INTERMEZZO A: Introduction to C Pointers

Gene Cooperman

Introduction to C Pointers

Understanding pointers is a fundamental milestone in learning C. Understanding pointers is also required for understanding the man pages of system calls. At its core, a pointer is simply a variable that stores the **memory address** of another variable.

---

1. Direct and Indirect Manipulation: `prog1.c`

To understand how memory works, imagine computer memory as a long row of numbered **boxes**. Each variable you declare is a box that holds a value.

```c
#include <stdio.h>
void addOne(int *myvariable) {
  *myvariable = *myvariable + 1;
}
// Three ways to increment a variable, including pointers.
int main() {
  int x = 17;
  int *y = &x;
  x = x+1;
  printf("x: %d\n", x);
  *y = *y + 1;
  printf("x: %d\n", x);
  addOne(&x);
  printf("x: %d\n", x);
  return 0;
}
```

Expected Output

```
x: 18
x: 19
x: 20
```

Memory Diagram (The "Box" Analogy)

The following diagram illustrates the relationship between the variable `x` and the pointer `y`. We assume for this example that the box for `x` is located at memory address **0x4a30**.

```
   int *y:                 0x4a30: int x:
+-----------+             +-----------+
|  0x4a30   | ---------> |     17    |
+-----------+             +-----------+
```

Explaining the Variables

- **The `x` Box:** When `int x = 17;` is executed, C sets aside a box labeled `int x:` at a specific address (e.g., **0x4a30**) and stores the value **17** inside it.
- **The `y` Box:** When `int *y = &x;` is executed, a second box labeled `int *y:` is created.
- **The Type:** In the declaration `int *y`, the type is `int *`, which means "pointer to int".
- **The Value:** The value inside the `y` box is **0x4a30**, the memory address of the `x` box. This address acts as a pointer to `x`.
- **The Address-of Operator (`&`):** The `&` symbol is an operator that returns the memory address of a variable—it tells you where that variable's box is located in memory.
- **Dereferencing (`*`):** In a statement using `*y`, the `*` operator tells the computer to "follow the pointer" from the `y` box to the box it points to. This process is called dereferencing.

---

2. Arrays and Strings: `prog2.c`

In C, an **array is a constant pointer** to its first element. Consequently, C does not have a native string type; it uses the type `char *` as a pointer to the beginning of an array of characters. By convention, an array of `char` representing a string must end with the **null character** (`\0`) to indicate the end of the string.

```c
#include <stdio.h>
int main() {
  char *mystring = "Help me, or I'm leaving.\n";
  // ALTERNATIVE:  char mystring[] = "...\n";
  printf("%s\n", mystring);
  printf("%s\n", mystring + 12);
  printf("%s\n", &mystring[12]);
  mystring[7] = '\0';
  printf("%s\n", mystring);
  return 0;
}
```

Expected Output

```
Help me, or I'm leaving.
I'm leaving.
I'm leaving.
Help me
```

Memory Diagram of the C String

This diagram shows the `mystring` pointer pointing to the array starting at address **0x6be0**.

```
  char *mystring:
+----------+
|  0x6be0  |
+----------+
     |
     |
     v
 0x6be0:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| H | e | l | p |   | m | e | , |   | o | r |...| n | g | . | \n| \0|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

An Array Name is a Constant Pointer

Finally, note that we could have used either of:

```
const char *mystring = "Help me, or I'm leaving.";
char mystring[] = "Help me, or I'm leaving.";
```

In this form, the diagram would no longer show a mystring box in memory. Now, mystring is a constant. Constants have a fixed value known only to the compiler, and the value of the constant cannot be changed at runtime.

Understanding Pointer Arithmetic and the Null Character as Terminator

1. **Pointers of `char *`:** The expression `mystring + 12` calculates a new address by moving 12 character-sized boxes forward from the start of the array.
2. **Addressing Indices:** The expression `&mystring[12]` is functionally identical to `mystring + 12`, as both return the memory address of the character at index 12. Similalry, `mystring[12]` returns a `char`, and the expression is equivalent to `(*mystring+12)`.
3. **The Null Character (`\0`) as Terminator:** By setting `mystring[7] = '\0';`, the code inserts a "stop sign" into the array. When `printf` processes the string, it stops immediately upon hitting the null character, even though the rest of the characters are still in memory.
4. **Pointers of `int`:** Now, let's consider an array of int: `int mynum[100]`. In this case, the expression `&mynum[12]` is functionally identical to `mynum + 12`, as both return the memory address of the character at index 12. Similalry, `mynum[12]` returns an `int`, and the expression is equivalent to `(*mynum+12)`.

With this insight, we can now discuss **pointer arithmetic**. What happens if you add an integer to a pointer? What happens if you subtract two pointers? The answer to these questions depends on the type of the pointer. Is it a pointer to a char (`char *`), a pointer to an int (`int *`), or something else? In particular, how many bytes does a `char` occupy and how many bytes does an `int` occupy.

In addition to the operators, '*' and '&', C provides a third operator, `sizeof`. In a typical C compiler for a typical CPU, we will usually find that `sizeof(char)==1` and `sizeof(int)==4`, where the size is measured in bytes. This means that a "char box" occupies 1 byte and an "int box" occupies 4 bytes.

Nevertheless, `&mynum[7]-&mynum[6]==(myunum+7)-(mynum+6)==1`. Intuitively, we are operating on pointers to `int`. When you add one to a pointer to `int`, then we are now pointing to the *next* "int box". An "int box" has size 4. So, we are now pointing to an address that is 4 bytes beyond the original address, but as a pointer to `int`, we have incremented the pointer to 1. Similarly, if we subtract two pointers to `int`, the result is the number of "int boxes" between the first pointer and the second pointer.

You can verify this with:

```c
#include <stdio.h>
// Adding and subtracting pointers to int.
int main() {
  int mynum[100];
  int *ptrA = mynum+6;
  int *ptrB = mynum+7;
  // Subtracting these two pointers yields the int, 1.
  printf("ptrB-ptrA: %ld\n", ptrB-ptrA);
  // Again subtracting these two pointers yields the int, 1.
  printf("(mynum+7)-(mynum+6): %ld\n", (mynum+7)-(mynum+6));
  // And again subtracting these two pointers yields the int, 1.
  printf("&mynum[7]-&mynum[6]: %ld\n", &mynum[7]-&mynum[6]);
  // Adding a pointer and the int 1 yields a new pointer that is 4 bytes higher.
  printf("ptrA: %p; ptrA+1: %p\n", ptrA, ptrA+1);
  // Again, the second pointer is 4 bytes higher.
  printf("&mynum[6]: %p; &mynum[7]: %p\n", &mynum[6], &mynum[7]);
  return 0;
}
```

And finally, C provides a **cast**, so that you can change the type. So, `(char *)(mynum+7)-(char *)(mynum+6)` is equal to 4, not 1. We can fit 4 "char boxes" between the two pointers that we subtracted.

3. System Calls and IN/OUT Parameters

A **system call** is a controlled entry point into the operating system kernel. User programs cannot directly access hardware or perform privileged operations—they must ask the kernel to do it on their behalf.

Common syscalls include:

- `read` - Read data from a file or input device
- `write` - Write data to a file or output device
- `open` - Open a file
- `close` - Close a file
- `time` - Get the current time
- `brk` - Allocate memory

Pointers in System Calls

Many syscalls require pointers as arguments. This is because:

1. **Efficiency**: Passing a pointer to a large buffer is cheaper than copying the entire buffer
2. **Kernel access**: The kernel needs to know where in your program's memory to read from or write to
3. **Variable-length data**: Strings and buffers have varying sizes

IN Parameters vs OUT Parameters

When a syscall takes a pointer argument, it can be used in two ways:

IN parameter: The pointer points to data that the syscall will *read*. Your program fills in the data before the syscall, and the kernel reads it.

OUT parameter: The pointer points to a buffer where the syscall will *write* results. Your program provides empty space, and the kernel fills it in.

Some parameters are **IN-OUT**: the kernel both reads the initial value and writes back a result.

Example 1: The `write` Syscall (IN Parameter)

The `write` syscall outputs data to a file descriptor. Its signature in C is:

```
ssize_t write(int fd, const void *buf, size_t count);
```

- `fd` - **IN parameter**: File descriptor (1 = stdout, 2 = stderr)
- `buf` - **IN parameter**: Pointer to the data to write
- `count` - **IN parameter**: Number of bytes to write
- Returns: Number of bytes actually written

The `buf` parameter is an IN parameter because the kernel *reads* from this memory location.

```
char message[] = "Hello, World!\n";
// Pass the address of our data using & (or array name decays to pointer)
int bytes_written = write(1, message, 14);
// The kernel reads FROM the memory at &message[0]
// Our data flows: message -> kernel -> screen
```

Notice that we pass the *address* of `message` to the kernel. The kernel will read 14 bytes starting at that address.

Example 2: The `read` Syscall (OUT Parameter)

The `read` syscall inputs data from a file descriptor. Its signature in C is:

```
ssize_t read(int fd, void *buf, size_t count);
```

- `fd` - **IN parameter**: File descriptor (0 = stdin)
- `buf` - **OUT parameter**: Pointer to buffer where data will be stored
- `count` - **IN parameter**: Maximum number of bytes to read
- Returns: Number of bytes actually read

The `buf` parameter is an OUT parameter because the kernel *writes* to this memory location.

```
char buffer[100];  // Reserve space for input (initially empty/undefined)
// Pass the address of our buffer.  An array name is also a pointer.
int bytes_read = read(0, buffer, 100);
// The kernel writes TO the memory at buffer (also could be: &buffer[0])
```

Before the syscall, `buffer` is uninitialized space. After the syscall, the kernel has filled it with whatever the user typed.

Example 3: The `time` Syscall (OUT Parameter)

The `time` syscall gets the current time. Its signature in C is:

```
time_t time(time_t *tloc);
```

- `tloc` - **OUT parameter**: Pointer to where the time should be stored (can be NULL)
- Returns: The current time (seconds since January 1, 1970)

This syscall demonstrates an interesting pattern: the return value and the OUT parameter contain the same information.

```
time_t tloc;
// Pass the address of time_val using &
time_t result = time(&tloc);
// The kernel writes TO the memory at &time_val
// After the call:
//   - result contains the current time
```

Using the `&` operator, we pass the *address* of `tloc`. The kernel then uses this address to write the current time into our variable.

Summary: Pointer Usage Patterns in Syscalls

Syscall	Pointer Parameter	Type	Purpose
`write`	`buf`	IN	Kernel reads data from your buffer
`read`	`buf`	OUT	Kernel writes data into your buffer
`time`	`tloc`	OUT	Kernel writes time value

Key Principles

1. **Syscalls are the interface to the OS** - They're how your program requests services from the kernel
2. **Pointers are essential for syscalls** - They allow the kernel to access your program's memory
3. **IN parameters point to data you provide** - The kernel reads from these locations
4. **OUT parameters point to space you allocate** - The kernel writes results to these locations
5. **Always allocate enough space for OUT parameters** - Buffer overflows cause crashes and security vulnerabilities
6. **Check return values** - Syscalls can fail; the return value typically indicates success or error

Understanding syscalls and pointer parameters reveals how all programs interact with the operating system. Every time you call `printf()` or `scanf()` in C, the library ultimately makes syscalls like `write` and `read`, passing pointers exactly as we've shown here!