Week 03

Data Representation (cont)

The story so far ...

2/36

We have looked at:

- character data
  - ASCII ... 7-bit encoding of English alphabet
  - UTF8 ... 7,11,16,21 bit encoding of all languages
- integer data
  - `unsigned int` ... 32-bit encoding of $0..2^{32}-1$
  - `int` ... 32-bit twos-complement encoding of $-2^{31}..2^{31}-1$

Pointers

3/36

Pointers represent memory addresses/locations

- number of bits depends on memory size, but typically 32-bits
- data pointers reference addresses in *data*/*heap*/*stack* regions
- function pointers reference addresses in *code* region

Many kinds of pointers, one for each data type, but

- `sizeof(int *) = sizeof(char *)`
  `= sizeof(double *) = sizeof(struct X *)`

Pointer *values* must be appropriate for data type, e.g.

- `(char *)` ... can reference any byte address
- `(int *)` ... must have *addr* `%4 == 0`
- `(double *)` ... must have *addr* `%8 == 0`

Exercise 1: Valid Pointers

4/36

Which of the following are likely to be valid pointers

```
0x00000000   0x00001000   0x00001001
0x7f000000   0x7f000001   0x7f000004
```

to objects of type

- `char`
- `int`
- `unsigned int`
- `double`
- `int (*f)()`

... Pointers

5/36

Can "move" from object to object by *pointer arithmetic*

For any pointer *T* `*p;`, `p++` increases `p` by `sizeof(`*T*`)`

Examples (assuming 16-bit pointers):

```
char   *p = 0x6060;  p++;  assert(p == 0x6061)
int    *q = 0x6060;  q++;  assert(q == 0x6064)
double *r = 0x6060;  r++;  assert(r == 0x6068)
```

A common (efficient) paradigm for scanning a string

```
char *s = "a string";
char *c;
// print a string, char-by-char
for (c = s; *c != '\0'; c++) {
   printf("%c", *c);
}
```

Exercise 2: Sum an array of `ints`

6/36

Write a function

```
int sumOf(int *a, int n)  { ... }
```

to sum the elements of array `a[]` containing `n` values.

Implement it two ways:

- using the "standard" approach with an index
- using a pointer that scans the elements

Floating Point Numbers

7/36

Floating point numbers model a (tiny) subset of $\mathbb{R}$

- many real values don't have exact representation (e.g. 1/3)
- results of calculations may contain small inaccuracies

Precision categorises how close to exact

- numbers close to zero have higher precision (more accurate)
- numbers further from zero have lower precision (less accurate)

... Floating Point Numbers

8/36

C has two floating point types

- **`float`** ... typically 32-bit quantity (lower precision, narrower range)
- **`double`** ... typically 64-bit quantity (higher precision, wider range)

Literal floating point values: `3.14159`, `1.0/3`, `1.0e-9`

Display via printf

```
printf("%W.Pf", (float)2.17828)
printf("%W.Plf", (double)2.17828)
```

`W` gives total width (blank padded), *`P`* gives #digits after dec point

```
printf("%10.4lf", (double)2.718281828459);
displays ␣␣␣␣2.7183
printf("%20.20lf", (double)4.0/7);
displays 0.57142857142857139685
```

... Floating Point Numbers

9/36

IEEE 754 standard ...

- scientific notation with *fraction F* and *exponent E*
- numbers have form $F \times 2^E$, where both *F* and *E* can be -ve

- `INFINITY` = representation for ∞ and -∞ (e.g. 1.0/0)
- `NAN` = representation for invalid value NaN (e.g. sqrt(-1.0))
- 32-bit single-precision, 64-bit double precision

Fraction part is *normalised* (i.e. 1.2345×10^2 rather than 123.45)

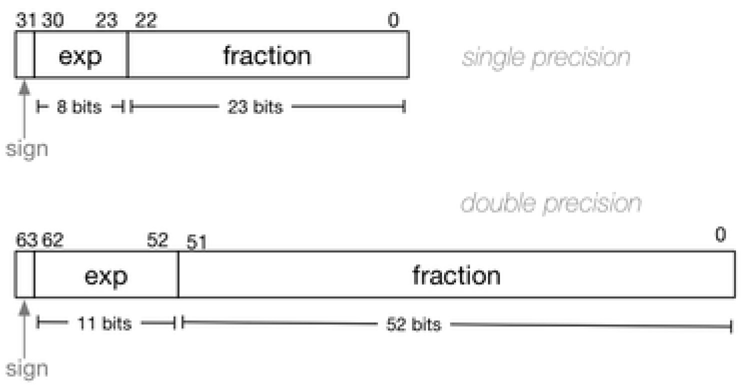
Example of normalising in binary:

- `1010.1011` is normalized as `1.0101011`$\times 2^{011}$
- `1010.1011` = 10 + 11/16 = 10.6875
- `1.0101011`$\times 2^{011}$ = (1 + 43/128) * 2^3 = 1.3359375 * 8 = 10.6875

... Floating Point Numbers

10/36

Internal structure of floating point values



More complex than `int` because *1.dddd e dd*

... Floating Point Numbers

11/36

Details of internal structure

- fraction part is always `1.`*bbbbbbbb*; don't store `1`
- exponent is offset relative to a baseline $-2^{b-1}-1$, b = #exponent bits

Ranges of values for 32-bit single-precision float:

Component	Min Value	Max Value
exponent	`00000000` = -127	`11111111` = 128
fraction	`00...00` = 0	`11...11` = $2^{-1}+2^{-2}+...+2^{-24}$

`00...00` = 24 zero bits, `11...11` = 24 one bits

Ranges of values for 64-bit double-precision float:

Component	Min Value	Max Value
exponent	`00000000000` = -2047	`11111111111` = 2048
fraction	`00...00` = 0	`11...11` = $2^{-1}+2^{-2}+...+2^{-51}$

`00...00` = 24 zero bits, `11...11` = 24 one bits

... Floating Point Numbers

12/36

Example (single-precision):

```
150.75 = 10010110.11
         // normalise fraction, compute exponent
       = 1.001011011 × 2^7
         // determine sign bit,
         // map fraction to 24 bits,
         // map exponent relative to baseline
       = 01000011000101101100000000000000
```

where red is sign bit, green is exponent, blue is fraction

Note:

- the baseline (aka bias) is 127, the exponent is 2^7
- so, in the exponent, we store 127+7 = 134 = `10000110`

Exercise 3: Floating point → Decimal

13/36

Convert the following floating point numbers to decimal.

Assume that they are in IEEE 754 single-precision format.

```
0 10000000 11000000000000000000000
```

```
1 01111110 10000000000000000000000
```

Arrays

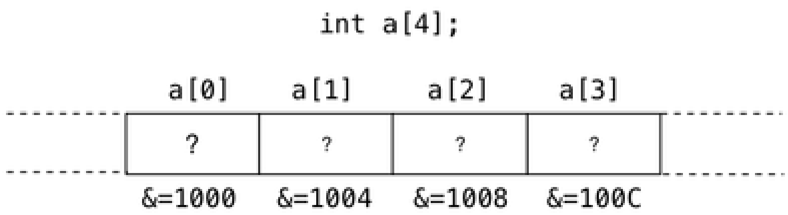
14/36

Arrays are defined to have *N* elements, each of type *T*

Examples:

```
int    a[100];    // array of 10 ints
char   str[256];  // array of 256 chars
double vec[100];  // array of 100 doubles
```

Elements are laid out adjacent in memory



... Arrays

15/36

Assuming an array declaration like *Type* `v[` *N* `]` ...

- individual array elements are accessed via indices 0..*N-1*
- total amount of space allocated to array *N* × `sizeof(` *Type* `)`

Name of array gives address of first element (e.g. `v` = `&v[0]`)

Name of array can be treated as a pointer to element type *Type*

Array indexing can be treated as `v[i]` ≅ `*(v+i)`

If have pointer to first element, can use it just like an array

Strings are just arrays of `char` with a `'\0'` terminator

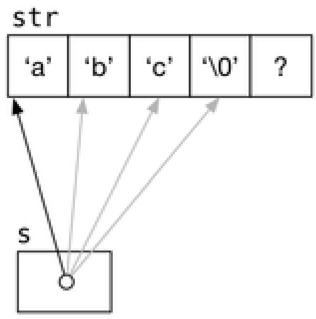
- constant strings have `'\0'` added automatically
- string buffers must allow for element to hold `'\0'`

... Arrays

16/36

When arrays are "passed" to a function, actually pass `&a[0]`

```
int main(void)
{
   char str[5] = "abc";
   f(str);
}

void f(char *s)
{
   while (*s != '\0')
      printf("%c", s++);
}
```



Exercise 4: Initialising Strings

17/36

Explain the difference between the following initialisers:

```
char a[9] = "a string";

char *b   = "a string";

char c[9] = {'a',' ','s','t','r','i','n','g','\0'};

char *d = {'a',' ','s','t','r','i','n','g','\0'};
```

... Arrays

18/36

Arrays can be created automatically or via `malloc()`

```
int main(void)
{
    char str1[9] = "a string";
    char *str2;  // no array object yet

    str2 = malloc(20*sizeof(char));
    strcpy(str2, str);
    printf("&str1=%p, str1=%s\n", str1, str1);
    printf("&str2=%p, str2=%s\n", str2, str2);

    free(str2);
    return 0;
}
```

Two separate arrays (different &'s), but have same contents

(except for the unitialised parts of the arrays)

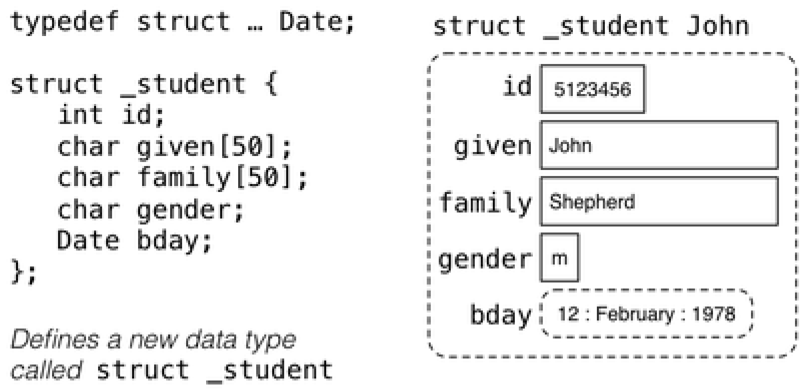
Structs

19/36

Structs are defined to have a number of components

- each component has a *Name* and a *Type*

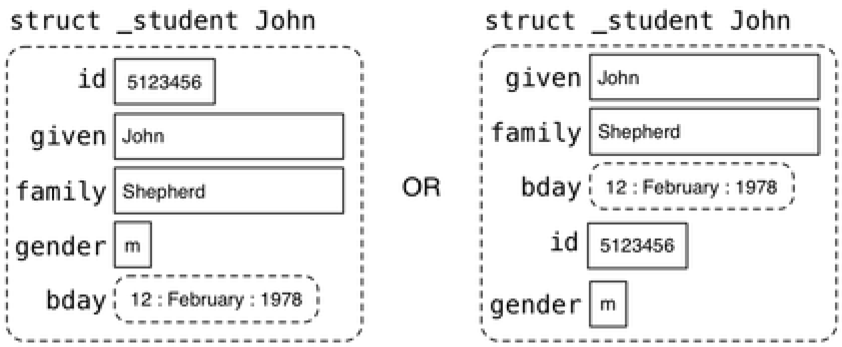
Example:



... Structs

20/36

Internal layout of `struct` components determined by compiler



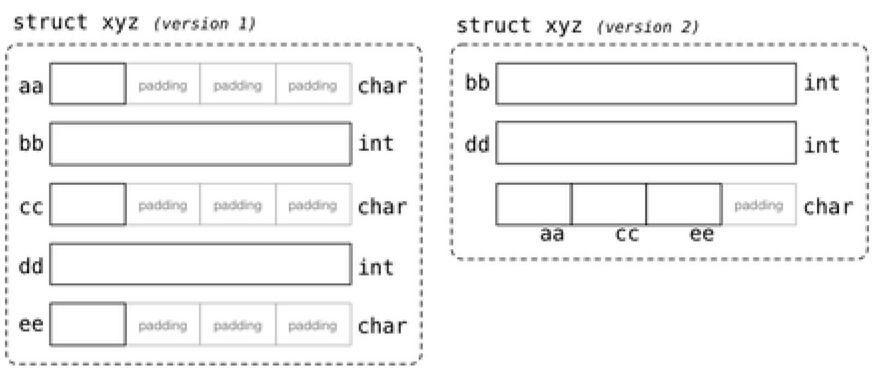
Each name maps to a byte offset within the `struct`

E.g. in first example `id` = offset 0, `given` = offset 4, `family` = offset 54, etc.

... Structs

21/36

To ensure alignment, internal "padding" may be needed



Padding wastes space; re-order fields to minimise waste.

`clang` has `-Wpadded` to warn about padding in `struct`s

Exercise 5: Struct Internal Layout

22/36

Determine the offsets of the fields in

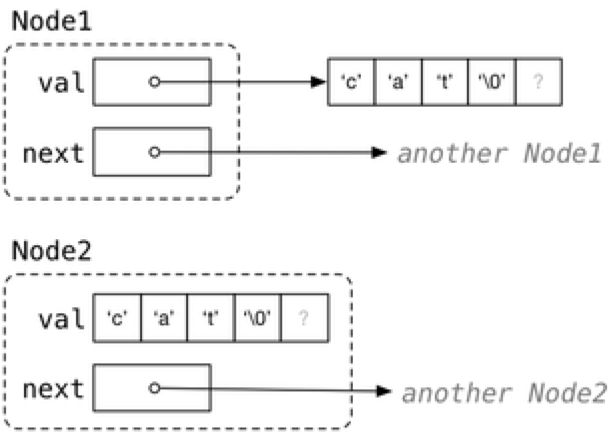
```
struct _s1 {
   char    a[6];  // array of 6 1-byte chars
   int     b;     // 4-byte int
   char    c;     // 1-byte char
```

```
    double d;      // 8-byte int
    int    e;      // 4-byte int
    char   f;      // 1-byte char
};
```

Exercise 6: Struct Alternatives

23/36

Consider these two possible representations of Nodes in a linked list of strings:



Show how they would be (a) defined, (b) initialised. How large is each?

Variable-length Structs

24/36

`Structs` can contain pointers to dynamic objects

But we can also "embed" one dynamic object in a malloc'd `struct`

- define the dynamic object as the last component
- malloc() more space than the `struct` requires
- to make the final component as large as required

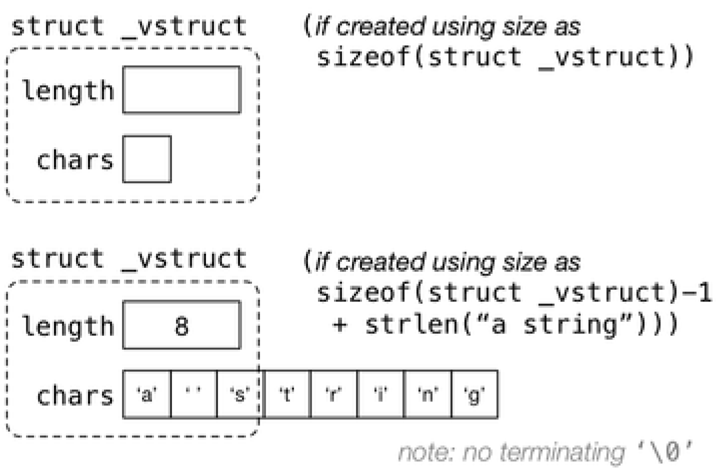
Example:

```
struct _vstruct {
    int  length;
    char chars[1]; // array whose real length
                   // is calculated when malloc'ing
}
```

... Variable-length Structs

25/36

Amount of memory allocated to `struct` is determined dynamically:



Bit-wise Structs

26/36

For fine-grained control over layout of fields in `structs`

- C allows programmers to specify `structs` bit-wise

Bit-field `structs` ...

- specify unnamed components using standard types
- specify named individual bit fields in each component

Example:

```
struct _bit_fields {
   unsigned int first_bit    : 1,
                next_7_bits  : 7,
                last_24_bits : 24;
};
```

Has one component and three bit fields within that component.

... Bit-wise Structs

27/36

Two ways of declaring bit fields:

```
struct _bit_fields {
   unsigned int first_bit    : 1,
                next_7_bits  : 7,
                last_24_bits : 24;
};
OR
struct _bit_fields {
   unsigned int first_bit    : 1;
   unsigned int next_7_bits  : 7;
   unsigned int last_24_bits : 24;
};
```

In both cases, `sizeof(struct _bit_fields)` is 4 bytes.

First way makes it clearer that a single `unsigned int` is used.

... Bit-wise Structs

28/36

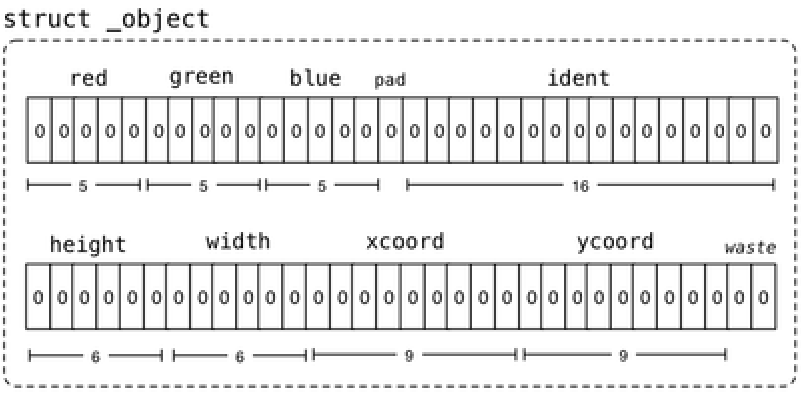
Another example (graphics objects):

```
struct _object { // comprised of two 32-bit words
   unsigned int  red    : 5,  // 5 bits for red
                 blue   : 5,  // 5 bits for blue
                 green  : 5,  // 5 bits for green
                 pad    : 1,  // 1 bit to pad to short
                 ident  : 16; // 16 bits for object ID
   unsigned int  height : 6,  // 6 bits for object height
                 width  : 6,  // 6 bits for object width
                 xcoord : 9,  // 9 bits for x-coordinate
                 ycoord : 9;  // 9 bits for y-coordinate
};

struct _object oval;
...
oval.red = 4; oval.blue = 31; oval.green = 15;
oval.height = 5; oval.width = 15;
```

... Bit-wise Structs

29/36

The graphics object would be stored in memory as:



... Bit-wise Structs

30/36

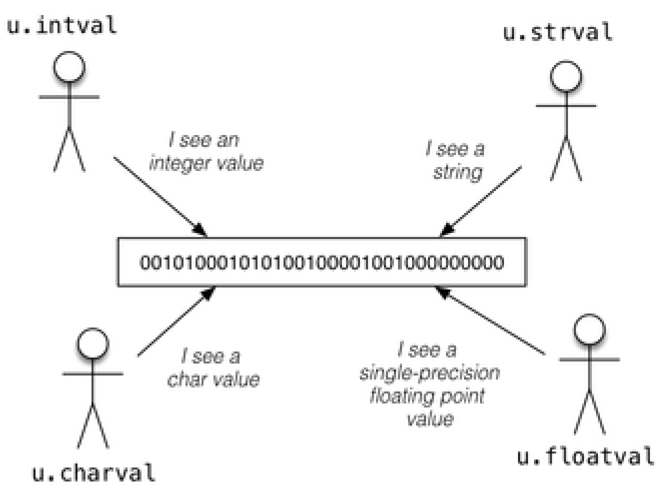
Bit-fields provide an alternative to bit operators and masks:

```
typedef unsigned int uint;          struct _privs {
typedef uint privs;                    unsigned int
#define OWNER_READ  (1 << 8)              owner_read  : 1,
#define OWNER_WRITE (1 << 7)              owner_write : 1,
#define OWNER_EXEC  (1 << 6)              owner_exec  : 1,
...                                       ...
#define OTHER_WRITE (1 << 1)              other_write : 1,
#define OTHER_EXEC  (1 << 0)              other_exec  : 1;
unsigned int myPrivs;               } myPrivs;

// give owner execute permission on file
myPrivs |= OWNER_EXEC;              myPrivs.owner_exec = 1;

// prevent others from writing on file
myPrivs &= ~OTHER_WRITE;            myPrivs.other_write = 0;

// check whether file is readable to all
open = myPrivs & OTHER_READ;        open = myPrivs.other_read;
```

Unions

31/36

Unions allow programmers to specify multiple interpretations for a single piece of memory.



... Unions

32/36

Example of defining a `union` type (cf. `struct`):

```
union _alltypes {
    int   intval;
    char  strval[4];
```

```
    char  charval;
    float floatval;
};

union _alltypes myUnion;
```

`myUnion` is a single 4-byte memory object

Programmers can specify how to interpret bits using field names.

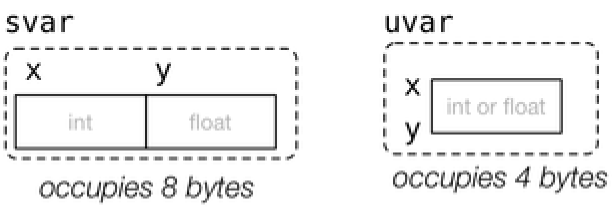
In the example above, all components are coincidentally the same size (4 bytes)

... Unions

33/36

Difference between a `struct` and a `union`

```
struct _s {              union _u {
   int   x;                 int   x;
   float y;                 float y;
} svar;                  } uvar;
```



... Unions

34/36

General syntax for defining `union` types and variables:

```
union Tag {
   Type1  Member1;
   Type2  Member2;
   Type3  Member3;
   ...
} uvar;
```

`Type` can be any C type; *`Member`* names must be distinct

`sizeof(`*Union*`)` is the size of the largest member

`&uvar.`$Member_1$ `==` `&uvar.`$Member_2$ `==` `&uvar.`$Member_3$ `...`

... Unions

35/36

Common use of `union` types: "generic" variables

```
#define IS_INT   1
#define IS_FLOAT 2
#define IS_STR   3

struct _generic {
   int   vartype;
   union { int ival; char *sval; float fval; };
};

struct _generic myVar;
// treat myVar as an integer
myVar.vartype = IS_INT;
myVar.ival    = 42;
printf("%d\n", myVar.ival);
// now treat myVar as a float
```

```
myVar.vartype = IS_FLOAT;
myVar.fval    = 3.14159;
printf("%0.5f\n", myVar.fval);
```

Enumerated Types

Enumerated types allow programmers to define a set of distinct named values

```
typedef enum { RED, YELLOW, BLUE } PrimaryColours;
```

```
typedef enum { LOCAL, INTL } StudentType;
```

The names are assigned consecutive `int` values, starting from 0

Above `PrimaryColors` type is equivalent to

```
#define RED    0
#define YELLOW 1
#define BLUE   2
```

Variables of type `enum...` are effectively `unsigned ints`.

Produced: 10 Aug 2017