Introduction to Pointers in C

Readings: CP:AMA 11, 17.7

Address operator

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

The ***address operator*** (&) produces the starting address of where the value of an identifier is stored in memory.

```
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: 0x68a9e0
```

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 declaration syntax, not the identifier itself.

```
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 *`).

The **value** of a pointer is an **address**.

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

printf("value of i     (i) = %d\n",  i);
printf("address of i  (&i) = %p\n", &i);
printf("value of p     (p) = %p\n",  p);
```

```
value of i     (i) = 42
address of i  (&i) = 0xf020
value of p     (p) = 0xf020
```

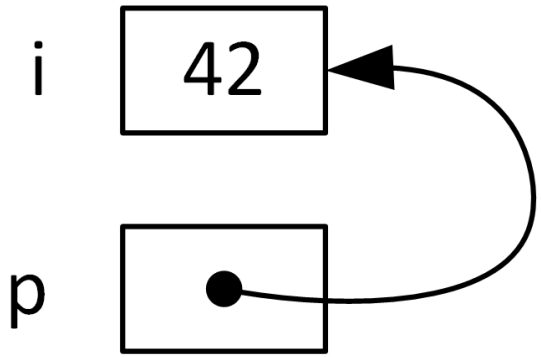
To make working with pointers easier in these notes, we often use shorter, simplified ("fake") addresses.

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

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

value
42
0xf020

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



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".

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

printf("value of p                  (p) = %p\n",  p);
printf("value of what p points at (*p) = %d\n", *p);
```

```
value of p                  (p) = 0xf020
value of what p points at (*p) = 42
```

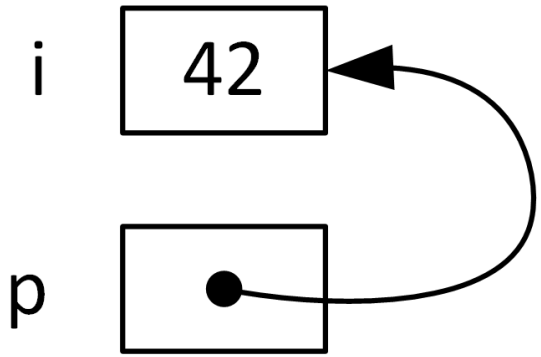
The value of ∗&i is simply the value of i.

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".



`*p` ⇒ 42

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

- as the *multiplication operator* between expressions

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

- in pointer *declarations* and pointer *types*

```
int *pi = &i;
sizeof(int *)
```

- as the *indirection operator* for pointers

```
j = *pi;
*pi = 5;
```

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

C mostly ignores white space, so these are equivalent

```
int *pi = &i;      // style A
int * pi = &i;     // style B
int* pi = &i;      // 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 `pi` is an "`int *`".

However, *in the declaration* the `*` "belongs" to the `pi`, not the `int`, and so style A is used in this course and in CP:AMA.

This is clear with multiple declarations: (not encouraged)

```
int i = 42, j = 23;
int *pi = &i, *pj = &j; // VALID
int* pi = &i, pj = &j;  // INVALID: pj is not a pointer
```

Pointers to pointers

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

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

This is actually a **type error**:

- `p` is declared as `(int *)`, a pointer to an `int`, but
- the type of `&p` is `(int **)`, a pointer to a pointer to an `int`.

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

```
int i = 42;
int *pi = &i;       // pointer pi points at i
int **ppi = &pi;    // pointer ppi points at pi
```

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

`(**ppi * **ppi)` 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", or is "invalid". Some functions return a `NULL` pointer to indicate an error. `NULL` is essentially "zero", but it is good practice to use `NULL` in code to improve communication.

If you *dereference* a `NULL` pointer, your program will likely crash.

Most functions should *require* that pointer parameters are not `NULL`.

```
assert (p != NULL);
assert (p);  // <-- because NULL is not true...
             // this is equivalent and common
```

`NULL` is defined in the `stdlib` module (and several others).

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.

A function pointer **declaration** includes the *return type* and all of the *parameter types*, which makes them a little messy.

```
int add1(int i) { return i + 1; }

int main(void) {
  int (*fp)(int) = add1;                    // OR = &add1;
  printf("add1(3) = %d\n", fp(3));
}
```

```
add1(3) = 4
```

The syntax to declare a function pointer with name `fpname` is:

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

examples: function pointer declarations

```
int functionA(int i) {...}
int (*fpA)(int) = functionA;

char functionB(int i, int j) {...}
char (*fpB)(int, int) = functionB;

int functionC(int *ptr, int i) {...}
int (*fpC)(int *, int) = functionC;

int *functionD(int *ptr, int i) {...}
int *(*fpD)(int *, int) = functionD;

struct posn functionE(struct posn *p, int i) {...}
struct posn (*fpE)(struct posn *, int) = functionE;
```

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

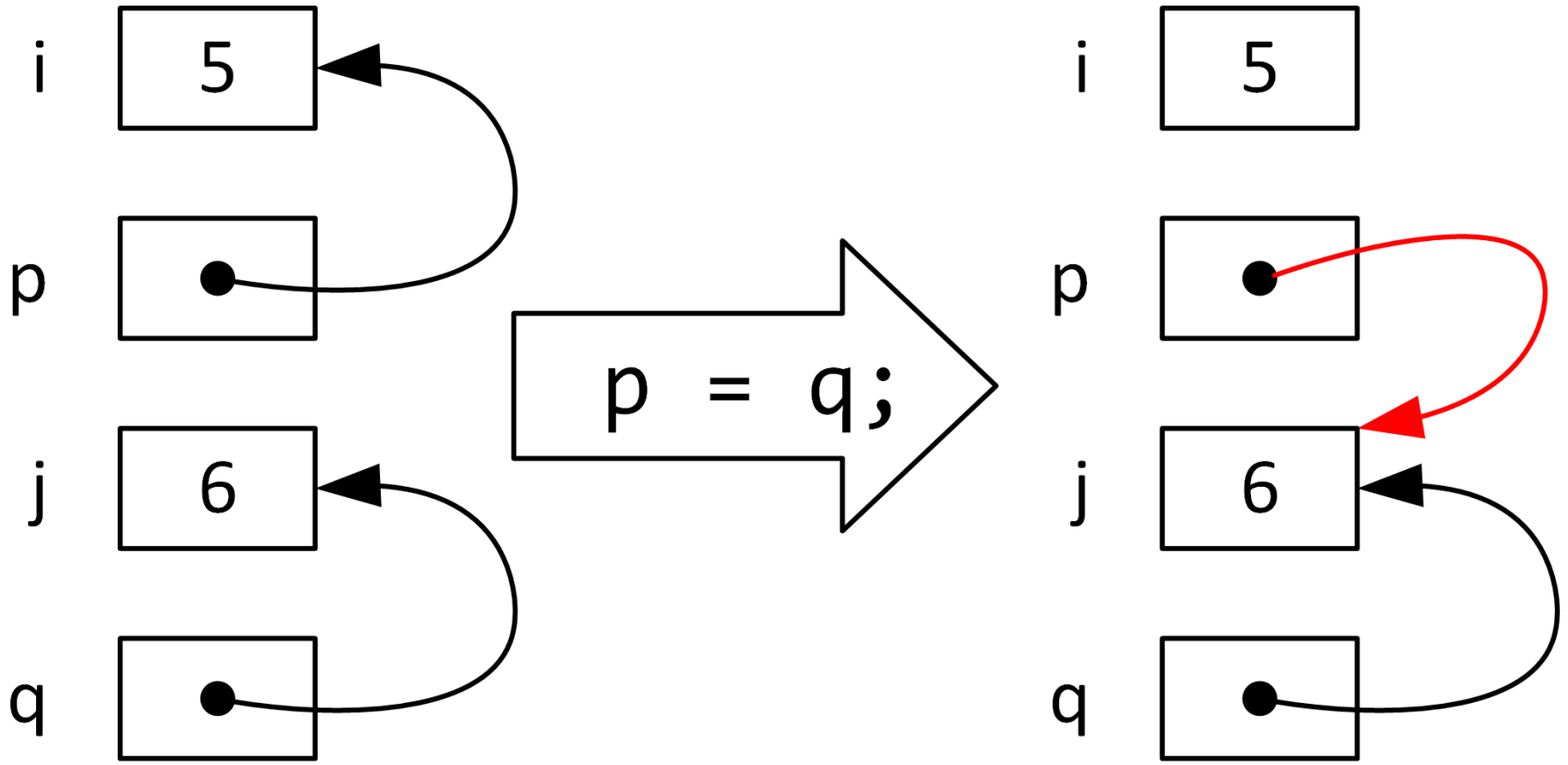
Pointer assignment

Consider the following code

```
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`**.


i
5
p
j
6
q
p = q;
i
5
p
j
6
q

Using the same initial values,

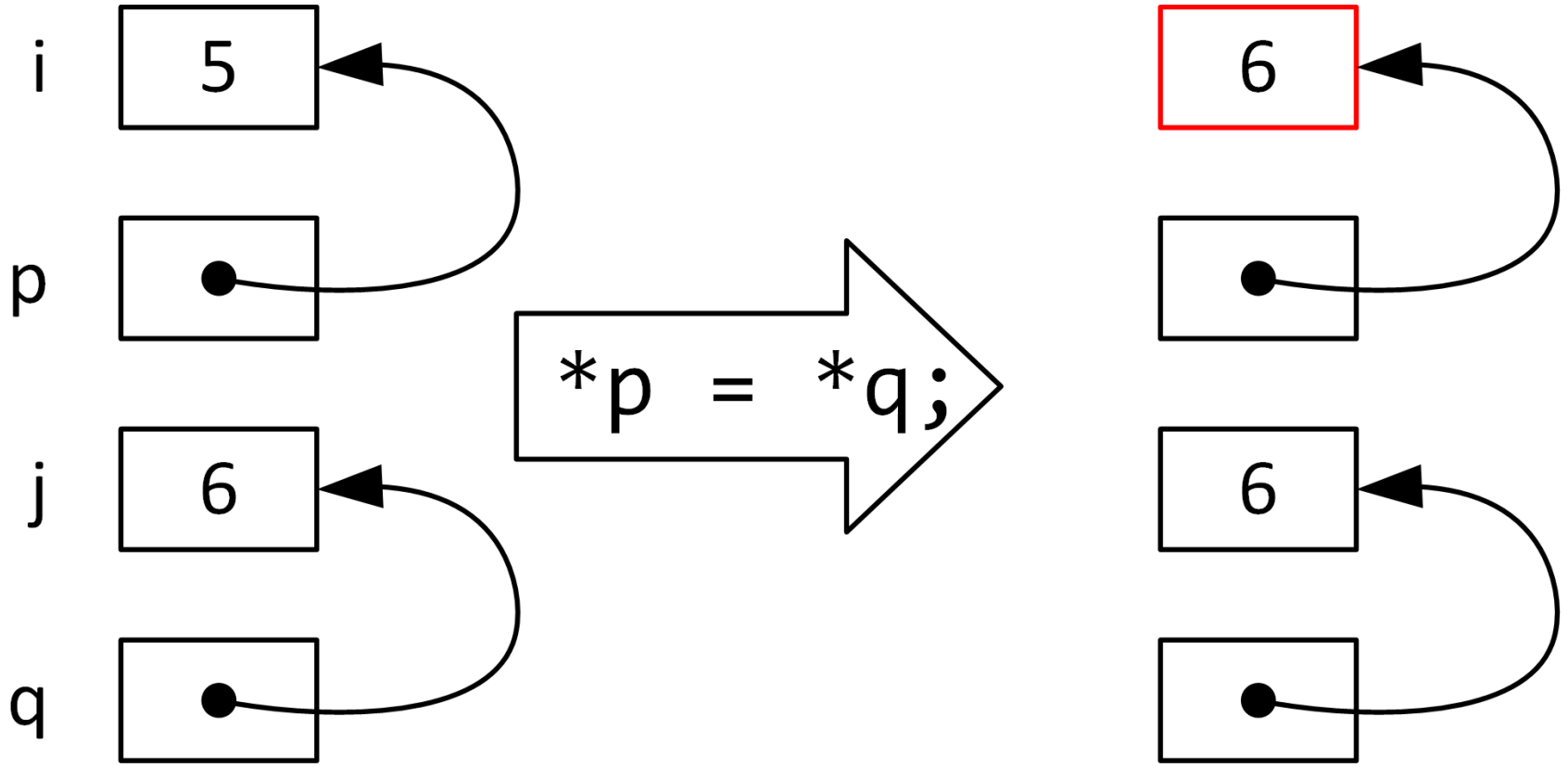
```
int i = 5;
int j = 6;

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

the statement

```
*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.


i
5
p
j
6
q
*p = *q;
6
6

example: aliasing

```
int i = 2;

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

printf("i = %d\n", i);
*p1 = 7;                    // i changes...
printf("i = %d\n", i);
*p2 = 100;                  // without being used directly
printf("i = %d\n", i);
```

```
i = 2
i = 7
i = 100
```

Mutation & parameters

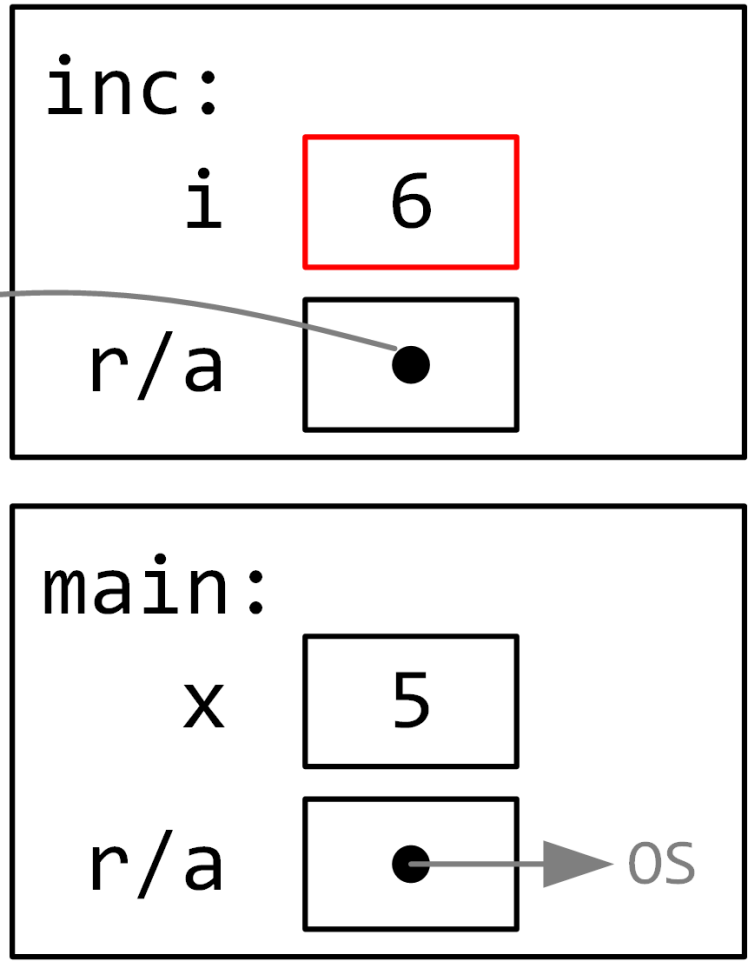
Consider the following C program:

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

int main(void) {
  int x = 5;
  inc(x);
  printf("x = %d\n", 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.

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

int main(void) {
  int x = 5;
  inc(x);
  printf("x = %d\n", 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? (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 considered “pass by value” because we pass the **value** of the address.

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

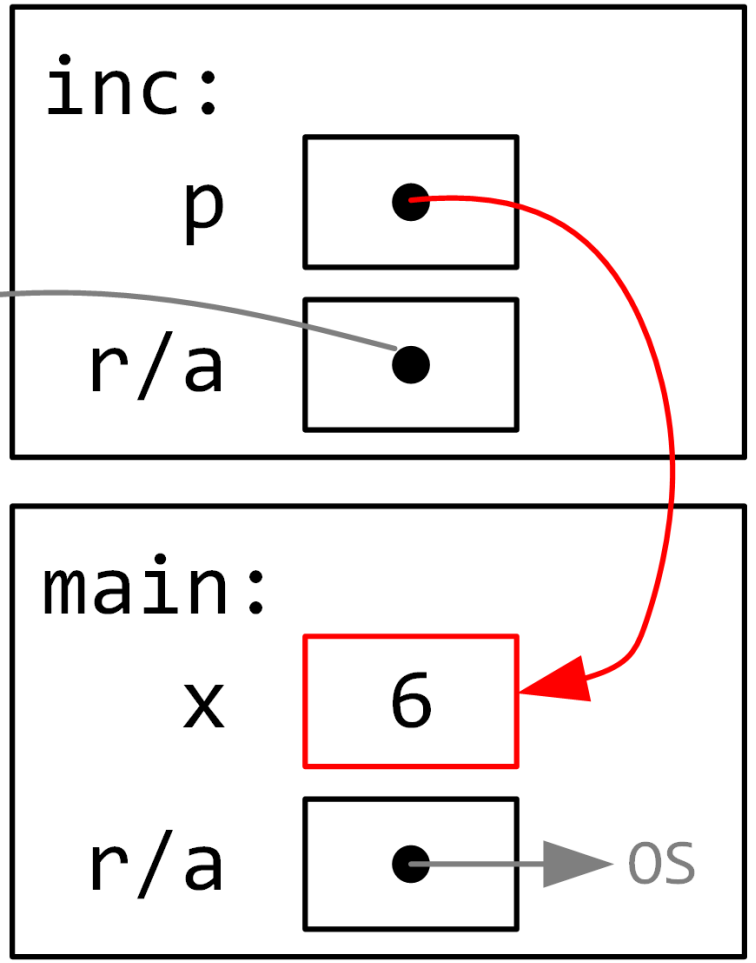
It is also common to say “pass a pointer to x”.

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

int main(void) {
  int x = 5;
  inc(&x);                    // note the &
  printf("x = %d\n", x);      // NOW it's 6
}

x = 6
```

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

The corresponding parameter type is an int pointer (int *).

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

int main(void) {
  int x = 5;
  inc(&x);
  printf("x = %d\n", x);
}
```



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

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

The parentheses are necessary.

Because of the order of operations, the ++ would have incremented the pointer `p`, not what it points at (`*p`).

C is a minefield of these kinds of issues: the best strategy is to use straightforward code.

example: mutation side effects

```
//  effects: swaps the contents of *x and *y
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main(void) {
  int x = 3;
  int y = 4;
  printf("x = %d, y = %d\n", x, y);
  swap(&x, &y);                          // Note the &
  printf("x = %d, y = %d\n", x, y);
}
```

```
x = 3, y = 4
x = 4, y = 3
```

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*.

Mutable structures can be modified by a function.

```
(struct mposn (x y) #:mutable #:transparent)

(define (swap! mp)
  (define oldx (mposn-x mp))
  (set-mposn-x! mp (mposn-y mp))
  (set-mposn-y! mp oldx))

(define my-posn (mposn 3 4))
(swap! my-posn)
my-posn ;; => (mposn 4 3)
```

Returning more than one value

Like Racket, 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 the variables.

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;
}

int main(void) {

  int q;       // this is a rare example where
  int r;       // no initialization is necessary

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

  assert(q == 2 && r == 3);
}
```

This “multiple return” technique is 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).

```
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.

example: pointer return types

The return type of a function can also be an address (pointer).

```
int *ptr_to_max(int *a, int *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);
}
```

Returning addresses become more useful in Section 10.

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

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

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.

```
struct bigstruct {
  int a; int b; int c; ... 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.

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

```
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};

  assert(sqr_dist(&p1, &p2) == 25);     // note the &
}
```

```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;
}
```

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 a awkward. Because it frequently occurs there is an *additional* selection operator for working with pointers to structures.

The ***arrow selection operator*** (`->`) combines the indirection and the selection operators.

`ptr->field` is equivalent to `(*ptr).field`

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

```
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.

```
// scale(p, f) scales the posn *p by f
// requires: p is not null
// effects:  changes the field values of p

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

If a function has a pointer parameter, the documentation should clearly communicate whether or not the function can mutate the pointer's destination ("what the pointer points at").

While all side effects should be properly documented, documenting the absence of a side effect may be awkward.

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
}
```

The `const` should be placed first, before the type (see the next slide).

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.

```
int *p;                 // p can change, can point at any int

const int *p;           // p can change,
                        // but must point at a const int

int * const p = &i;     // p must always point at i,
                        // but i can change

const int * const p = &i;  // p is constant and i is constant
```

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".

Note: the following are equivalent and a matter of style.

```
const int i = 42;
int const i = 42;
```

const parameters

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

```
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*.

```
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 that **the copy** will not be mutated.

For simple parameters, `const` is meaningless in a function **declaration**.

The caller (client) does not need to know if the function will mutate the **copy** of the argument value.

```
int my_function(int x);                    // DECLARATION
                                           // (no const)

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

It is good style to use `const`ant parameters in **definitions** to improve communication.

In the notes, we often omit `const` in parameters to save space.

Opaque structures in C

C supports **opaque structures** through ***incomplete declarations***, where a structure is *declared* without any fields. With incomplete declarations, only *pointers* to the structure can be defined.

```
struct posn;                    // INCOMPLETE DECLARATION

struct posn my_posn;            // INVALID
struct posn *posn_ptr;          // VALID
```

If a module only provides an *incomplete declaration* in the **interface**, the client can not directly access any of the fields.

The module must provide a function to *create* an instance of the structure. This will be explored more in Section 10.

Goals of this Section

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

- declare and de-reference pointers
- use the two new operators (& $*$)
- use function pointers
- describe aliasing
- use pointers to structures as parameters and explain why parameters are often pointers to structures