Summer 2013

CSE2421 Systems1

Introduction to Low-Level Programming and Computer Organization

Kitty Reeves

MTWR 10:20am-12:25pm
Understanding Pointers

- See page 252 of systems book
- [http://boredzo.org/pointers/](http://boredzo.org/pointers/)
Address **1001**: The system knows where the `var` variable is located (symbol table)

```c
int var = 50;  //
int *ptr, i;  // i is an integer variable
ptr = &var;  // *ptr == var
i = *ptr;  //
*ptr = 1;  // var == 1
```

- Address in memory
- Value
- Variable
Key points about pointers

- A pointer is a memory address
- Every pointer has an associated type
- Every pointer has a value
- NULL indicates that a pointer does not point anywhere
- Pointers are given a value with the & operator
- Pointers are dereferenced with the * operator
  - Get the value at that address
- Arrays and pointers are closely related
- Casting from one type of pointer to another changes its type but not its value
- Pointers can also point to functions
Values of variables are stored in memory, at a particular location
A location is identified and referenced with an address
  Analogous to identifying a house’s location via an address
A pointer is a variable that contains the address of another variable
  * is used in the declaration of a pointer type
  int *p means variable p is a pointer that points to an integer
& (unary operator) gives the “address of” an object
  p = &c means the address of c is assigned to the variable p
* (unary not arithmetic operator) is a dereferencing operator when applied to pointers
  When applied to a pointer, it accesses the object the pointer points to
  * in front of a pointer variable means “get the value at that address” i.e. “contents of”
  int a = *p means get the value at the address designated by p and assign it to a
  *p = 1 means assign the value of 1 to the memory location designated by the address of p
Every pointer points to a specific data type
  Exception = void (a generic pointer); pointer to void holds any type of pointer but can’t be dereferenced (i.e. cannot get the “contents of”)

Ah, yes. POINTERS. At last, we arrive at THE MOST DREADED WORD in the lexicon of the C student. Pointers are indeed so dreaded that Java has completely done away with pointers and wrapped their functionality into the (admittedly safer) concept of references. C++, as a transitional step, has both pointers and references.
Declaring Pointers

- int* ptr_a;
- int *ptr_a;
- The first style leads to mistakes
  - int* ptr_b, ptr_c, ptr_d
    - b is a pointer but c and d are integers
  - int *ptr_b, *ptr_c, *ptr_d
    - 3 pointers are declared here

Char example
- char ch = 'c';
- char *chptr = &ch;
- char *ptr = chptr;
- Both chptr and ptr have the same initial value
Pointer example

Reminders:
- * in a declaration says “I am a pointer” that points to a certain type of value
- & “address of”
- * In front of a pointer type says “get the value at that address” i.e. “contents of” operator

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>ADDRESS (in decimal)</th>
<th>MEMORY (assuming 4 bytes per word and each block is a byte)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>0</td>
<td>Is a pointer; holds an addr; 8... 16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>8</td>
<td>1... 0</td>
</tr>
<tr>
<td>y</td>
<td>12</td>
<td>2... 1</td>
</tr>
<tr>
<td>z</td>
<td>16</td>
<td>z[0]</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>z[1]</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>z[2]</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>etc</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

* not going to worry about "size" right now

EXAMPLE:
int x=1, y=2, z[10];
int *ip;
ip = &x;
y = *ip;
*ip = 0;
ip = &z[0];
Every pointer points to a specific data type.
- one exception:
  - a “pointer to void” is used to hold any type of pointer but cannot be dereferenced itself.
- If ip points to the integer x (ip=&x) then *ip can occur in any context where x could
  - Example: *ip = *ip + 10 \(\Rightarrow\) x=x+10; increments the contents of the address at ip by 10.
- The unary operators * and & bind more tightly than arithmetic operators
  - Example: y = *ip + 1 takes whatever ip points at, adds 1, and assigns the result to y.
  - Other ways to increment by 1:
    - *ip += 1 \(\Rightarrow\) *ip = *ip + 1
    - ++*ip
    - (*ip)++
      - The parentheses are necessary; without them, the expression would increment ip instead of what it points to, because unary operators like * and ++ associate right to left.
- Pointers are variables so can be used without dereferencing.
  - Example: int x, *iq, *ip=&x;
    - iq = ip;
    - copies the contents of ip (an address) into iq, making iq point to whatever ip points to.
You try...

REMINDER: Every pointer points to a specific data type EXCEPTION: void (a generic pointer); pointer to void holds any type of pointer but can’t be dereferenced (i.e. cannot get the “contents of”)

/* EXAMPLE 1 */
#include <stdio.h>
int main() {
    float i=10, *j;
    void *k;
    k=&i;
    j=k;
    printf("%f\n", *j);
    return 0; }

/* EXAMPLE 2 */
#include <stdio.h>
#include <stdlib.h>
main() {
    int x, *p;
    p = &x;
    *p = 0;
    printf("x is %d\n", x);
    printf("*p is %d\n", *p);
    *p += 1;
    printf("x is %d\n", x);
    (*p)++;
    printf("x is %d\n", x);
    return 0; }

/* EXAMPLE 3 */
#include <stdio.h>
int main(void) {
    char ch = 'c';
    char *chptr = &ch;
    int i = 20;
    int *intptr = &i;
    float f = 1.20000;
    float *fptr = &f;
    char *ptr = "I am a string";
    printf("\n [%c], [%d], [%f], [%c], [%s]\n", *chptr, *intptr, *fptr, *ptr, ptr);
    return 0; }

Void parameter means there are no arguments
### Pointer Review – In Class Assignment

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>Question</th>
</tr>
</thead>
</table>
| 1. | int *a;  
  *a = 12; | ? Creates memory for the storage of an integer?  
  ? What is the value of a? |
|   |      |          |
| 2. | int a = 12;  
  int *d = &a;  
  *d = 10 - *d;  
  d = 10 - *d; | OK?  
  OK? |
|   |      |          |
| 3. | int a;  
  *&a = 25; | OK? What does it do? |
|   |      |          |
| 4. | *100 = 25 | OK? |
|   |      |          |
| 5. | Store 25 at address 100 | Can you do this? If so, how? |
|   |      |          |
| 6. | How is c declared?  
 int a = 12;  
 int *b = &a;  
 c = &b;  
 a = ?  
 b = ?  
 *b = ?  
 c = ?  
 *c = ?  
 * *c = ? | ![Diagram of variables and pointers] |

<table>
<thead>
<tr>
<th></th>
<th>var</th>
<th>ptr</th>
<th>ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>12</td>
<td>b</td>
<td>&amp;a</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>c</td>
<td>&amp;b</td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>**c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
No matter how complex a pointer structure gets, the list of rules remains short:

- A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.
- The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.
- Allocating a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.
- Assignment between two pointers makes them refer to the same pointee which introduces sharing.

NOTE: A “pointee” is a variable whose address is assigned to be the value of the pointer.
Other Pointers

- Wild Pointer
  - one that has not been initialized

- Dangling Pointer problem
  - Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory

NOTE: Dangling/wild pointer bugs frequently become security holes
Seg fault?


You know when you're falling asleep, and you imagine yourself walking or something.

And suddenly you misstep, stumble, and jolt awake?

Yeah!

Well, that's what a segfault feels like. Double-check your damn pointers, okay?
In-class exercise

A. Write a loop that uses a pointer reference as the loop counter. In other words, convert the variable `x` to a pointer reference in the below code so that it works exactly the same:

```c
int x;
for (x = 1; x < 5; x++)
    printf("loop counter value is %d \n", x);
```

B. What is the value of the loop counter once the for loop has finished executing? Write a `printf` statement to output this value using the pointer variable. Write another `printf` statement to output this value using the variable `x`.

C. What is the value of the pointer variable; not the value that it points to, but the value of the pointer itself (i.e. the address)? Write a `printf` statement to output this value using the pointer variable. Write another `printf` statement to output this value using the variable `x`. 
Function definition

- Function prototype
  - Return type*
  - Argument definition
  - return_type function_name (type1 arg1,type2 arg2,...,typen argn)
- Function calls
  - Basic syntax
  - Parameter passing*
- Standard library and function calls

NOTES:
- Functions should be short and sweet.
- Functions do not nest.
- Variables have to be communicated through function arguments or global variables.
- * If no return type or no argument type, then it defaults to int

Sample: http://www.cprogrammingexpert.com/images/Function.gif
Pointers and Functions

- **Pass By Value**
  - Passing a variable by value *makes a copy* of the variable before passing it onto a function. This means that if you try to modify the value inside a function, it will only have the modified value inside that function. Once the function returns, the variable you passed it will have the same value it had before you passed it into the function.

- **Pass By Reference**
  - There are two instances where a variable is passed by reference:
    - When you modify the value of the passed variable locally (inside the callee) and the value of the variable in the *calling* function as well.
    - To avoid making a copy of the variable for efficiency reasons.
  - Technically, C does not “pass by reference” as typically seen in other programming languages. Actually, when you pass a pointer (an address), a copy is made of that variable so two variables point to the same address (one from the callee and one from the caller).
C functions

- Similar to Java methods but...
  - Not part of a class
  - Not associated with an object
  - No “this”
- Only one function with a given name may be defined.
  - Unlike Java, C does not support overloading (i.e., two functions with the same name but different signatures).
A number of statements grouped into a single logical unit are called a function.

**REMINDER** → It is necessary to have a single function ‘main’ in every C program.

A function prototype is a function declaration or definition which includes:

- Information about the number of arguments
- Information about the types of the arguments

Although you are allowed *not to specify* any information about a function's arguments in a declaration, it is purely because of backwards compatibility with Old C and should be avoided (poor coding style).

- A declaration without any information about the arguments is *not* a prototype.

To declare or not to declare... That is the question!
Since C passes arguments to functions by value and make a copy local to swap; so there is no direct way for the called function (callee) to alter a variable in the calling function (caller).
Because of call by value, swap can’t affect the arguments a and b in the routine that called it.
The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed:
Since the operator & produces the address of a variable, &a is a pointer to a. In swap itself, the parameters are declared as pointers, and the operands are accessed indirectly through them.
NOW KNOW WHY SCANF NEEDS & SYMBOLS!!!
The RETURN statement

- Every function except those returning void should have at least one, each return showing what value is supposed to be returned at that point.
- Although it is possible to return from a function by falling through the last }, unless the function returns void, an unknown value will be returned, resulting in undefined behavior.
- The type of expression returned must match the type of the function, or be capable of being converted to it as if an assignment statement were in use.
- Following the return keyword with an expression is *not* permitted if the function returns void.
 Another Function example

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

void pmax(int first, int second); /* declaration prototype */

main () {int i,j;
    for(i = -10; i <= 10; i++)
    {
        for(j = -10; j <= 10; j++)
        {
            pmax(i,j);
        }
    }
    return 0;
}

/* Prints larger of its two arguments. */
void pmax (int a1, int a2) /* definition */
{
    int biggest;
    if (a1 > a2)
        { biggest = a1; }
    else{ biggest = a2; }
    printf("larger of %d and %d is %d
", a1, a2, biggest);
}
```

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

void printtotal(int total);
void addxy(int x, int y, int total);
void subxy(int x, int y, int *total);

void main() {
  int x, y, total;
  x = 10;
  y = 5;
  total = 0;
  printtotal(total);
  addxy(x, y, total);
  printtotal(total);
  subxy(x, y, &total);
  printtotal(total);
}

Program continued...

void printtotal(int total) {
  printf("Total in Main: %dn", total);
}

void addxy(int x, int y, int total) {
  total = x + y;
  printf("Total from inside addxy: %dn", total);
}

void subxy(int x, int y, int *total) {
  *total = x - y;
  printf("Total from inside subxy: %dn", *total);
}
Another Function example

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

void date(int *, int *); /* declare the function */

main() {
    int month, day;
    date (&day, &month);
    printf("day is %d, month is %d\n", day, month);
    return 0;
}

void date(int *day_p, int *month_p) {
    int day_ret, month_ret;
    /* * At this point, calculate the day and month * values in day_ret and month_ret respectively. */
    *day_p = day_ret;
    *month_p = month_ret; }
```
Function Summary

- Functions can be called recursively.
- Functions can return any type that you can declare, except for arrays and functions (you can get around that restriction to some extent by using pointers).
- Functions returning no value should return void.
- Always use function prototypes.
- Undefined behavior results if you call or define a function anywhere in a program unless either
  - a prototype is always in scope for every call or definition, or
  - you are very, very careful.
- Assuming that you are using prototypes, the values of the arguments to a function call are converted to the types of the formal parameters exactly as if they had been assigned using the = operator.
- Functions taking no arguments should have a prototype with (void) as the argument specification.
C built-in functions (revisted)


Now that you know more about functions, check out:

- The definition
- The parameters
- The return value
- Any side effects

Try:

- `<string.h>` strcpy, strcat, strlen
- `<ctype.h>` isspace, iscntrl, isupper
- `<stdio.h>` fopen, fgetc, getc, getchar, perror, feof
Java arrays vs C arrays

- Java programmers should immediately note a difference. Arrays are declared directly with a given size in C rather than using a two-step process of declaring a reference to an array then using the new operator like in Java.

  NOTE: Java’s approach is really a use of pointers.

- C arrays are not objects. They don’t have methods, and most importantly, they don’t know their own size (i.e., there is no array.length method).

- For this reason, C arrays are not bounds-checked. Accessing an array element beyond the end (or before the beginning) of a C array may well crash the program rather than generate an orderly exception, as with Java.
Declaration of Arrays

- An array is a way to store many values under the same name in adjacent memory locations.
- Arrays must be declared before they can be used in the program.
- Standard array declaration is as
  - `<type> <name> [<size>];`
  - `<size>` elements i.e. values of the array, are stored using an index/subscript number from 0 to `<size>`-1
- Examples
  - `double height[10]; // height[0] to height[9]`
  - `float width[20]; // width[0] to width[19]`
  - `int min[9]; // etc`
  - `char name[20]; // a string!`
- Why first index/subscript=0???
  - Address of min = address of min[0]

<table>
<thead>
<tr>
<th>in memory:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>min --&gt;</td>
<td>[0]</td>
<td>[1]</td>
<td>[2]</td>
<td>[3]</td>
<td>[4]</td>
<td>[5]</td>
<td>[6]</td>
<td>[7]</td>
<td>[8]</td>
</tr>
<tr>
<td>address --&gt;</td>
<td>+0</td>
<td>+4</td>
<td>+8</td>
<td>+12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the square brackets [] actually represent an operator (known as array dereference) which has a relative priority and associativity.
Initializing Arrays

- The initializing values are enclosed within the curly braces in the declaration and placed following an equal sign after the array name.
- Initialize an individual array location (name[sub]) like any other variable/memory location.
- An array location can be used like any other single variable:
  - `x = array[3]`
  - `array[5] = x + y`

```c
int studentAge[4];
studentAge[0] = 14;
studentAge[1] = 13;
studentAge[2] = 15;
studentAge[3] = 16;

// initialize and print all the elements of the array
int myArray[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++)
{
  printf("%d", myArray[i]);
}
```

If you omit the size of the array, an array just big enough to hold the initialization is created:
```c
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```
Size of operator

- **Unary operator**
- Returns the *number of bytes*
- The size of primitive data types in C are implementation defined/dependent
- Used to determine:
  - The number of bytes per primitive data type
  - The size, in bytes, of arrays
  - The size of memory to be allocated
- Number of array elements that avoids you needing to know the base type of the array:
  - `sizeof(array) / sizeof(*array)`
  - Or define a macro that has base type involved → `#define LENGTH(x) (sizeof(x)/sizeof(*x))`

```c
main() {
  int a; short b; double c; char d[10];
  printf("Line 1 - Size of variable a = %d\n", sizeof(a ));
  printf("Line 2 - Size of variable b = %d\n", sizeof(b ));
  printf("Line 3 - Size of variable c= %d\n", sizeof(c ));
  printf("Line 4 - Size of variable d= %d\n", sizeof(d ));
  printf("Line 5 - # array elements in d= %d\n", LENGTH(d ));
}
```

% sizeck2
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c= 8
Line 4 - Size of variable d= 10
Line 5 - Size of variable d= 10
There is no such statement in C language which can directly copy an array into another array. So we have to copy each item separately into another array.*

```c
#include <stdio.h>
int main()
{
    int iMarks[4] = {78, 64, 66, 74};
    int newMarks[4];
    int i,j;
    for(i=0; i<4; i++)
        newMarks[i]=iMarks[i];
    for(j=0; j<4; j++)
        printf("%d\n", newMarks[j]);
    return 0; }  
```

*But there is a function ;o)
Index checking

- Index access is not checked by the compiler
  - Check for valid range manually
  - Especially important for user entered indices

- Index checking means that, in all expressions indexing an array, first check the index value against the bounds of the array which were established when the array was defined, and should an index be out of bounds, further execution is suspended via some sort of error (buffer overflow, segmentation fault, bug).

- Important to understand how arrays are used “behind the scenes”
- Performing bounds checking during every usage is time-consuming
- C never performs automatic bounds checking in order to raise speed
- It depends on the OS to ensure that you are accessing valid memory.
- There’s a difference in being outside array bounds but inside your allotted memory; and outside the array bounds and outside your allotted memory!
- Yet... sizeof (array) works, but that’s the total number of bytes not the index bounds themselves
Manipulating Arrays

- C Language treats the name of the array as if it were a pointer to the first element
  - see handout ArrayInOutSwapReverse.docx
- The name of the array refers to the whole array. It works by representing a pointer to the start of the array.

Prototype/Call

```c
void intSwap(int *x, int *y)
intSwap(&a[i], &a[n-i-1]);

void printIntArray(int a[], int n)
printIntArray(x, hmny);

int getIntArray(int a[], int nmax, int sentinel)
hmny = getIntArray(x, 10, 0);

void reverseIntArray(int a[], int n)
reverseIntArray(x, hmny);
```

When we pass arrays into functions, the compiler automatically converts the array into a pointer to the first element of the array. In short, the array without any brackets will act like a pointer. So we just pass the array directly without using the ampersand.
Multi-dimensional Arrays

Declarations – [row][col] subscript order

- float table [50] [50];
- char line [24] [40];
- int values [3] [4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 }

How stored? ➔ row order

<table>
<thead>
<tr>
<th>rows</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

in memory:

<table>
<thead>
<tr>
<th>values --&gt;</th>
<th>[0][0]</th>
<th>[0][1]</th>
<th>[0][2]</th>
<th>[0][3]</th>
<th>[1][0]</th>
<th>[1][1]</th>
<th>[1][2]</th>
<th>[1][3]</th>
<th>[2][0]</th>
<th>[2][1]</th>
<th>etc</th>
</tr>
</thead>
<tbody>
<tr>
<td>address --&gt;</td>
<td>+0</td>
<td>+4</td>
<td>+8</td>
<td>+12</td>
<td>etc</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#include <stdio.h>
int main()
{
    int x; int y; int array[8][8]; /* Declares an array like a gameboard or matrix*/
    for ( x = 0; x < 8; x++ )
    {
        for ( y = 0; y < 8; y++ )
        {
            array[x][y] = x * y; /* Set each element to a value */
        }
    }
    printf( "Array Indices:\n" );
    for ( x = 0; x < 8;x++ )
    {
        for ( y = 0; y < 8; y++ )
        {
            printf( "[\%d][\%d]=\%d", x, y, array[x][y] );
        }
        printf( "\n" );
    }
    getchar();
    return 0;
} //twodarr.c
Character Arrays i.e. Strings

- Declarations:
  - char arr[] = {'c','o','d','e','\0'};
    - The null byte is required as a terminating byte when string is read as a whole.
  - char arr[] = "code";
    - Implies that there are 4 characters along with the NUL byte (i.e. the \0 character) so a “length” of 5.

- This type of array allocation, where the size of the array is determined at compile-time, is called **static allocation**.

- What about this declaration...
  - char *ch = “code”;
  - Implied constant... can’t change it
A string is an array of characters.

- So we have no string pointers in C. Its the character pointers that are used in case of strings too.
- When we point a pointer to a string, by default it holds the address of the first character of the string (just like an array)

Gives the memory address without a reference operator (&)

```
char *ptr;
char str[40];
ptr = str;
```

Strings end with an implied \0 by default

```
"I am a string" = l_am_a_string\0
```

- sizeof operator says size = ??
- strlen() function is in the string.h header file

- The strlen function returns the length of the null-terminated string s in bytes. In other words, it returns the offset (i.e. starting at position zero) of the terminating null character within the array.

```
char string[32] = "hello, world";
sizeof (string) ⇒ 32
strlen (string) ⇒ 12
```

- this will not work unless string is the character array itself, not a pointer to it
### C String functions `<string.h>`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strcat</code></td>
<td>Appends a string</td>
</tr>
<tr>
<td><code>strchr</code></td>
<td>Finds first occurrence of a given character</td>
</tr>
<tr>
<td><code>strcmp</code></td>
<td>Compares two strings</td>
</tr>
<tr>
<td><code>strcmpi</code></td>
<td>Compares two strings, non-case sensitive</td>
</tr>
<tr>
<td><code>strcpy</code></td>
<td>Copies one string to another</td>
</tr>
<tr>
<td><code>strlen</code></td>
<td>Finds length of a string</td>
</tr>
<tr>
<td><code>strlwr</code></td>
<td>Converts a string to lowercase</td>
</tr>
<tr>
<td><code>strncat</code></td>
<td>Appends n characters of string</td>
</tr>
<tr>
<td><code>strncpy</code></td>
<td>Copies n characters of one string to another</td>
</tr>
<tr>
<td><code>strnset</code></td>
<td>Sets n characters of string to a given character</td>
</tr>
<tr>
<td><code>strstr</code></td>
<td>Finds last occurrence of given character in string</td>
</tr>
<tr>
<td><code>strrev</code></td>
<td>Reverses string</td>
</tr>
<tr>
<td><code>strset</code></td>
<td>Sets all characters of string to a given character</td>
</tr>
<tr>
<td><code>strspn</code></td>
<td>Finds first substring from given character set in string</td>
</tr>
<tr>
<td><code>strstr</code></td>
<td>Returns a pointer to the first occurrence of string s2 in string s1</td>
</tr>
<tr>
<td><code>strupr</code></td>
<td>Converts string to uppercase</td>
</tr>
</tbody>
</table>
Built-in functions for character handling

- **isalnum**: Tests for alphanumeric character
- **isalpha**: Tests for alphabetic character
- **isascii**: Tests for ASCII character
- **iscntrl**: Tests for control character
- **isdigit**: Tests for 0 to 9
- **isgraph**: Tests for printable character
- **islower**: Tests for lowercase
- **isprint**: Tests for printable character
- **ispunct**: Tests for punctuation character
- **isspace**: Tests for space character
- **isupper**: Tests for uppercase character
- **isxdigit**: Tests for hexadecimal
- **toascii**: Converts character to ascii code
- **tolower**: Converts character to lowercase
- **toupper**: Converts character to uppercase
String Function example

```c
#include <stdio.h>
#include <string.h>
int main () {
    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[15];
    int len;
    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy(str3, str1): %s\n", str3);
    /* concatenates str1 and str2 */
    strcat(str1, str2);
    printf("strcat(str1, str2): %s\n", str1);
    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1): %d\n", len);
    return 0; }
```

OUTPUT:
% strfuncs
strcpy(str3, str1): Hello
strcat(str1, str2): HelloWorld
strlen(str1): 10
Character Array (i.e. string) example

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

int main(void)
{
    char arr[4];  // for accommodating 3 characters and one null '\0' byte
    char *ptr = "abc";  // a string containing 'a', 'b', 'c', '\0'
    printf("print string with pointer var %s \n",ptr);
    char *ptr2 = ptr;  printf("print string with pointer var %s \n",ptr2);
    memset(arr, '\0', sizeof(arr));
    strncpy(arr, ptr, sizeof("abc"));  // Copy the string "abc" into the array arr
    printf("%s \n",arr);  // print the array as string
    arr[0] = 'p';  // change the first character in the array
    printf("%s \n",arr);
    printf("arr1 is %s \n",arr1);
    printf("arr2 is %s \n",arr2);
}
```

```c
char arr1[4]= "abc";
char arr2[4]= "def";
// arr1 = arr2; WHY? incompatible types when assigning to type 'char[4]' from type 'char *'
strncpy(arr1,arr2,sizeof(arr));
printf("\narr1 is %s\n",arr1);
printf("arr2 is %s\n",arr2);
return 0; 
```
When you add to or subtract from a pointer, the amount by which you do that is multiplied by the size of the type the pointer points to.

In the case of our three increments, each 1 that you added was multiplied by sizeof(int).

```c
int array[] = { 45, 67, 89 };
int *array_ptr = array;
printf(" first element: %i\n", *(array_ptr++));
printf("second element: %i\n", *(array_ptr++));
printf(" third element: %i\n", *array_ptr);
```

Output:
first element: 45
second element: 67
third element: 89

NOTE 1: 1==4 (programmer humor?!) *(array_ptr++) == *array_ptr++

*`(array_ptr++)`* ¹

vs

*(array_ptr)++

find the value at that address, output, then add "1" to the address

VS
Find the value at the address, output, then add one to the value at that address
## Pointer Arithmetic (cont)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Assuming p is a pointer to a...</th>
<th>... and the size of *p is...</th>
<th>Value added to the pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>p+1</td>
<td>char</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p+1</td>
<td>short</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p+1</td>
<td>int</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>p+1</td>
<td>double</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>p+2</td>
<td>char</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>p+2</td>
<td>short</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>p+2</td>
<td>int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p+2</td>
<td>double</td>
<td>8</td>
<td>16</td>
</tr>
</tbody>
</table>
**Pointer Arithmetic (again)**

- **pointer (+ or -) integer**
  - Only for pointers that are pointing at an element of an array
  - Also works with `malloc`
  - Watch for bounds (begin and end)
    - Ok to go one beyond the array but not a valid dereference

- **pointer#1 – pointer#2**
  - Only allowed when both point to elements of the same array and p1 index < p2 index
  - Measured in array elements not bytes
  - If `p1 \to array[i]` and `p2 \to array[j]` then `p2-p1 == j - i`
The subscript operator (the [] in array[0]) has nothing to do with arrays. In most contexts, arrays decay to pointers. This is one of them: That's a pointer you passed to that operator, not an array.

```c
int array[] = { 45, 67, 89 }; printf("%i\n", array[0]);    // output is 45
// array and array[0] point to same thing
```

- array points to the first element of the array;
  - `array[1] == *(array + 1)`
- array_ptr is set to &array[1], so it points to the second element of the array.
  - So array_ptr[1] is equivalent to array[2]
#include<stdio.h>
int main()
{
    char *ptr1 = "Himanshu";
    char *ptr2 = "Arora";
    char *ptr3 = "TheGeekStuff";

    char* arr[3];
    arr[0] = ptr1;
    arr[1] = ptr2;
    arr[2] = ptr3;

    printf("\n [%s]\n", arr[0]);
    printf("\n [%s]\n", arr[1]);
    printf("\n [%s]\n", arr[2]);
    return 0;
} // Declaring/Initializing 3 characters pointers

//Declaring an array of 3 char pointers

// Initializing the array with values

//Printing the values stored in array
# Pointers to Arrays

## Examples

### Declaring a Pointer to an Array of 5 Integers

```c
#include<stdio.h>

int main(void)
{
    int (*ptr)[5];
    // Declares a pointer ptr to an array of 5 integers.
}
```

### Declaring and Initializing an Array and a Pointer to an Array of 3 Characters

```c
#include<stdio.h>

int main(void)
{
    char arr[3];
    char (*ptr)[3];
    arr[0] = 'a';
    arr[1] = 'b';
    arr[2] = 'c';
    ptr = &arr;
    return 0;
}
```

### Declaring an Array of Int Pointers and a Pointer to an Array of Integers

```c
int *arr[8];  // An array of int pointers.
int (*arr)[8];  // A pointer to an array of integers
// same as using the name of the array
```
It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries.

```c
#include <stdio.h>
int main () {
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", &ptr);
    return 0; }
```

The value of `ptr` is 0

Note: The macro "NULL" is defined in locale.h, stddef.h, stdio.h, stdlib.h, string.h, time.h, wchar.h
On most operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows:

```c
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```
NULL vs 0 vs ‘\0’

- NULL is a macro defined in several standard headers
- 0 is an integer constant
- '\0' is a character constant, and
  - nul is the name of the character constant.

**All of these are *not* interchangeable**

- NULL is to be used for pointers only since it may be defined as ((void *) 0), this would cause problems with anything but pointers.
- 0 can be used anywhere, it is the generic symbol for each type's zero value and the compiler will sort things out.
- '\0' should be used only in a character context.
  - nul is not defined in C or C++, it shouldn't be used unless you define it yourself in a suitable manner, like:
  - #define nul '\0'
NULL pointer and VOID

- 0 (an integer value) is convertible to a null pointer value if assigned to a pointer type
- VOID – no value at all – literally means “nothing”
  - So it is type-less (no type defined) so can hold any type of pointer
  - We cannot perform arithmetic on void pointers (no type defined)
  - Cannot dereference (can’t say, “get the value at that address” – no type defined)
- NULL is defined as 0 cast to a void * pointer
  - #define NULL (void *) 0;
- FYI: However, NULL and zero are not the same as no returned value at all, which is what is meant by a void return value (see your first C program examples)

- Is there any difference between the following two statements?
  - `char *p=0;`
  - `char *t=NULL;`
  - NO difference. NULL is defined as 0 in the 'stdio.h' file. Thus, both p and t are NULL pointers.

- Is this a correct way for NULL pointer assignment?
  - `int i=0;`
  - `char *q=(char*) i;` // char * cannot point to an int type... even for a moment in time
  - NO. Correct → `char *q=0` (or) `char *q=(char*)0`

- Is the NULL pointer same as an uninitialized pointer? NO
Static and Dynamic Arrays

Static arrays are used when we know the amount of bytes in array at compile time.
- Static arrays are ones that reside on the stack
  - char arr[10];

A dynamic array is used where we come to know about the size on run time.
- Dynamic arrays is a popular name given to a series of bytes allocated on the heap.
  - char *ptr = (char*) malloc(10);
  - allocates a memory of 10 bytes on heap and we have taken the starting address of this series of bytes in a character pointer ptr.
  - Fine if know number of characters, but what if don’t?
    - Read in one char/byte at a time until the user presses the enter key

malloc (memory allocation) is used to dynamically allocate memory at run time. Possible uses for this function are:
- Read records of an unknown length.
- Read an unknown number of database records.
- Link lists.
The stack is the memory set aside as scratch space for a thread of execution. When a function is called, a block is reserved on the top of the stack for local variables and some bookkeeping data. When that function returns, the block becomes unused and can be used the next time a function is called. The stack is always reserved in a LIFO order; the most recently reserved block is always the next block to be freed. This makes it really simple to keep track of the stack; freeing a block from the stack is nothing more than adjusting one pointer.

The heap is memory set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time; there are many custom heap allocators available to tune heap performance for different usage patterns.
# Address Space... a quick look

- An array of 8-bit bytes
- A pointer is just an index into this array

<table>
<thead>
<tr>
<th>ADDRESS SPACE</th>
<th>Description/info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel virtual memory</td>
<td>Memory invisible to user code</td>
</tr>
<tr>
<td>User stack (created at run time)</td>
<td>Implements function calls</td>
</tr>
<tr>
<td>Memory mapped region for shared libraries</td>
<td>Ex. printf function</td>
</tr>
<tr>
<td>Run-time heap (created at run time by malloc/calloc)</td>
<td>Dynamic in size</td>
</tr>
<tr>
<td>Read/write data</td>
<td>Program (executable file)</td>
</tr>
<tr>
<td>Read-only code and data</td>
<td>Fixed size</td>
</tr>
</tbody>
</table>

32/64 bit starting address

Address 0

Notice symbolically drawn with memory “starting” at the bottom
Address Space... another look
The Stack

- Stored in computer RAM just like the heap.
- Variables created on the stack will go out of scope and automatically deallocate.
- Much faster to allocate in comparison to variables on the heap.
- Implemented with an actual stack data structure.
- Stores local data, return addresses, used for parameter passing.
- Can have a stack overflow when too much of the stack is used. (mostly from infinite (or too much) recursion, very large allocations)
- Data created on the stack can be used without pointers.
- You would use the stack if you know exactly how much data you need to allocate before compile time and it is not too big.
- Usually has a maximum size already determined when your program starts.
The Heap

- Stored in computer RAM just like the stack.
- Variables on the heap must be destroyed manually and never fall out of scope. The data is freed with `free`.
- Slower to allocate in comparison to variables on the stack.
- Used on demand to allocate a block of data for use by the program.
- Can have fragmentation when there are a lot of allocations and deallocations.
- Data created on the heap will be pointed to by pointers and allocated with `calloc/malloc`.
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- Responsible for memory leaks.
Dynamic Memory Functions

Can be found in the stdlib.h library:

- To allocate space for an array in memory you use
  - `calloc()`

- To allocate a memory block you use
  - `malloc()`

- To de-allocate previously allocated memory you use
  - `free()`

Each function is used to initialize a pointer with memory from free store (a section of memory available to all programs i.e. the heap)
The function `malloc()` will allocate a block of memory that is size bytes large. If the requested memory can be allocated a pointer is returned to the beginning of the memory block.

- **Note:** the content of the received block of memory is not initialized.

**Usage of `malloc()`:**

```c
void * malloc ( size_t size );
```

**Parameters:**

- Size of the memory block in bytes.

**Return value:**

- If the request is successful then a pointer to the memory block is returned.
- If the function failed to allocate the requested block of memory, a null pointer is returned.

**Example**


**Another example:**

```c
#include <stdlib.h>
int *ptr = malloc( sizeof (int) );
    set ptr to point to a memory address of size int
int *ptr = malloc( sizeof (*ptr) );
    is slightly cleaner to write malloc statements by taking the size of the variable pointed to by using the pointer directly
float *ptr = malloc( sizeof (*ptr) );
    float *ptr;
/* hundreds of lines of code */
ptr = malloc( sizeof(*ptr) );
```
Usage of calloc():

```c
void * calloc( size_t num, size_t size );
```

Parameters:

- Number of elements (array) to allocate and the size of elements.

Return value:

- Will return a pointer to the memory block. If the request fails, a NULL pointer is returned.

Example:

- note: ptr_data = (int*) calloc ( a,sizeof(int) );
**Difference Between Malloc and Calloc**

- The number of arguments. `malloc()` takes a single argument (memory required in bytes), while `calloc()` needs two arguments.
- `malloc()` does not initialize the memory allocated, while `calloc()` initializes the allocated memory to ZERO.
The free function returns memory to the operating system.

```c
free(ptr);
```

After freeing a pointer, it is a good idea to reset it to point to 0.

NOTE: When 0 is assigned to a pointer, the pointer becomes a null pointer...in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.
## Main Pointer concepts

<table>
<thead>
<tr>
<th>C - Pointer arithmetic</th>
<th>There are four arithmetic operators that can be used on pointers: ++, --, +, -</th>
</tr>
</thead>
<tbody>
<tr>
<td>C - Array of pointers</td>
<td>You can define arrays to hold a number of pointers.</td>
</tr>
<tr>
<td>C - Pointer to pointer</td>
<td>C allows you to have pointer on a pointer and so on.</td>
</tr>
<tr>
<td>Passing pointers to functions in C</td>
<td>Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.</td>
</tr>
<tr>
<td>Return pointer from functions in C</td>
<td>C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.</td>
</tr>
</tbody>
</table>
# Main Array Concepts

<table>
<thead>
<tr>
<th>CONCEPT</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multi-dimensional arrays</strong></td>
<td>C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.</td>
</tr>
<tr>
<td><strong>Passing arrays to functions</strong></td>
<td>You can pass to the function a pointer to an array by specifying the array's name without an index.</td>
</tr>
<tr>
<td><strong>Return array from a function</strong></td>
<td>C allows a function to return an array (by passing back the address of the array).</td>
</tr>
<tr>
<td><strong>Pointer to an array</strong></td>
<td>You can generate a pointer to the first element of an array by simply specifying the array name, without any index.</td>
</tr>
</tbody>
</table>
C programming code to open a file and print its contents to the screen, one character at a time.
//fileio1.c

#include<stdio.h>
#include<stdlib.h>
int main() {
    char ch;
    FILE *fp;
    fp = fopen("lab2p2in","r"); // read mode
    if( fp == NULL ) {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE); }
    printf("The contents of the file is : \n\n");
    while( ( ch = fgetc(fp) ) != EOF )
        printf("%c",ch);
    fclose(fp);
    return 0; }

(1) fgetc returns the value of an int that is converted from the character
(2) What happens if delete lab2p2in file? i.e. it can’t be found to open?
FYI... the exit function

Syntax:
```c
#include <stdlib.h>
void exit( int exit_code );
```

Description:
The `exit()` function stops the program. `exit_code` is passed on to be the return value of the program, where usually zero indicates success and non-zero indicates an error.

Example:
```
```
File function summary

- Open/Close files
  - fopen() – open a stream for a file
  - fclose() – closes a stream

- One character at a time:
  - fgetc() – similar to getchar()
  - fputc() – similar to putchar()

- One line at a time:
  - fprintf()/fputs() – similar to printf()
  - fscanf()/fgets() – similar to scanf()

- File errors
  - perror() – reports an error in a system call
File constants `<stdio.h>`

- `FILE` – a variable type suitable for string information for a file stream
- `fpos_t` – a variable type suitable for starting any position in a file
- `NULL` – value of a null pointer constant
- `EOF` – negative integer which indicates end-of-file has been reached
- `FOPEN_MAX` – integer which represents the maximum number of files that the system can guarantee that can be opened simultaneously
- `FILENAME_MAX` – integer which represents the longest length of a char array
- `stderr/stdin/stdout` – pointers to FILE types which correspond to the standard streams
File usage

- When a program begins, there are already three available streams which are predefined and need not be opened explicitly and are of type “pointer to FILE”
  - standard input
  - standard output
  - standard error
- Files are associated with streams and must be opened to be used.
- The point of I/O within a file is determined by the file position.
- When a file is opened, the file position points to the beginning of the file (unless the file is opened for an append operation in which case the position points to the end of the file).
- The file position follows read and write operations to indicate where the next operation will occur.
- When a file is closed, no more actions can be taken on it until it is opened again.
- Exiting from the main function causes all open files to be closed.
File open and close

- FILE *fopen(const char *filename, const char *mode);
- Mode... (lots more!)
  - r – read text mode
  - w – write text mode (truncates file to zero length or creates a new file)
  - If the file does not exist and it is opened with read mode (r), then the open fails → need to check for this
- Declaration: int fclose(FILE *stream);
  - Closes the stream.
  - If successful, it returns zero.
  - On error it returns EOF.
- perror
  - void perror(const char *str);
  - Prints a descriptive error message to stderr. First the string str is printed followed by a colon then a space (your error message). Then an error message based on the current setting of the variable errno is printed (system error message).
fgetc and fputc

Declaration: `int fgetc(FILE *stream);`
- Gets the next character (an `unsigned char`) from the specified stream and advances the position indicator for the stream.
- On success the character is returned.
- If the end-of-file is encountered, then `EOF` is returned and the end-of-file indicator is set.
- If an error occurs then the error indicator for the stream is set and `EOF` is returned.

Declaration: `int fputc(int char, FILE *stream);`
- Writes a character (an `unsigned char`) specified by the argument `char` to the specified stream and advances the position indicator for the stream.
- On success the character is returned.
- If an error occurs, the error indicator for the stream is set and `EOF` is returned.
C programming code to open a file and print its contents to the another file, one character at a time.

//fileio2.c
Example

FILE *infp, *outfp;
char * mode = "r";
char outfile[] = "lab2p2out";
char input[101], save_first_letter;
char *inptr;
int first_letter = TRUE, n=101;

infp = fopen("lab2p2in","r");
if (infp == NULL) {
    fprintf(stderr, "can't open input file lab2p2in!\n");
    exit(EXIT_FAILURE); }

outfp = fopen(outfile,"w");
if (outfp == NULL) {
    fprintf(stderr, "Can't open output file %s!\n", outfile);
    exit(EXIT_FAILURE); }

fgets(input,n,infp);
while (!feof(infp)) {
    // etc
    fgets(input,n,infp);
}
//close files

- `fgets(buffer,size,stdin);`
  
  `buffer` is the location of your string storage space or buffer.

- `size` is the number of characters to input. This sets a limit on input.

- Note that `fgets()` also reads in the carriage return (enter key; newline character) at the end of the line. That character becomes part of the string you input.

- `fscanf(infp,"%s",input);`

- `while (!feof(infp))`
# fgets vs fscanf

<table>
<thead>
<tr>
<th>Declaration: <code>char *fgets(char *str, int n, FILE *stream);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Reads a line from the specified stream and stores it into the string pointed to by <code>str</code>.</td>
</tr>
<tr>
<td>- It stops when either (n-1) characters are read, the newline character is read, or the end-of-file is reached, whichever comes first.</td>
</tr>
<tr>
<td>- <strong>The newline character is copied to the string.</strong></td>
</tr>
<tr>
<td>- A null character is appended to the end of the string.</td>
</tr>
<tr>
<td>- On error a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Declaration: <code>int fscanf(FILE *stream, const char *format, ...);</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied.</td>
</tr>
<tr>
<td>- On success the number of input fields converted and stored are returned. If an input failure occurred, then EOF is returned.</td>
</tr>
<tr>
<td>- Returns EOF in case of errors or if it reaches eof</td>
</tr>
</tbody>
</table>
fprintf and feof

- Declaration:
  - int fprintf(FILE *stream, const char *format, ...);
  - sends formatted output to a stream
  - Just like printf, but puts file pointer as first argument
  - In lab2p2:
    - fprintf(outfp, “Your sipher coded message is %s\n”,input);

- Declaration: int feof(FILE *stream);
  - Tests the end-of-file indicator for the given stream
  - If the stream is at the end-of-file, then it returns a nonzero value. If it is not at the end of the file, then it returns zero.
```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fpin;
    fpin = fopen("fileio3in","r"); // read mode
    if( fpin == NULL ) {
        perror("Error while opening the input file.\n");
        exit(EXIT_FAILURE); }
    char a[15], b[15]; float c, d; int e, f;
    fscanf(fpin,"%s %s %f %d %f %d",
            a,b,&c,&e,&d,&f);
    while (!feof(fpin)) {
        printf("a= %s",a);
        printf(" b= %s",b);
        printf(" c= %f",c);
        printf(" e= %d",e);
        printf(" d= %f",d);
        printf(" f= %d\n",f);
        fscanf(fpin,"%s %s %f %d %f %d",
               a,b,&c,&e,&d,&f);
        //getchar();
    }
    fclose(fpin); return 0; } }
```
Stop here

Didn’t cover the following slides
**R and L values**

- **L-value** = something that can appear on the left side of an equal sign
  - A place i.e. memory location for a value to be stored
- **R-value** is something that can appear on the right side of an equal sign
  - A value

**Example:**

- `a = b+25` vs `b+25 = a`

**Example:**

- `int a[30];`
- `a[b+10]=0;`

**Example:**

- `int a, *pi;`
- `pi = &a;`
- `*pi = 20;`
### Problem Expression

<table>
<thead>
<tr>
<th>Problem</th>
<th>Expression</th>
<th>R-value</th>
<th>L-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ch</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>&amp;ch</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>3</td>
<td>cp</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>&amp;cp</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>5</td>
<td>*cp</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>*c+1</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>7</td>
<td>*(c+1)</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>++cp</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>9</td>
<td>cp++</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>10</td>
<td>*++cp</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>11</td>
<td>*cp++</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>++*cp</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>13</td>
<td>(*cp)++</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>14</td>
<td>++*++cp</td>
<td>yes</td>
<td>illegal</td>
</tr>
<tr>
<td>15</td>
<td>+++*cp++</td>
<td>yes</td>
<td>illegal</td>
</tr>
</tbody>
</table>

**Given:**

- `char ch = 'a';`
- `char *cp = &ch;`

**NOTE:** the `?` is the location that follows `ch`
Files accessed through the FILE mechanism provided by `<stdio.h>`


Text streams are composed of lines.
Each line has zero or more characters and are terminated by a new-line character which is the last character in a line.

Conversions may occur on text streams during input and output.

Text streams consist of only printable characters, the tab character, and the new-line character.

Spaces cannot appear before a newline character, although it is implementation-defined whether or not reading a text stream removes these spaces.