1. **Pointers (ptr)**: special types of variables that provide direct access to memory.
   1. C and C++ are two of the few high-level languages that permit direct access to memory locations.
      1. C: often called **mid-level programming language**.
      2. Java: very similar to C++, but does not permit pointers.
         1. Does not have main().
   2. Normal variables vs. Pointer variables
      1. Normal variables: contain values of variables.
         1. Addresses are hidden from programmers.
         2. Names directly reference a variable.
      2. Pointer variables: contain memory address of other variables.
         1. Allow call by reference.
         2. Permit dynamic allocation of memory (at runtime).
            1. Create variables not specified beforehand.
         3. Permit creation of dynamic data structures.
         4. **Indirect reference/indirection**: pointers indirectly reference variables.
            1. Without using variable name.
            2. With type of variable.

Size of variable depends upon its type.

* + - 1. I/O: %p
  1. Double & Triple pointers
     1. Point to addresses of other pointers.
  2. Direction of Pointer Variables
     1. Pointers must be declared with the same naming rules.
        1. Include type of variable they point to
     2. Declaration
        1. int \* ptr;
           1. Declares a ptr that points to an int value.
     3. Pointers should always be initialized.
        1. The \* does not distribute.
        2. Can be set to NULL (preferred) or 0.
           1. Pointers assigned a value of 0 have the value 0, not an address.
           2. NULL is a symbolic constant defined in <stdio.h>.

00000000

Used in linked lists.

* 1. Operators that only apply to pointers
     1. **Address-of-operator (&)**: unary operator that returns the address of its operand.
        1. Example
           1. int y = 5;
           2. int \* ptr = NULL;
           3. ptr = &y;
           4. return(ptr);
           5. Returns address of y
     2. **Indirection/dereferencing operator (\*)**: unary operator that returns the value of the variable pointed at by the pointer.
        1. Example
           1. int y = 5;
           2. int \* ptr = NULL;
           3. ptr = \*y;
           4. return(\*ptr);
           5. Returns value of y… 5
        2. Also allows giving a value to y.
  2. Examples of differences between int and ptr
     1. void main()
     2. {
     3. int a;
     4. int \* aptr;
     5. a = 7;
     6. aptr = &a;
     7. printf(“The value of a = %d”, a);
     8. printf(“The address of a = %p, &a);
     9. printf(“The value of aptr is the address of a = %p”, aptr);
     10. printf(“The indirect value of a = %p”, \*aptr);
     11. }
  3. Modifying Variables Through Pointers
     1. Example
        1. int = 5, y = 10;
        2. int \* yptr = &y, \* xptr = &x;
        3. y = y + x; //changes the value of y directly
        4. \*yptr = \*yptr + \*xptr; //changes the value of y indirectly
     2. Call by Value vs. Call by Reference
        1. void cube\_it (int);
        2. void main()
        3. {
        4. int number = 5;
        5. printf(“Original value = %d\n”, number);
        6. cube\_it(number);
        7. printf(“New value = %d\n”, number);
        8. }
        9. void cube\_it(int n)
        10. {
        11. n\*n\*n;
        12. }
     3. **Call-by-reference**: passing variable address to a function to indirectly modify the variable’s value.
        1. Example
           1. void cube\_it\_2(int \*);
           2. void main()
           3. {
           4. int number = 5;
           5. printf(“Original value = %d\n, number);
           6. cube\_it\_2 (&number);
           7. printf(“New value = %d\n”, number);
           8. }
           9. void cube\_it\_2 (int \* nptr)
           10. {
           11. \*nptr = (\*nptr) \* (\*nptr) \* (\*nptr);
           12. }
  4. Functions Returning Pointers
     1. Functions can also return pointers to variables.
        1. Meaningless address, given that the variable ceases to exist outside of the function.
        2. Example
           1. int x = 5;
           2. int \* xptr = &x;
           3. return xptr;
     2. More commonly done with dynamically-allocated variables.
        1. Link lists
  5. *const* and Pointer Passing
     1. A programmer can specify that variables be made constant.
     2. *const* qualifier: tells compiler that the variable following it is not to be changed by any program statements.
        1. Measure of security when passing addresses of variables whose values are not to be modified under any circumstances.
     3. 4 cases when passing pointers
        1. **Non-constant pointer to non-constant data** (*highest data access, lowest data security*): declaration does not include *const*
           1. Values of variables can be modified through pointers
           2. Pointer can be modified to point to other variables if desired
           3. Example

int a, \* ptr = &a;

* + - 1. **Non-constant pointer to constant data**: pointer can be modified to point to any location, but variable cannot be modified.
         1. “A pointer to an integer constant”
         2. Example

const int \* a;

* + - 1. **Constant pointer to non-constant data**: pointer always points to same memory location, and data can be modified.
         1. Pointer must be initialized.
         2. “Constant pointer to an integer”
         3. Example

int x;

int \* const aptr = &x;

* + - 1. **Constant pointer to constant data**: pointer always points to same location and data cannot be modified.
         1. Pointer must be initialized.
         2. “Constant pointer to an integer constant”
         3. Example

int x = 5;

const int \* const aptr = &x;

* 1. Pointer Arithmetic
     1. Pointers are valid operands
     2. Not all operators are valid
        1. /
        2. \*
        3. / and \* only make sense in arrays
     3. Operators that are valid do not always work the same way
        1. **+**, +=, -and
           1. Value added/subtracted is the number of **memory elements** to be moved.

Depends on variable type

int = 4 bytes in a 32-bit computer

* + - * 1. Example

int \* yptr = 3000;

yptr += 2;

Output

yptr = 3008, assuming 4-byte int for a 32-bit computer

3000 + 2 memory blocks(4 bytes/int)

3008

* + - 1. ++
      2. –
  1. **Double pointers**: pointer 🡪 pointer 🡪 variable
     1. Commonly used when call-by-reference is desired and variable to be modified is itself a pointer.
     2. Declaration
        1. int \* \* dbl\_ptr
     3. “Pointer to a pointer to an integer”
     4. Dereferencing
        1. Results in an address
        2. Obtain the value of the variable being pointed at twice removed, the double pointer must be de-referenced twice
        3. Example
           1. var = \*(\*dbl\_ptr);
  2. Outputting
     1. printf() can be used to output address
        1. %p
     2. Address will be returned in hexadecimal
  3. Pointers to Functions
     1. Pointers that contain the address of the function in memory.
     2. Addressing the code segment of the main memory.
     3. Pointers to functions can be…
        1. Passed to functions
        2. Returned from functions
        3. Stored in arrays
        4. Assigned to other function pointers
     4. Commonly used in menu-driven system
        1. Choice made can result in calling different functions
     5. Example
        1. double (\* foo) (double);
           1. foo: name of the pointer that points to a function.
           2. Second double: type of variable returned
           3. First double: type of variable it is passed to.
        2. Once declared, a pointer to a function can be set to the address of a function.
           1. double square(double x)
           2. {
           3. return x\*x;
           4. }
           5. foo = &square;