Section 4: Intro to Pointers in C

4.1: Introduction to Pointers

Address Operator

C was designed to give programmers "low-level" access to memory and **expose** the underlying memory model.

The **address operator** (`&`) produces the **location** of an identifier in memory (the **starting** address of where its value is stored). The `printf` format specifier, `%p`, displays an address (in hex).

Example:

```
int g = 42;

int main(void) {
  printf("the value of g is    %d\n",  g);
  printf("the address of g is  %pCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &g);
}
```

Pointers

A ***pointer*** is a variable that stores a memory address.

To **define** a pointer, place a *star* (*) before the identifier (name).

The **type** of a pointer is the type of memory address it can store (or "point at").

Example:

```
struct posn {
  int x;
  int y;
};

int main(void) {
  int i;                  // i is an integer [uninitialized]
  i = 42;
  char c = 'z';
  struct posn p1 = {3, 4};

  int *pi;                // pi is an pointer to an integer [uninitialized]
  pi = &i;                // pi now stores the address of i or "pi points at i"
  char *pc = &c;          // pc points at c
  struct posn *pp = &p1;  // pp points at p1

  printf("The value of integer i is %d and the address is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", i, pi);
  printf("The value of character c is %c and the address is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", c, pc);
  printf("The address of the structure posn is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", pp);
}
```

The pointer definition syntax can be a bit overwhelming at first, especially with initialization.

The * is part of the definition and is **not part of the variable name**. The name of the above variable is simply q, not *q.

In C, we can define a **pointer to a pointer**.

Example:

```
int main(void) {
  int i = 42;
  int *p = &i;        // p is defined and initialized

  int *q;             // q is defined [uninitialized]
  q = &i;             // q now points at i

  //*q = &q;          // q points at q ?!?

  int *p1 = &i;       // pointer p1 points at i
  int **p2 = &p1;     // pointer p2 points at p1

```

```
  printf("The address of i is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &i);
  printf("The address of i is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", p);
  printf("The address of i is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", p1);
  printf("The address of p is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &p);
  printf("The address of q is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &q);
  printf("The address of p1 is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &p1);
  printf("The address of p1 is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", p2);
  printf("The address of p2 is %p.CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", &p2);
}
```

Remember, pointers are variables, and variables store values. A pointer is only “special” because the **value** it stores is an **address**.

Because a pointer is a variable, it *also* has an address itself.

Example:

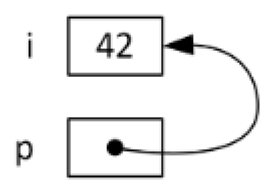
```
int main(void) {
  int i = 42;
  int *p = &i;

  trace_int(i);
  trace_ptr(&i);
  trace_ptr(p);
  trace_ptr(&p);

  int *p1 = NULL;       // GOOD
  trace_ptr(p1);

  if (p) {
    printf("The pointer p is not NULL!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  }

  if (p1 != NULL) {
    printf("The pointer p1 is not NULL!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  } else {
    printf("The pointer p1 is NULL!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  }

  int j = 99;
  trace_ptr(&j);
  int *p2 = NULL;       // p2 is initialized to NULL
  trace_ptr(p2);
  p2 = &j;              // p2 now points at j
  trace_ptr(p2);
  p2 = NULL;            // p2 now points at nothing
  trace_ptr(p2);
  p2 = &i;              // p2 now points at i
  trace_ptr(p2);
}
```

The diagram below illustrates that while the value of p is equal to the address of i, the address of p is different:

```
int i = 42;
int *p = &i;
```

identifier	type	address	value
i	int	0xf020	42
p	int *	0xf024	0xf020

When drawing a *memory diagram*, we rarely care about the value of the pointer (the address of whatever it is pointing at), and visualize a pointer with an arrow (that “points”).



In most k-bit systems, memory addresses are k bits long, so pointers require k bits to store an address. In our 64-bit edX environment, the sizeof a pointer is always 64 bits (8 bytes).

The `sizeof` a pointer is **always the same size**, regardless of the type of data stored at that address.

The ***indirection operator*** (`*`) , also known as the *dereference* operator , is the **inverse** of the *address operator* (`&`).

`*p` produces the **value** of what pointer p "points at".

The structure operator (`.`) has higher precedence than the indirection operator (`*`) which means awkward parenthesis are required to access a field of a pointer to a structure: `(*ptr).field`.

Fortunately, the ***indirection selection operator***, also known as the "arrow" operator (`->`) combines the indirection and the selection operators.

Example:

```
struct posn {
  int x;
  int y;
};

int main(void) {
  struct posn my_posn = {0, 0};
  struct posn *ptr = &my_posn;

  (*ptr).x = 3;                   // awkward
  ptr->y = 4;                     // much better

  trace_int(ptr->x);
  trace_int(ptr->y);
}
```

NULL value

`NULL` is a special **value** that can be assigned to a pointer to represent that the pointer points at "nothing". If the value of a pointer is unknown at the time of definition, or what the pointer points at becomes *invalid*, it's good style to assign the value of `NULL` to the pointer. A pointer with a value of `NULL` is often called a "NULL pointer".

Recall that in C, false is defined to be zero. In fact, `NULL` is also considered to be "`false`" when used in a Boolean context.

Multiple uses of *

The * symbol is used in three different ways in C:

- as the *multiplication operator* between expressions as in line 17 of Example 4.1.5 below
- in pointer *definitions* and pointer *types* as in line 11
- as the *indirection operator* for pointers as in line 12

Aliasing

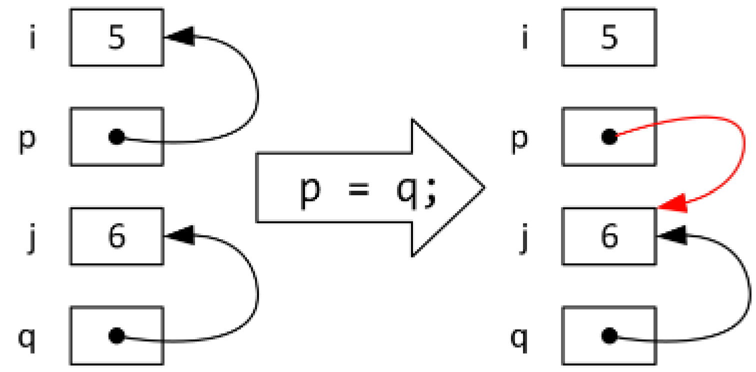
Aliasing occurs when the same memory address can be accessed from more than one pointer variable.

Example:

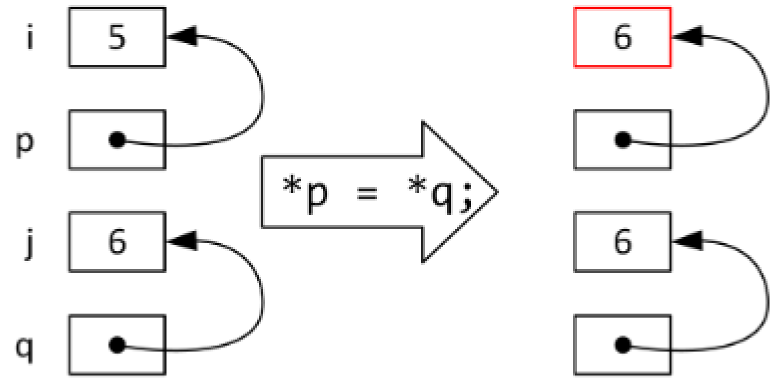
```
int main(void) {
  int i = 5;
  int j = 6;
  int *p = &i;
  int *q = &j;

  p = q;         // the address of p changes, but i does not
  trace_ptr(p);
  trace_ptr(q);
  trace_int(i);

  p = &i;
  *p = *q;       // i changes, but p does not
  trace_ptr(p);
  trace_ptr(q);
  trace_int(i);

    i = 1;
    int *p1 = &i;
    int *p2 = p1;
    int **p3 = &p1;

    trace_int(i);
    *p1 = 10;       // i changes...
```

```
    trace_int(i);
    *p2 = 100;      // without being used directly
    trace_int(i);
    **p3 = 1000;    // same as *(*p3)
    trace_int(i);
}
```

Consider the code in Example 4.1.7. Notice that in line 11 the statement p = q; is a ***pointer assignment***. It means "change p to point at what q points at". It changes the value of p to be the value of q. In this example, the address of j is assigned to p, but the ***value of i is unchanged*** :



After redefining p to point at i in line 16 and running the statement *p = *q;, notice the program does **not** change the value of p. Instead, it changes the value *of what p points at*. In this example, it **changes the value of** i to 6, *even though i was not used in the statement*:



4.2: Pointers - Mutation

Mutation & parameters

The "pass by value" convention of C is where a **copy** of an argument is passed to a function.

The alternative convention to "pass by value" is "pass by reference", where a variable passed to a function can be changed by the function. Some languages support both conventions.

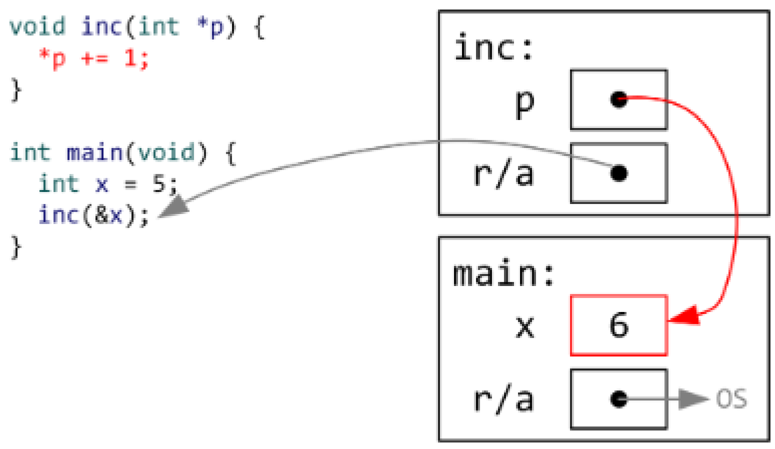
In C we can emulate "pass by reference" by passing **the address** of the variable we want the function to change. Note, that this is actually still technically "pass by value" because we **pass** the value of the address.

However, by passing the address of x, we can change the value of x. It is also common to say "pass a pointer to x".

To pass the address of x use the **address operator** (&x),

Most pointer parameters should be **required** to be valid (*e.g.*, non-NULL).

The following memory diagram illustrates what is happening to the value of the variable x when you get to line 11 of the code (shown in red font in the diagram):



Example:

```
// inc(p) increments the value of *p
// effects: modifies *p
// requires: p is a valid pointer
void inc(int *p) {
  assert(p);
  *p += 1;
```

```
}

int main(void) {
  int x = 5;
  trace_int(x);
  inc(&x);            // note the &
  trace_int(x);
}
```

We now have a fourth side effect that a function may have:

- produce output
- read input
- mutate a global variable
- **mutate a variable through a pointer parameter**

Returning an address

A function may return an address.

As soon as the function returns, the stack frame “disappears”, and all memory within the frame is considered **invalid**.

A function must **never** return an address to a variable within its stack frame.

Example:

```
// bad_idea(n) returns a pointer to the bad_idea frame (BAD!)
int *bad_idea(int n) {
  return &n;            // NEVER do this
}

// bad_idea2(n) returns a pointer to the bad_idea2 frame (BAD!)
int *bad_idea2(int n) {
  int a = n * n;
  return &a;            // NEVER do this
}

// ptr_to_max(a, b) returns either a or b: whichever points to
//   the larger value
// requires: a, b are valid pointers
int *ptr_to_max(int *a, int *b) {
  assert(a);
  assert(b);
  if (*a >= *b) {
    return a;
  }
  return b;
}

int main(void) {
  int x = 3;
  int y = 4;

  int *p = ptr_to_max(&x, &y);        // note the &
  assert(p == &y);
  printf("%pCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", bad_idea(13));
  printf("%pCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", bad_idea2(13));
}
```

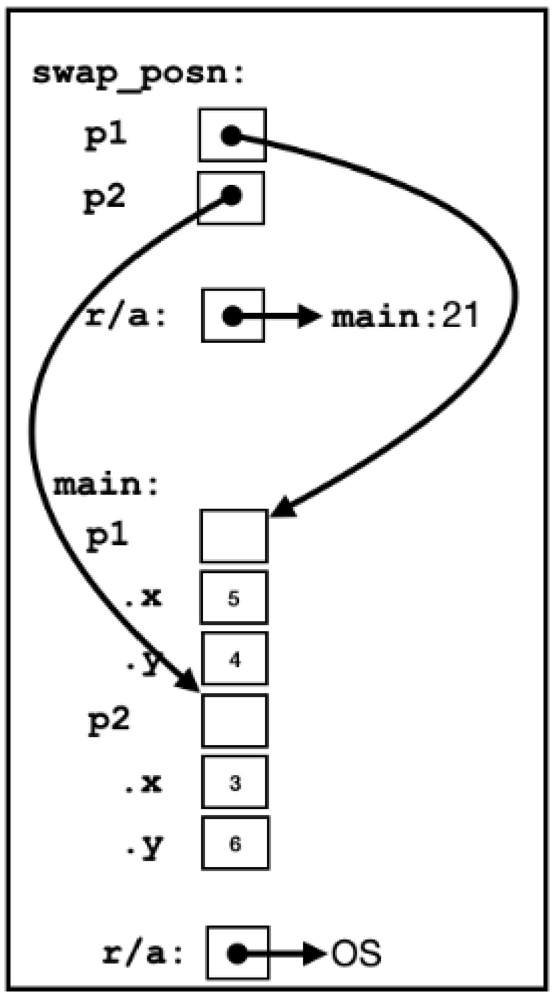
Modelling pointers

Example below demonstrates how you would draw a memory snapshot with pointers and structures. Since we don't know the actual memory addresses, we use [addr_1] for p1 and [addr_2] for p2.

```
void swap_int(int *i, int *j) {
   assert(i);
   assert(j);
   int temp = *i;
   *i = *j;
   *j = temp;
}
```

```
void swap_posn(struct posn *p1,
               struct posn *p2) {
   assert(p1);
   assert(p2);
   swap_int(&(p1->x), &(p2->x));
   // Snapshot
   swap_int(&(p1->y), &(p2->y));
}

int main(void) {
   struct posn p1 = {3, 4};
   struct posn p2 = {5, 6};
   swap_posn(&p1, &p2);
}
```

```
=========================
swap_posn:
  p1: addr_1
  p2: addr_2
  return address: main:21
=========================
main:
  p1: [addr_1]
    .x = 5
    .y = 4
  p2: [addr_2]
    .x = 3
    .y = 6
  return address: OS
=========================
```



STACK

4.3: C Input & Pointers

C input: scanf

We can now use the built-in `scanf` function.

`scanf` takes two required parameters: the format specifier for the type of data to be read in (`%d` in the case of an integer) and a **pointer** to a variable to **store** the value read in from input.

The **return value** of `scanf` is an integer, and either:

- the quantity (count) of values *successfully* read, or
- the constant `EOF`: the **E**nd **O**f **F**ile (`EOF`) has been reached.

If input is not formatted properly a zero is returned (*e.g.*, the input is hello and we try to scanf an int with `%d`).

It is important that you always check the return value of scanf to be sure that you have successfully scanned in a single character (if you are following our advice to read one value per scanf).

Example:

```
retval = scanf("%d", &i); // read in an integer, store it in i
if (retval != 1) {
  printf("Fail! I could not read in an integer!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
}
```

Always check the return value of `scanf`: one is "success".

Consider the following statement: `retval = scanf("%d", &i);`. There are three separate side effects:

- a value is read from input

- `i` is mutated
- `retval` is mutated

Whitespace

When reading an int with `scanf ("%d")` C **ignores any whitespace** (spaces and newlines) that appears before the next int. However, when reading in a char, you *may or may not* want to ignore whitespace: it depends on your application.

The difference between these two programs is subtle. There is a space before in the format specifier if you want whitespace to be ignored(`scanf(" %c", &c)`), whereas it does not include the space if you want to include the whitespace(`scanf("%c", &c)`).

Using pointers to "return" multiple values

C functions can only return a single value. However, recall how scanf is used: `retval = scanf("%d", &i);` We "receive" two values: the return value, *and* the value read in (stored in `i`).

In fact, pointer parameters can be used to *emulate* "returning" more than one value. The addresses of several variables can be passed to a function, and the function can change the value of those variables.

This "multiple return" technique is also useful when it is possible that a function could encounter an error.

In C, we can use the emulated pass-by-reference functionality to return multiple values.

4.4: Pointers - Structures, Constants, and Functions

Passing structures

Recall that when a function is called, a **copy** of each argument value is placed into the stack frame. For structures, the *entire* structure is copied into the frame. For large structures, such as the bigstruct example below, this can be inefficient.

```
struct bigstruct {
  int a;
  int b;
  int c;
  int d;
  ...
  int y;
  int z;
};
```

To avoid structure copying, it is very common to pass the *address* of a structure to a function

We now have **two** different reasons for passing a structure pointer to a function:

- to avoid copying the structure
- to mutate the contents of the structure

While it would be good to communicate whether or not there is a side effect (mutation), documenting the **absence** of a side effect ("no side effect here") is awkward.

Example:

```
struct posn {
  int x;
  int y;
};


// sqr_dist(p1, p2) calculates the square of
// the distance between p1 and p2
// requires: p1, p2 are not null
int sqr_dist(struct posn *p1, struct posn *p2) {
  assert(p1);
  assert(p2);
  int xdist = p1->x - p2->x;
  int ydist = p1->y - p2->y;
  return xdist * xdist + ydist * ydist;
}

// scale(p, f) scales the posn p by f
// requires: p is not null
// effects: modifies p
void scale(struct posn *p, int f) {
```

```
  assert(p);
  p->x *= f;
  p->y *= f;
}

int main(void) {
  struct posn a = {1, 2};
  struct posn b = {4, 6};

  trace_int(sqr_dist(&a, &b));
  scale(&a, 2);
  trace_int(a.x);
  trace_int(a.y);
}
```

const pointers

Adding the const keyword to the *start* of a pointer definition prevents the pointer's **destination** (the variable it points at) from being mutated through the pointer.

It is **good style** to add const to a pointer parameter to communicate (and enforce) that the pointer's destination does not change.

A pointer definition that *begins* with const prevents the **pointer's destination** from being mutated *via the pointer*.

However, the pointer variable itself is still mutable, and can point to another int.

`int *p;`	`p` can point at any mutable integer, you can modify the int (via `*p`)
`const int *p;`	`p` can point at any integer, you can NOT modify the integer (via `*p`)
`int * const p = &i;`	`p` always points at the integer `i`, `i` must be mutable and can be modified (via *p)
`const int * const p = &i;`	`p` always points at the integer `i`, you can not modify `i` (via `*p`)

Minimizing mutative side effects

Previously in the course we used *mutable* global variables to demonstrate mutation and how functions can have mutative side effects.

Global mutable variables make your code harder to understand, maintain and test. On the other hand, global **constants** are "good style" and encouraged.

Global mutable variables are strongly discouraged and considered "poor style".

Your preference for function design should be:

1. **"Pure" function**: No side effects or dependencies on global *mutable* variables.
2. **Only I/O side effects**: If possible, avoid any mutative side effects.
3. **Mutate data through pointer parameters**: If mutation is necessary, use a pointer parameter.
4. **Dependency on global mutable variables**: Mutable global variables should be avoided.
5. **Mutate global data**: Only when absolutely necessary (it rarely is).

Function pointers

In C, functions are not first-class values, but ***function pointers*** are. A significant difference between C and Racket is that **new** Racket functions can be created during program execution, while in C they cannot.

A function pointer can only point to a function that already exists.

A *function pointer* stores the (starting) address of a function, which is an address in the code section of memory. The type of a function pointer includes the *return type* and all of the *parameter types*, which makes the syntax a little messy.