Module 4: Pointers

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 **address** of an identifier: the location in memory where it is stored. An address is just a number, but is traditionally written in hexadecimal.

We use the conversion specifier `%p` to print an address, and `trace_ptr` to trace it.

```c
int main(void) {
  int g = 42;
  printf("the value of g is:   %d\n",  g); // use %d to print an int (as we know).
  printf("the address of g is: %p\n", &g); // use %p to print an address.
  trace_ptr(&g);   // display this address for debugging.
}
```

We see something like:

```
the value of g is:   42
the address of g is: 0x7ffec2eef410
>>> [pointer-demo.c|main|7] >> &g => 0x7ffec2eef410
```

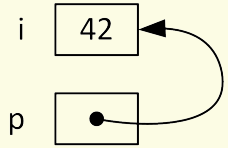
Pointers

In C, there is also a kind of *type* for storing an address: a **pointer**.
If we add a `*` after some type `T`, it is a new type. "`T *`" is a pointer to a `T`.

```
int i = 42;   // a variable named i that stores 42
int *p = &i;  // a variable named p that stores the address of i.
```

The *type* of `p` is "`int *`" which means "pointer to an `int`".
The numeric value of a pointer is almost never directly useful.
We will draw a pointer as an arrow pointing to a value.
The pointer **is** the arrow.

Like this →



Exercise

Draw a picture of what this looks like:

```
int x = 5;
int *p = &x;
int **q = &p;
```

Exercise

Write a function `show_int_ptr` that takes an `int`, and a pointer to another `int`. The function shall print the address of the `int`, and the value of the pointer. Use `%p`.

Exercise

What do we see when we call `show_int_ptr(x, &x)` for some `x`? Why?

This definition:

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

is comparable to the following definition and assignment:

```
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.

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

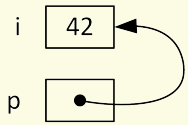
Indirection operator

The *indirection operator* (`*`), is the inverse of the *address operator* (`&`).
Use of the indirection operator is referred to as **dereferencing**.

When `p` is a pointer, the value of `*p` is the value of what pointer `p` "points to".

- "`&x`" can be thought of as "get the address of box `x`".
- "`*x`" can be thought of as "get the contents of the box that `x` points to".

```
int i = 42;
int *p = &i;
trace_ptr(p);  ⇒ 0x7ffe16d57550
trace_int(*p); ⇒ 42
```



Exercise: Consider: what do we get from `*&i` ?

Exercise: Consider: what do we get from `&*i` ?

Indirection operator

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

- as the *multiplication operator* between expressions

```
k = i * i;
```

- in pointer *definitions* and pointer *types*

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

- as the *indirection operator* for pointers

```
j = *p;
*p = 5;
```

Exercise Create variables `x` and `y` so that `x ** y` $\Rightarrow$ `81`.

Exercise Create what is needed so that `(**p) * (**p)` $\Rightarrow$ `81`.

Pointers to pointers

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

```
int *p = &p;      // pointer p points at p ???
```

Exercise: Try to run this code. What error message do you get?

This is **type error**:

- `p` must contain a value of type `(int *)`; that is, it must point at an `int`.
- The type of `&p` is `(int **)`, a pointer to a pointer to an `int`.

Pointers to pointers

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

```
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.

> A `void` pointer (`void *`) can point at anything, including itself.
> We won’t have any use for them until Section 09.

Pointer assignment

Consider the following code:

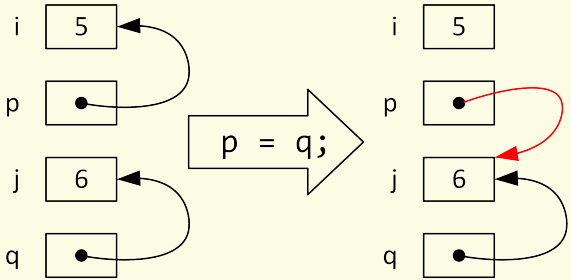
```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;
```

Exercise: Draw a memory diagram for this code.

Exercise: Update your memory diagram as if we ran: `p = q;`

Pointer assignment

Notice: this does does not change the value of `i` or `j`.



Pointer assignment

Consider that code again:

```
int i = 5;
int j = 6;

int *p = &i;
int *q = &j;
```

Exercise: Update your memory diagram as if we ran: `*p = *q;`

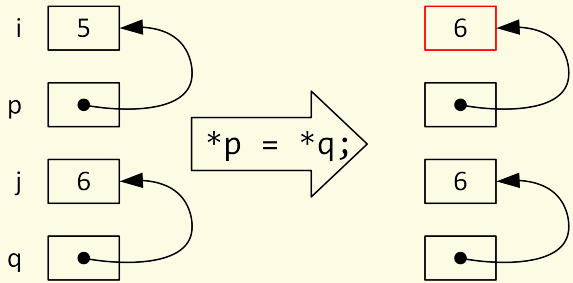
An "alias" is an "alternative name" used for some purpose.
"James Bond" sometimes goes by the name of "Arlington Beech" or "Robert Sterling"; these are "aliases" that he uses. If a villain could kill "Arlington Beech", they would also kill "James Bond", because they are the same person.

Here `i` and `*p` are two different "names" for that particular slot in memory. Since they refer to the same slot in memory, we say they are **aliases**.

If we change `*p`, we also change `i`, because they are the same slot.

Pointer assignment

Notice: this does **not** change the value of `p`: it changes the value *of what `p` points to*. In this example, it **changes the value of `i`** to `6`, *even though `i` was not used in the statement*.



Pointer assignment

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

trace_int(i);
*p1 = 10;
trace_int(i);
*p2 = 100;
trace_int(i);
**p3 = 1000;
trace_int(i);
```

Exercise

Carefully trace this code. What value does `i` have at the 4 calls to `trace_int`?

Here too, we see aliasing.

Consider the following C program:

```
1  void inc(int i) {
2    ++i;
3  }
4
5  int main(void) {
6    int x = 5;
7    inc(x);
8    trace_int(x);     // 5 or 6 ?
9  }
```

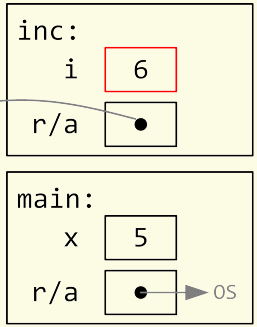
Exercise: Draw a stack frame immediately before line 2 is executed.

When `inc(x)` is called, the value of the variable `x` is placed in the stack frame — a “copy”.

The function `inc` can change this copy, stored in `i`, but not `x`.

Mutation & parameters

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

int main(void) {
  int x = 5;
  inc(x);
}
```



Consider the following apparently similar program:

```
void inc(int *p) {
  (*p)++;
}
int main(void) {
  int x = 5;
  trace_int(x);
  inc(&x);
  trace_int(x);
}
```

Exercise Draw a stack frame immediately before line 2 is executed. Work out what values are traced.

x ⇒ 5
x ⇒ 6

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

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.

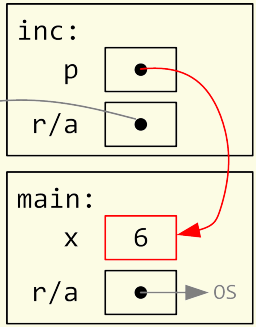
What if we want a C function to change a variable passed to it?

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.

Mutation & parameters

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

int main(void) {
  int x = 5;
  inc(&x);
}
```



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.

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

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

The NULL pointer

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

The following two are equivalent:

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

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

```
p = NULL;
i = *p;       // crash!
```

The NULL pointer

It can be hard to debug crashes involving `NULL` pointers.

Whenever you have a parameter that is a pointer, you should **require** that it is a valid (e.g. non-`NULL`) pointer.

```
// inc(p) increment the value of *p
// effects: modifes *p
// requires: p is a valid pointer
void inc(int *p) {
  assert(p != NULL);
  (*p)++;
}
```

Exercise: Why did we write `(*p)++` instead of `*p++`?

Refer again to the C operator precedence chart.

Side Effects

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

int main(void) {
  int a = 3;
  int b = 4;
  printf("a:%d; b:%d\n", a, b);
  swap(&a, &b);
  printf("a:%d; b:%d\n", a, b);
}
```

Exercise

Draw stack frames to trace this code. What is printed?

```
a:3; b:4
a:4; b:3
```

Documenting side effects

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

1. produce output (`printf`)
2. read input (`read_int` etc.)
3. mutate a global variable
4. ~~mutate a global variable, but not in CS136~~
5. mutate a variable through a pointer parameter

Add a *`//effects:`* section to any function that modifies anything outside its own stack frame:

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

Pass by value and by reference comparison

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*.
(In Python, everything is always passed by reference. But numbers are immutable.)

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.

Returning more than one value

example: "returning" more than one value

This function performs division and "returns" both the quotient and the remainder.

```
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:

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

```
q ⇒ 2
r ⇒ 3
```

Returning more than one value

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 (division by zero).

```
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.

Reading input: `scanf`

So far we have been using our tools (e.g. `read_int`) to read input.
We can now use the standard `scanf` function:

```
int count = scanf("%d", &i);
```

`scanf` **mutates** a parameter, and also **returns** an integer showing how many things read.

- 0 indicates that there was something to read, but it did not look like an integer.
- the constant `EOF` (end-of-file) indicates that there was nothing more to read.

Reading input: `scanf`

Consider this example:

```
void read_once(void){
  int i = 12345;
  int count = scanf("%d", &i);
  printf("i is %d; count is %d\n", i, count);
}
int main(void) {
  read_once();
  read_once();
}
```

If we give as input `42 24601`, we see:

```
i is 42; count is 1
i is 24601; count is 1
```

If we give as input `17`, we see:

```
i is 17; count is 1
i is 12345; count is -1
```

If we give as input `foo 42`, we see:

```
i is 12345; count is 0
i is 12345; count is 0
```

Exercise

Use `scanf` to write a function `int my_read_int(void)` that

- returns an integer that it read, when possible, and
- returns `READ_INT_FAIL` when it cannot read.

Reading input: `scanf`

As with `printf`, you can use multiple placeholders to read in more than one value. This is valid:

```
int count = scanf("%d%d", &i, &j);
if (2 != count) {
  printf("Tried to read 2 things, but got %d\n", count);
}
```

It is usually wise to read only one value per `scanf`. That is, **don't** do what we did above. Have only one `%` pattern in your string.

This will make your code easier to debug, and facilitate our testing.

> **!** In this course, you must read only one value per `scanf`.

Reading input: `scanf`

If you are entering input at a terminal (e.g. a bash prompt), you can send “end of file” by typing `Ctrl-D` (“Control D”).

Reading characters with `scanf`

We can read numbers using `scanf`. What about things are aren't numbers?

Recall that with `printf` we can use `%c` to print a character:

`printf("Character (%d) is [%c]\n", 42, 42);` displays `Character (42) is [*]`

Similarly, we can read a single character with `scanf` using `%c`:

```
void read_once_char(void) {
  char c = 'Q';
  int count = scanf("%c", &c);
  printf("c is %c; count is %d\n", c, count);
}
int main(void) {
  read_once_char();
  read_once_char();
}
```

Exercise

Write a function that reads all characters from input and prints them, turning all `'a'` characters into `'A'`.

For example, given input of

```
A man a plan a canal: Panama!
```

it prints

```
A mAn A plAn A cAnAl: PAnAmA!
```

Reading characters with `scanf`

Sometimes you might want to ignore whitespace (spaces, tabs, newlines, etc.)

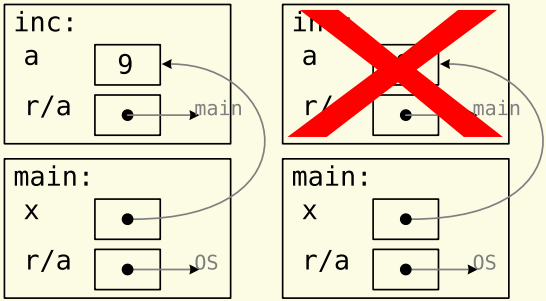
There are features built in to `scanf` to support this;

```
// 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 an address

```c
int *bad_idea(int n) {
  int a = n * n;
  return &a;
}
int main(void) {
  int *x = bad_idea(3);
  printf("%p -> %d\n", x, *x);
}
```



Exercise

Draw a stack trace at the moment when `printf` is called.

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

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

In Section 09, we will create functions that return an address. Until then, don’t.

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. This can take a lot of space.

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

There is a stack frame for each call to a function; calling a recursive function may create many stack frames. If each stack frame is large, we can run out of stack space: a condition called a *stack overflow*.

Passing structures

To avoid structure copying, we can instead pass a pointer to a `struct`:

```
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;
}

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

  trace_int(sqr_dist(&p1, &p2));
```

```
>>> [sqr_dist.c|main|18] >> sqr_dist(&p1, &p2) => 25
```

Exercise

Draw the stack frame just before `sqr_dist` returns.

Pointers to structures: the arrow selection operator ->

Consider this:

```
int xdist = (*p1).x - (*p2).x;
//          ^   ^     ^   ^
```

Why do we have those brackets?

Refer to the C operator precedence chart.
We do “Structure member access” before “Indirection”.

So `*p.x` means `*(p.x)`. But `p` is a “`struct posn *`”; it does not have a “`.x`”.

We can use the `->` operator in such situations. `X->Y` is shorthand for `(*X).Y`

Exercise

Rewrite `sqr_dist` so it contains no brackets.

```
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;
}
```

Exercise

Write a function `void scale(posn *p, int f)` that mutates `*p` so it is scaled by `f`.
Use the `->` operator. Then make an alternative version that does not use this operator.
For example,

```
struct posn q = {.x = 3, .y = 4};
scale(&q, 2);
assert(q.x == 6 && q.y == 8);
scale(&q, 10);
assert(q.x == 60 && q.y == 80);
```

! Remember to document the side effects of your function!

`const` pointers

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

```
void cannot_change(const struct posn *p) {
  p->x = 5;   // INVALID; we promised not to change the struct posn.
}
```

> **!** Use `const` with parameters, especially `struct` parameters, whenever you can.

- Only omit the `const` if you intend need to mutate something.
- When you omit `const`, add an `effects:` section to document what you will mutate.

There is a little subtlety for working with pointers and `const`. The rule is:

> "If `const` is the first thing in a type, it applies only to the thing that follows it. Otherwise, it applies to the thing that comes before it."

Exercise

Imagine we have `int i = 3;` and `int j = 4;`. For each variable `a, b, c`:

- find an expression that makes some change, if possible
- find an expression that the compiler rejects.

```
const int *a = &i;
int * const b = &i;
const int * const c = &i;
```

```
a = &j; ⇒ OK;        *a = 5; ⇒ error: assignment of read-only location '*a'
*b = 5; ⇒ OK;        b = &j; ⇒ error: assignment of read-only variable 'b'
// "const int" means we can't change the int; "* const" means we can't change the *.
```

`const` pointers

For a simple value, the `const` keyword indicates that the parameter is immutable *within the function*.

```
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.

First Class Values

In Racket, a function is a *first class value*, meaning that anything you can do with other values, you can do with functions. Remember:

- A function can consume a function:
  `(map add1 (list 2 4 6 0 1))` ⇒ `(list 3 5 7 1 2)`
- We can store a function in a variable or list:
  ```
  (define myfunc add1)
  (myfunc 41) ⇒ 42
  (define funcs (list add1 sqr map))
  ```
- We can make a function that returns a function:
  ```
  (define (make-adder n) (lambda (x) (+ x n)))
  ((make-adder 5) 1) ⇒ 6
  ```
- etc.

You cannot do these things in C. But you can do some of them with **function pointers**.

Function Pointers: meaning

A function pointer is a pointer.
It points to memory that contains the machine code representation of the function.

```c
int main(void) {
  int i = 0x11223344;
  int (*p)(void) = &main;
  // p points to the machine code of main.
  // This madness prints some of it.
  for (i=0; i<30; ++i){
    printf(" %02x", *(((unsigned char *) p) + i));
  } // Don't worry about what this means.
}
```

In the machine code, we see:

```
f3 0f 1e fa
55
48 89 e5
48 83 ec 10
45 f4 44 33 22 11
8d 05 e6 ff ff ff
48 89 45 f8
```

When we run this, it prints:

```
f3 0f 1e fa 55 48 89 e5 48 83 ec 10 c7 45 f4 44 33 22 11 48 8d 05 e6 ff ff ff 48 89 45 f8
```

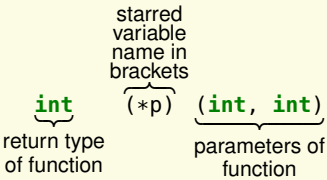
We cannot really use this information!
Message: a function pointer is just a pointer to memory, no different from any other pointer.

Function Pointers: syntax

In Racket, we describe a function that consumes a `X`, a `Y`, and a `Z` and returns a `R` as a `(X Y Z -> R)`. For example, `substring` is a `(Str Nat Nat -> Str)`; `add1` is a `(Num -> Num)`.

A similar amount of information is needed to describe a function point in C.

To define a variable or parameter that is a pointer to a function, we write something like:



That's why we stored a pointer to our `int main(void)` function in a variable defined like `int (*p)(void)`.

Exercise

Write a function `foo` so that `int (*p)(int, int) = &foo;` compiles.

Function Pointers: syntax

We can't do much by working with the machine code of a function.

The only reasonable way to use a function pointer it to call the function.

Here we have a simple function:

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

We can save it in a variable:

```
int (*f)(int, int) = &add;
```

And use it:

```
assert(f(3, 4) == 7);
```

Here are a few simple functions:

```
int add1(int n)  { return n + 1; }
int twice(int x) { return x * 2; }
int sqr(int i)   { return i * i; }
```

Exercise

Write a function `countdown` so that:

`countdown(5, &add);` prints `6 5 4 3 2 1`

`countdown(5, &twice);` prints `10 8 6 4 2 0`

`countdown(5, &sqr);` prints `25 16 9 4 1 0`

Function Pointers: syntax

In C, the `&` is redundant when working with function pointers. You **can** omit it.

If `f` is a function, and `p` is a function pointer where the statement makes sense, then these two statements are equivalent:

```
p = f;
p = &f;
```

I think it's clearer to include the `&`.

- It helps you remember that it's a pointer;
- It makes it clear that you didn't mean to call the function, like `p = f();`
- It's necessary under certain conditions in C++.

Goals of this Section

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

- define pointers and and apply indirection them
- use the new operators (`&`, `*`, `->`)
- describe aliasing
- use `scanf` 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

> **!** In class we work with the key ideas of the module. We sometimes skip a few details. Review the official CS136 slides to ensure you see all the material.