Pointers

A Pointer is Simply a Memory Address

As you know, it’s often convenient
◦ to use **pointers** to values
  (memory addresses)
◦ rather than the values themselves.

Examples of use include
◦ arguments that can be modified,
◦ strings, and
◦ “events” (or any structured data).

Pointer Types Used in the Same Way as Primitive Types

In C,
◦ a **pointer to a type X**
◦ has type X*.

The following thus declare...

```c
int* iptr; // pointer to int, and
cchar* cptr; // pointer to char.
```

Note: read pointer types from right to left.

Declaring a Pointer Only Makes Space for a Pointer

```c
int* iptr; // iptr points to an int
```

Three important points about pointer types:
◦ **iptr is a memory address** (bits required
  depends on addressability of memory);
◦ **compiler** knows type and thus **can
  interpret bits at memory address iptr**;
◦ **if program needs storage for int**
  (something to which iptr might point),
  declare it separately.
**char* Used to Point to NUL-Terminated Strings**

```c
char* cptr = "My favorite string";
```

In C, a `char*` can point to a string, (or just to a single character in memory), but does not include space for the string. In declaration above, `cptr` is then written with ... *what?* ... the address of the letter 'M'.

**Pitfall: * Associates with Variable, Not Type**

If one declares variables in one line, as in

```c
int * A, B;
```

* A has type int*.

What about B?

* B has type int.

(Be careful, and be clear in your code.)

**Dereferencing Produces Value to Which Pointer Points**

C provides two operators for pointers:

* the dereference operator

    ```c
    *  
    ```

* the address operator

Dereferencing a pointer evaluates to the value to which the pointer points.

```c
char* cptr = "My favorite string";
```

For example, `*cptr` evaluates to 'M'.

**Pitfall: Avoid Condensing Expressions to Illegibility**

One cannot dereference a non-pointer type (meaningless, so compiler gives error).

Dereference and multiply use same character. Compiler chooses operator from context:

* dereference is unary: `*<a pointer>`, but

* multiplication is binary: `<expr> * <expr>`.

Write your code so that humans need not pretend to be compilers!

Example: `(A) * (B)`, not `A**B`
& Produces Address at Which Expression is Stored

- the address operator
You have used address operator with `scanf`.
Address operator evaluates to
  - the address of an expression
  - (usually a variable).
  ```c
  char* cptr = "My favorite string";
  ```
For example, `&cptr` evaluates to the address at which `cptr` is stored.

& Only Usable with Expressions that Have Addresses

```c
char* cptr = "My favorite string";
```
What about this one?
```c
&&cptr
```
&cptr not known to be stored anywhere, so expression above gives error.

However, `*(&cptr)` evaluates to 'M'.

Can Also Use Pointers to Pointers

```c
char* cptr = "My favorite string";
```
What if we want to store `&cptr`?
  - What is the type?
  - Pointer to pointer to char.
  (remember LDI/STI?)
So:
```c
char** cptr_ptr = &cptr;
```
And `**cptr` evaluates to what?
  - 'M'

Understanding Pointers is Critical

How useful are pointers?

- Rare to find anything but toy programs that does not use pointers (albeit hidden by many high-level languages).

How useful are pointers to pointers?

- Useful in a wide range of applications; you will use them often (but as above, you may not know it).
Don’t Overdo It: You Know What a Memory Address Is

How useful are pointers to pointers to pointers? I think I’ve seen them used.

How useful are pointers to pointers to pointers to pointers? Great tool for testing whether students understand pointers. Otherwise useless.

Example: Compare Two Strings

Let’s do an example.
Let’s compare two strings.

```c
// Return 1 if equal, 0 otherwise.
int string_equal (char* s1, char* s2);
```

String comparison
◦ is available in C’s standard library,
◦ but used to be a routine interview question
◦ to check whether the applicant had a clue.

Good Code for Comparing Two Strings?

```c
int string_equal (char* s1, char* s2)
{
    return (s1 == s2);
}
```

What do you think?
Maybe not what we want.

Code for Comparing Two Strings

```c
int string_equal (char* s1, char* s2)
{
    while (*s1 != *s2) { return 0; }
    s1++;
    s2++;
}
```

End of s1 yet?

```
if (*s1 != *s2) { return 0; }
```  

ASCII NUL in C

```
s1++;  // if characters differ...
s2++;  // ...strings also differ.
```

Advance string pointers.

Also at end of s2?

**Example Use of String Comparison Code**

What is printed by the code below?

```c
char* w = "word1";
char* x = "word2";
printf ("%d
", string_equal (w, x));
printf ("%s %s
", w, x);
```

First, let’s execute the function.

**Execution Example for String Comparison**

```c
while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
}
return ('\0' == *s2);
```

**Execute Loop and If Statement Tests**

```c
while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
}
```

**Advance s1 and s2 to Point to Next Characters**

```c
while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
}
```
Execute Loop and If Statement Tests

```c
while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
}
return ('\0' == *s2);
```

What is *s1? 'o' (not NUL)
What is *s2? 'o' (not NUL)
also 'o' (don't return 0)

Advance s1 and s2 to Point to Next Characters

```c
while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
}
return ('\0' == *s2);
```

char* w = "word1";
char* x = "word2";

Advance s1.
And advance s2.

```
```
Execute Loop and If Statement Tests

while ('\0' != *s1) {
    if (*s1 != *s2) { return 0; }
    s1++; s2++;
} return ('\0' == *s2);

What is *s1? 'd' (not NUL)
What is *s2? also 'd' (don't return 0)

Advance s1.
And advance s2.

Now We Know the First Line of Output

What is printed by the code below?
char* w = "word1";
char* x = "word2";
printf ("%d\n", string_equal (w, x));
printf ("%s %s\n", w, x);
first line of output: 0
What Does the Second `printf` Print?

What is printed by the code below?

```c
char* w = "word1";
char* x = "word2";
printf ("%d\n",
    string_equal (w, x));
printf ("%s %s\n", w, x);
```

First line of output: 0
Second line of output: word1 word2

---

Changes to Parameters Do Not Affect Caller's Variables

```c
printf ("%d",
    string_equal (w, x));

But `string_equal` changes `s1` and `s2`!

Why don’t `w` and `x` change?
Remember: C uses call by value.
Values of `w` and `x` are passed.

`w` and `x` cannot be changed.
But `*w` and `*x` can be changed...
```

---

Function Can Modify Bits at Addresses Passed by Value

```c
while (\'\0\' != *s1) {
    if (*s1 != *s2) {
        *s1 = *s2 = \'\0\';
        return 0;
    }
    s1++; s2++;
}
```

Add some new code!

---

What Does the Second `printf` Print Now?

```c
How does the change affect the output?
char* w = "word1";
char* x = "word2";
printf ("%d",
    string_equal (w, x));
printf ("%s %s\n", w, x);
```

First line of output: 0
Second line of output: word1 word2
Use `const` to Indicate Read-Only Behavior

```c
int string_equal(const char* s1, const char* s2) {
    while (*s1 != '\0') {
        if (*s1 != *s2) { return 0; }
        s1++;
        s2++;
    }
    return (*'\0' == *s2);
}
```

One last pointer topic: NULL pointers.

**What's the bug in this code?**

```c
int* ptr;
scanf("%d", ptr);
```

Hint: `ptr` has automatic storage class.

**What's in `ptr` when `scanf` is called?**

Bits.

Two Ways to Fix the Bug

Two ways to fix.
1. Our traditional way: don’t use pointers...
   ```c
   int value;
   scanf("%d", &value);
   ```
2. Declare an `int`, too:
   ```c
   int value;
   int* ptr = &value;
   scanf("%d", ptr);
   ```

Motivation for a Special Pointer Value: Point to Nothing

What if we want to initialize an `int*` pointer, but we don’t have an `int` yet?

Leave the `int*` filled with bits?

How can a C function tell that a pointer parameter points to nothing?

Generally, it can’t.

(Nearly any bit pattern can be a memory address.)
Using the 0 Bit Pattern for NULL Has Several Benefits

Define NULL as pointer that points to nothing.

Benefits (assuming initialization to NULL)

1. Compare with NULL to check for invalid pointers.
2. Use all 0 bit pattern (so a pointer is true if valid, false if not valid).
3. Dereferencing NULL on most systems* crashes the program.
   *Not true on many microcontrollers, however.

Pitfall: Mental Overload of Nullification

Keep in mind
- NUL is an ASCII character.
- NULL is a pointer (to nothing).
- "null" is an English word.
- 0 is a number.
They are all associated with 0 and bit patterns containing only 0s.
But they're not the same.*
Don't confuse them.
   *In some languages, “NULL” is written “null.” Go figure.