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.

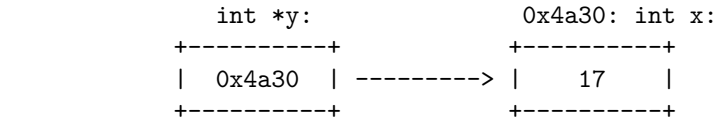
```c
#include <stdio.h>
void addOne(int *myvariable) {
  *myvariable = *myvariable + 1;
}
// Three ways to increment a variable, including pointers.
int main() {
  int x = 17;
  int *y = &x;
  x = x+1;
  printf("x: %d\n", x);
  *y = *y + 1;
  printf("x: %d\n", x);
  addOne(&x);
  printf("x: %d\n", x);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



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.

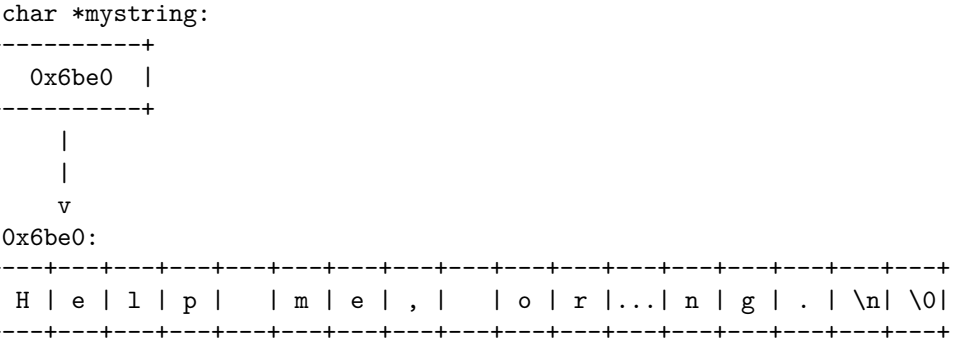
```c
#include <stdio.h>
int main() {
  char *mystring = "Help me, or I'm leaving.\n";
  // ALTERNATIVE:  char mystring[] = "...\n";
  printf("%s\n", mystring);
  printf("%s\n", mystring + 12);
  printf("%s\n", &mystring[12]);
  mystring[7] = '\0';
  printf("%s\n", mystring);
  return 0;
}
```

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**.



Understanding Pointer Arithmetic and the Null Terminator

- **Pointer Arithmetic:** The expression `mystring + 12` calculates a new address by moving 12 character-sized boxes forward from the start of the array.
- **Addressing Indices:** The expression `&mystring[12]` is functionally identical to `mystring + 12`, as both return the memory address of the character at index 12.
- **The Null Terminator (`\0`):** 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.

An Array 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.