Pointers and Structures in C

CS 350: Computer Organization & Assembler Language Programming

Files: The zip file containing this pdf should also contain `basic_ptr1.c`, `basic_ptr2.c`, `swap.c`, `complex1.c`, `complex2.c`, and `complex3.c`

A. Why?

- C is an important ancestor of C++ and Java.
- Pointers are an efficient way to share large memory objects.
- Structures give us a way to define data values that contain named components.

B. Outcomes

After this lecture, you should

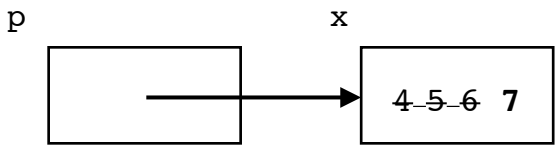
- Know how to declare, initialize, assign, and use pointer variables in C.
- Declare and use structures in C and define routines that take (pointers to) structure values.

C. Pointers in C

- A **pointer** is a memory address. A **pointer variable** is a variable whose value is an address. In C and C++ you can declare pointer variables, assign them values, and use them.
- **Declaration**: `int *p;` declares `p` to have the type pointer-to-integer.
- **Address-of operator**: If `x` is a variable, then `&x` is the address `x` is stored at. The assignment `p = &x;` makes "`p` point to `x`": The value of `p` is set to the address of `x`. You can take the `&` of anything that has an address (= can be assigned to = "has an **lvalue**"). E.g., `&17` is illegal; if `b` is an array then `&b[0]` (i.e, `&(b[0])`) is legal.
- **Printing addresses**: The `%p` format code is for addresses: if we set `p = &x;` then `printf("%p, %p", &x, p);` will print the same address twice.
- **Dereferencing operator**: If expression *e* is an address, then `*`*e* is the thing *e* points to. If `p` points to `x`, then the value of `p` is an address; the value of `*p` is the value

stored there. So `printf("%d, %d\n", x, *p);` prints out the value of `x` twice. Similarly `*p = 6;` sets `x` to 6.

```
// basic_ptr1.c
//
#include <stdio.h>
int main(void) {
    int x = 4;                  // integer variable
    int *p = NULL;              // pointer to integer
    // (NULL is used to point to nowhere)

    p = &x;                     // Make p point to x
    printf("%p %p\n", &x, p);   // print same address twice
    printf("%d %d\n", x, *p);   // Print 4 and 4

    x = 5;                      // Change x (and also *p)
    printf("%d %d\n", x, *p);   // Print 5 and 5

    *p = 6;                     // Change *p (and also x)
    printf("%d %d\n", x, *p);   // Print 6 and 6

    *p = *p+1;                  // x and *p = 7 (= 6+1)
    printf("%d %d\n", x, *p);   // Print 7 and 7

    return 0;
}
```



- **Dereferencing** `NULL`: Initializing `p` to `NULL` (a built-in constant) sets `p` but doesn't actually make it point to anything. But if you try to use `*p` when `p` is `NULL`, you'll get a runtime error. (Probably "Segmentation fault".)
- **Declarations; Initialization:** Pointer declaration syntax is tricky. If you have multiple pointers in one declaration, put `*` before each one: `int *p, *q;` . You can initialize a pointer with `int *p =` e_1`, *q =` e_2`;` but the expressions have to be of type pointer-to-int, so the types look weird.
  - **Example 1:** `int *p = &x;` means `int *p; p = &x;` not `int *p; *p = &x;` (which gets a type error)

  - **Example 2:** Similarly, `int *q = p;` is like `int *q; q = p;` not `int *q; *q = p;` (which gets another type error).

- **Pointer Comparisons:** You can compare two pointers for <, ==, etc. E.g., `&b[0] < &b[1]` yields true.
  - You can't compare a pointer and an integer.
- **Pointer Equality and Aliasing**: Two pointers are **aliased** if they are == (that is, they point to the same location) Note `p == q` implies `*p == *q` . On the other hand, even if `p != q`, it's possible for `*p == *q` . (I.e., `p` and `q` point to different locations that happen to contain the same value.)

```
// basic_ptr2.c
//
#include <stdio.h>
int main(void) {
    int x = 4, *p, *q;
    p = q = &x;    // Make p, q point to x
    printf("%p %p %p\n", p, q, &x);         // all the same address
    printf("%d %d %d\n", *p, *q, (p == q)); // 4 4 true

    int y = 4;
    q = &y;        // Make q point to y
    printf("%d %d %d\n", *p, *q, (p == q)); // 4 4 false

    return 0;
}
```

- Pointers to non-primitive data: In addition to pointers to the built-in primitive types of data (int, double, char, etc.), you can also have pointers to more complicated types of data. We'll see structures and pointers to structures in the next lecture, but we won't discuss pointers to pointers or pointers to arrays.

D. Passing Pointer Arguments as Parameters

- **Call-by-Reference** (**CBR**, a.k.a. pass-by-reference) is a style of function call where parameter variables are aliased to argument expressions (which must have lvalues). So changes to parameters cause changes to arguments.
- In C pass-by-reference is emulated using pointers. In the caller, we pass `&`*argument*. The subroutine takes pointer arguments; everywhere we want to refer to the argument, we use `*`*parameter*.

- Here's an example: The `swap` routine takes pointers to the variables whose values are to be swapped, and it works correctly. The `badswap` routine swaps the values of its two `int` parameters, which works fine within `badswap` but doesn't affect the `main` program.
- (Programming note: Since we're defining the bodies of `swap` and `badswap` after their uses in `main`, we declare their prototypes before the `main` program so that the compiler can be sure about what types of arguments / type of result they have. If we don't do this, we'll get compile-time errors.)

```
// swap.c: Emulate call-by-reference using pointers
//
#include <stdio.h>

// Function prototypes
void swap(int *v1, int *v2);
void badswap(int x1, int x2);

int main() {
    int x = 2, y = 3;
    printf("x and y are at %p and %p\n", &x, &y);
    printf("Before swap, x = %d, y = %d\n\n", x, y);

    swap(&x, &y);          // Now x == 3, y == 2
    printf("After swap, x = %d, y = %d\n\n", x, y);

    badswap(x, y);         // Doesn't affect x or y
    printf("Back in main, after bad swap, x = %d, y = %d\n", x, y);

    return 0;
}
void swap(int *v1, int *v2) {
    printf("Swapping values at %p and %p\n", v1, v2);
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
void badswap(int x1, int x2) {
    printf("Bad swap of values at %p and %p\n", &x1, &x2);
    printf("At start of bad swap, x1 = %d, x2 = %d\n", x1, x2);
    int temp = x1;
    x1 = x2;
    x2 = temp;
    printf("At end   of bad swap, x1 = %d, x2 = %d\n\n", x1, x2);
}
```


- The `scanf` routine is an example of a library routine that uses simulated call-by-reference: In `scanf("%d", &x);` we pass the address of `x` so that the `scanf` code can modify `x`.

E. Structures in C

- For combining different kinds of data into one logical (and physical) record, C uses "structures" ("structs" for short). They are similar to classes where all members are public data members, but they don't have constructors, member functions, interfaces, or inheritance. The fields of a `struct` are laid out consecutively in memory.
- **Example 3**: The sample program `complex1.c` contains the definition of a structure type for complex numbers (of the form $a + b\,i$) and shows how to declare and manipulate them. The `struct complex { … };` declaration says that a `complex` value has two fields `real_part` and `imag_part`. The main program declares a `struct complex val` and manipulates the fields of val using "dot" notation: `val.`*field*, where field is `real_part` or `imag_part`.

```
// complex1.c
//
#include <stdio.h>

struct complex {
    double real_part;
    double imag_part;
};

int main(void) {
    struct complex val;
    val.real_part = 1.1;
    val.imag_part = 2.2;
    printf("%f + %f i\n", val.real_part, val.imag_part);
    return 0;
}
// Output: 1.100000 + 2.200000 i
```

F. Functions With Structure Arguments

- If we declare a function that takes a `struct` parameter, then when we call the function, we'll actually copy all the fields of the actual argument to the parameter variable. (This can be expensive for large structures; `struct complex` isn't too bad, since it just contains two floating-point values.)

- In practice, to pass a structure to a function, we pass a pointer to it. This lets us avoid copying the whole structure to the called routine, and it also lets us pass structures by reference, so we can modify them in-place.
- **Example 4**: The sample program `complex2.c` declare the same structure as in `complex1.c`, but it uses functions to set and print complex values. The `set_cpx` function takes a pointer `x` to a struct `complex` and two double fields `a` and `b`, and it sets the real and imaginary fields of `*x` to `a` and `b`. The `cpx_print` routine takes a pointer `x` and prints out the two fields of the structure value `*x`.
- **C Syntax**: Because of the precedences involved, we need the parentheses in (e.g.) `(*x).real_part = a`. Without them, we get `*(x.real_part) = a`, which would typecheck only if `x` were a structure variable containing a field `real_part` that returns a pointer to `double`.

```
// complex2.c
//
#include <stdio.h>

struct complex {
    double real_part;
    double imag_part;
};  // <-- note semicolon here

// Prototypes
void set_cpx(struct complex *p, double a, double b);
void cpx_print(struct complex *x);

int main(void) {
    struct complex val, *x = &val;
    set_cpx(x, 1.1, 2.2);
    cpx_print(x);
    return 0;
}
// set_cpx(x, a, b) sets *x to a + bi
//
void set_cpx(struct complex *x, double a, double b) {
    (*x).real_part = a;
    (*x).imag_part = b;
}
// cpx_print(x) prints *x in a + bi format.
//
void cpx_print(struct complex *x) {
    printf("%f + %f i\n", (*x).real_part, (*x).imag_part );
}
```


G. Syntactic Abbreviations

- Because they come up so often, we can define an abbreviation for `struct complex` and for (`*`*ptr*)`.`*field*
- The declaration `typedef struct complex Complex;` lets us use `Complex` instead of `struct complex`
- The expression *ptr* `->` *field* means (`*`*ptr*)`.`*field* (The hyphen-greater-than is supposed to look like an arrow; the pointer points to a structure that has the requested field.) You can only use the `->` operator in the context of pointing to a structure; it's not an abbreviation for `*`*ptr* in general.
- **Example 5**: The sample program `complex3.c` is the same as `complex2.c` but uses these abbreviations:

```
// complex3.c
//
#include <stdio.h>

struct complex {
    double real_part;
    double imag_part;
};  // <-- note semicolon here
typedef struct complex Complex;

// Prototypes
void set_cpx(Complex *p, double a, double b);
void cpx_print(Complex *x);

int main(void) {
    Complex x_val, *x = &x_val;
    set_cpx(x, 1.1, 2.2);
    cpx_print(x);
    return 0;
}

void set_cpx(Complex *x, double a, double b) {
    x -> real_part = a;
    x -> imag_part = b;
}

void cpx_print(Complex *x) {
    printf("%f + %f i\n", x -> real_part, x -> imag_part );
}
```


Pointers and Structures in C

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Pointers are an efficient way to share large memory objects without copying them.

B. Outcomes

After this activity, you should

- Be able to hand-execute code that uses the `*` and `&` operators in C.

C. Questions

1. Draw a memory diagram for the state at the end of execution of

```
int w=3, z=4, *p, *q;
p = &w;
*p = z;
z = 5;
q = &z;
p = q;
*p = 6;
```

2. Draw a memory diagram for the state at the end of execution of

```
int x = 7, y = 10;
int *p = &x, *q = p;
*p = *q+1;
q = &y;
*q = *q+1;
```

3. Modify `cpx_print` so that
   (1) Instead of `0 + 0 i`, it just prints `0`
   (2) Instead of *a* `+ 0 i` (where $a \neq 0$), it just prints *a*
   (3) Instead of `0 +` *b* `i` (where $b \neq 0$), it just prints *b* `i`

4. Write some C code that establishes the memory diagram below. Declare `x` to be an array of `struct S` ; each S value has two fields: an integer `n` and a char `c`. (So

below, `5` is the value of the `n` field of `x[0]`.) Declare `p` so that it points to struct S values; `q` should point to integers. There exist multiple right answers.