DPST1092 23T2 — Data Representation Part 2

https://www.cse.unsw.edu.au/~dp1092/23T2/

Character Data

Character data has several possible representations (encodings)

The two most common:

- ASCII (ISO 646)
  - 7-bit values, using lower 7-bits of a byte (top bit always zero)
  - can encode roman alphabet, digits, punctuation, control chars
- UTF-8 (Unicode)
  - 8-bit values, with ability to extend to multi-byte values
  - can encode all human languages plus other symbols

  e.g.:

$\surd$ $\sum$ $\forall$ $\exists$ 🤑 🤗 🤓 😎 🤡 🤠

ASCII Character Encoding

Uses values in the range `0x00` to `0x7F` (0..127)

Characters partitioned into sequential groups

- control characters (0..31) ... e.g. `'\0'`, `'\n'`
- punctuation chars (32..47,91..96,123..126)
- digits (48..57) ... `'0'`..`'9'`
- upper case alphabetic (65..90) ... `'A'`..`'Z'`
- lower case alphabetic (97..122) ... `'a'`..`'z'`

Sequential nature of groups allows for things like `(ch - '0')` Eg.

See `man 7 ascii`

Unicode

Basically, a 32-bit representation of a wide range of symbols

- around 140K symbols, covering 140 different languages

Using 32-bits for *every* symbol would be too expensive

- e.g. standard roman alphabet + punctuation needs only 7-bits

More compact character encodings have been developed (e.g. UTF-8)

UTF-8 Character Encoding

UTF-8 uses a variable-length encoding as follows

#bytes	#bits	Byte 1	Byte 2	Byte 3	Byte 4
1	7	`0xxxxxxx`	-	-	-
2	11	`110xxxxx`	`10xxxxxx`	-	-
3	16	`1110xxxx`	`10xxxxxx`	`10xxxxxx`	-
4	21	`11110xxx`	`10xxxxxx`	`10xxxxxx`	`10xxxxxx`

The 127 1-byte codes are compatible with ASCII

The 2048 2-byte codes include most Latin-script alphabets

The 65536 3-byte codes include most Asian languages

The 2097152 4-byte codes include symbols and emojis and ...

ASCII Character Encoding

UTF-8 examples

ch	unicode	bits	simple binary	UTF-8 binary
$	U+0024	7	`010 0100`	`00100100`
¢	U+00A2	11	`000 1010 0010`	`11000010 10100010`
€	U+20AC	16	`0010 0000 1010 1100`	`11100010 10000010 10101100`

Unicode strings can be manipulated in C (e.g.”안녕하세요”)

Like other C strings, they are terminated by a 0 byte (i.e. `'\0'`)

Warning: Functions like strlen may not work as expected.

Exercise 1: UTF-8 Unicode Encoding

For each of the following symbols, with their Unicode value

- show the bit-string that would be used to represent them

Symbols:

- & `U+00026`
- µ `U+000B5`

Given that ♥ has the code `U+02665`

- Convert it into the bitstring that would represent it
- Write a C program to print
  ♥ beats

Fractions in different Bases

The decimal fraction 0.75 means

- $7*10^{-1} + 5*10^{-2} = 0.7 + 0.05 = 0.75$
- or equivalently $75/10^2 = 75/100 = 0.75$

Similary 0b0.11 means

- $1*2^{-1} + 1*2^{-2} = 0.5 + 0.25 = 0.75$
- or equivalently $3/2^2 = 3/4 = 0.75$

Similarly 0x0.C means

- $12*16^{-1} = 0.75$
- or equivalently $12/16^1 = 3/4 = 0.75$

Note: We call the `.` a radix point rather than a decimal point when we are dealing with other bases.

Fractions in different Bases

If we want to convert 0.75 in decimal to binary, it may be easy to look and and realise we need 0.5 + 0.25 which gives us 0b0.11. Sometimes it is not that easy and we need a systematic approach.

The algorithm to convert a decimal fraction to another base is:

- take the fractional component and multiply by the base
- the whole number becomes the next digit to the right of the radix point in our fraction.
- We now disregard the whole number part of the previous result and repeat this process until the fractional part becomes exhausted or we have sufficient digits (this process is not guaranteed to terminate).

Fractions in different Bases

For example if we want to convert 0.3125 to base 2

- 0.3125 * 2 = **0**.625
- 0.625 * 2 = **1**.25
- 0.25 * 2 = **0**.5
- 0.5 * 2 = **1**.0

Therefore 0.3125 = 0b0.0101

Exercise 2: Fractions: Decimal → Binary

Convert the following decimal values into binary

- 12.625
- 0.1

Floating Point Numbers

Floating point numbers model a (tiny) subset of real numbers

- many real values don't have exact representation (e.g. 1/3)
- numbers close to zero have higher precision (more accurate)

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`

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

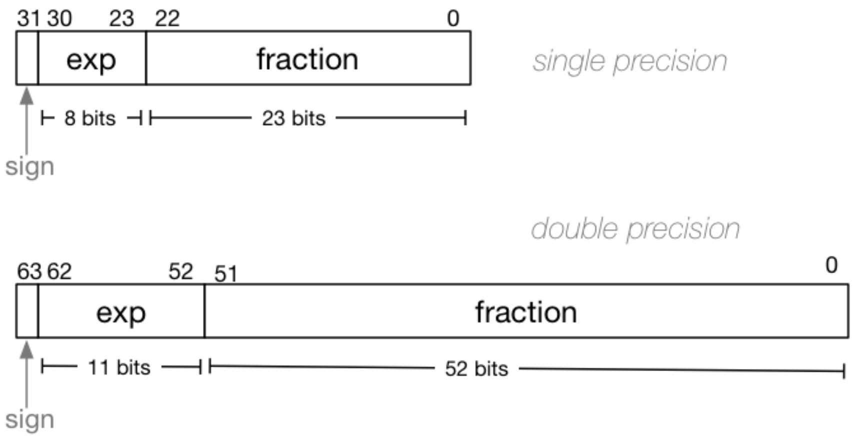
Floating Point Numbers

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 (e.g. sqrt(-1.0))

Floating Point Numbers

IEEE 754 standard internal structure of floating point values



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

Floating Point Numbers

Example of normalising the fraction part in binary:

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

The normalised fraction part always has `1` before the decimal point.

Example of determining the exponent in binary:

- assume an 8-bit exponent, then bias $B = 2^{8-1}-1 = 127$
- valid bit patterns for exponent `00000001` .. `11111110` (1..254)
- exponent values -126 .. 127

Floating Point Numbers

Example (single-precision):

$150.75 = 10010110.11$

`// normalise fraction, compute exponent`

$= 1.001011011 \times 2^7$

`// sign bit = 0`

`// exponent = 10000110`

`// fraction = 00101101100000000000000`

`= 01000011000101101100000000000000`

Note: B=127, e=2^7, so exponent = 127+7 = 134 = `10000110`

Floating Point Numbers

Convert the decimal numbers 1 to a floating point number in IEEE 754 single-precision format.

Convert the following floating point numbers to decimal.

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

`0 10000000 11000000000000000000000`

`1 01111110 10000000000000000000000`

You can try out more examples with this Floating Point Calculator

Floating Point Numbers

Special cases:

- If every bit (except the sign bit) is 0 then we have the number `0`. This means we can have positive and negative 0.
- If every bit of the exponent is 1 and the fraction is 0 then we have `infinity` (positive or negative)
- If every bit of the exponent is 1 and the fraction is not 0 then we have `NaN` (not a number).
- **Underflow**: If the exponent has minimum value (all zero), special rules for denormalized values are followed. The exponent value is set to 2^{-126} and the "invisible" leading bit for the fraction part is no longer used.

Pointers

Pointers represent memory addresses/locations

- number of bits depends on memory size, 64-bits on cse machines
- 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 *)`

Pointers

Code and data is aligned and is machine dependant. For example:

- `char` ... can be stored at any byte address
- `int` ... must be stored at an address *addr* `%4 == 0`
- `double` ... often must be stored at an address *addr* `%8 == 0`

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

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

Pointer arithmetic

Pointers 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);
}
```

Function Pointers

In C you may point to anything in memory.

- The compiled program is in memory
- The compiled program is made up of functions
- Therefore you can point at functions

Function pointers ...

- are references to memory address of a function
- are pointer values and can be assigned/passed

Function Pointers

Syntax of declaring a function pointer:

*return_t (*var) (arg_t, ...)*

Examples of declaring a function pointer:

```
// variable fp is a pointer to a function with
// one int parameter and an int return value
int (*fp) (int);

// variable fp2 is a pointer to a function with
// a char and an int parameters and a void return value
void (*fp2) (char, int);
```


Function Pointers

Examples of use:

```
int square (int x) { return x*x; }
int timesTwo (int x) { return x*2; }

int (*fp) (int);
//Point to the square function and use it
fp = &square;
int n = (*fp)(10);

//It also works without the '&'
fp = timesTwo;
n = (*fp)(2);

//Normal function notation also works
n = fp(2);
```

Function Pointers

Can traverse a collection such as an array, applying the function to all values

```
void traverse(int len, int a[], int (*f)(int)){
    for(int i = 0; i < len; i++){
        a[i] = f(a[i]);
    }
}

int main(void){
    int a[3] = {1,2,3};
    traverse(3,a,square);
    traverse(3,a,timesTwo);
    return 0;
}
```

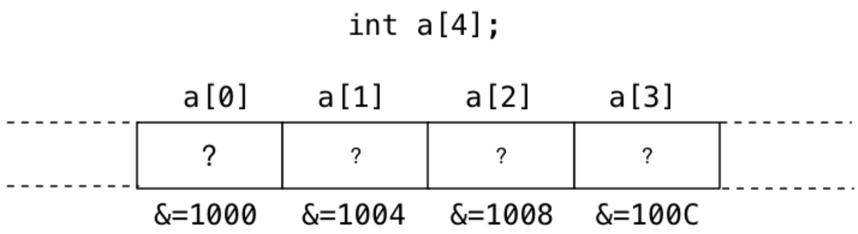

Arrays

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

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*`)`
- array name gives address of first element (e.g. `v` = `&v[0]`)
- `v[i]` is the same as `*(v+i)`

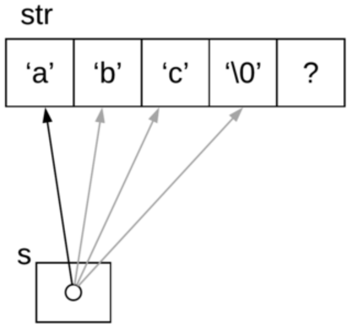
Strings are just arrays of `char` with a `'\0'` terminator

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

Arrays

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);
      s++;
   }
}
```



Arrays

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, str1);
    printf("&str1=%p, %s\n", &str1, str1);
    printf("&str2=%p, %s\n", &str2, str2);
    printf("str1=%p, str2=%p\n",str1,str2);
    free(str2);
    return 0;
}
```

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

(except for the unitialised parts of the arrays)

structs

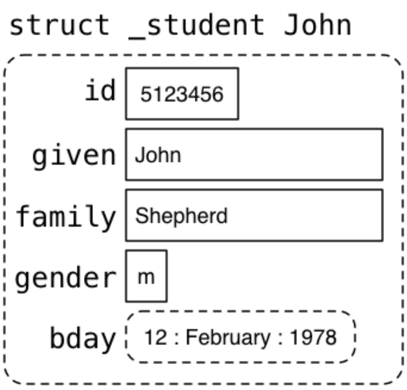
Structs are defined to have a number of components

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

Example:

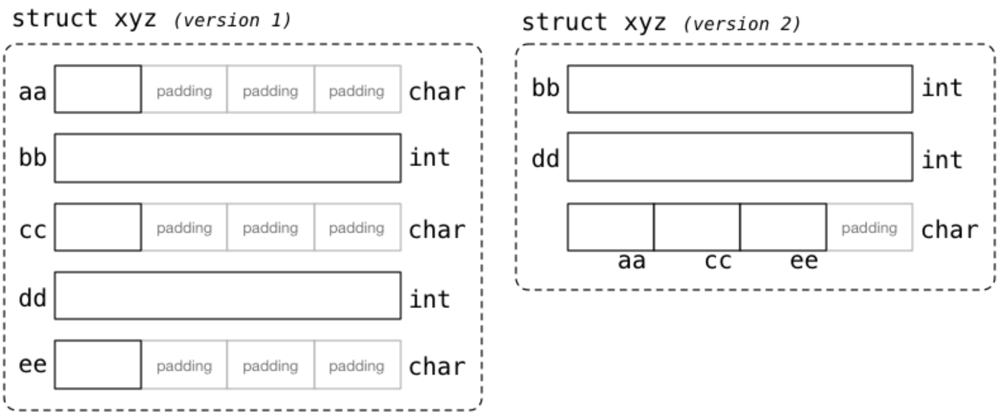
```
typedef struct … Date;

struct _student {
    int id;
    char given[50];
    char family[50];
    char gender;
    Date bday;
};
```

Defines a new data type called `struct _student`



structs

To ensure *alignment* for the fields and for the struct itself, internal



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

unions

A **union** is a special data type available in C that allows storing different data types in the **same** memory location.

The size of a union is equal to the size of its largest member (plus any padding).

An example of declaring a union

```
union MyUnion {
    unsigned long long value;
    char    s[8];
};
```

This union can store either an unsigned long long value, or a string of size 8 (including the '\0' terminator).

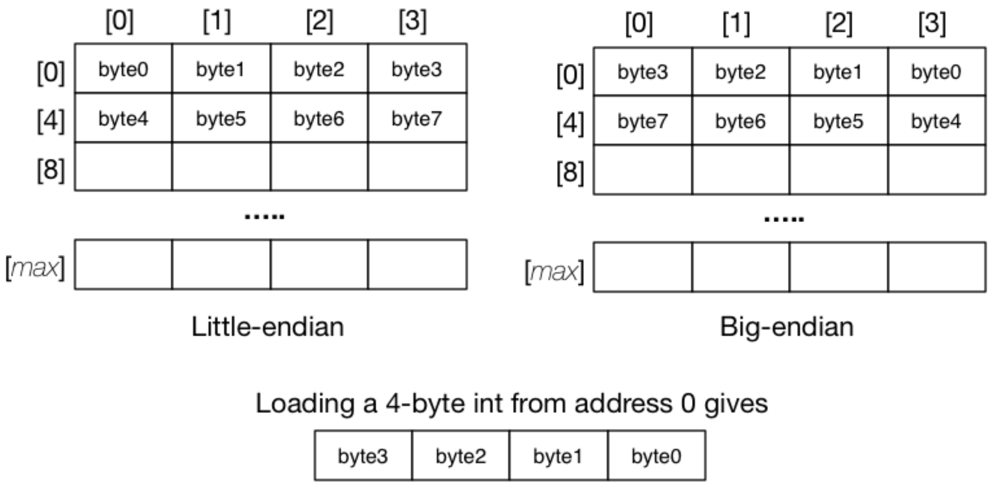
unions

Example usage

```
union MyUnion u;
printf("%d\n",sizeof(union MyUnion)); //prints out 8
u.value = 999999;
printf("%llu\n",u.value); //prints out 999999
strcpy(u.s,"hello");
printf("%s\n",u.s);       //prints out hello
printf("%llu\n",u.value); //Does NOT print out 999999
                          //as it has been (partly) overwritten
```

Memory and Endianness

Memories can be categorised as *big-endian* or *little-endian*



Exercise: Endianness

Write code to print out an int, byte by byte. Is your system big or little endian?