before.

If pa points to a[0], then *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].

By definition, the value of a variable or expression of type array is the address of element zero of the array.

The name of an array is a synonym for the location of the initial element of the array.

Thus, after the assignment pa = &a[0]; pa and a have identical values. The assignment can also be written as pa = a;

a[i] can be written as *(a+i)

&a[i] is equivalent to a+i

pa[i] is equivalent to *(pa+i)

Difference between array name and a pointer: a pointer is a variable, but an array name is not a variable. So, pa = a and pa++ are legal, while a=pa and a++ are illegal.

Try the above program, make sense of the output.

In which expressions do p and a interchangeable?
Character pointers

char *pmessage;
/* below statement assigns to pmessage a pointer to the character array
this is not a string copy, only pointers are involved. */
pmessage = "I am a string";

/* below definitions are different
amessage is an array, pmessage is a pointer initialized to point to a string constant */
char amessage[] = "I am a string";
char *pmessage = "I am a string";

String copy function:

/* copy string t to string s */
void strcpy(char *s, char *t){
    int i = 0;
    while((s[i] = t[i]) != '\0'){
        i++;
    }
}

• Write a main function to test the strcpy function.
• Rewrite the strcpy function by using pointer arithmetic instead of array indexing
Exercise

Run the program, observe the results and make sense of what happens in each step.

```c
#include <stdio.h>
int main() {
    char s[] = "1234567890";
    char *p = "Iamastring";
    char *ps = s;
    printf("%s %s %s \n", s, ps, p);
    s[1] = 'A';
    ps[2] = 'B';
    printf("%s %s %s \n", s, ps, p);
    ps = p;
    p = s;
    printf("%s %s %s \n", s, ps, p);
    ps = &s[3]; // now the reference to "Iamastring" is lost
    printf("%s %s %s \n", s, ps, p);
    return 0;
}
```

- What are the differences between the two strings, "1234567890" and "Iamastring"?
- The former has memory allocated for it within the function, the content can be modified freely, and the reference (i.e., the array s) will not be lost until the function ends.
- The latter has memory allocated in some area that is read-only to the function, the content of the string cannot be changed, and the reference to this memory location (thus the string) may be lost.
Pointers to char array and string literal

```c
char a[] = {'I', '*', 'a', 'm', '#', '\0'};
```

is equivalent to

```c
char a[] = "I*am#";
```

but not the same as

```c
char *p = "I*am#";
```

The value of each element of this string (or char array) can be changed; e.g., `a[0] = 'U';` is legal

A string literal is created, and its address is assigned to `p`

In most compilers, a string literal is treated as a constant and stored as read-only data, thus `*p = 'U';` is illegal.

```
#include <stdio.h>
int main()
{
    char a[] = {'I', '*', 'a', 'm', '#', '\0'};
    char b[] = "I*am#";
    char *p = "I*am#";
    printf("%s %s %s\n", a, b, p);
    printf("%d %d %d\n", strlen(a), strlen(b), strlen(p));
    printf("%d %d %d\n", sizeof(a), sizeof(b), sizeof(p), sizeof(char*));
}
```

• Try this program and make sense of the results.
• Remove the element ‘\0’ in the initializer of `a[]`, re-run and observe the difference in results.
Pointer Arrays

- Pointers are variables, so they can be stored in arrays just as other variables can.
- In the declaration of an array (of any type), either the size must be provided, as in char a[10]; or an initializer is provided, as in char a[] = {'a', 'b'}; so the compiler can count the elements in the initializer and deduce the array size.
- A slash \\ at the end of a line tells the C preprocessor that the current statement is not finished yet, and is extended to the next line.

```c
#include <stdio.h>

void writelines(char * lineptr[], int nlines){
    for(int i = 0; i < nlines; i++){
        printf("%s\n", lineptr[i]);
    }
}

int main(){
    char * textlines[] = {"first line", "second line"};
    int n = sizeof(textlines)/sizeof(char*);
    writelines(textlines, n);
    for(int i = 0; i < n; i++){
        printf("%c\n", *textlines[i]);
    }
}
```

- The most frequent use of arrays of pointers is to store character strings of different lengths.

```c
#include <stdio.h>

char * month_name(int n){
    return (n < 1 || n > 12) ? name[0] : name[n];
}

int main(){
    int month = 5;
    printf("month %d is %s\n", month, month_name(month));
}
```
The \textit{static} keyword

- means different things in different places:
  - For functions and variables declared outside any function, the \textit{static} keyword means that the declared function or variable is only visible to the current source file in which the declaration is made.
  - For variables within a function, the \textit{static} keyword means that the declared variable maintains its value between invocations of the function, acting as a global variable whose scope is only the function it is declared in.
Function Pointers

```c
#include <stdio.h>
int product(int u, int v) {
    return u*v;
}

void main() {
    int (*p)(int, int);  // declare a function pointer p
    p = &product;       // assign the address of a function to p
    int x2 = (*p)(12, 4);  // call the function whose address is stored in p
    printf("%d\n", x2);
}
```

typedef is usually used to ease the usage:

```c
#include <stdio.h>
typedef int (*t_somefunc)(int, int);
int product(int u, int v) {
    return u*v;
}

void main() {
    t_somefunc afunc = &product;
    int x2 = (*afunc)(12, 4);
    printf("%d\n", x2);
}
```
Pointers to constant and constant pointers

```c
#include <stdio.h>

int main()
{
    int a = 1;
    int *pa = &a; /* non-constant pointer to a non-constant */
    const int b = 2;
    const int *pb = &b; /* non-constant pointer to a constant */
    int c = 3;
    int * const pc = &c; /* constant pointer to a non-constant */
    const int d = 4;
    const int * const pd = &d; /* constant pointer to a constant */

    printf("%d %d %d %d\n", *pa, *pb, *pc, *pd);
    pa = &b; /* change pa */
    *pa = 22; /* change the value of the variable that pa points to */
    pb = &a; /* change the address stored in pb */
    *pc = 33; /* change the value pointed to by pc */
    printf("%d %d %d %d\n", *pa, *pb, *pc, *pd);
    printf("%d %d %d %d\n", a, b, c, d);
}
```
#include <stdio.h>
/* Method 1: declare a struct with tag */
struct point {
    float x;
    float y;
};
/* Method 2: declare a named type */
typedef struct {
    double x;
    double y;
} Point;

int main(){
    struct point pt1 = {2.0, 3.0};
    Point pt2 = {4.0, 5.0};
    printf("%5.2f %5.2f\n", pt1.x, pt1.y);
    printf("%5.2f %5.2f\n", pt2.x, pt2.y);
    struct point * p1; // declare a pointer to a structure of type struct point
    Point * p2;       // declare a pointer to an object of type Point
    p1 = &pt1;
    p2 = &pt2;
    printf("%5.2f %5.2f\n", p1->x, p1->y);
    printf("%5.2f %5.2f\n", p2->x, p2->y);
}

Exercises:
- Define a structure to represent a line segment on the plane
- Write a function to calculate the length of a given line
Bit-fields

```c
#include <stdio.h>

struct myReg {
    unsigned int PIN1 : 4;
    unsigned int PIN2 : 4;
    unsigned int PIN3 : 4;
    unsigned int PIN4 : 4;
    unsigned int PIN5 : 4;
    unsigned int PIN6 : 4;
    unsigned int PIN7 : 4;
    unsigned int PIN8 : 1;
    unsigned int PIN9 : 1;
    unsigned int PIN10 : 2;
};

int main()
{
    struct myReg r1 = {0,0,1};
    printf("%ld\n", sizeof(r1));
    r1.PIN1 = 1;
    r1.PIN5 = 15;
    r1.PIN8 = 1;
    printf("%d %d %d %d\n", r1.PIN1, r1.PIN2, r1.PIN5, r1.PIN8);
}
```

- In embedded programming, we want to reduce the RAM footprint of a program so that more data and computing can be handled in the limited RAM space.
- Many frequently used variables in a program represent the state of something, for which only a few unique values are needed.
- But a 2-byte int can represent $2^{16} - 1 = 65,535$ unique values.
- If we used int for a state variable which can take only a few possible values, it would be a waste of memory space.
- A C structure with bit-fields allows us to pack more variables in a machine word (say, 4-byte memory slot).
- The declaration of each unsigned int member is suffixed by a width specifier : \text{n}, which specifies that the member needs \text{n-bit}.
Exercise

- Write a utility function to print the content of a structure into a character buffer

```c
/**
 * Describes the capabilities of a SPI peripherals
 */
typedef struct {
    uint32_t minimum_frequency;
    uint32_t maximum_frequency;
    uint32_t word_length;
    uint16_t slave_delay_between_symbols_ns;
    uint8_t clk_modes;
    bool support_slave_mode;
    bool hw_cs_handle;
    bool async_mode;
    bool tx_rx_buffers_equal_length;
} spi_capabilities_t;

/* Write a function to print the member names and values of a spi_capabilities_t instance referenced by s, one member per line, into a character buffer reference by buf. The total length (in number of characters) of the print should not exceed lim. You can make use of the sprintf function, which is available through stdio.h */
void printSPIcapabilities(spi_capabilities_t * s, char * buf, int lim){
}
```