Introduction to Pointers in C

Readings: CP:AMA 11, 17.7

The primary goal of this section is to be able use pointers in C.

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).

```c
int g = 42;
int main(void) {
    printf("the value of g is: %d\n", g);
    printf("the address of g is: %p\n", &g);
}
the value of g is: 42
the address of g is: 0x71a0a0
```

The printf placeholder to display an address (in hex) is "%p".

Pointers

In C, there is also a type for storing an address: a pointer.

A pointer is defined by placing a star (*) before the identifier (name). The * is part of the definition syntax, not the identifier itself.

```c
int i = 42;
int *p = &i;  // p "points at" i
```

The type of p is an “int pointer” which is written as “int *”.

For each type (e.g., int, char) there is a corresponding pointer type (e.g., int *, char *).
This definition:

```c
int *p = &i; // p "points at" i
```

is comparable to the following definition and assignment:

```c
int *p;       // p is defined (not initialized)
p = &i;       // p now "points at" i
```

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

As with any variable, its value can be changed.

```c
p = &j;       // p now "points at" j
p = &i;       // p now "points at" i
```

The value of a pointer is an address.

```c
int i = 42;
int *p = &i;
trace_int(i);
trace_ptr(&i);
trace_ptr(p);
trace_ptr(&p);
```

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>int</td>
<td>0xf020</td>
<td>42</td>
</tr>
<tr>
<td>p</td>
<td>int *</td>
<td>0xf024</td>
<td>0xf020</td>
</tr>
</tbody>
</table>

To make working with pointers easier in these notes, we often use shorter, simplified (“fake”) addresses.

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

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

```
```

i 42

p
**sizeof a pointer**

In most $k$-bit systems, memory addresses are $k$ bits long, so pointers require $k$ bits to store an address.

In our 64-bit Seashell 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.

```
sizeof(int *) ⇒ 8
sizeof(char *) ⇒ 8
```

**Indirection operator**

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".

```c
int i = 42;
int *p = &i; // pointer p points at i

trace_ptr(p);
trace_int(*p);

p => 0xf020
*p => 42
```

The value of *&i is simply the value of i.

The **address operator** (\&) can be thought of as:
"get the address of this box".

The **indirection operator** (\*) can be thought of as:
"follow the arrow to the next box and get its contents".
The * symbol is used in three different ways in C:

- as the **multiplication operator** between expressions
  
  \[ k = i \times i; \]

- in pointer **definitions** and pointer **types**
  
  \[
  \text{int } *p = &i; \\
  \text{sizeof(int *)}
  \]

- as the **indirection operator** for pointers
  
  \[
  j = *p; \\
  *p = 5;
  \]

\((p * *p)\) is a confusing but valid C expression.

C mostly ignores whitespace, so these are equivalent
\[
\text{int } *p = &i; \quad \text{// style A} \\
\text{int } *p = &i; \quad \text{// style B} \\
\text{int* } p = &i; \quad \text{// style C}
\]

There is some debate over which is the best style. Proponents of style B & C argue it's clearer that the type of \(p\) is an "int *".

However, **in the definition** the * "belongs" to the \(p\), not the \text{int},
and so style A is used in this course and in CP:AMA.

This is clear with multiple definitions: (not encouraged)
\[
\text{int } i = 42, j = 23; \\
\text{int } *p1 = &i, *p2 = &j; \quad \text{// VALID} \\
\text{int } *p1 = &i, p2 = &j; \quad \text{// INVALID: p2 is not a pointer}
\]

**Pointers to pointers**

A common question is: "Can a pointer point at itself?"

\[
\text{int } *p = &p; \quad \text{// pointer p points at p }???
\]

This is actually a **type error**:

- \(p\) is defined as \((\text{int } *)(\text{int }*)\), a pointer to an \text{int}, but
- the type of \&p is \((\text{int } **)\), a pointer to a pointer to an \text{int}.
In C, we can define a **pointer to a pointer**:

```c
int i = 42;
int *p1 = &i;    // pointer p1 points at i
int **p2 = &p1; // pointer p2 points at p1
```

C allows any number of pointers to pointers. More than two levels of “pointing” is uncommon.

```c
(*p * *p) is a confusing but valid C expression.
```

A **void** pointer (**void**) can point at anything, including a **void** pointer (itself).

---

**The NULL pointer**

**NULL** is a special pointer **value** to represent that the pointer points to “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.

```c
int *p;       // BAD (uninitialized)
int *p = NULL; // GOOD
```

Some functions return a **NULL** pointer to indicate an error.

**NULL** is considered “false” when used in a Boolean context (**false** is defined to be zero or **NULL**).

The following two are equivalent:

```c
if (p) ...
if (p != NULL) ...
```

If you try to **dereference** a **NULL** pointer, your program will crash.

```c
p = NULL;
i = *p;     // crash!
```
Pointer assignment

Consider the following code

```c
int i = 5;
int j = 6;
int * p = &i;
int * q = &j;
p = q;
```

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, it assigns the address of `j` to `p`.

It does not change the value of `i`.

Using the same initial values,

```c
int i = 5;
int j = 6;
int * p = &i;
int * q = &j;

*p = *q;
```

does not change the value of `p`; 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.

This is an example of **aliasing**, which is when the same memory address can be accessed from more than one variable.
example: aliasing

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

trace _ int(i);
*p1 = 10; // i changes...
trace _ int(i);
**p3 = 1000;
trace _ int(i);

i => 1
i => 10
i => 100
i => 1000
```

Mutation & parameters

Consider the following C program:

```c
void inc(int i) {
    ++i;
}

int main(void) {
    int x = 5;
    inc(x);
    trace _ int(x);  // 5 or 6 ?
}
```

It is important to remember that when `inc(x)` is called, a copy of `x` is placed in the stack frame, so `inc` cannot change `x`.

The `inc` function is free to change it’s own copy of the argument (in the stack frame) without changing the original variable.
In the “pass by value” convention of C, a **copy** of an argument is passed to a function.

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

What if we want a C function to change a variable passed to it? (this would be a side effect)

In C we can *emulate* “pass by reference” by passing the **address** of the variable we want the function to change.

This is still actually “pass by value” because we pass the **value** of the address.

By passing the **address** of \( x \), we can change the **value** of \( x \).

It is also common to say “pass a pointer to \( x \)”.  

```c
void inc(int * p) {
    * p += 1;
}

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

\( x \) => 5
\( x \) => 6

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

The corresponding parameter type is an **int pointer** (int *).
Whenever you have a parameter that is a pointer, you should **require** that it is a valid (*e.g.*, non-**NULL**) pointer.

```c
void inc(int *p) {
    *p += 1;
}
```

The parentheses are necessary because of the order of operations: `++` would have incremented the pointer `p`, not what it points at (`*p`).

Note that instead of `*p += 1;` we could have written `(*p)++;

**example: mutation side effects**

```c
void swap(int *px, int *py) {
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

```c
int main(void) {
    int a = 3;
    int b = 4;
    trace_int(a); trace_int(b);  // Note the &
    swap(&a, &b);               // Note the &
    trace_int(a); trace_int(b);
}
```

```
a => 3
b => 4
a => 4
b => 3
```
Documenting side effects

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**

```c
// effects: modifies *px and *py
void swap(int *px, int *py) {
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

In the *functional paradigm*, there is no observable difference between “pass by value” and “pass by reference”.

In Racket, simple values (*e.g.*, numbers) are passed by *value*, but structures are passed by *reference*.

---

C input: scanf

So far we have been using our tools (*e.g.*, `read_int`) to read input. We are now capable of using the built-in `scanf` function.

```c
scanf("%d", &i); // read in an integer, store it in i
```

`scanf` requires a **pointer** to a variable to **store** the value read in from input.

Just as with `printf`, you use multiple placeholders to read in more than one value.

**However, in this course only read in one value per `scanf`.**

This will help you debug your code and facilitate our testing.
The return value of `scanf` is the number (count) of values successfully read.

The return value can also be the special constant value `EOF` to indicate that the End Of File (EOF) has been reached.

A Ctrl-D (“Control D”) keyboard sequence sends an `EOF`.

Always check the return value of `scanf`: one is “success”. (if you’re following our advice to read one value per `scanf`).

```c
    retval = scanf("%d", &i); // read in an integer, store it in i
```

```c
    if (retval != 1) {
        printf("Fail! I could not read in an integer!\n");
    }
```

As with our `read_int`, `scanf("%d", &i)` ignores whitespace (spaces and newlines) when reading the next integer.

If the next non-whitespace input to be read is not a valid integer (e.g., a letter), it stops reading and returns zero.

When reading in a `char`, you may or may not want to ignore whitespace.

```c
    // reads in next character (may be whitespace character)
    count = scanf("%c", &c);

    // reads in next character, ignoring whitespace
    count = scanf(" %c", &c);
```

The extra leading space in the second example indicates that whitespace should be ignored.

Returning more than one value

C functions can only return a single value.

Pointer parameters can be used to emulate “returning” more than one value.

The addresses of several variables can be passed to the function, and the function can change the value of those variables.
This function performs division and "returns" both the quotient and the remainder.

```c
void divide(int num, int denom, int * quot, int * rem) {
    * quot = num / denom;
    * rem = num % denom;
}
```

Here is an example of how it can be used:

```c
divide(13, 5, &q, &r);
trace_int(q);
trace_int(r);
```

```
q => 2
r => 3
```

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

For example, the previous `divide` example could return `false` if it is successful and `true` if there is an error (i.e., division by zero).

```c
bool divide(int num, int denom, int * quot, int * rem) {
    if (denom == 0) return true;
    * quot = num / denom;
    * rem = num % denom;
    return false;
}
```

Some C library functions use this approach to return an error. Other functions use "invalid" sentinel values such as -1 or `NULL` to indicate when an error has occurred.

---

**Returning an address**

In Section 10, we use functions that return an address (pointer).

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

```c
int *bad_idea(int n) {
    return &n; // NEVER do this
}
```

```c
int *bad_idea2(int n) {
    int a = n*n;
    return &a; // NEVER do this
}
```

As soon as the function **returns**, the stack frame "disappears", and all memory within the frame should be considered **invalid**.
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, this can be inefficient.

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

Large structures also increase the size of the stack frame. This can be especially problematic with recursive functions, and may even cause a stack overflow to occur.

```c
int sqr_dist(struct posn *p1, struct posn *p2) {
    int xdist = (*p1).x - (*p2).x;
    int ydist = (*p1).y - (*p2).y;
    return xdist * xdist + ydist * ydist;
}
```

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

```c
int main(void) {
    struct posn p1 = {2, 4};
    struct posn p2 = {5, 8};

    trace_int(sqr_dist(&p1, &p2)); // note the &
}
```

The parentheses () in the expression (*p1).x are used because the structure operator (.) has higher precedence than the indirection operator (*).

Without the parentheses, *p1.x is equivalent to *(p1.x) which is a “type” syntax error because p1 does not have a field x.

Writing the expression (*ptr).field is awkward.

There is an additional selection operator for working with pointers to structures.
The **arrow selection operator** (\(\rightarrow\)) combines the indirection and the selection operators.

\[
\text{ptr->field} \text{ is equivalent to } (*\text{ptr}).\text{field}
\]

The arrow selection operator can only be used with a **pointer to a structure**.

```c
int sqr_dist(struct posn *p1, struct posn *p2) {
    int xdist = p1->x - p2->x;
    int ydist = p1->y - p2->y;
    return xdist * xdist + ydist * ydist;
}
```

Passing the address of a structure to a function (instead of a copy) also allows the function to mutate the fields of the structure.

```c
// scale(p, f) scales the posn p by f
// requires: p is not null
// effects: modifies p

void scale(struct posn *p, int f) {
    p->x *= f;
    p->y *= f;
}
```

In the above documentation, we used \(p\), where \(\ast p\) would be more correct. It is easily understood that \(p\) represents the structure.

```c
// this is more correct, but unnecessary:

// scale(p, f) scales the posn \(\ast p\) by f
// effects: modifies \(\ast p\)
```

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

It would be good to communicate whether or not there is a side effect (mutation).

However, documenting the **absence** of a side effect (“no side effect here”) is awkward.
**const pointers**

Adding the `const` keyword to a pointer definition prevents the pointer's destination from being mutated through the pointer.

```c
void cannot_change(const struct posn *p) {
    p->x = 5; // INVALID
}
```

The `const` should be placed first, before the type.

---

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

---

The syntax for working with pointers and `const` is tricky.

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

The rule is "`const` applies to the type to the left of it, unless it's first, and then it applies to the type to the right of it".

```c
const int i = 42; // these are equivalent
int const i = 42; // but this form is discouraged
```

---

**const parameters**

As we just established, it is good style to use `const` with pointer parameters to communicate that the function does not (and can not) mutate the contents of the pointer.

```c
void can_change(struct posn *p) {
    p->x = 5; // VALID
}

void cannot_change(const struct posn *p) {
    p->x = 5; // INVALID
}
```

What does it mean when `const` is used with simple (non-pointer) parameters?
For a simple value, the `const` keyword indicates that the parameter is immutable within the function.

```c
int my_function(const int x) {
    // mutation of x here is invalid
    // ...
}
```

It does not require that the argument passed to the function is a constant.

Because a `copy` of the argument is made for the stack, it does not matter if the original argument value is constant or not.

A `const` parameter communicates (and enforces) that the copy of the argument will not be mutated.

---

**Minimizing mutative side effects**

In Section 03 we used `mutable` global variables to demonstrate mutation and how functions can have mutative side effects.

In practice, `mutable` global variables are **strongly discouraged** and considered “bad style”.

They make your code harder to understand, maintain and test.

On the other hand, global `constants` are “good style” and encouraged.

There are rare circumstances where global mutable variables are necessary.

---

Your preference for function design should be:

1. **No mutative side effects**
   - Functions are much easier to understand and test.

2. **Mutate data through pointer parameters**
   - When mutation of data is required, use a pointer parameter to access the data.

3. **Mutate global data**
   - This should only be used when absolutely necessary.

† In addition, the behaviour of a function should not depend on a global mutable variable. Multiple calls to a function with identical arguments and data should have identical results.
Function pointers

In Racket, functions are *first-class values*.

For example, Racket functions are values that can be stored in variables and data structures, passed as arguments and returned by functions.

In C, functions are not first-class values, but *function pointers* are.

A significant difference 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.

The syntax to define a function pointer with name *fpname* is:

```
return_type (*fpname)(param1_type, param2_type, ...)
```

In an exam, we would not expect you to remember the syntax for defining a function pointer.

**example: function pointer**

```c
int my_add(int x, int y) {
    return x + y;
}

int my_sub(int x, int y) {
    return x - y;
}

int main(void) {
    int (*fp)(int, int) = NULL;
    fp = my_add;
    trace_int(fp(7, 3));
    fp = my_sub;
    trace_int(fp(7, 3));
}
```

`fp(7, 3) => 10`

`fp(7, 3) => 4`
Goals of this Section

At the end of this section, you should be able to:

• define and dereference pointers
• use the new operators (&, *, -)
• describe aliasing
• use the scanf function to read input

• use pointers to structures as parameters and explain why parameters are often pointers to structures
• explain when a pointer parameter should be const
• use function pointers