Structures and Unions in C

CS 350: Computer Organization & Assembler Language Programming
Lecture 15

Note: Files: The zip file containing this pdf should also contain `complex1.c`, `complex2.c`, `complex3.c`, `union.c`, and `mips_union.c`.

Structures

- For combining different kinds of data into one logical (and physical) record, C uses “structures” (“structs” for short). They are similar to C++ 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` and `imag`. The main program declares a `struct complex val` and manipulates the fields of val using "dot" notation: `val.`*field*, where field is `real` or `imag`.

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

struct complex {
    double real;
    double imag;
};  // <-- note semicolon here

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

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 = a`. Without them, we get `*(x.real) = a`, which would typecheck only if `x` were a structure variable containing a field `real` that returns a pointer to `double`.)

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

struct complex {
    double real;
    double imag;
};  // <-- 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 = a;
    (*x).imag = b;
}
// cpx_print(x) prints *x in a + bi format.
//
void cpx_print(struct complex *x) {
    printf("%f + %f i\n", (*x).real, (*x).imag );
}
```

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;
      double imag;
};  // <-- 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 = a;
      x -> imag = b;
}

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

Representing MIPS Instructions Using Structures

- Here are the three kinds of instruction formats we've seen. (I'm omitting the two for floating-point instructions.)

Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	Type of Instruction
R-Format	opcode	rs	rt	rd	shamt	funct	Arithmetic, Logic
I-Format	opcode	rs	rt	immediate value (16 bits)			Branch, Immediate, Data Transfer
J-Format	opcode	target address (26 bits)					Jump

- It would be nice if we could (in C) declare a structure that gives us exact access to the fields of a 32-bit string, something like

```
struct R_Format { 6-bits opcode; 5-bits rs, rt, rd, shamt; 6-bits funct; }
```

- Unfortunately, we can't do that because we can't use "5 bits" or "6 bits" etc. as a datatype.

- We could use Java-style `get` and `set` functions and use only them to access and modify the different fields of an instruction. For example, for the opcode field we might have:

```
unsigned char get_opcode(unsigned int instruction) {
    ... build appropriate 6-bit mask for positions 26-31
    ... take & of mask and instruction
    ... shift result right to positions 0-5
    ... return the byte at positions 0-7 of instruction
}
// and similarly for
void set_opcode(unsigned int instruction, unsigned char new_opcode) {
    ... use appropriate mask and & to set positions 26-31 of
        instruction to 0's
    ... copy new opcode to a temporary 32-bit unsigned int and
        shift it left to positions 26-31
    ... shift new opcode left to positions 26-31
    ... take | of shifted opcode and instruction
}
```

- It's a bit awkward to use these.
- Another possibility is to use an intermediate format that represents an instruction without exactly matching the 32 bit string. We can't declare a 6- or 5-bit field, but we can declare a `unsigned char` (and waste 2 or 3 bits of space). For 16 bits, we can use an `short integer` which takes 16 bits, so there's no waste. For the 26-bit address field, we can use an `unsigned int` (32 bits, wastes 6 bits of space).
- For example,[1]

```
typedef unsigned char byte; // make "byte" mean "unsigned char"
struct R_Format { byte opcode, rs, rt, rd, shamt, funct; };
struct I_Format { byte opcode, rs, rt; short immediate; };
struct J_Format { byte opcode; unsigned int address; };
```

- We would still need functions to translate between 32-bit (unsigned int) values and the appropriate format, of course. In order to modify the fields of instruction, we use our usual pointer trick for simulating call-by-reference.

```
build_R_Format(unsigned int bitstring, struct R_Format *instruction) {
    ... calculate and set instruction -> opcode;
    ... calculate and set instruction -> rs;
    ... etc. [3/25]
}
```

- This all works fine if we just want one particular instruction of one format, but memory can contain a sequence of instructions where the instructions have of possibly-different formats. How can we handle this? One way is to define a jack-of-all-trades structure that has the possible fields for an instruction:

```
struct instr_all_fields {
    byte opcode, rs, rt, rd, shamt, funct;
    short immediate;
    unsigned int address;
};
```

[1] You can abbreviate `short int` to just `short` and `unsigned int` to just `unsigned`, if you like.

- For any particular format of instruction, we'd use the `opcode` field and some other fields but not all other fields.
  - For an R-format instruction, we would use the `rs`, `rt`, `rd`, `shamt`, and `funct` fields, but we wouldn't use the `immediate` or `address` fields.
  - For an I-format instruction, we would use the `rs`, `rt`, and `immediate` fields but not the `rd`, `shamt`, or `funct` fields.
  - For a J-format instruction, we would use the `address` field but not any of the others (except for `opcode`).
- This technique works -- we could create an array of these structures to model instructions sitting in memory, but we waste space with every instruction.

Unions vs Structs

- What we really want is a way to say "We have an `opcode` field and either (1) `rs`, `rt`, `rd`, `shamt`, and `funct` fields, or (2) `rs`, `rt`, and `immediate` fields, or (3) an `address` field."
- In C, the technique for declaring that we have only one of various choices for a piece of data is called `union`.
- Except for the keyword `union` vs `struct`, the two kinds declaration have the same syntax. They differ in how memory gets laid out.
  - The memory layout for `struct` fields is consecutive: first field, then second field, etc.
  - The memory layout for `union` fields uses **overlaying**: we have one piece of memory but call it by one of the different names.
  - For example, take `union i_or_f { int i; float f; }`;. If we have a variable `x` of type `i_or_f`, then the `x.i` field and `x.f` field start at the same location.
- **Example 6**: Here's a declaration for an `instruction` type for MIPS instructions. Note the opcode has been moved out of the R-, I-, and J-format structures and into the instruction structure. The test program will print the same address for the three union fields.

```
// mips_union.c
//
typedef unsigned char byte; // declare "byte" to mean "unsigned char"
struct R_Format { byte rs, rt, rd, shamt, funct; };
struct I_Format { byte rs, rt; short immediate; };
struct J_Format { unsigned int address; };

struct instruction {
    byte opcode;
    union {
        struct R_Format r_fmt;
        struct I_Format i_fmt;
        struct J_Format j_fmt;
    };
};
```



```
int main(void) {
    struct instruction instr;
    printf("Address of instr.opcode:          %p\n", &instr.opcode);
    printf("Address of instr.r_fmt:           %p\n", &instr.r_fmt);
    printf("Address of instr.i_fmt:           %p\n", &instr.i_fmt);
    printf("Address of instr.j_fmt:           %p\n", &instr.j_fmt);
    printf("\n");
    printf("Address of instr.r_fmt.rs:        %p\n", &instr.r_fmt.rs);
    printf("Address of instr.r_fmt.rt:        %p\n", &instr.r_fmt.rt);
    printf("Address of instr.r_fmt.rd:        %p\n", &instr.r_fmt.rd);
    printf("Address of instr.r_fmt.shamt:     %p\n", &instr.r_fmt.shamt);
    printf("Address of instr.r_fmt.funct:     %p\n", &instr.r_fmt.funct);
    printf("\n");
    printf("Address of instr.i_fmt.rs:        %p\n", &instr.i_fmt.rs);
    printf("Address of instr.i_fmt.rt:        %p\n", &instr.i_fmt.rt);
    printf("Address of instr.i_fmt.immediate: %p\n",&instr.i_fmt.immediate);
    printf("\n");
    printf("Address of instr.j_fmt.address:   %p\n",&instr.j_fmt.address);
    return 0;
}

/* OUTPUT:
Address of instr.opcode:          0x7fff5893b910
Address of instr.r_fmt:           0x7fff5893b914
Address of instr.i_fmt:           0x7fff5893b914
Address of instr.j_fmt:           0x7fff5893b914

Address of instr.r_fmt.rs:        0x7fff5893b914
Address of instr.r_fmt.rt:        0x7fff5893b915
Address of instr.r_fmt.rd:        0x7fff5893b916
Address of instr.r_fmt.shamt:     0x7fff5893b917
Address of instr.r_fmt.funct:     0x7fff5893b918

Address of instr.i_fmt.rs:        0x7fff5893b914
Address of instr.i_fmt.rt:        0x7fff5893b915
Address of instr.i_fmt.immediate: 0x7fff5893b916

Address of instr.j_fmt.address:   0x7fff5893b914
*/
```


Structures and Unions in C

CS 350: Computer Organization & Assembler Language Programming

Why?

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

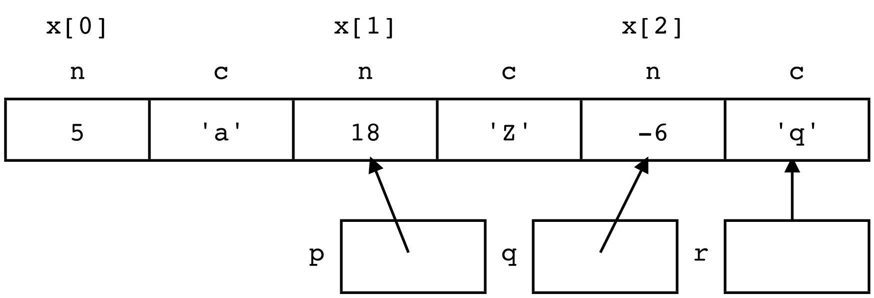
Outcomes

After this activity, you should

- Be able to hand-execute code that uses the `*` and `&` operators in C.
- Be able to compare the notions of structures and unions and when to use one over the other.

Questions

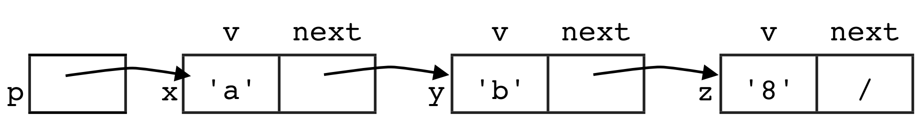
1. Modify `cpx_print` so that
   (1) Instead of `0 + 0 i`, it just prints `0`
   (2) Instead of a + `0 i` (where $a \neq 0$), it just prints a
   (3) Instead of `0` + b `i` (where $b \neq 0$), it just prints b `i`
2. Write some C code that establishes the memory diagram below. Declare `x` to be an array 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.



3. Write a diagram showing the state of memory after executing

```
struct S {int n; struct S *p;};
struct S x, y;
x.n = 1;
x.p = &y;
x.p -> n = 2;
y.p = NULL;
```

4. Write some code to declare a structure `T` and values `x`, `y`, and `z` of type `struct T` and a pointer `p` to `struct T` that sets up the following memory diagram



5. (Review) How are structures and unions similar? How do they differ? Why use one over the other?

Activity 15 Solution

1. (Modified `cpx_print`)

```
void cpx_print(Complex *x) {
    if (x -> imag == 0.0 & x -> real == 0.0)
        printf("0\n");
    else if (x -> 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);
}
```

2. (Code for memory diagram)

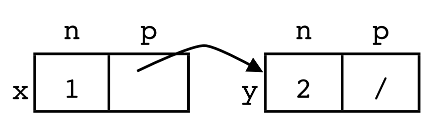
```
struct S { int n; char c; };
struct S x[3];
x[0].n = 5;
x[0].c = 'a';
x[1].n = 18;
x[1].c = 'Z';
x[2].n = -6;
x[2].c = 'q';

struct S *p = NULL;
int *q = NULL;
char *r = NULL;

p = &x[1];
q = &x[2].n;
r = &x[2].c;
```

There are many other possible answers, such as

```
p = x+1;
q = (int *) (p+1);
r = &(p[1].c);
```

The expression `(int *)` `(p+1)` is a **cast** that tells the compiler to treat the value of `p+1` as being the address of an integer. Since the address of a structure value is the address of its first field, `p+1` and `&((p+1).n)` are equal, so there's no problem at runtime. In general, overriding the compiler this way means the compiler's type-checker can be wrong if it says that your program is type-correct.

3. (Memory after code)



4. (Code for memory diagram)

```
struct T {char v; struct T *next;};
struct T x, y, z, *p;
p = &x;
x.next = &y;
y.next = &z;
z.next = NULL;
x.v = 'a';
y.v = 'b';
z.v = '8';
```

There are other possibilities. E.g., since `p == &x`, we can also use `(*p).v = 'a';` and `(*p).next = &y;` which in turn can be written as `p -> v = 'a';` and `p -> next = &y;` Even more complicated code works, such as `p -> next -> next -> v = '8';`

5. Structures and unions both declare collections of data and give them field names. Structure fields are laid out consecutively in memory; union fields all overlap the same memory. Structures are used when you have multiple data fields simultaneously; unions are used when you have only one data field at a time.