1. Pointer Basics

- In C, pointers are a way of referring to the location (or the address) of a variable.
- Pointers enable a variable’s value to be a “pointer” to a completely different variable.
- Programmers can access what a pointer points to and redirect the pointer to point to something else.
- This is an example of indirection, a powerful programming concept.

1. Pointer Basics

- Pointers require introducing new types and new operators.
- Every type $T$ has a corresponding pointer type $T*$.
- A variable of type $T*$ contains a pointer to a variable of type $T$.

```c
int* a_ptr; // pointer to a variable of type int
char* b_ptr; // pointer to a variable of type char
float* c_ptr; // pointer to a variable of type float
```

- The address-of operator (&) evaluates to the location of a variable.
- The address-of operator is used to initialize/assign to pointers.

```c
int a; // variable of type int
int* a_ptr; // pointer to a variable of type int
a_ptr = &a; // assign location of a to a_ptr
```

- The dereference operator (*) evaluates to the value of the variable the pointer points to.

```c
int b = 42; // initialize variable of type int to 42
int* b_ptr = &b; // pointer to a variable of type int
int c = *b_ptr; // initialize c with what b_ptr points to
```
Example declaring, initializing, RHS dereferencing pointers

```c
int a = 3;
int* a_ptr;
a_ptr = &a;
int b = 2;
int c = b + (*a_ptr);
```

Example illustrating aliasing

```c
int a = 3;
int* a_ptr0 = &a;
int* a_ptr1 = a_ptr0;
int c = (*a_ptr0) + (*a_ptr1);
```

Example declaring, initializing, LHS dereferencing pointers

```c
int a = 3;
int b = 2;
int c;
int* c_ptr = &c;
*c_ptr = a + b;
```
2. Call by Value vs. Call by Pointer

- Be careful – three very different uses of the * symbol!
  - Multiplication operator \( int \ a = b * c; \)
  - Pointer type \( int* \ d = &a; \)
  - Dereference operator \( int \ e = *d; \)

2. Call by Value vs. Call by Pointer

- So far, we have always used call by value
- Call by value copies values into parameters
- Changes to parameters by callee are not seen by caller

```c
void sort( int x, int y )
{
    if ( x > y ) {
        int temp = x;
        x = y;
        y = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( a, b );
    return 0;
}
```
• **Call by pointer** uses pointers as parameters
• Callee can read and modify parameters by dereferencing pointers
• Changes to parameters by callee *are seen* by caller

```c
void sort( int* x_ptr, int* y_ptr )
{
    if ( (*x_ptr) > (*y_ptr) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( &a, &b );
    return 0;
}

https://repl.it/@cbatten/ece2400-T04-ex1
```
2. Call by Value vs. Call by Pointer

Draw a state diagram corresponding to the execution of this program

```c
void avg( int* result_ptr, int x, int y )
{
    int sum = x + y;
    *result_ptr = sum / 2;
}

int main( void )
{
    int a = 10;
    int b = 20;
    int c;
    avg( &c, a, b );
    return 0;
}
```
3. Mapping Conceptual Storage to Machine Memory

- Our current use of state diagrams is conceptual
- Real machine uses memory to store variables
- Real machine does not use “arrows”, uses memory addresses
3. Mapping Conceptual Storage to Machine Memory

- Can visualize memory using a “byte” or “word” view
- Stack stored at high addresses, stack grows “down”
- As a simplification, assume we only have 128 bytes of memory

```c
int a = 3;
int* a_ptr;
a_ptr = &a;
int b = 2;
int c;
c = b + (*a_ptr);
```
3. Mapping Conceptual Storage to Machine Memory

- Both code and stack are stored in 128 bytes of memory
- Stack stored at high addresses, stack grows “down”
- Code stored at low addresses, execution moves “up”
- **Stack Frame**: collection of data on the stack associated with function call including return value, return addr, parameters, local variables

```c
void sort( int* x_ptr, int* y_ptr )
{
    if ( (*x_ptr) > (*y_ptr) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

int main( void )
{
    int a = 9;
    int b = 5;
    sort( &a, &b );
    return 0;
}
```

**Memory**
(4B word addr)
4. Pointers to Other Types

In addition to pointing to primitive types, pointers can also point to other pointers, to structs, or even functions.

4.1. Pointers to struct

- Pointer to a struct is declared exactly as what we have already seen
- Be careful to dereference the pointer first, then access a field

```c
typedef struct _node_t
{
    int         value;
    struct _node_t* next_ptr;
} node_t;

int main( void )
{
    // First node
    node_t node0;
    node0.value = 3;

    node_t* node_ptr = &node0;
    (*node_ptr).value = 4;

    // Second node
    node_t node1;
    node1.value = 5;
    node1.next_ptr = &node0;

    node_ptr = &node1;
    (*node_ptr).value = 6;
    (*((node_ptr).next_ptr)).value = 7;

    return 0;
}
```
4.2. Pointers to Nothing

- **NULL** is defined in `stdlib.h` to be a pointer to nothing
- **NULL** can be used to indicate “there is no answer” or “error"
- Simply write **NULL** in state diagrams
- In previous example, **NULL** can mean there is no next node
```c
#include <stddef.h>
#include <stdio.h>

typedef struct _node_t
{
    int value;
    struct _node_t* next_ptr;
} node_t;

int main( void )
{
    node_t node0;
    node0.value = 3;
    node0.next_ptr = NULL;

    node_t node1;
    node1.value = 4;
    node1.next_ptr = &node0;

    node_t node2;
    node2.value = 5;
    node2.next_ptr = &node1;

    int sum = 0;
    node_t* node_ptr = &node2;
    while ( node_ptr != NULL ) {
        sum += node_ptr->value;
        node_ptr = node_ptr->next_ptr;
    }
    return 0;
}
```
4.3. Pointers to Pointers

```c
int a = 3;
int* a_ptr = &a;
int** a_pptr = &a_ptr;
int*** a_ppptr = &a_pptr;
int b = ***a_ppptr + 1;
```

4.4. Pointers to Functions

- Code is also stored in memory, so a function pointer points to code
- Enables passing a function as a parameter to a different function
- Consider the following functions:

```c
int lt( int x, int y ) { return x < y; }
int gt( int x, int y ) { return x > y; }
```

- The type of a function is the function’s signature
- A function signature includes the parameter types and return type
- We don’t really care about the function or parameter names
- lt and gt have the same function signature and thus the same type
- lt and gt are essentially two “values” of the same type
- We can write the type of these functions as follows:

```c
int ( int, int )
```
• Recall that every type $T$ has a corresponding pointer type $T*$
• So the type of a pointer to this function might look like this:
  \[
  (\text{int ( int, int )})*
  \]
• ... and declaring a variable to hold a function pointer like this:
  \[
  (\text{int ( int, int )})* \text{cmp_ptr};
  \]
• This makes sense and would be consistent, but C actually uses a slightly different syntax for declaring a function pointer:
  \[
  \text{int (\text{cmp_ptr}) ( int, int );}
  \]
• This also makes sense since now there is a direct connection between a function declaration and a function pointer:
  \[
  \text{int lt ( int x, int y );}
  \]
  \[
  \text{int (\text{cmp_ptr}) ( int, int );}
  \]
• The type of a function pointer is complex, so use a typedef
  \[
  \text{typedef int (\text{cmp_ptr}) ( int, int );}
  \]
  \[
  \text{cmp_ptr_t cmp_ptr;}
  \]
• The address-of operator ($\&$) applied to a function name evaluates to a pointer to that function
  \[
  \text{typedef int (\text{cmp_ptr}) ( int, int );}
  \]
  \[
  \text{cmp_ptr_t cmp_ptr = \&lt ;}
  \]
• We can dereference a function pointer and use the call operator ($()$) to call a function via function pointer
  \[
  \text{typedef int (\text{cmp_ptr}) ( int, int );}
  \]
  \[
  \text{cmp_ptr_t cmp_ptr = \&lt ;}
  \]
  \[
  \text{int result = (\text{cmp_ptr})( 3, 4 );}
  \]
#include <stdio.h>

int lt( int x, int y ) { return x < y; }
int gt( int x, int y ) { return x > y; }

typedef int (*cmp_ptr_t)( int, int );

void sort( int* x_ptr, int* y_ptr, cmp_ptr_t cmp_ptr )
{
    // Dereference function pointer, then call function
    if ( (*cmp_ptr)( (*x_ptr), (*y_ptr) ) ) {
        int temp = *x_ptr;
        *x_ptr = *y_ptr;
        *y_ptr = temp;
    }
}

void sort3( int* x_ptr, int* y_ptr, int* z_ptr, cmp_ptr_t cmp_ptr )
{
    sort( x_ptr, y_ptr, cmp_ptr );
    sort( y_ptr, z_ptr, cmp_ptr );
    sort( x_ptr, y_ptr, cmp_ptr );
}

int main( void )
{
    int a = 9; int b = 5; int c = 3;
    sort3( &a, &b, &c, &gt );
    printf(" %d < %d < %d \n", a, b, c );
    sort3( &a, &b, &c, &lt );
    printf(" %d > %d > %d \n", a, b, c );
    return 0;
}