Pointers

CSC100 / Introduction to programming in C/C++

README

- This script introduces C pointers in theory and practice.
- Practice workbooks and input files in GDrive.
- PDF version of this file and of the completed practice workbooks is available in GitHub.
- This section, including some sample code, is based on: chapter 11 in King (2008), and chapter 7 in Davenport/Vine.

Overview

- Computer memory is like a list of *locations*
- Each chunk of memory has an *address* to a location

- *Pointers* point to these addresses

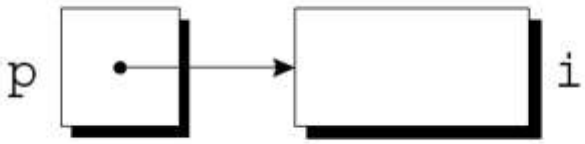

Figure 1: pointer p points to address of i

- The *address* is not the house, it's a *reference*

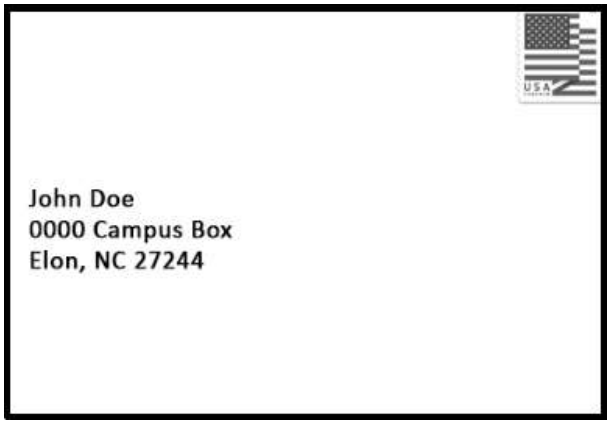

Figure 2: envelope = pointer to an address

- C#, Java, Pascal, Python…do not offer pointers (easily)[1]
- C and C++ offer pointer variables and operators naturally
- This gives you a lot more control over the computer

- Examples:

    - **String manipulation** (working with text - e.g. when creating

    fast-performing chat bots or AI agents)

    - **Dynamic memory allocation -** the process of assigning memory during the execution time (when a program typically competes with thousands of other processes)
- It's mind control: You can essentially decide what the computer should think with which part of its "brain" (great potential to mess up, too), e.g. when you mis-allocate resources.

Indirection (concept)

- Imagine you have a *variable* `iResult` that contains the *value* `75`.
- The variable is *located* at a memory address `0061FEC8`.
- Imagine you have a *pointer variable* `myPointer` that contains the address `0061FEC8` of the variable `iResult`.
- This means that `myPointer` *indirectly* points to the value `75`.

- You already worked with pointers: an *array name* `a` is a pointer to the start of the array, the address of `a[0]`.

```
int a[2] = {0,1};
printf("a[0]=%d\na[1]=%d\n&a=%d\n", a[0], a[1], &a);
```

```
a[0]=0
a[1]=1
&a=6422216
```

- You already worked with pointers: arguments in the call of `scanf~` are *pointers*: without the `&`, the function would be supplied with the *value* of `i`, not the *address*.

```
int i;
scanf("%d", &i);
```

Indirection (code)

- There are two *unary* pointer operators:
    - the *address* (or referencing) operator `&`
    - the *indirection* (or dereferencing) operator `*`
- The unary *address* operator `&` returns a computer memory address, e.g. `&iResult = 6422216` - it *references* the memory location

- The unary *indirection* operator `*` returns a value, e.g. `*myPointer = 75` if `myPointer` points at `&iResult`. T

```
int iResult; // declare an integer variable
iResult = 75; // iResult now has the value 75
```

```
int *myPointer; // declare an integer pointer variable
myPointer = &iResult; // myPointer points at iResult's address

printf("iResult = %d and lives at &iResult = %p\n",
       iResult, &iResult);

printf("myPointer = %p points to iResult = %d\n",
       myPointer, *myPointer);
```

```
iResult = 75 and lives at &iResult = 0061FEC8
myPointer = 0061FEC8 points to iResult = 75
```

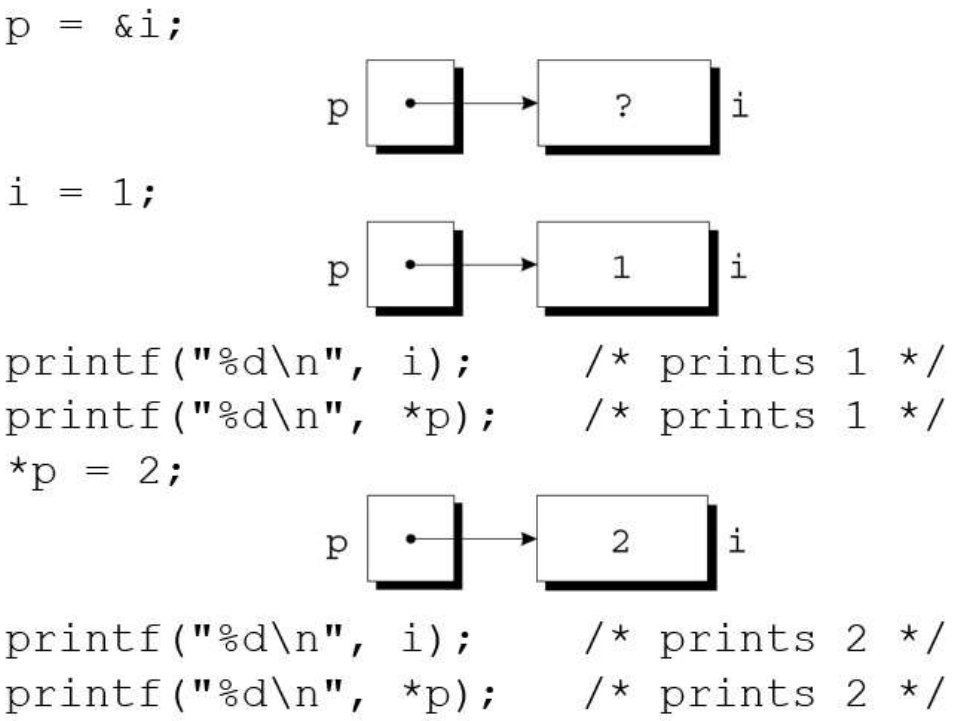

Figure 3: Graphical illustration of the indirection operator (Source: King)

* and & are inverse to one another

- Adress and indirection operator are *inverse* to one another (i.e. they reverse each other's operation - applying both amounts to doing nothing).

```
int iResult = 75, *myPointer = &iResult; // declaring and initializing

// print variable and dereferenced pointer
printf("iResult=%d => *&iResult=%d\n",
       iResult, *&iResult);
// print pointer and address of pointer
printf("myPointer=%p => &*myPointer=%p\n",
       myPointer, &*myPointer);
```

```
iResult=75 => *&iResult=75
myPointer=0061FEC8 => &*myPointer=0061FEC8
```

- Applying `&` to a variable produces a pointer to the variable
- Applying `*` to the pointer takes us back to the original variable

```
j = *&i  // same as j = i
```

Pointers must be initialized

- Not initialized pointers lead to invalid data or expressions.
- Pointer variables should always be initialized with:
  - another variable's memory address (e.g. `&i`), or
  - with 0, or
  - with the keyword `NULL`.
- Here are some *valid* pointer initializations - `printf` uses the format specifier `%p` for pointers.

```
int *ptr1;  // declarations
int *ptr2;
int *ptr3;
double x = 3.14; // initialize variable

ptr1 = &x; // initialize with address
ptr2 = 0;  // initialize with 0
ptr3 = NULL; // initialize with NULL

printf("%p %p %p\n", ptr1, ptr2, ptr3);
```

```
0061FEB8 00000000 00000000
```

- `[ ]`

  Here are a few non-valid initializations:

  - can you tell why?
  - can you right the wrongs?

```
int i = 5; // declare and initialize i
int *iPtr; // declare pointer iPtr

iPtr = i;   // wrong because...
iPtr = 7;   // wrong because...
```

- Solution:

```
int i = 5; //
int *iPtr;

iPtr = &i;   // pointer initialized with memory address
```

```
*iPtr = 7;   // value of i indirectly changed

printf("%p %p %d\n", iPtr, &i, i);
```

```
0061FEC8 0061FEC8 7
```

Let's practice!

- [ ] Head over to GDrive for the Emacs Org workbook `pointers_practice.org`. Download also the image file `indirection.png`.

References

- Davenport/Vine (2015) C Programming for the Absolute Beginner (3ed). Cengage Learning.
- Kernighan/Ritchie (1978). The C Programming Language (1st). Prentice Hall.
- King (2008). C Programming - A modern approach (2e). W A Norton. URL: knking.com.
- Orgmode.org (n.d.). 16 Working with Source Code [website]. URL: orgmode.org

Footnotes:

[1] Python e.g. is actually written in C - its default implementation is called CPython. However, in Python, usability was favored over machine performance, so pointers are not implemented at the user level. C underlies most of the much-used modern programming languages and their (internal) memory management.

Author: Marcus Birkenkrahe
Created: 2022-04-27 Wed 10:07