C Structures & Pointers

CS 350: Computer Organization & Assembly Language Programming

A. Why?

- Structures give us a way to define data values that contain named components.
- Pointers to structures are an efficient way to pass around large structures.
- In C, to let a routine modify a structure value, we pass a pointer to the value.

B. Outcomes

After this lecture, you should know how to

- Declare a structure in C.
- Declare routines that take (pointers to) structure values.

C. 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**: The sample program `complex.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 `typedef` defines the id `Complex` to be the name of a type that is a structure where each value has two fields `real` and `imag`, both of type `double`.

```
typedef struct {
    double real;
    double imag;
} Complex;
```

- The main program starts by declaring a complex number `x`, setting it to $0 + i$, and printing it out. Note that to access the fields of `x`, we use dot notation: `cpx_value.real` and `cpx_value.imag`.


```
Complex cpx_value;
cpx_value.real = 0.0;
cpx_value.imag = 1.0;
printf("x.real = %f, x.imag = %f\n",
    cpx_value.real, cpx_value.imag );
```

- **Output**: `cpx_value: 0.000000 + 1.000000 i`

D. Functions With Structure Arguments

- To pass a structure to a function, we pass a pointer to it, not the actual structure. The pointer usually takes less space than the structure value; also, passing a pointer lets us change the structure fields in the function and have the changes seen in the caller.
- E.g., for `Complex` numbers, `complex.c` contains a `cpx_print` routine that takes a pointer to a `Complex` value and prints it out. Below
  - The parameter `p` is a pointer to a `Complex`
  - `*p` is a `Complex`
  - `(*p).real` and `(*p).imag` are the two fields of the value.
    - (The parentheses are necessary.)
  - Because `(*`*ptr*`).`*field* comes up so often, C supports an abbreviation for it: *ptr* `->` *field* (where `->` is hyphen greater-than).
- Here's the `cpx_print` routine:

```
// Print a complex number as (real + imag i)
//
void cpx_print(Complex *p) {
   printf("(%f + %f i)\n", p->real, p->imag);
}
```

- In the main program, we pass a pointer to the complex value to print:

```
int main() {
  Complex cpx_value;
  ...
  Complex *cpx;
  cpx = &cpx_value;
  cpx_print(cpx);
```



```
   ...
}
```

- For another example of a function that takes a structure parameter, here's a function that sets the fields of a complex value: `set_cpx(&x,`*a*`,`*b*`);` sets `x.real =` *a* and `x.imag =` *b*.

```
// set_cpx(&x, a, b) sets x to a + bi
//
void set_cpx(Complex *x, double a, double b) {
   x->real = a;
   x->imag = b;
}
```

E. Returning A Struct Result Is Harder

- With `set_cpx`, we pass the struct we want to change. What if we want a routine that returns a new struct value? We'd call it with something like

```
Complex *p;
p = make_cpx(0.0, 1.0);   // *p = 0 + 1 i
```

- It turns out to be tricker to create such a routine than you might think. Your first attempt might be

```
Complex * bad_make_cpx(double x, double y) {
   Complex result;
   result.real = x;
   result.imag = y;
   return &result;  // BUG
}

int main() {
   Complex *p;
   ...
   p = bad_make_cpx(1.1, 2.2);
}
```

- The problem with this routine is that it returns a pointer to a variable that's local to `bad_make_cpx`.
  - Space for a function's local variables is allocated on entry to the function and deallocated (returned to the OS) on return.

- If we return `&result`, then the caller will have a pointer to a bad memory location (it could be reallocated and overwritten by some other declaration).
- To write this kind of routine correctly, we'll have to use **dynamic storage allocation**; we'll look at this next class.

C Structures & Pointers I

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Structures give us a way to define data values that contain named components.
- In C, to let a routine modify a structure value, we pass a pointer to the value.

B. Outcomes

After this activity, you should be able to

- Write simple routines that use pointers to structures as parameters.

C. Questions

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

```
// Print a complex number as (real + imag i)
//
void cpx_print(Complex *p) {
   printf("(%f + %f i)\n", p->real, p->imag);
}
```

2. Complete the definition of `cpx_add(p, q, r)` below so that a call like `cpx_add(&x, &y, &z);` sets `x` = `y + z` (where `x`, `y`, and `z` are `Complex` values.

```
// cpx_add(&x,&y,&z) sets x = y+z for
// Complex x, y, and z.
//
void cpx_add(Complex *p, Complex *q, Complex *r) {
  ...
}
```

3. Write a `main` program that (1) Creates 3 complex numbers: *i*, -1, and 0, (2) Prints them out using `cpx_print`, (2) Adds *i* and -1 using `cpx_add`, and (3) Prints the result using `cpx_print`.

Solution

Here's everything wrapped up into one program.

```
#include <stdio.h>
typedef struct { double real; double imag; } Complex;

// Print a complex number as (real + imag i). Check for
// real or imaginary parts = 0 as special cases; print
// just real or imag i in those cases.  Print zero as 0.
//
void cpx_print(Complex *p) {
  if (p->imag == 0.0) {
    printf("%f\n", p->real);
  }
  else if (p->real == 0.0) {
    printf("%f i\n", p->imag);
  }
  else {
    printf("(%f + %f i)\n", p->real, p->imag);
  }
}

// cpx_add(r,x,y) sets r = x+y where r, x, and y are
// (pointers to) Complex numbers.
//
void cpx_add(Complex *r, Complex *x, Complex *y) {
  r->real = x->real + y->real;
  r->imag = x->imag + y->imag;
}

int main() {
  Complex a, b, c, *p=&a, *q=&b, *r=&c;
  a.real = 0.0;
  a.imag = 1.0;
  b.real = -1.0;
  b.imag = 0.0;
  c.real = c.imag = 0.0;
  cpx_print(p); cpx_print(q), cpx_print(r);
  cpx_add(r, p, q);
  cpx_print(r);
}
```

