

Introduction to Pointers

CP:AMA Readings: 11, 17.7

CHTP Readings: 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 that 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 4)
- working with dynamic memory (Section 8)
- working with linked data structures (*e.g.*, lists and trees)
(Section 9)

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 main(void) {  
    int g = 42;  
    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
float p2;        // [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"
```

~~p = 0x5559~~

Pointer types

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

```
int i = 42;  
char c = 'z';
```

```
int * pi = &i;           // pi points at i  
char * pc = &c;         // pc points at c
```

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

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

float pointer float ✗

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 = &i;
```

```
// q is defined [uninitialized]
```

```
// 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 the textbooks CP:AMA and CHTP.

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]
```

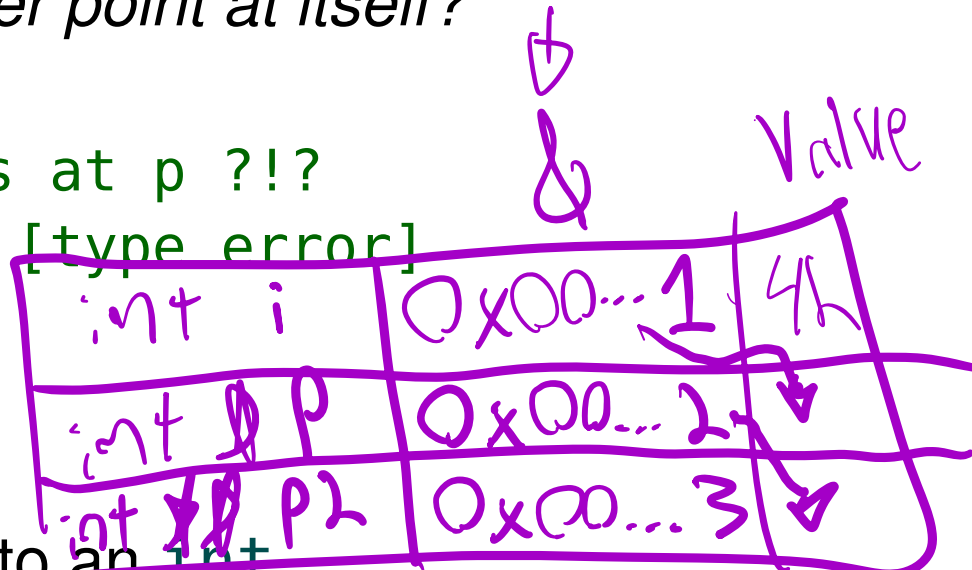


This is actually a **type error**:

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**?



int i	0x00...1	4h
int * p	0x00...2	↓
int ** p2	0x00...3	↓

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

```
int i = 42;
```

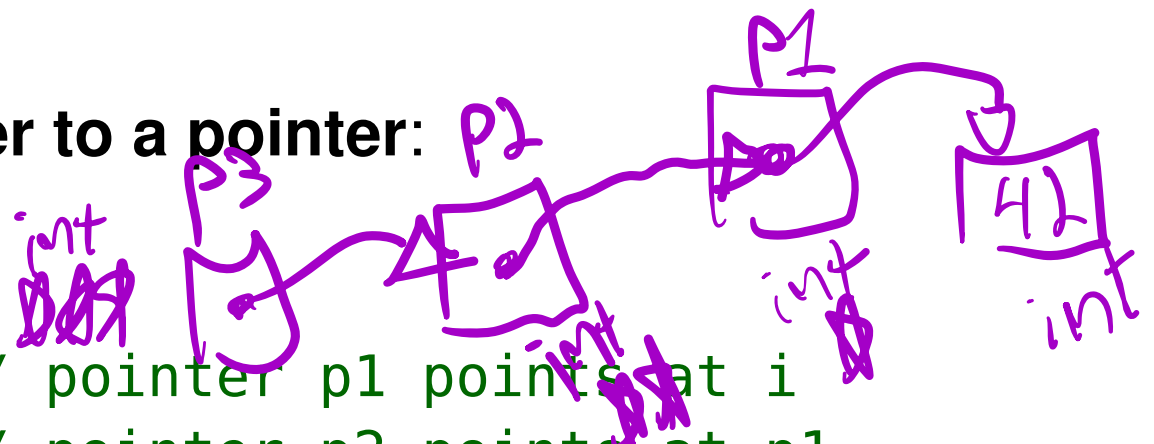
```
int * p1 = &i;
```

```
int ** p2 = &p1;
```

```
int *** p3 = &p2;
```

```
// pointer p1 points at i
```

```
// 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;
```

i => 42
&i => 0xF020
p => 0xF020
&p => 0xF024

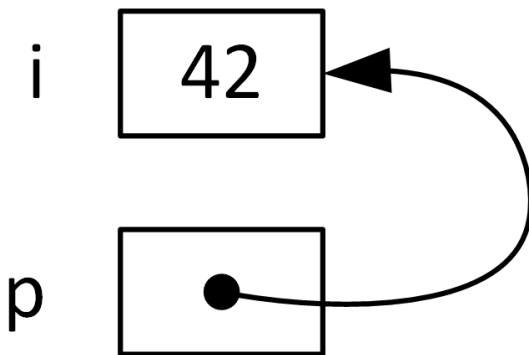
Because a pointer is a variable, it *also* has an address itself.

The address 0xF020 represented as a decimal number is 61472, but the standard convention is to represent addresses in hex.

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

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 two are equivalent:

```
if (p) ...
```

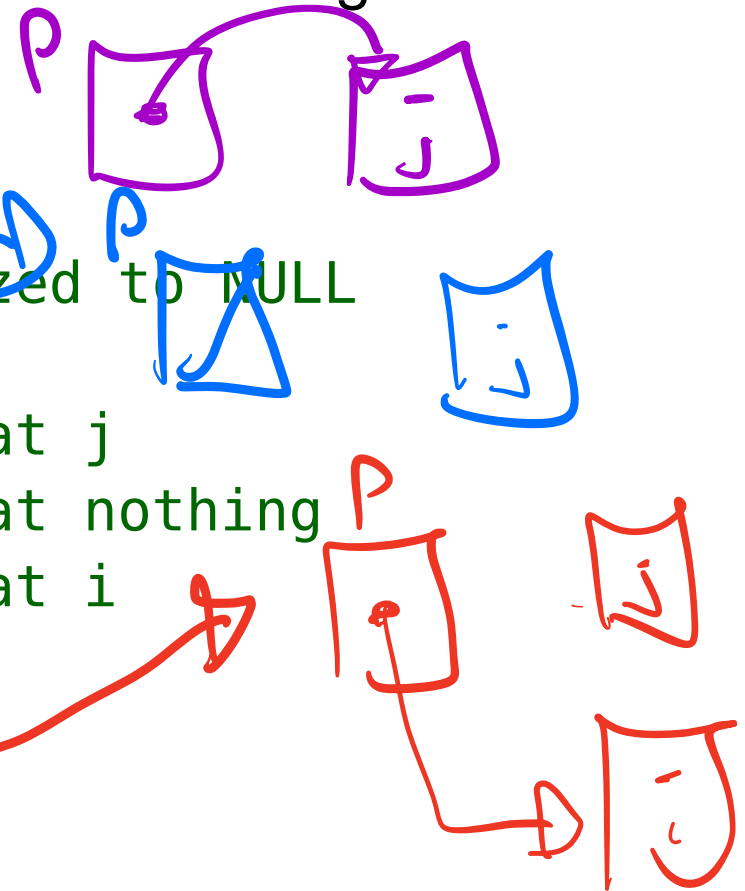
```
if (p != NULL) ...
```

$p = \text{NULL}$
 $\text{if} (!p)$
 $\text{if} (!\text{NULL})$

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.

For example, the sizeof a pointer is 64 bits (8 bytes) in a 64-bit environment.

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

sizeof(int *) \Rightarrow 8

sizeof(char *) \Rightarrow 8

sizeof(float *) \Rightarrow 8

32 bits \rightarrow 4 bytes
64 bits \rightarrow 8 bytes

Indirection operator

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

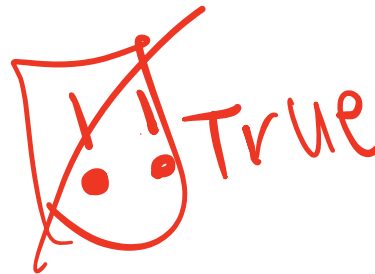
*p produces the **value** of what pointer p “points at”.

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



```
i => 42  
*p => 42
```

$\&i == p$
 $i == *p$



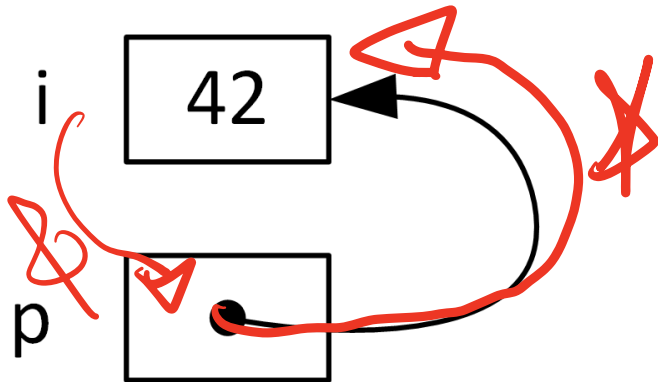
The value of ~~*&i~~ or *&*&*&i is simply the value of i.

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 is set to be a NULL pointer
```

```
p = NULL;
```

```
// Using *p will cause the program to crash
```

```
printf("The value pointed at by p: %d\n", *p);
```

add checks for NULL

Multiple uses of *

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

- as the *multiplication operator* between expressions

`k = i * i;`

→ binary operator (RHS)

- in pointer *definitions* and pointer *types*

`int * p = &i;`

→ definition (LHS)

`sizeof(int *)`

→ Type

- as the *indirection operator* for pointers

`i = *p;`

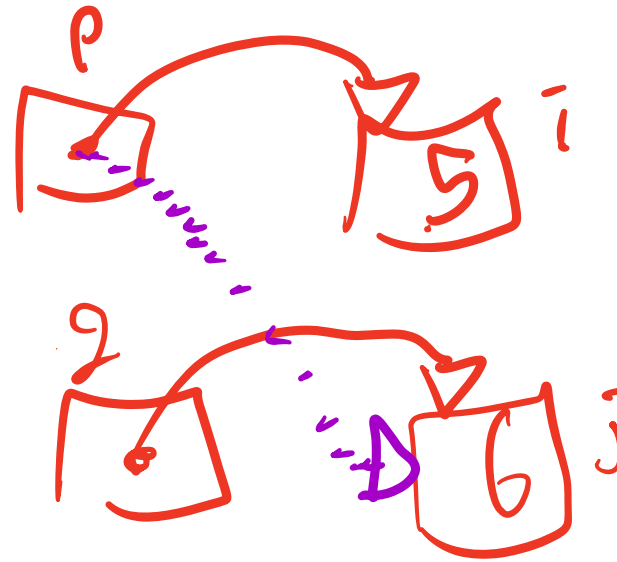
→ Unary operators (RHS)

pointers

Pointer assignment

Consider the following code

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

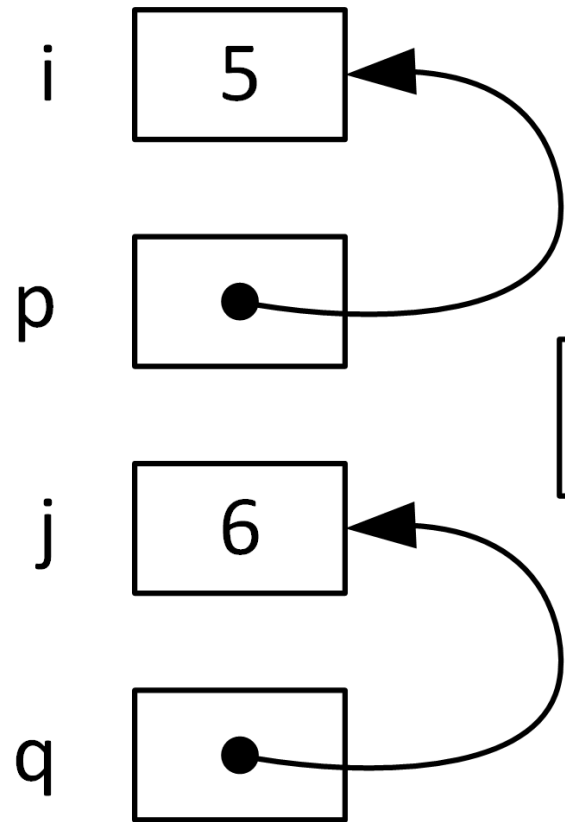


```
p = q;
```

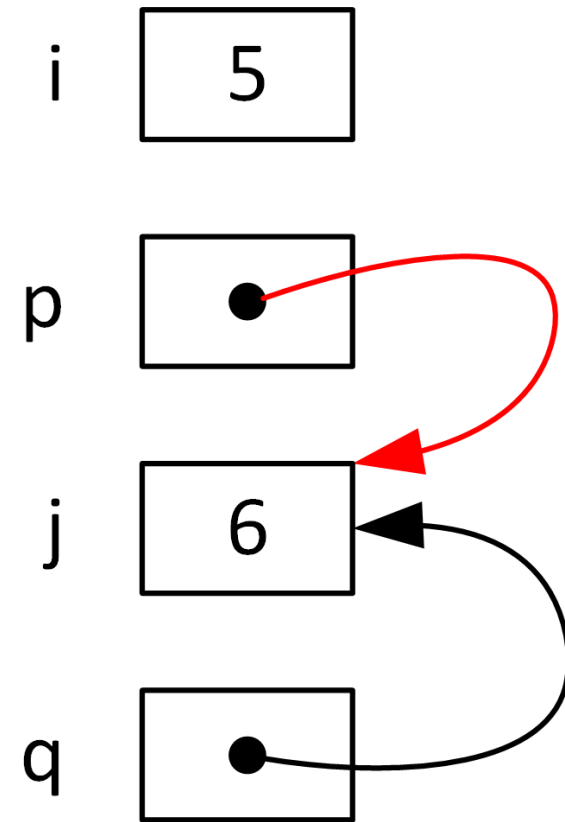
The statement `p = q;` is a **pointer assignment**. It means “change `p` to point at what `q` points at”. It changes the *value* of `p` to be the value of `q`. In this example, it assigns the *address* of `j` to `p`.

It does not change the value of `i`.

$p == \&j$
 ~~$p == j$~~



`p = q;`



Using the same initial values,

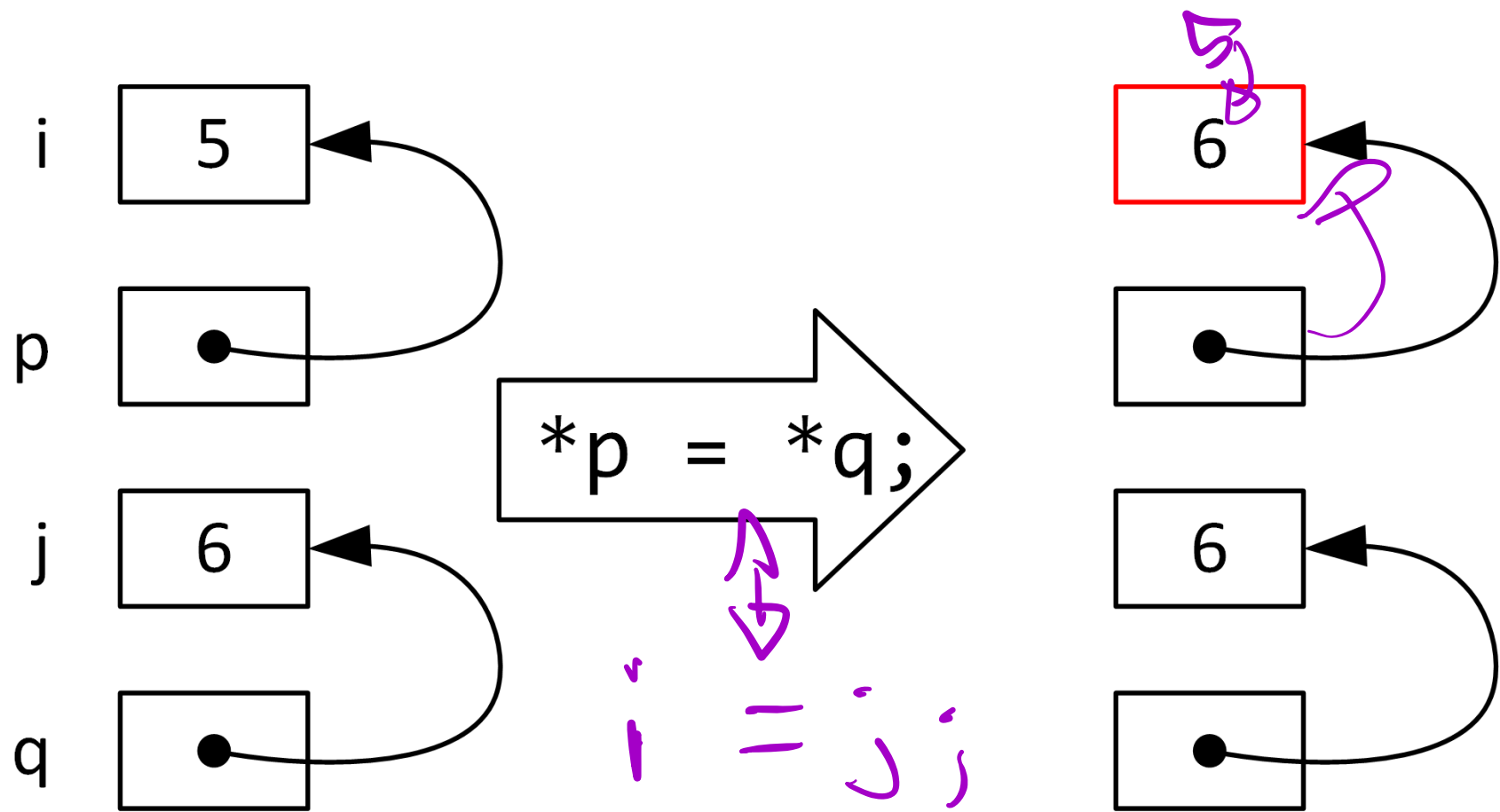
```
int i = 5;  
int j = 6;  
  
int * p = &i;  
int * q = &j;
```

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.



example: aliasing

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

```
printf("i => %d\n", i);
*p1 = 10;           // i changes...
printf("i => %d\n", i);

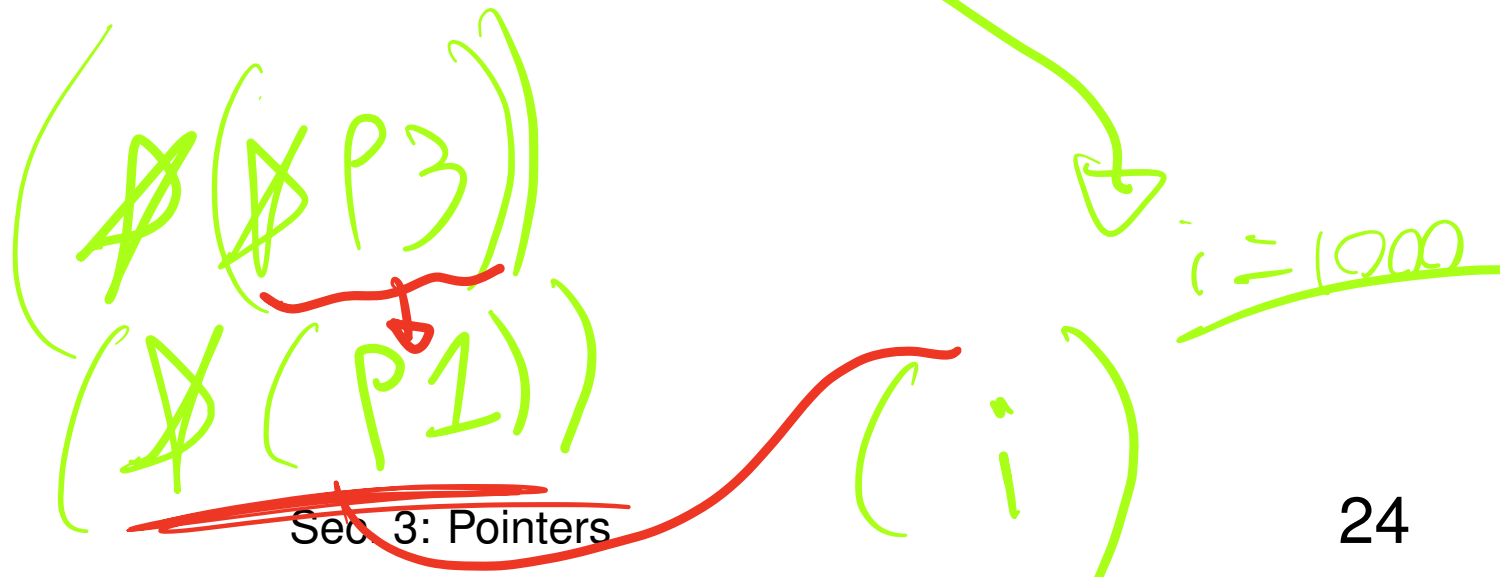
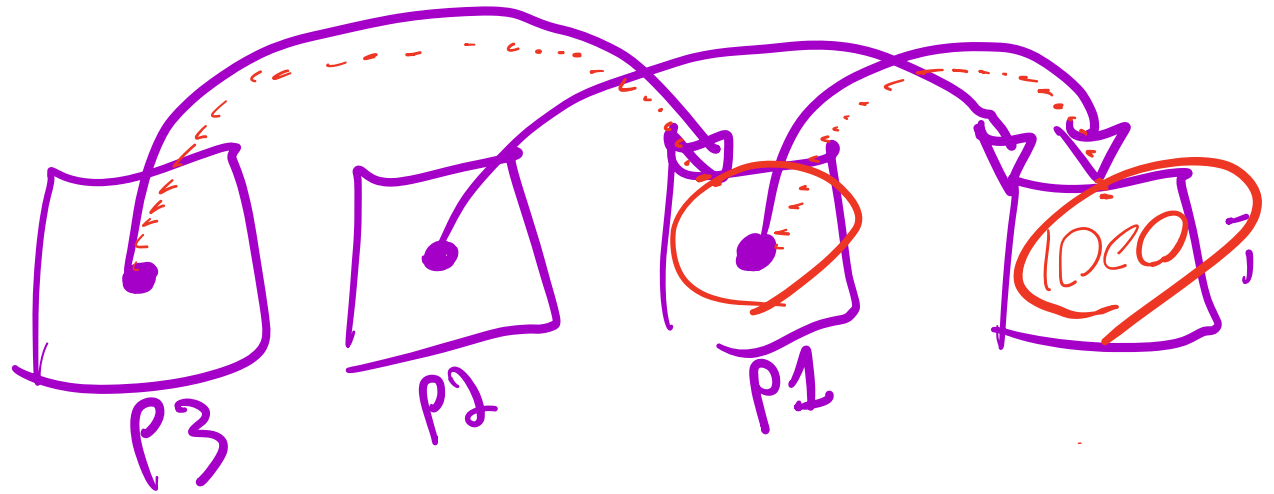

*p2 = 100;         // without being used directly


printf("i => %d\n", i);


**p3 = 1000;       // same as *(*p3)


printf("i => %d\n", 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);  
    printf("x => %d\n", x);  
}
```

// 5 or 6 ?

Pass by Value

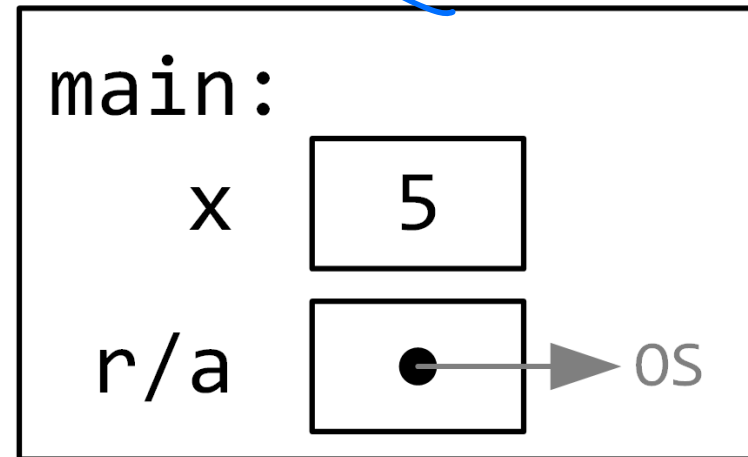
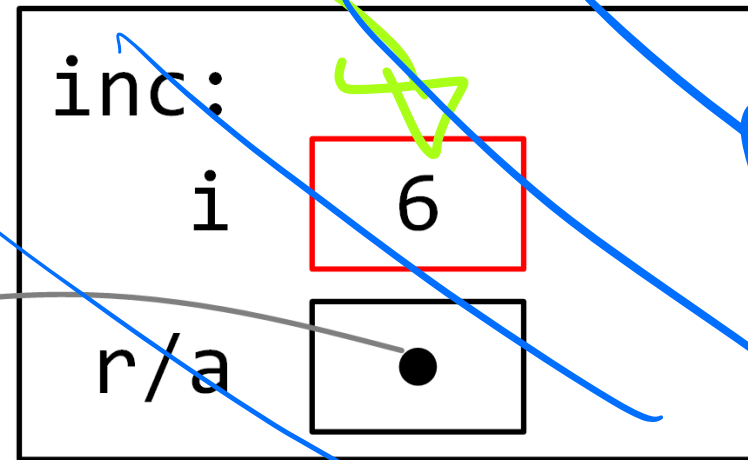
5

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);  
}
```

when it returns



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*.

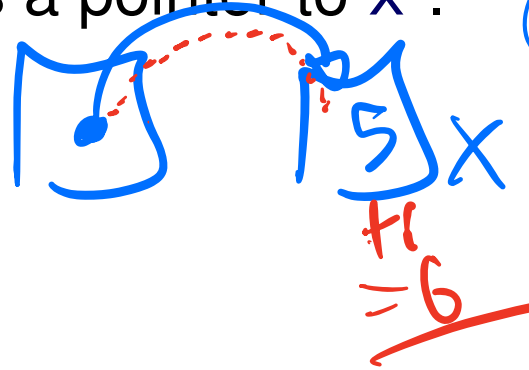
It is also common to say “pass a pointer to *x*”.

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

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

x => 5

x => 6



(pass by ref)

→ 5

// note the &

→ 6

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

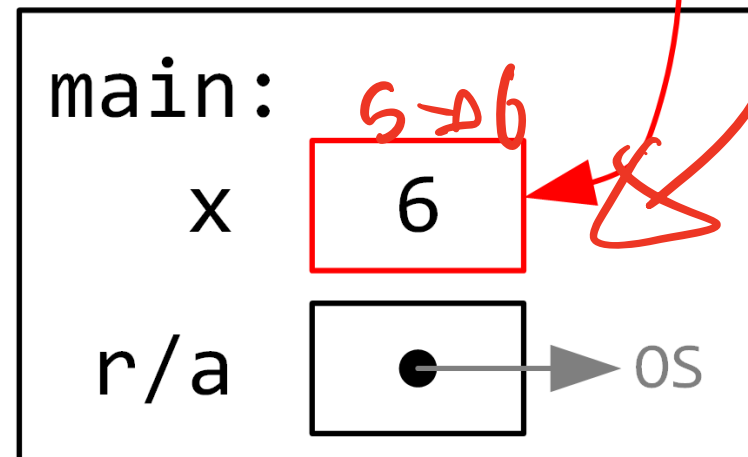
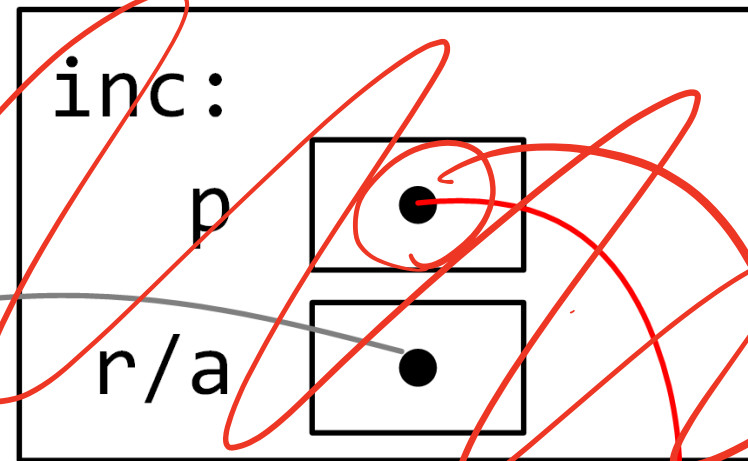
The corresponding parameter type is an *int* pointer (*int* *).

```

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

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

```



Same
 $(\cancel{p}) + 1$

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
// requires: p is a valid pointer
```

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

if (!p)

i += 1;
i++;
++i;
ret.

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: swapping two values

// swap(px, py) switches the values *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);
```

```
}
```

a => 3, b => 4

a => 4, b => 3

~~a, b = b, a~~

~~pass by value~~

int swap(int x, int y)

int temp = x;

x = y;

y = temp;

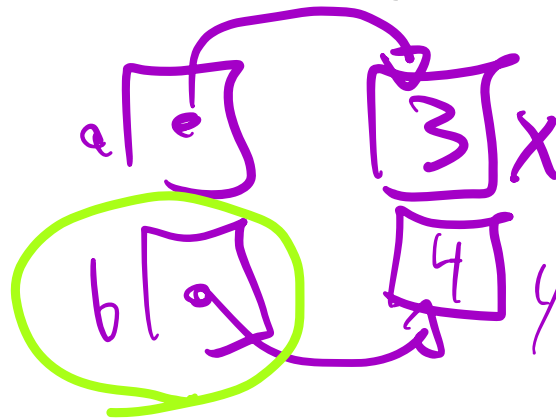
// Note the &

Returning an address

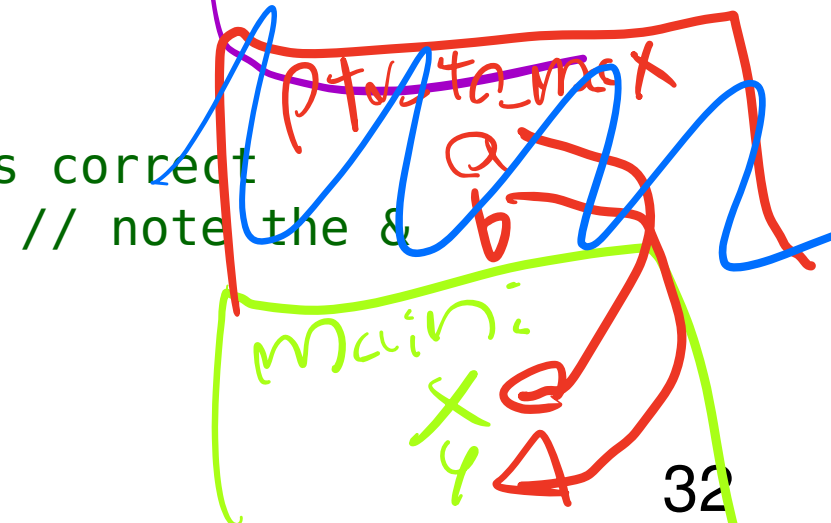
A function may **return** an address (this becomes more useful in Section 8).

```
// ptr_to_max(a, b) returns either a or b,  
// whichever points to the larger int  
int * ptr_to_max(int * a, int * b) {  
    if (*a >= *b) {  
        return a;  
    }  
    return b;  
}
```

```
int main(void) {  
    int x = 3;  
    int y = 4;  
    // Test the output of the function is correct  
    int * p = ptr_to_max(&x, &y);  
    assert(p == &y);  
}
```



int max(int a, int b)
if (a >= b)
 return a;
return b;



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

As soon as the function **returns**, 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


Previously you may have used the built-in `scanf` function to read input. We are now capable of understanding how it works.

```
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.

scanf("%d %d", &i, &j);



Just as with `printf`, multiple format specifiers can be used to read in more than one value. However, it is recommended that you **only read in one value per `scanf`**. This will help you debug your code and facilitate your 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.

EOF
= -1

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 the CLion “Run” tool window a Ctrl - D (“Control D”) keyboard sequence sends an `EOF`.

`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`).

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

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

number of arguments

```
if (retval == EOF)  
    // give error msg  
    // clear EOF
```

example: reading integers

Write a function `read_sum` that reads in `ints` from input (until `EOF` or an unsuccessful read occurs) and returns their sum.


```
// read_sum() reads ints from input and returns their sum
int read_sum(void) {
    int sum = 0;
    int n = 0;
    while (scanf("%d", &n) == 1) {
        sum += n;
    }
    return sum;
}
```

Handwritten annotations:


- A red box around the condition `scanf("%d", &n) == 1` with the text `== 1` written in red.
- A purple bracket under the `scanf` function call.
- A red underline under the `sum += n;` line.
- A red underline under the `return sum;` line.
- An arrow pointing from the red box to the text "fails if".
- Two numbered points:
 - ① input is char
 - ② ctrl + d (EOF)

Whitespace

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

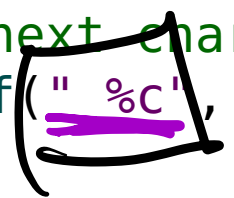


When reading in a `char`, you *may* or *may not* want to ignore whitespace: it depends on your application.



```
// 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`).

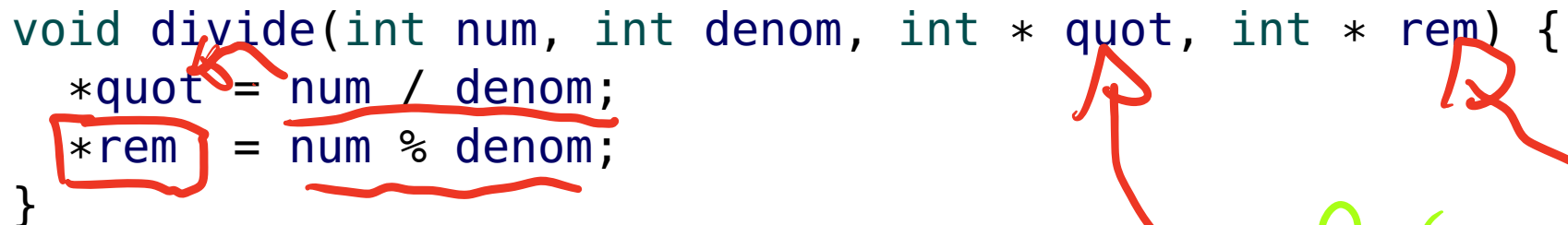
Pointer parameters can be used to *emulate* “returning” more than one value.

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

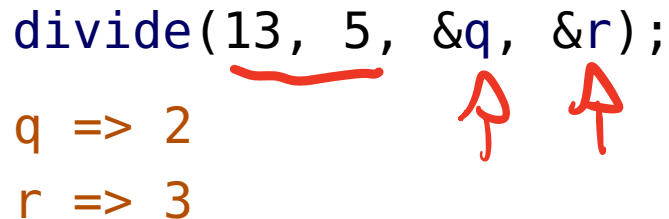
This function performs division and “returns” both the quotient and the remainder.

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

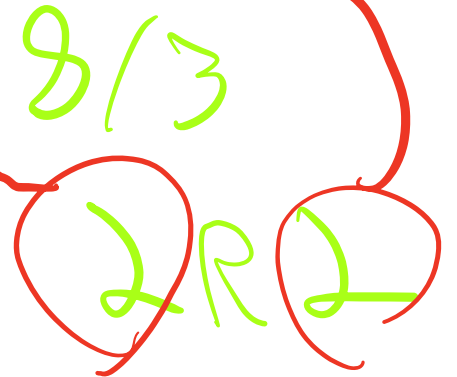


Here is an example of how it can be used:

```
divide(13, 5, &q, &r);  
q => 2  
r => 3
```



8/3
2R2



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.

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 char * p) {  
    *p = 'a';    // 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 `constant`".

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

mutable
↓
int * const p = &i; // p always points at i, i must be
                   // mutable and can be modified via *p
    [p is a constant pointer]
    p = 10; ✓      p = &i; ✗

const int * const p = &i; // p must always point at i
                          // you can not modify i via *p
```

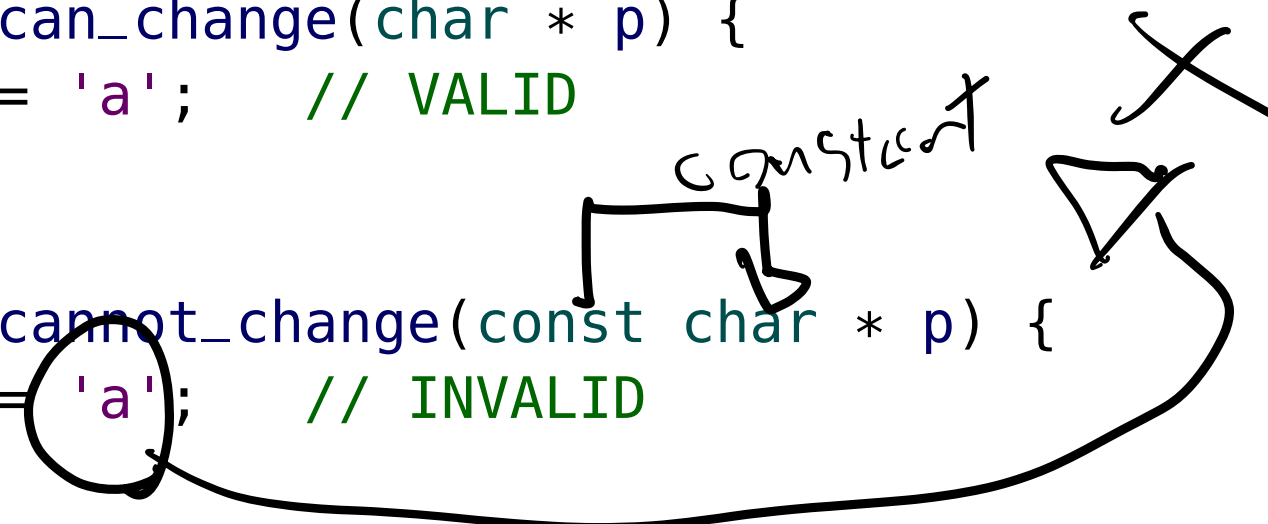
The rule is “`const` applies to the type to the left of it, unless it’s first, and then it applies to the type to the right of it”.

```
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 can not) mutate the contents of the pointer.

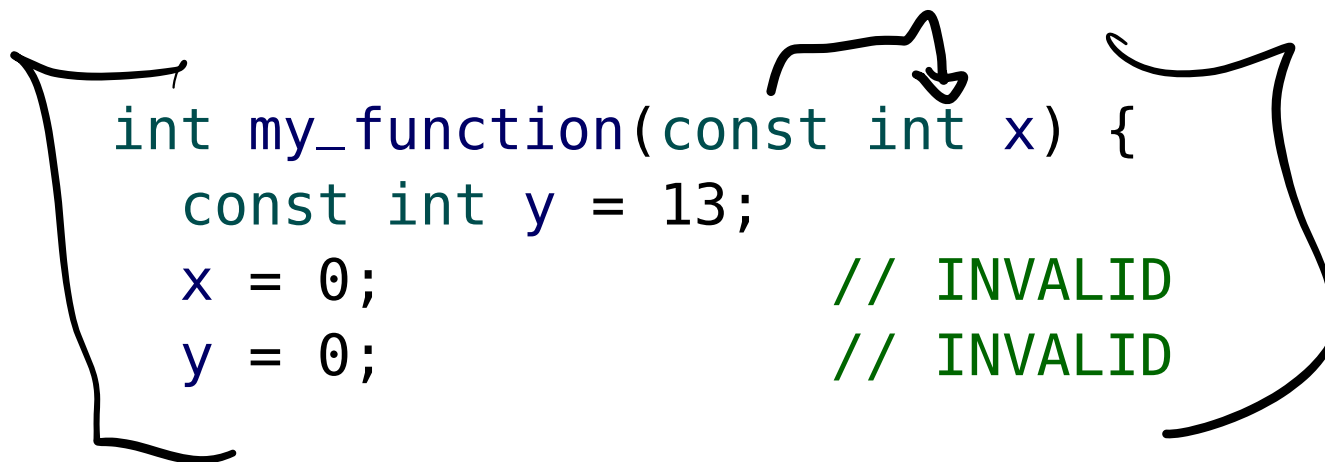
```
void can_change(char * p) {  
    *p = 'a';    // VALID  
}  
  
void cannot_change(const char * p) {  
    *p = 'a';    // INVALID  
}
```



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.

Function pointers

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.

int (ptr) (int) = fact; int fact(int);

The diagram shows a handwritten function pointer declaration `int (ptr) (int) = fact;` and a handwritten function definition `int fact(int);`. Arrows indicate the correspondence: an arrow from `ptr` to `int` in the definition, an arrow from `int` to `int`, and an arrow from `fact` to `fact`.

The syntax to define a function pointer with name `fptr` is:

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

On an exam we would not expect you to memorize this syntax.

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;  
    printf("fp(7, 3) => %d\n", fp(7, 3));  
    fp = my_sub;  
    printf("fp(7, 3) => %d\n", fp(7, 3));  
}
```

fp(7, 3) => 10
fp(7, 3) => 4

my_add(7, 3)

fp(7, 3)

fp(7, 3)

In the previous example:

```
fp = my_add;
```

We could have also used:

```
fp = &my_add;
```

This is because C cannot get the “value” of a function so instead it uses the *address* of the function.

This is often explained by saying that functions in C are not “first class values”. Other programming languages such as Lisp do consider functions to be values that can be stored in variables.

example: a function that modifies a pointer

How could you write a function that modifies a pointer (*e.g.*, sets it to `NULL`)?

A function with parameter `int * p` would not work as `p` is **passed by value** and therefore modifications to `p` would not be visible to the caller. Instead, you can simulate “pass by reference” by passing the *address* of `p` which is of type “`int **`”.

```
// clear_pointer(p) sets the pointer *p to NULL
void clear_pointer(int ** p) {
    *p = NULL;
}
```

Goals of this Section

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

- define and dereference pointers
- use the pointer operators `&` and `*`
- use pointers to simulate pass by reference
- use pointers to perform aliasing
- use the `scanf` function to read input
- explain when a pointer parameter should be `const`
- explain function pointers