Introduction to Pointers in C

Address operator

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

The ***address operator*** (&) produces the 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"`.

memory sections (so far)

We can see how the different memory sections are arranged.

```
const int r = 42;
int g = 15;

int main(void) {

  int s = 23;

  printf("the address of main is:   %p\n", &main); // CODE
  printf("the address of    r is:   %p\n", &r);    // READ-ONLY
  printf("the address of    g is:   %p\n", &g);    // GLOBAL DATA
  printf("the address of    s is:   %p\n", &s);    // STACK
}
```

```
  the address of main is:   0x46c060
  the address of    r is:   0x477620
  the address of    g is:   0x68a9e0
  the address of    s is:   0x7fff4f6031a0
```

To illustrate the behaviour of the stack “growing down”, we can display the address of the parameter n in each recursive call:

```c
int sum_first(int n) {
  printf("the address of n is: %p\n", &n);
  if (n <= 0) return 0;
  return n + sum_first(n-1);
}
```

```
the address of n is: 0x7fff2f7037d0
the address of n is: 0x7fff2f703700
the address of n is: 0x7fff2f703630
the address of n is: 0x7fff2f703560
the address of n is: 0x7fff2f703490
the address of n is: 0x7fff2f7033c0
the address of n is: 0x7fff2f7032f0
the address of n is: 0x7fff2f703220
...
```

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.

```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 *`).

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

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

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

```
address of i  (&i) = 0xf004
value of p     (p) = 0xf004
value of q     (q) = 0xf004
```

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

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.

Note: `sizeof(int *)` and `sizeof(char *)` are both 8 bytes.

```
int i = 42;
char c = 'c';
int *pi = &i;
char *pc = &c;

printf("sizeof(i)      = %zd\n", sizeof(i));
printf("sizeof(c)      = %zd\n", sizeof(c));
printf("sizeof(pi)     = %zd\n", sizeof(pi));
printf("sizeof(pc)     = %zd\n", sizeof(pc));
printf("sizeof(int *)  = %zd\n", sizeof(int *));
printf("sizeof(char *) = %zd\n", sizeof(char *));
```

```
sizeof(i)      = 4
sizeof(c)      = 1
sizeof(pi)     = 8
sizeof(pc)     = 8
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
int j = *p;        // integer j is 42
```

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

example: indirection

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

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

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

```
address of i                (&i) = 0xf004
value of i                   (i) = 42

address of p                (&p) = 0xf008
value of p                   (p) = 0xf004
value of what p points at (*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;
s = sizeof(int *);
```

- as the *indirection operator* for pointers

```
i = *pi;
```

`(*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

In the following code, "`pi` points at `i`", but what if we want a pointer to "point at `pi`"?

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

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

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

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

example: pointers to pointers

```
int i     = 42;
int *pi   = &i;
int **ppi = &pi;
```

```
address of i                              (&i) = 0xf004
value of i                                 (i) = 42

address of pi                            (&pi) = 0xf008
value of pi                               (pi) = 0xf004
value of what pi points at               (*pi) = 42

address of ppi                          (&ppi) = 0xf00C
value of ppi                             (ppi) = 0xf008
value of what ppi points at             (*ppi) = 0xf004
value of what ppi points at points at (**ppi) = 42
```

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

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).

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 “`p` now points 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,

```
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 = 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 it's 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”.

```
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 *).

```
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 bugs: 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
```

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).

```c
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 `return`s, 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) {
  const int xdist = (*p1).x - (*p2).x;
  const int ydist = (*p1).y - (*p2).y;
  return xdist * xdist + ydist * ydist;
}

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

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

```
int sqr_dist(struct posn *p1, struct posn *p2) {
  const int xdist = (*p1).x - (*p2).x;
  const 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) {
  const int xdist = p1->x - p2->x;
  const 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 p->x and p->y by multiplying by f

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

The `const` should be placed 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, must point at 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;
```

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 pointers to structures as parameters and explain why parameters are often pointers to structures