Introduction to Pointers in C

Optional Textbook Readings: CP:AMA 11, 17.7

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

We are learning about pointers in this course *early* (before we really “need” them) so you are more comfortable with them when they are required later.

In this section we mostly focus on *syntax* and simple applications. Later we will have more practical applications:

- understanding arrays (Section 07)
- working with dynamic memory (Section 10)
- working with linked data structures (*e.g.*, lists and trees) (Section 11)

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).

```
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: 0x725520
```

The `printf` format specifier to display an address (in hex) is `"%p"`.

Pointers

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

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

```
int i;          // i is an integer [uninitialized]
int *p;         // p is a pointer to an integer
                // [uninitialized]
```

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

The pointer variable `p` above can store the address of an `int`.

```
p = &i;         // p now stores the address of i
                // or "p points at i"
```

Pointer types

For *each type* there is a corresponding *pointer type*.

```
int i = 42;
char c = 'z';
struct posn p1 = {3, 4};

int *pi = &i;              // pi points at i
char *pc = &c;             // pc points at c
struct posn *pp = &p1;     // pp points at p1
```

The *type* of `pi` is an *"int pointer"* which is written as "`int *`".

The *type* of `pc` is a *"char pointer"* or "`char *`".

The *type* of `pp` is a *"struct posn pointer"* or "`struct posn *`".

Pointer initialization

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

Remember, that the following definition:

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

is comparable to the following definition and assignment:

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

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`.

C mostly ignores whitespace, so these are equivalent

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

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

This is clear with multiple definitions: (not encouraged)

```
int i = 42, j = 23;
int *p1 = &i, *p2 = &j; // VALID
int * p1 = &i, p2 = &j;  // INVALID: p2 is not a pointer
```

Pointers to pointers

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

```
int *p = &p;        // p points at p ?!?
                    // INVALID [type error]
```

This is actually a **type error**:

The type of `p` is `(int *)`, a pointer to an `int`.

`p` can only point at an `int`, but `p` itself is **not** an `int`.

What if we wanted a variable that points at `p`?

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
```

The type of `p2` is "`int **`" or a "pointer to a pointer to an `int`".

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 a `void` pointer (itself).

Pointer values

Remember, pointers are variables, and variables store values.

A pointer is only “special” because the **value** it stores is an **address**.

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

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

```
i => 42
&i => 0xf020
p => 0xf020
&p => 0xf024
```

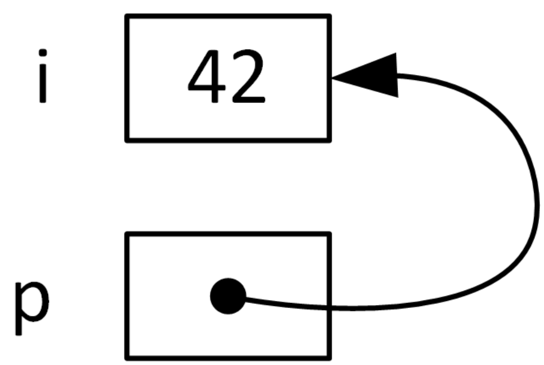
Because a pointer is a variable, it *also* has an address itself.

```
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").



The 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”.

```
int *p;               // BAD (uninitialized)

int *p = NULL;        // GOOD
```

NULL is false

`NULL` is considered “false” when used in a Boolean context.

In C, **false** is defined to be zero *or* `NULL`.

The following are equivalent:

```
if (p) ...

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

Both are considered “good style”.

Pointer assignment

As with any variable, the value of a pointer can be changed (mutated) with the assignment operator.

```
int *p = NULL;   // p is initialized to NULL

p = &j;          // p now points at j
p = NULL;        // p now points at nothing
p = &i;          // p now points at i
```

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".

```
i = 42;
p = &i;

trace_int(*p);
```

```
*p => 42
```

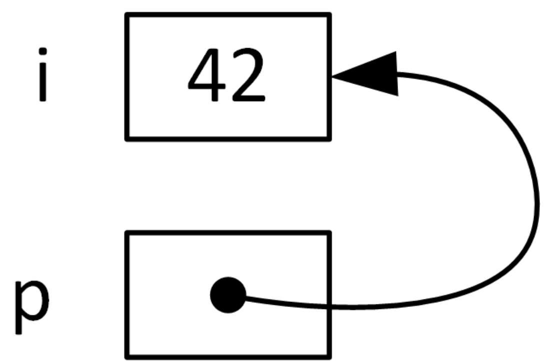
The value of `*&i` or `*&*&*&i` is simply the value of `i`.

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".



$*p \Rightarrow 42$

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

```
p = NULL;
trace_int(*p);        // crash!
```

Multiple uses of *

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;
sizeof(int *)
```

- as the *indirection operator* for pointers

```
trace_int(*p);
j = *p;
```

Dereferencing pointers to structures

Unfortunately, the structure operator (`.`) has higher precedence than the indirection operator (`*`).

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.

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

example: indirection selection operator

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

int main(void) {
  struct posn my_posn = {3, 4};
  struct posn *ptr = &my_posn;

  trace_int((*ptr).x)              // awkward
  trace_int(ptr->x);               // much better

}
```

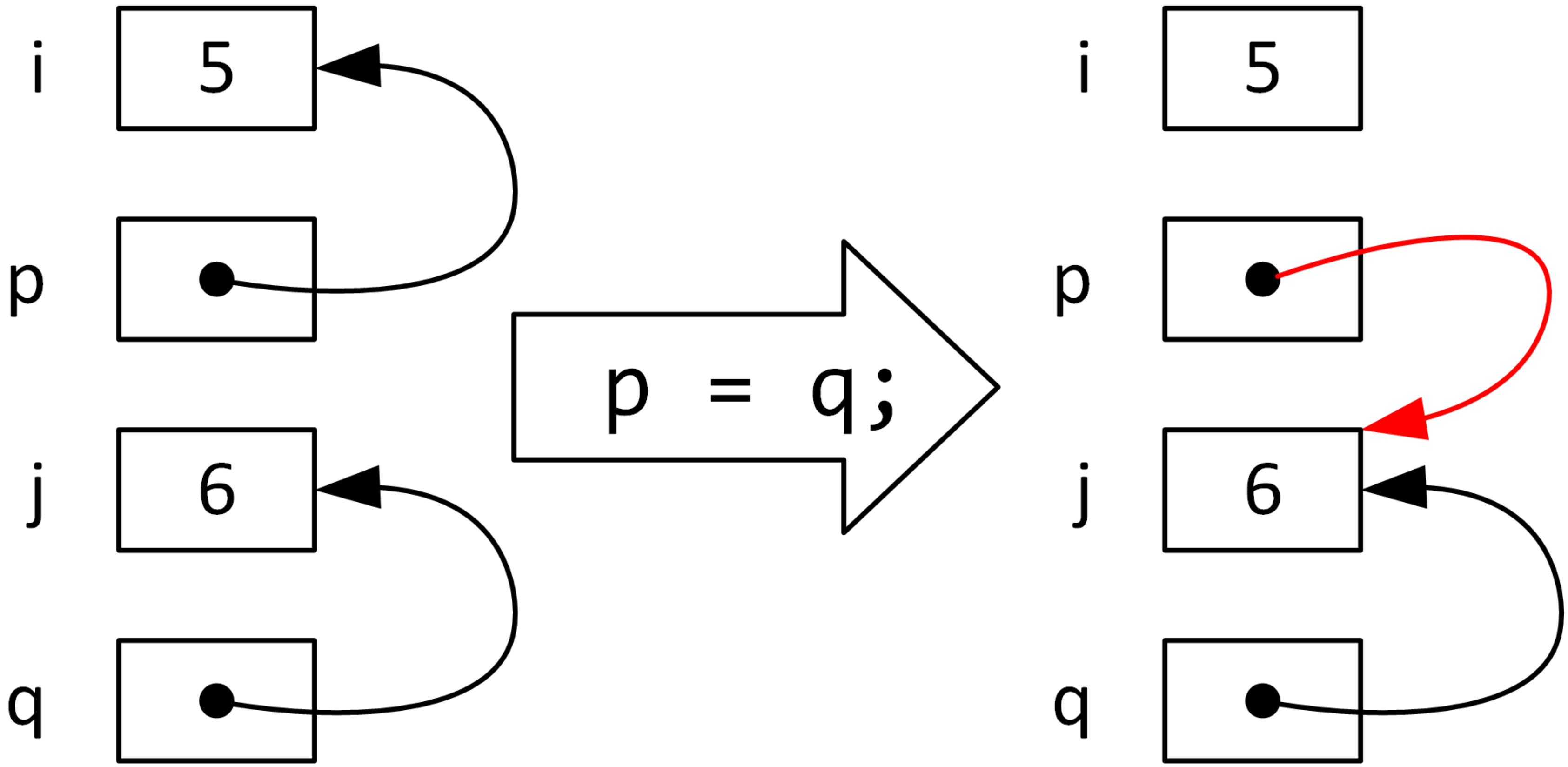
Pointer assignment

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,

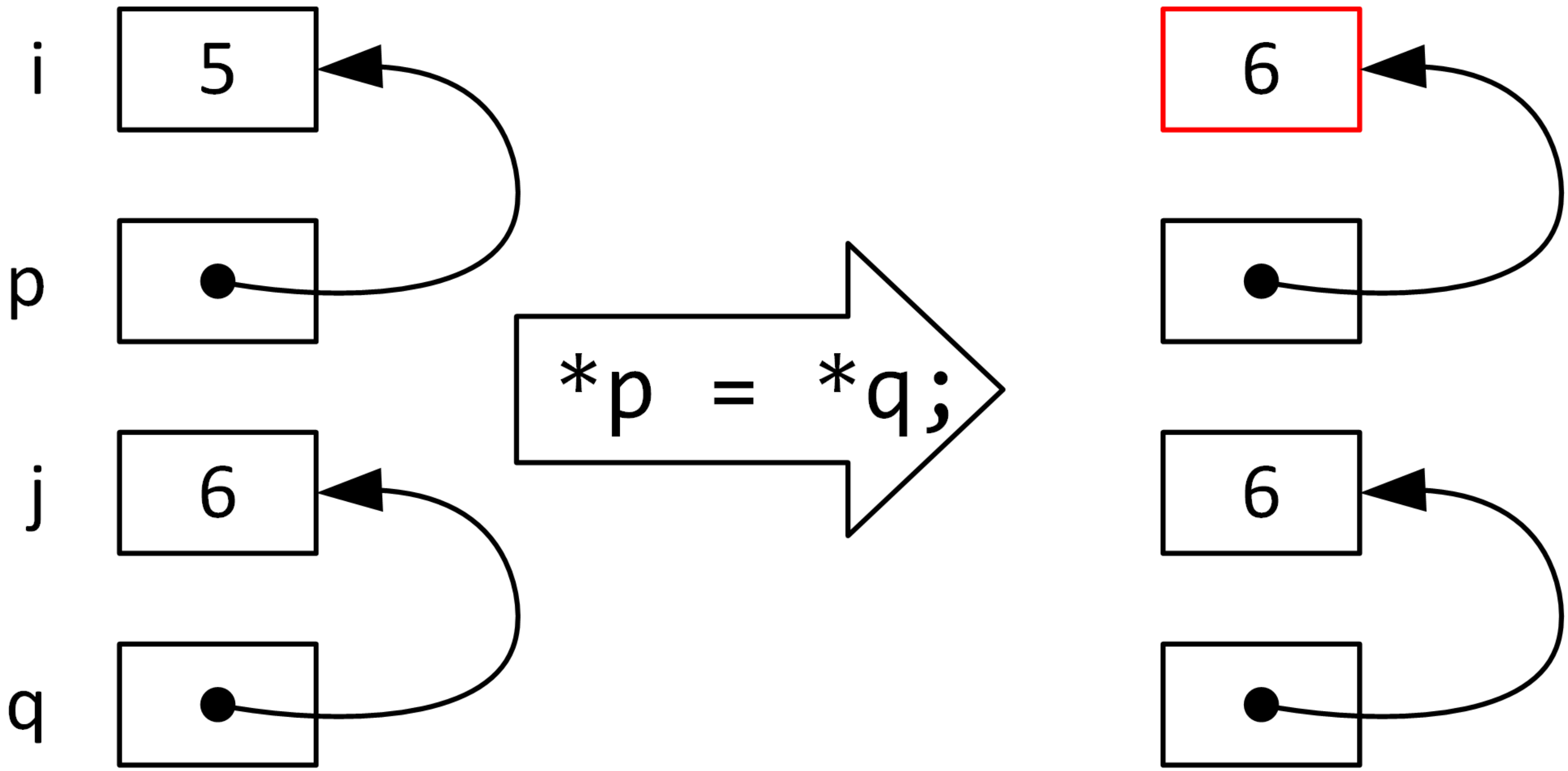
```
int i = 5;
int j = 6;

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
*p = *q;
j
6
q
6
6

example: aliasing

```
int 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);
```

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

Mutation & parameters

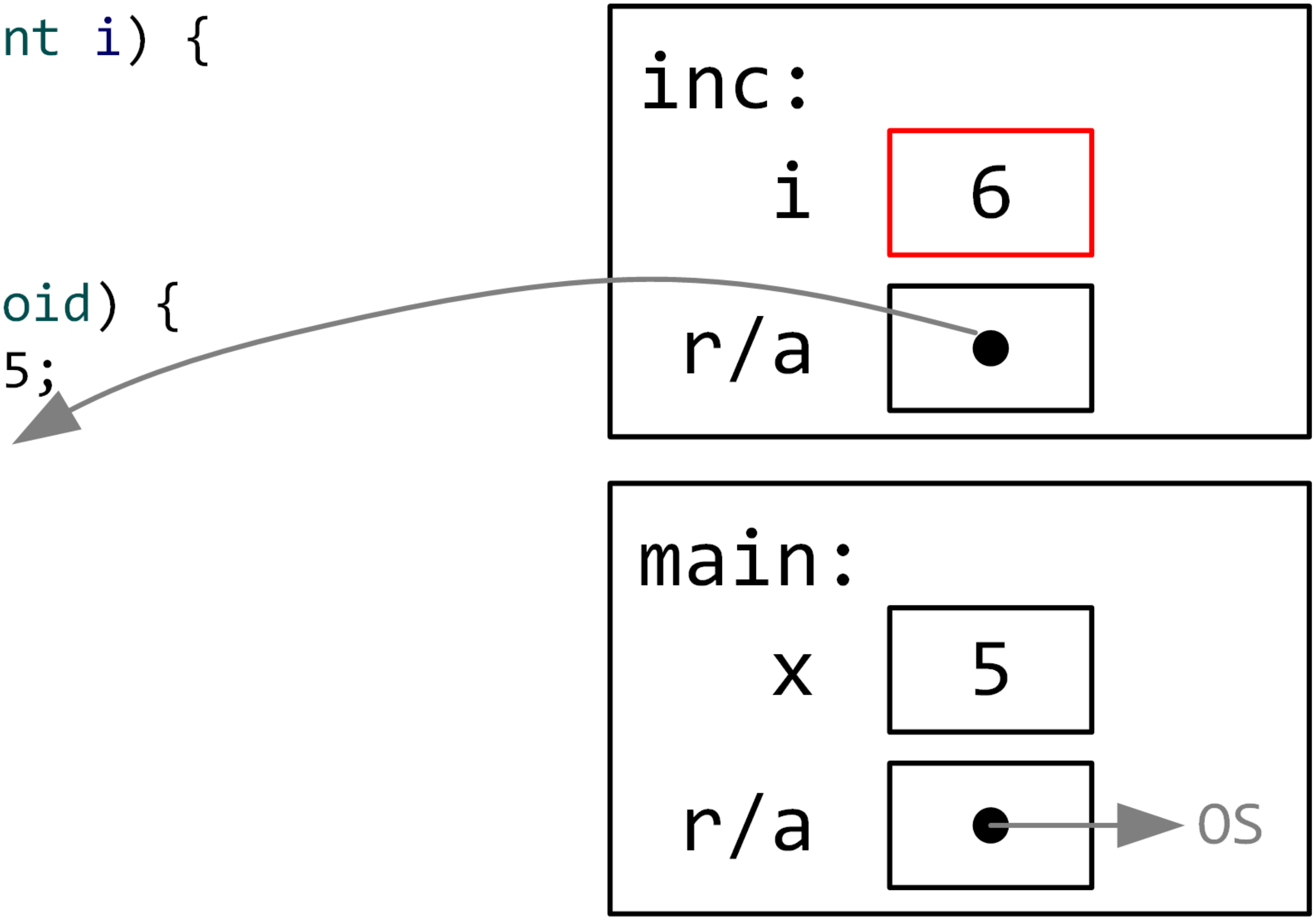
Consider the following C program:

```
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 its own copy of the argument (in the stack frame) without changing the original variable.

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

int main(void) {
  int x = 5;
  inc(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 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”.

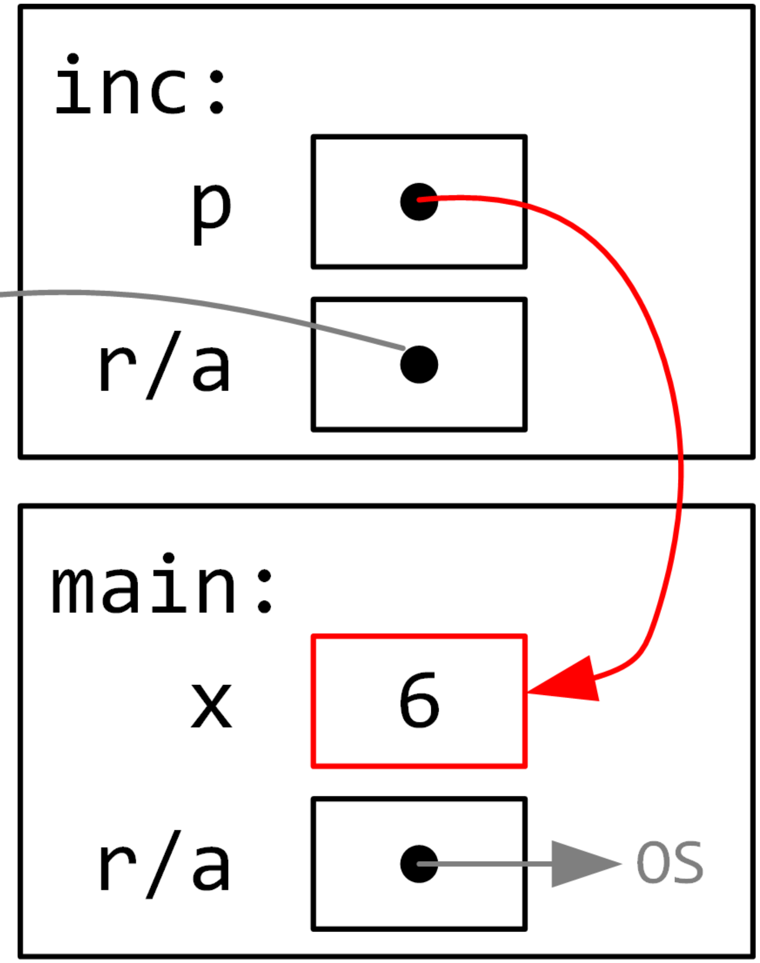
```
void inc(int *p) {
  *p += 1;
}

int main(void) {
  int x = 5;
  trace_int(x);
  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 *`).

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

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



Most pointer parameters should be **required** to be valid (*e.g.*, non-NULL). In the slides it is often omitted to save space.

```
// inc(p) increments the value of *p
// effects:  modifies *p
// requires: p is a valid pointer

void inc(int *p) {
  assert(p);            // or assert(p != NULL);
  *p += 1;
}
```

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

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

example: mutation side effects

```
//  effects: modifies *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;
  trace_int(a); trace_int(b);
  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**

```
//  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*.

Returning an address

A function may `return` an address.

```
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 to a variable within its stack frame.

As soon as the function `return`s, the stack frame "disappears", and all memory within the frame is considered **invalid**.

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

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

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.

```
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`, multiple format specifiers can be used 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.

scanf return value

The **return value** of `scanf` is an `int`, 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"`).

In `Seashell`, a `Ctrl-D` ("Control D") keyboard sequence (or the `[EOF]` button) sends an `EOF`.

In our environment, `EOF` is defined as `-1`, but it is much better style to use the constant `EOF` instead of `-1`.

Invalid input

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

(if you are following our advice to read one value per `scanf`).

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

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

Multiple side effects

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

Earlier we encouraged you to only have *one side effect per statement*. Unfortunately, when using `scanf` it is impossible.

example: reading integers

This function reads in `int`s from input (until `EOF` or an unsuccessful read occurs) and returns their sum.

```
int read_sum(void) {
  int sum = 0;
  int n = 0;
  while (scanf("%d", &n) == 1) {
    sum += n;
  }
  return sum;
}
```

example: read_int

We can now see how the `read_int` function is implemented.

```
const int READ_INT_FAIL = INT_MIN;

int read_int(void) {
  int i = 0;
  int result = scanf("%d", &i);
  if (result == 1) {
    return i;
  }
  return READ_INT_FAIL;
}
```

On assignments and exams, you will now be using `scanf` instead of `read_int`.

Whitespace

When reading an `int` with `scanf` (`"%d"`) C **ignores any whitespace** (spaces and newlines) that appears before the next `int`.

When reading in a `char`, you *may* or *may not* want to ignore whitespace: it depends on your application.

```
// 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 leading whitespace is ignored.

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`).

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.

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
```

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).

```
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.

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 d;
  ...
  int y;
  int z;
};
```

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

```
// sqr_dist(p1, p2) calculates the square of
//   the distance between p1 and p2

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:  modifies p

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

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

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

// scale(p, f) scales the posn *p by f
// effects:  modifies *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 the *start* of a pointer definition prevents the pointer's **destination** (the variable it points at) from being mutated through the pointer.

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

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

Remember, a pointer definition that *begins* with `const` prevents the pointer's **destination** from being mutated *via the pointer*.

```
int i = 5;
const int *p = &i;

*p = 10;                    // INVALID
i = 10;                     // still valid
```

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

```
p = &j;         // valid
*p = 10;        // INVALID
```

A handy tip is to read the definition **backwards**:

`const int *p` $\Rightarrow$ *"p is a pointer to an* `int` *that is* `const`*ant"*.

See the following advanced slide for more details.

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

```
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".

```
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 cannot) 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*.

Remember that parameters behave the same as local variables and are stored in the stack frame.

```
int my_function(const int x) {
  const int y = 13;
  x = 0;                 // INVALID
  y = 0;                 // 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.

Minimizing mutative side effects

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

Global mutable variables are **strongly discouraged** and considered “poor 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. **“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 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

```
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
```

In the previous example:

```
fp = my_add;
```

We could have also used:

```
fp = &my_add;
```

Because functions are not “first class values” C cannot get the “value” of a function. Instead it uses the *address* of the function.

Since both are equivalent, in practice `my_add` is used more often than `&my_add` because it is “cleaner”, even though `&my_add` is more correct.

example: passing a function to a function

```
// io_apply(f) reads in each int [n] from input
//   and prints out f(n)
// effects: produces output
//          reads input
void io_apply(int (*f)(int)) {
  int n = 0;
  while (scanf("%d", &n) == 1) {
    printf("%d\n", f(n));
  }
}

int sqr(int i) {
  return i * i;
}

int main(void) {
  io_apply(sqr);
}
```

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