Structures and Unions in C

Alan L. Cox
alc@rice.edu
Administrivia

Assignment 1 is due tonight

Textbook

• Lectures begin covering material that is also covered by the textbook on 1/30
• Assignment 3 (assigned 2/1) requires use of the textbook
Objectives

Be able to use compound data structures in programs

Be able to pass compound data structures as function arguments, either by value or by reference

Be able to do simple bit-vector manipulations
Structures

Compound data:

A date is

* an int month and
* an int day and
* an int year

Unlike Java, C doesn’t automatically define functions for initializing and printing …

```
struct ADate {
    int month;
    int day;
    int year;
};

struct ADate date;

date.month = 1;
date.day = 18;
date.year = 2018;
```
Structure Representation & Size

sizeof(struct ...) =

sum of sizeof(field)

+ alignment padding

Processor- and compiler-specific

```
struct CharCharInt {
    char  c1;
    char  c2;
    int   i;
} foo;
```

```
foo.c1 = 'a';
foo.c2 = 'b';
foo.i  = 0xDEADBEEF;
```

<table>
<thead>
<tr>
<th>c1</th>
<th>c2</th>
<th>padding</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>61</td>
<td>62</td>
<td></td>
<td>EF BE AD DE</td>
</tr>
</tbody>
</table>

x86 uses “little-endian” representation
Typedef

Mechanism for creating new type names

- New names are an alias for some other type
- *May* improve clarity and/or portability of the program

```c
typedef long int64_t;
typedef struct ADate {
    int month;
    int day;
    int year;
} Date;

int64_t i = 1000000000000;
Date d = { 1, 18, 2018 };
```

Overload existing type names for clarity and portability

Simplify complex type names
Constants

Allow consistent use of the same constant throughout the program

• Improves clarity of the program
• Reduces likelihood of simple errors
• Easier to update constants in the program

Preprocessor directive

#define SIZE 10

Constant names are capitalized by convention

Define once, use throughout the program

int array[10];
for (i=0; i<10; i++) {
    ...
}

int array[SIZE];
for (i=0; i<SIZE; i++) {
    ...
}
Arrays of Structures

```c
Date birthdays[NFRIENDS];

bool check_birthday(Date today) {
    int i;
    for (i = 0; i < NFRIENDS; i++) {
        if ((today.month == birthdays[i].month) &&
            (today.day == birthdays[i].day))
            return (true);
    }
    return (false);
}
```
Pointers to Structures

```c
Date
create_date1(int month,
             int day,
             int year)
{
    Date d;
    d.month = month;
    d.day   = day;
    d.year  = year;
    return (d);
}

void
create_date2(Date *d,
             int month,
             int day,
             int year)
{
    d->month = month;
    d->day   = day;
    d->year  = year;
}
```

Copies date
Pass-by-reference

```c
Date today;
today = create_date1(1, 18, 2018);
create_date2(&today, 1, 18, 2018);
```
void create_date2(Date *d, int month, int day, int year)
{
    d->month = month;
    d->day   = day;
    d->year  = year;
}

void fun_with_dates(void)
{
    Date today;
    create_date2(&today, 1, 18, 2018);
}
```c
Date *
create_date3(int month,
    int day,
    int year)
{
    Date *d;
    d->month = month;
    d->day = day;
    d->year = year;
    return (d);
}
```

What is d pointing to?!?!
(more on this later)
Abstraction in C

From the #include file widget.h:

```c
struct widget;
struct widget *widget_create(void);
int            widget_op(struct widget *widget, int operand);
void           widget_destroy(struct widget *widget);
```

From the file widget.c:

```c
#include "widget.h"

struct widget {
    int x;
    ...
};
```
Collections of Bools (Bit Vectors)

Byte, word, ... can represent many Booleans

One per bit, e.g.,

00100101 = false, false, true, ..., true

Bit-wise operations:

Bit-wise AND: 00100101 & 10111100 == 00100100
Bit-wise OR: 00100101 | 10111100 == 10111101
Bit-wise NOT: ~ 00100101 == 11011010
Bit-wise XOR: 00100101 ^ 10111100 == 10011001
Operations on Bit Vectors

const unsigned int low_three_bits_mask = 0x7;
unsigned int bit_vec = 0x15;

A *mask* indicates which bit positions we are interested in:

0...00 0111 == 0...01 0101 & 0...00 0111

Always use C’s *unsigned* types for bit vectors.

Selecting bits:

```
important_bits = bit_vec & low_three_bits_mask;
```

Result = ?

0...00 0101 == 0...01 0101 & 0...00 0111
Operations on Bit Vectors

const unsigned int low_three_bits_mask = 0x7;  \[0...00 \ 0111\]
unsigned int bit_vec = 0x15;  \[0...01 \ 0101\]

Setting bits:

\(\text{bit_vec} \mid= \text{low\_three\_bits\_mask};\)

\(\text{Result} = ?\)

\(0...01 \ 0111 \ \text{==} \ 0...01 \ 0101 \ | \ 0...00 \ 0111\)
Operations on Bit Vectors

const unsigned int low_three_bits_mask = 0x7;
unsigned int bit_vec = 0x15;

Clearing bits:

bit_vec &= ~low_three_bits_mask;

Result = ?

0...01 0000 == 0...01 0101 & ~0...00 0111
Bit-field Structures

Special syntax packs structure values more tightly

Similar to bit vectors, but arguably easier to read

- Nonetheless, bit vectors are more commonly used.

Padded to be an integral number of words

- Placement is compiler-specific.

```
struct Flags {
    int f1:3;
    unsigned int f2:1;
    unsigned int f3:2;
} my_flags;

my_flags.f1 = -2;
my_flags.f2 = 1;
my_flags.f3 = 2;
```
Unions

Choices:

An element is
  • an int i or
  • a char c

sizeof(union ...) =
  maximum of sizeof(field)

union AnElt {
  int   i;
  char  c;
} elt1, elt2;

elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;

<table>
<thead>
<tr>
<th>c</th>
<th>padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF</td>
<td>BE</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
Unions

A union value doesn’t “know” which case it contains

```c
union AnElt {
    int   i;
    char  c;
} elt1, elt2;
```

```c
elt1.i = 4;
elt2.c = 'a';
elt2.i = 0xDEADBEEF;
```

```c
if (elt1 currently has a char) ...
```

How should your program keep track whether `elt1`, `elt2` hold an `int` or a `char`?

Basic answer: Another variable holds that info
Tagged Unions

*Tag* every value with its case

I.e., pair the type info together with the union

Implicit in Java, Scheme, ML, …

```c
enum Union_Tag { IS_INT, IS_CHAR };
struct TaggedUnion {
    enum Union_Tag tag;
    union {
        int   i;
        char  c;
    } data;
};
```

- Enum must be external to struct, so constants are globally visible.
- Struct field must be named.
Next Time

Memory Allocation