STRUCTURES AND UNIONS IN C

SHAILESH S
ASST. PROFESSOR
DEPT. OF COMPUTER SCIENCE
SH COLLEGE

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` <u>and</u>
- an `int day` <u>and</u>
- an `int year`

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

struct ADate date;

date.month = 1;
date.day = 18;
date.year = 2018;
```

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

STRUCTURE REPRESENTATION

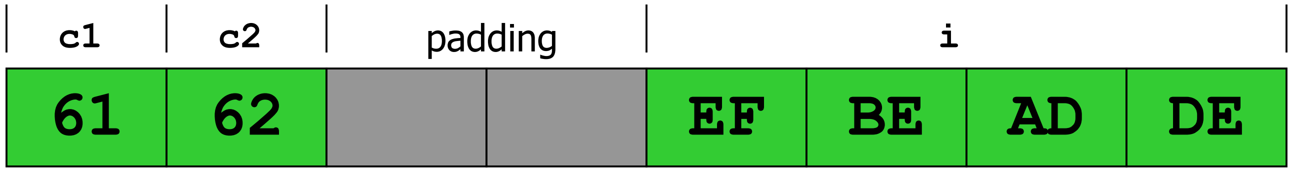
`sizeof(struct …)` =

  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;
```



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

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

int64_t i = 100000000000;
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

Constant names are capitalized by convention

```
int array[10];

for (i=0; i<10; i++) {
   …
}
```

```
#define SIZE 10

int array[SIZE];

for (i=0; i<SIZE; i++) {
   …
}
```

Define once, use throughout the program

ARRAYS OF STRUCTURES

Array declaration

Constant

Array index, then structure field

```
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

```
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;
}
```

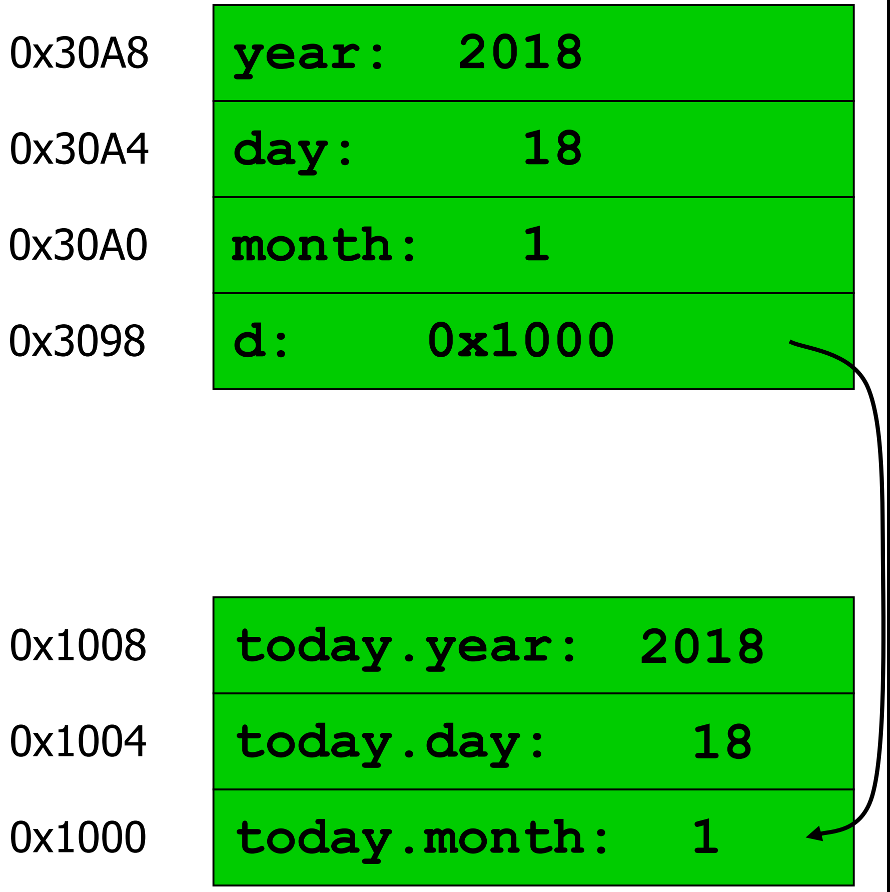
Pass-by-reference

```
Date today;

today = create_date1(1, 18, 2018);
create_date2(&today, 1, 18, 2018);
```

Copies date

POINTERS TO STRUCTURES

```
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);
}
```



POINTERS TO STRUCTURES

```
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)

COLLECTIONS OF BOOLS

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;
```

0…00 0111
0…01 0101

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

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;
unsigned int        bit_vec = 0x15;
```

```
0…00 0111
0…01 0101
```

Setting bits:

```
bit_vec |= low_three_bits_mask;
```

Result = ?

```
0…01 0111 == 0…01 0101 | 0…00 0111
```

OPERATIONS ON BIT VECTORS

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

0…00 0111

0…01 0101

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

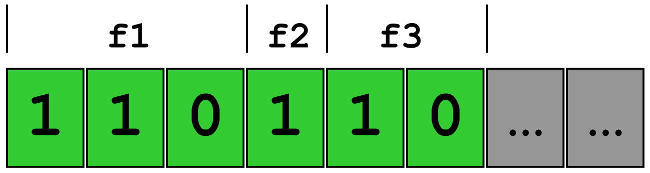
Similar to bit vectors, but arguably easier to read

- 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:

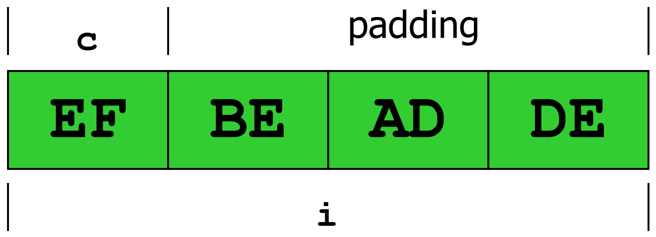
An element is

- 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;
```



UNIONS

A union value doesn't "know" which case it contains

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

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

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, …

```
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.

THANK YOU