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.

```c
int g = 42;

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

The value of g is: 42
the address of g is: 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.

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

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

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

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

is equivalent to the following definition and assignment:

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

The `*` is part of the definition/declaration of `p` and is not part of the variable name. It is part of the type specification. The name of the variable is simply `p`, not `* p`.

As with any variable, its value can be changed.

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

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

```c
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.

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

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

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 *) $\Rightarrow$ 8
sizeof(char *) $\Rightarrow$ 8

Indirection operator

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

*p produces the value of what pointer p "points at".

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

printf("value of p (p) = %p
", p);
printf("value of what p points at (*p) = %d
", *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".
The * symbol is used in three different ways in C:

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

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

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

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

Note that the pointer's identifier (the variable name) can be any valid identifier.

\[
\begin{align*}
\text{int } i &= 2; \\
\text{int } j &= 3; \\
\text{int } p_1 &= \&i; & \text{ // } p_1 \text{ is a pointer to } i \\
\text{int } p_2 &= \&j; & \text{ // } p_2 \text{ is a pointer to } j \\
\text{int } p_3 &= \&i; & \text{ // } p_3 \text{ also points at } i \\
\text{int } p_4 &= \&i; & \text{ // } p_4 \text{ is a pointer to } i \\
\text{int } p_5 &= \&j; & \text{ // } p_5 \text{ is a pointer to } j \\
\text{int } jenny &= \&j; & \text{ // } jenny \text{ is a pointer to } j
\end{align*}
\]

\[
\begin{align*}
p_i &= \&j; & \text{ // the variable name does not restrict its use} \\
p_j &= \&i;
\end{align*}
\]

C mostly ignores white space, so these are equivalent

\[
\begin{align*}
\text{int } *p_1 &= \&i; & \text{ // style A} \\
\text{int } *p_2 &= \&i; & \text{ // style B} \\
\text{int* } p_3 &= \&i; & \text{ // style C}
\end{align*}
\]

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

This is clear with multiple declarations: (not encouraged)

\[
\begin{align*}
i &= 42, \ j &= 23; \\
*p_1 &= \&i, \ *p_2 &= \&j; & \text{ // VALID} \\
*p_3 &= \&i, \ *p_4 &= \&j; & \text{ // INVALID: } p_j \text{ is not a pointer}
\end{align*}
\]
Pointers to pointers

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

```c
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:

```c
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.

```c
(*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.

```c
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.

```c
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**

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

**It does not change the value of \( i \).**

Using the same initial values,

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

p = q;
```

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

```c
int i = 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:

```c
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 its own copy of the argument (in the stack frame) without changing the original variable.
In the “pass by value” convention of C, a **copy** of an argument is passed to a function.

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

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

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

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

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

---

element: mutation side effects

// effects: swaps the contents of * 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); // Note the &
    printf("a = %d, b = %d\n", a, b);
}
a = 3, b = 4
a = 4, b = 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.

```scheme
(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.

```c
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).

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

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

---

**example: pointer return types**

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

```c
int *ptr_to_max(int *a, int *b) {
    if (*a >= *b) return a;
    return b;
}
```

```c
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.**

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

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

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

Recall that when a function is called, a copy of each argument value is placed into the stack frame.

For structures, the entire structure is copied into the frame. For large structures, this can be inefficient.

```
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;
}
```

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.

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

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

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

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

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

```c
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.

```c
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.

```c
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.

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

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

What does it mean when const is used with simple (non-pointer) parameters?

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

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

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

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

A const parameter communicates 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.

```c
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 constant 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.

```c
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 new operators (`&, *, ->`)
- explain when a pointer parameter should be `const`
- use function pointers
- describe aliasing
- use pointers to structures as parameters and explain why parameters are often pointers to structures