Chapter 22: Pointers

A pointer is a type of variable which can store the address of another object or a function.

Section 22.1: Introduction

A pointer is declared much like any other variable, except an asterisk (*) is placed between the type and the name of the variable to denote it is a pointer.

```
int *pointer; /* inside a function, pointer is uninitialized and doesn't point to any valid object
yet */
```

To declare two pointer variables of the same type, in the same declaration, use the asterisk symbol before each identifier. For example,

```
int *iptr1, *iptr2;
int *iptr3,  iptr4;  /* iptr3 is a pointer variable, whereas iptr4 is misnamed and is an int */
```

The address-of or reference operator denoted by an ampersand (&) gives the address of a given variable which can be placed in a pointer of appropriate type.

```
int value = 1;
pointer = &value;
```

The indirection or dereference operator denoted by an asterisk (*) gets the contents of an object pointed to by a pointer.

```
printf("Value of pointed to integer: %d\n", *pointer);
/* Value of pointed to integer: 1 */
```

If the pointer points to a structure or union type then you can dereference it and access its members directly using the `->` operator:

```
SomeStruct *s = &someObject;
s->someMember = 5; /* Equivalent to (*s).someMember = 5 */
```

In C, a pointer is a distinct value type which can be reassigned and otherwise is treated as a variable in its own right. For example the following example prints the value of the pointer (variable) itself.

```
printf("Value of the pointer itself: %p\n", (void *)pointer);
/* Value of the pointer itself: 0x7ffcd41b06e4 */
/* This address will be different each time the program is executed */
```

Because a pointer is a mutable variable, it is possible for it to not point to a valid object, either by being set to null

```
pointer = 0;     /* or alternatively */
pointer = NULL;
```

or simply by containing an arbitrary bit pattern that isn't a valid address. The latter is a very bad situation, because it cannot be tested before the pointer is being dereferenced, there is only a test for the case a pointer is null:

```
if (!pointer) exit(EXIT_FAILURE);
```

A pointer may only be dereferenced if it points to a *valid* object, otherwise the behavior is undefined. Many modern implementations may help you by raising some kind of error such as a segmentation fault and terminate execution, but others may just leave your program in an invalid state.

The value returned by the dereference operator is a mutable alias to the original variable, so it can be changed, modifying the original variable.

```
*pointer += 1;
printf("Value of pointed to variable after change: %d\n", *pointer);
/* Value of pointed to variable after change: 2 */
```

Pointers are also re-assignable. This means that a pointer pointing to an object can later be used to point to another object of the same type.

```
int value2 = 10;
pointer = &value2;
printf("Value from pointer: %d\n", *pointer);
/* Value from pointer: 10 */
```

Like any other variable, pointers have a specific type. You can't assign the address of a `short int` to a pointer to a `long int`, for instance. Such behavior is referred to as type punning and is forbidden in C, though there are a few exceptions.

Although pointer must be of a specific type, the memory allocated for each type of pointer is equal to the memory used by the environment to store addresses, rather than the size of the type that is pointed to.

```
#include <stdio.h>

int main(void) {
    printf("Size of int pointer: %zu\n", sizeof (int*));      /* size 4 bytes */
    printf("Size of int variable: %zu\n", sizeof (int));      /* size 4 bytes */
    printf("Size of char pointer: %zu\n", sizeof (char*));    /* size 4 bytes */
    printf("Size of char variable: %zu\n", sizeof (char));    /* size 1 bytes */
    printf("Size of short pointer: %zu\n", sizeof (short*));  /* size 4 bytes */
    printf("Size of short variable: %zu\n", sizeof (short));  /* size 2 bytes */
    return 0;
}
```

(NB: if you are using Microsoft Visual Studio, which does not support the C99 or C11 standards, you must use `%Iu`1 instead of `%zu` in the above sample.)

Note that the results above can vary from environment to environment in numbers but all environments would show equal sizes for different types of pointer.

Extract based on information from Cardiff University C Pointers Introduction

Pointers and Arrays

Pointers and arrays are intimately connected in C. Arrays in C are always held in contiguous locations in memory. Pointer arithmetic is always scaled by the size of the item pointed to. So if we have an array of three doubles, and a pointer to the base, `*ptr` refers to the first double, `*(ptr + 1)` to the second, `*(ptr + 2)` to the third. A more convenient notation is to use array notation `[ ]`.

```
double point[3] = {0.0, 1.0, 2.0};
double *ptr = point;
```

```
/* prints x 0.0, y 1.0 z 2.0 */
printf("x %f y %f z %f\n", ptr[0], ptr[1], ptr[2]);
```

So essentially ptr and the array name are interchangeable. This rule also means that an array decays to a pointer when passed to a subroutine.

```
double point[3] = {0.0, 1.0, 2.0};

printf("length of point is %s\n", length(point));

/* get the distance of a 3D point from the origin */
double length(double *pt)
{
   return sqrt(pt[0] * pt[0] + pt[1] * pt[1] + pt[2] * pt[2])
}
```

A pointer may point to any element in an array, or to the element beyond the last element. It is however an error to set a pointer to any other value, including the element before the array. (The reason is that on segmented architectures the address before the first element may cross a segment boundary, the compiler ensures that does not happen for the last element plus one).

Footnote 1: Microsoft format information can be found via `printf()` and format specification syntax.

Section 22.2: Common errors

Improper use of pointers are frequently a source of bugs that can include security bugs or program crashes, most often due to segmentation faults.

Not checking for allocation failures

Memory allocation is not guaranteed to succeed, and may instead return a `NULL` pointer. Using the returned value, without checking if the allocation is successful, invokes undefined behavior. This usually leads to a crash, but there is no guarantee that a crash will happen so relying on that can also lead to problems.

For example, unsafe way:

```
struct SomeStruct *s = malloc(sizeof *s);
s->someValue = 0; /* UNSAFE, because s might be a null pointer */
```

Safe way:

```
struct SomeStruct *s = malloc(sizeof *s);
if (s)
{
    s->someValue = 0; /* This is safe, we have checked that s is valid */
}
```

Using literal numbers instead of sizeof when requesting memory

For a given compiler/machine configuration, types have a known size; however, there isn't any standard which defines that the size of a given type (other than `char`) will be the same for all compiler/machine configurations. If the code uses 4 instead of `sizeof(int)` for memory allocation, it may work on the original machine, but the code isn't necessarily portable to other machines or compilers. Fixed sizes for types should be replaced by

`sizeof(that_type)` or `sizeof(*var_ptr_to_that_type)`.

Non-portable allocation:

```
int *intPtr = malloc(4*1000);    /* allocating storage for 1000 int */
long *longPtr = malloc(8*1000);  /* allocating storage for 1000 long */
```

Portable allocation:

```
int *intPtr = malloc(sizeof(int)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(long)*1000);  /* allocating storage for 1000 long */
```

Or, better still:

```
int *intPtr = malloc(sizeof(*intPtr)*1000);     /* allocating storage for 1000 int */
long *longPtr = malloc(sizeof(*longPtr)*1000);  /* allocating storage for 1000 long */
```

Memory leaks

Failure to de-allocate memory using `free` leads to a buildup of non-reusable memory, which is no longer used by the program; this is called a memory leak. Memory leaks waste memory resources and can lead to allocation failures.

Logical errors

All allocations must follow the same pattern:

1. Allocation using `malloc` (or `calloc`)
2. Usage to store data
3. De-allocation using `free`

Failure to adhere to this pattern, such as using memory after a call to `free` (dangling pointer) or before a call to `malloc` (wild pointer), calling `free` twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program.

These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.

On systems where it works, Valgrind is an invaluable tool for identifying what memory is leaked and where it was originally allocated.

Creating pointers to stack variables

Creating a pointer does not extend the life of the variable being pointed to. For example:

```
int* myFunction()
{
    int x = 10;
    return &x;
}
```

Here, `x` has *automatic storage duration* (commonly known as *stack* allocation). Because it is allocated on the stack, its lifetime is only as long as `myFunction` is executing; after `myFunction` has exited, the variable `x` is destroyed. This function gets the address of `x` (using `&x`), and returns it to the caller, leaving the caller with a pointer to a non-existent variable. Attempting to access this variable will then invoke undefined behavior.

Most compilers don't actually clear a stack frame after the function exits, thus dereferencing the returned pointer often gives you the expected data. When another function is called however, the memory being pointed to may be overwritten, and it appears that the data being pointed to has been corrupted.

To resolve this, either `malloc` the storage for the variable to be returned, and return a pointer to the newly created storage, or require that a valid pointer is passed in to the function instead of returning one, for example:

```
#include <stdlib.h>
#include <stdio.h>

int *solution1(void)
{
    int *x = malloc(sizeof *x);
    if (x == NULL)
    {
        /* Something went wrong */
        return NULL;
    }

    *x = 10;

    return x;
}

void solution2(int *x)
{
    /* NB: calling this function with an invalid or null pointer
       causes undefined behaviour. */

    *x = 10;
}

int main(void)
{
    {
        /* Use solution1() */

        int *foo = solution1();
        if (foo == NULL)
        {
            /* Something went wrong */
            return 1;
        }

        printf("The value set by solution1() is %i\n", *foo);
        /* Will output: "The value set by solution1() is 10" */

        free(foo);    /* Tidy up */
    }

    {
        /* Use solution2() */

        int bar;
        solution2(&bar);

        printf("The value set by solution2() is %i\n", bar);
        /* Will output: "The value set by solution2() is 10" */
    }

    return 0;
```

```
}
```

Incrementing / decrementing and dereferencing

If you write `*p++` to increment what is pointed by `p`, you are wrong.

Post incrementing / decrementing is executed before dereferencing. Therefore, this expression will increment the pointer `p` itself and return what was pointed by `p` before incrementing without changing it.

You should write `(*p)++` to increment what is pointed by `p`.

This rule also applies to post decrementing: `*p--` will decrement the pointer `p` itself, not what is pointed by `p`.

Section 22.3: Dereferencing a Pointer

```
int a = 1;
int *a_pointer = &a;
```

To dereference `a_pointer` and change the value of a, we use the following operation

```
*a_pointer = 2;
```

This can be verified using the following print statements.

```
printf("%d\n", a); /* Prints 2 */
printf("%d\n", *a_pointer); /* Also prints 2 */
```

However, one would be mistaken to dereference a `NULL` or otherwise invalid pointer. This

```
int *p1, *p2;

p1 = (int *) 0xbad;
p2 = NULL;

*p1 = 42;
*p2 = *p1 + 1;
```

is usually undefined behavior. `p1` may not be dereferenced because it points to an address `0xbad` which may not be a valid address. Who knows what's there? It might be operating system memory, or another program's memory. The only time code like this is used, is in embedded development, which stores particular information at hard-coded addresses. `p2` cannot be dereferenced because it is `NULL`, which is invalid.

Section 22.4: Dereferencing a Pointer to a struct

Let's say we have the following structure:

```
struct MY_STRUCT
{
    int my_int;
    float my_float;
};
```

We can define `MY_STRUCT` to omit the `struct` keyword so we don't have to type `struct MY_STRUCT` each time we use it. This, however, is optional.

```
typedef struct MY_STRUCT MY_STRUCT;
```

If we then have a pointer to an instance of this struct

```
MY_STRUCT *instance;
```

If this statement appears at file scope, `instance` will be initialized with a null pointer when the program starts. If this statement appears inside a function, its value is undefined. The variable must be initialized to point to a valid `MY_STRUCT` variable, or to dynamically allocated space, before it can be dereferenced. For example:

```
MY_STRUCT info = { 1, 3.141593F };
MY_STRUCT *instance = &info;
```

When the pointer is valid, we can dereference it to access its members using one of two different notations:

```
int a = (*instance).my_int;
float b = instance->my_float;
```

While both these methods work, it is better practice to use the arrow `->` operator rather than the combination of parentheses, the dereference `*` operator and the dot `.` operator because it is easier to read and understand, especially with nested uses.

Another important difference is shown below:

```
MY_STRUCT copy = *instance;
copy.my_int    = 2;
```

In this case, `copy` contains a copy of the contents of `instance`. Changing `my_int` of `copy` will not change it in `instance`.

```
MY_STRUCT *ref = instance;
ref->my_int    = 2;
```

In this case, `ref` is a reference to `instance`. Changing `my_int` using the reference will change it in `instance`.

It is common practice to use pointers to structs as parameters in functions, rather than the structs themselves. Using the structs as function parameters could cause the stack to overflow if the struct is large. Using a pointer to a struct only uses enough stack space for the pointer, but can cause side effects if the function changes the struct which is passed into the function.

Section 22.5: Const Pointers

Single Pointers

- Pointer to an `int`

  The pointer can point to different integers and the `int`'s can be changed through the pointer. This sample of code assigns b to point to `int b` then changes b's value to `100`.

  ```
  int b;
  int* p;
  p = &b;    /* OK */
  *p = 100;  /* OK */
  ```

- Pointer to a `const int`

  The pointer can point to different integers but the `int`'s value can't be changed through the pointer.

```
int b;
const int* p;
p = &b;    /* OK */
*p = 100;  /* Compiler Error */
```

- `const` pointer to `int`

  The pointer can only point to one `int` but the `int`'s value can be changed through the pointer.

```
int a, b;
int* const p = &b; /* OK as initialisation, no assignment */
*p = 100;  /* OK */
p = &a;    /* Compiler Error */
```

- `const` pointer to `const int`

  The pointer can only point to one `int` and the `int` can not be changed through the pointer.

```
int a, b;
const int* const p = &b; /* OK as initialisation, no assignment */
p = &a;   /* Compiler Error */
*p = 100; /* Compiler Error */
```

Pointer to Pointer

- Pointer to a pointer to an `int`

  This code assigns the address of `p1` to the to double pointer `p` (which then points to `int* p1` (which points to `int`)).

  Then changes `p1` to point to `int a`. Then changes the value of a to be 100.

```
void f1(void)
{
  int a, b;
  int *p1;
  int **p;
  p1 = &b; /* OK */
  p = &p1; /* OK */
  *p = &a; /* OK */
  **p = 100; /* OK */
}
```

- Pointer to pointer to a `const int`

```
 void f2(void)
{
  int b;
  const int *p1;
  const int **p;
```

```
  p = &p1; /* OK */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- Pointer to `const` pointer to an `int`

```
void f3(void)
{
  int b;
  int *p1;
  int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* OK */
}
```

- `const` pointer to pointer to `int`

```
void f4(void)
{
  int b;
  int *p1;
  int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
  **p = 100; /* OK */
}
```

- Pointer to `const` pointer to `const int`

```
void f5(void)
{
  int b;
  const int *p1;
  const int * const *p;
  p = &p1; /* OK */
  *p = &b; /* error: assignment of read-only location '*p' */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` pointer to pointer to `const int`

```
void f6(void)
{
  int b;
  const int *p1;
  const int ** const p = &p1; /* OK as initialisation, not assignment */
  p = &p1; /* error: assignment of read-only variable 'p' */
  *p = &b; /* OK */
  **p = 100; /* error: assignment of read-only location '**p' */
}
```

- `const` pointer to `const` pointer to `int`

```
void f7(void)
{
```

```
    int b;
    int *p1;
    int * const * const p = &p1; /* OK as initialisation, not assignment */
    p = &p1; /* error: assignment of read-only variable ‘p’  */
    *p = &b; /* error: assignment of read-only location ‘*p’ */
    **p = 100; /* OK */
}
```

Section 22.6: Function pointers

Pointers can also be used to point at functions.

Let's take a basic function:

```
int my_function(int a, int b)
{
    return 2 * a + 3 * b;
}
```

Now, let's define a pointer of that function's type:

```
int (*my_pointer)(int, int);
```

To create one, just use this template:

```
return_type_of_func (*my_func_pointer)(type_arg1, type_arg2, ...)
```

We then must assign this pointer to the function:

```
my_pointer = &my_function;
```

This pointer can now be used to call the function:

```
/* Calling the pointed function */
int result = (*my_pointer)(4, 2);

...

/* Using the function pointer as an argument to another function */
void another_function(int (*another_pointer)(int, int))
{
    int a = 4;
    int b = 2;
    int result = (*another_pointer)(a, b);

    printf("%d\n", result);
}
```

Although this syntax seems more natural and coherent with basic types, attributing and dereferencing function pointers don't require the usage of `&` and `*` operators. So the following snippet is equally valid:

```
/* Attribution without the & operator */
my_pointer = my_function;

/* Dereferencing without the * operator */
int result = my_pointer(4, 2);
```

To increase the readability of function pointers, typedefs may be used.

```
typedef void (*Callback)(int a);

void some_function(Callback callback)
{
    int a = 4;
    callback(a);
}
```

Another readability trick is that the C standard allows one to simplify a function pointer in arguments like above (but not in variable declaration) to something that looks like a function prototype; thus the following can be equivalently used for function definitions and declarations:

```
void some_function(void callback(int))
{
    int a = 4;
    callback(a);
}
```

See also

Function Pointers

Section 22.7: Polymorphic behaviour with void pointers

The `qsort()` standard library function is a good example of how one can use void pointers to make a single function operate on a large variety of different types.

```
void qsort (
    void *base,                                 /* Array to be sorted */
    size_t num,                                 /* Number of elements in array */
    size_t size,                                /* Size in bytes of each element */
    int (*compar)(const void *, const void *)); /* Comparison function for two elements */
```

The array to be sorted is passed as a void pointer, so an array of any type of element can be operated on. The next two arguments tell `qsort()` how many elements it should expect in the array, and how large, in bytes, each element is.

The last argument is a function pointer to a comparison function which itself takes two void pointers. By making the caller provide this function, `qsort()` can effectively sort elements of any type.

Here's an example of such a comparison function, for comparing floats. Note that any comparison function passed to `qsort()` needs to have this type signature. The way it is made polymorphic is by casting the void pointer arguments to pointers of the type of element we wish to compare.

```
int compare_floats(const void *a, const void *b)
{
    float fa = *((float *)a);
    float fb = *((float *)b);
    if (fa < fb)
        return -1;
    if (fa > fb)
        return 1;
    return 0;
}
```

Since we know that qsort will use this function to compare floats, we cast the void pointer arguments back to float pointers before dereferencing them.

Now, the usage of the polymorphic function qsort on an array "array" with length "len" is very simple:

```
qsort(array, len, sizeof(array[0]), compare_floats);
```

Section 22.8: Address-of Operator (&)

For any object (i.e, variable, array, union, struct, pointer or function) the unary address operator can be used to access the address of that object.

Suppose that

```
int i = 1;
int *p = NULL;
```

So then a statement `p = &i;`, copies the address of the variable `i` to the pointer `p`.

It's expressed as `p` ***points to*** `i`.

`printf("%d\n", *p);` prints 1, which is the value of `i`.

Section 22.9: Initializing Pointers

Pointer initialization is a good way to avoid wild pointers. The initialization is simple and is no different from initialization of a variable.

```
#include <stddef.h>

int main()
{
    int *p1 = NULL;
    char *p2 = NULL;
    float *p3 = NULL;

         /* NULL is a macro defined in stddef.h, stdio.h, stdlib.h, and string.h */

    ...
}
```

In most operating systems, inadvertently using a pointer that has been initialized to `NULL` will often result in the program crashing immediately, making it easy to identify the cause of the problem. Using an uninitialized pointer can often cause hard-to-diagnose bugs.

Caution:

The result of dereferencing a `NULL` pointer is undefined, so it *will not necessarily cause a crash* even if that is the natural behaviour of the operating system the program is running on. Compiler optimizations may mask the crash, cause the crash to occur before or after the point in the source code at which the null pointer dereference occurred, or cause parts of the code that contains the null pointer dereference to be unexpectedly removed from the program. Debug builds will not usually exhibit these behaviours, but this is not guaranteed by the language standard. Other unexpected and/or undesirable behaviour is also allowed.

Because `NULL` never points to a variable, to allocated memory, or to a function, it is safe to use as a guard value.

Caution:

Usually `NULL` is defined as `(void *)0`. But this does not imply that the assigned memory address is `0x0`. For more clarification refer to C-faq for NULL pointers

Note that you can also initialize pointers to contain values other than NULL.

```
int i1;

int main()
{
   int *p1 = &i1;
   const char *p2 = "A constant string to point to";
   float *p3 = malloc(10 * sizeof(float));
}
```

Section 22.10: Pointer to Pointer

In C, a pointer can refer to another pointer.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &pA;
  int*** pppA = &ppA;

  printf("%d", ***pppA); /* prints 42 */

  return EXIT_SUCCESS;
}
```

But, reference-and-reference directly is not allowed.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  int A = 42;
  int* pA = &A;
  int** ppA = &&A; /* Compilation error here! */
  int*** pppA = &&&A;  /* Compilation error here! */

  ...
```

Section 22.11: void* pointers as arguments and return values to standard functions

Version > K&R

`void*` is a catch all type for pointers to object types. An example of this in use is with the `malloc` function, which is declared as

```
void* malloc(size_t);
```

The pointer-to-void return type means that it is possible to assign the return value from `malloc` to a pointer to any other type of object:

```
int* vector = malloc(10 * sizeof *vector);
```

It is generally considered good practice to *not* explicitly cast the values into and out of void pointers. In specific case of `malloc()` this is because with an explicit cast, the compiler may otherwise assume, but not warn about, an incorrect return type for `malloc()`, if you forget to include `stdlib.h`. It is also a case of using the correct behavior of void pointers to better conform to the DRY (don't repeat yourself) principle; compare the above to the following, wherein the following code contains several needless additional places where a typo could cause issues:

```
int* vector = (int*)malloc(10 * sizeof int*);
```

Similarly, functions such as

```
void* memcpy(void *restrict target, void const *restrict source, size_t size);
```

have their arguments specified as `void *` because the address of any object, regardless of the type, can be passed in. Here also, a call should not use a cast

```
unsigned char buffer[sizeof(int)];
int b = 67;
memcpy(buffer, &b, sizeof buffer);
```

Section 22.12: Same Asterisk, Different Meanings

Premise

The most confusing thing surrounding pointer syntax in C and C++ is that there are actually two different meanings that apply when the pointer symbol, the asterisk (`*`), is used with a variable.

Example

Firstly, you use `*` to **declare** a pointer variable.

```
int i = 5;
/* 'p' is a pointer to an integer, initialized as NULL */
int *p = NULL;
/* '&i' evaluates into address of 'i', which then assigned to 'p' */
p = &i;
/* 'p' is now holding the address of 'i' */
```

When you're not declaring (or multiplying), `*` is used to **dereference** a pointer variable:

```
*p = 123;
/* 'p' was pointing to 'i', so this changes value of 'i' to 123 */
```

When you want an existing pointer variable to hold address of other variable, you **don't** use `*`, but do it like this:

```
p = &another_variable;
```

A common confusion among C-programming newbies arises when they declare and initialize a pointer variable at the same time.

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

Since `int i = 5;` and `int i; i = 5;` give the same result, some of them might thought `int *p = &i;` and `int *p; *p = &i;` give the same result too. The fact is, no, `int *p; *p = &i;` will attempt to deference an **uninitialized** pointer which will result in UB. Never use `*` when you're not declaring nor dereferencing a pointer.

Conclusion

The asterisk (`*`) has two distinct meanings within C in relation to pointers, depending on where it's used. When used within a **variable declaration**, the value on the right hand side of the equals side should be a ***pointer value*** to an ***address*** in memory. When used with an already **declared variable**, the asterisk will ***dereference*** the pointer value, following it to the pointed-to place in memory, and allowing the value stored there to be assigned or retrieved.

Takeaway

It is important to mind your P's and Q's, so to speak, when dealing with pointers. Be mindful of when you're using the asterisk, and what it means when you use it there. Overlooking this tiny detail could result in buggy and/or undefined behavior that you really don't want to have to deal with.