目录

Data structures: memory allocation

A character is stored in a byte (8 bits)

- a *char* takes 1 byte (8 bits)
- an *int* takes 4 bytes (32 bits)
- ***how much memory does a pointer need?***

Commonly, computers are 32 bit

- means they support 32-bit addresses
    - an address is hence 4 bytes (same as an *int*)
- 4GB memory can be addressed

More modern computers are 64 bit

- addresses are hence 64 bits (= 8 bytes)
- *chars* are (still) 1 byte
- *ints* are (still) 4 bytes

Consider the program

切换行号显示

```c
#include <stdio.h>

int main(void)
{
  int i;
  char c;

  printf("i and c are uninitialised\n");
  printf("&i = %p\n", (void*)&i);
  printf("&c = %p\n", (void*)&c);

  printf("sizeof(&i) = %ld, sizeof(i) = %ld\n", sizeof(&i), sizeof(i));
  printf("sizeof(&c) = %ld, sizeof(c) = %ld\n", sizeof(&c), sizeof(c));

  printf("now give i and c values\n");
  i = 58;
  c = 'X';
  printf("i = %d\n", i);
  printf("c = %c\n", c);
  return 0;
}
```

Aside:

- why the cast (void *) in the program?

This program has two variables *i* and *c*.

- the address of both are taken and printed
- the *sizeof()* each variable, and their addresses are then printed
- the variables are then initialised
- their values are then printed

The output is

```
i and c are uninitialised
&i = 0x7ffd8383dc00
&c = 0x7ffd8383dc10
sizeof(&i) = 8, sizeof(i) = 4
sizeof(&c) = 8, sizeof(c) = 1
now give i and c values
i = 58
c = X
```

Notice:

- chars are 1 byte, ints are 4 bytes, and addresses are 8 bytes long
  - ... 8 bytes because it is a 64-bit machine (the address space)
  - we normally do not worry about the size of data

sizeof structs

Consider the following program:

切换行号显示

```
// padding.c
#include <stdio.h>
#include <stdlib.h>

#define SIZEBYTES 5
#define S(x)     sizeof(x)  // this is a C macro that is substituted by the preprocessor
                            // I use it in the program just to save line-length space
struct s1 {
    char  c;
    int   i;
    char *s;
};
struct s2 {
    char  c;
    char *s;
    int   i;
};

int main() {

   struct s1 a;
   struct s2 b;

   printf("sizeof(a.c)=%lu, sizeof(a.i)=%lu, sizeof(a.s)=%lu\n", S(a.c), S(a.i), S(a.s));
   printf("sizeof(b.c)=%lu, sizeof(b.s)=%lu, sizeof(b.i)=%lu\n", S(b.c), S(b.s), S(b.i));
```

```

    printf("sizeof(a) = %lu, sizeof(b) = %lu\n\n", sizeof(a), sizeof(b));

    a.s = malloc(SIZEBYTES); // returned value not checked
    printf("&a.c = %p\n", (void*)&a.c);
    printf("&a.i = %p\n", (void*)&a.i);
    printf("&a.s = %p\n\n", (void*)&a.s);

    b.s = malloc(SIZEBYTES); // returned value not checked
    printf("&b.c = %p\n", (void*)&b.c);
    printf("&b.s = %p\n", (void*)&b.s);
    printf("&b.i = %p\n", (void*)&b.i);

    return 0;
}
```

Compile "'padding.c"' with "gcc"

The output will look like:

```
sizeof(a.c)=1, sizeof(a.i)=4, sizeof(a.s)=8
sizeof(b.c)=1, sizeof(b.s)=8, sizeof(b.i)=4
sizeof(a) = 16, sizeof(b) = 24

&a.c = 0x7fadedface30
&a.i = 0x7fadedface34
&a.s = 0x7fadedface38

&b.c = 0x7fadedface10
&b.s = 0x7fadedface18
&b.i = 0x7fadedface20
```

Notice:

- both structs contain 3 fields: a char, a pointer and an int
  - total is 1+8+4 = 13 bytes
- order of fields is different in the two structs
- fields have addresses in declaration order (low to high)
- structs are in reverse declaration order (high to low)

But:

- var *a* uses 16 bytes
- var *b* uses 24 bytes
- ***why different? why larger?***

Let's examine the fields in variable *a*:

- *a.c* (1 byte) occupies 30, so **bytes 31...33 are padding (3 bytes)**
- *a.i* (4 bytes) occupies 34...37
- *a.s* (8 bytes) occupies 38 ..3f

- Total: 13 bytes actual data + 3 bytes padding

Let's examine the fields in variable *b*:

- *b.c* (1 byte) occupies 10, so **bytes 11..17 are padding (7 bytes)**
- *b.s* (8 bytes) occupies 18..1f
- *b.i* (4 bytes) occupies 20..23, so **bytes 24..27 are padding (4 bytes)**

- Total: 13 bytes actual data + 11 bytes padding

Why is the compiler inserting padding?

It is a 64-bit address computer. A 'word' is hence 8 bytes (i.e. 64 bits)

- variable *a*
    - the *char* is padded to an *int* boundary (3 bytes inserted)
        - the *char* and the *int* share the same word
    - address *char* * naturally occupies one word
    - the *struct* has finished on a word boundary
- variable *b*
    - the *char* is padded to a word boundary (7 bytes inserted)
    - address *char* * naturally occupies one word
    - the *int* is padded to a word boundary (4 bytes inserted)
    - the *struct* has finished on a word boundary

Between *a* and *b* there is <u>also</u> padding

- *b* finishes at 0x7fadedface27, *a* starts at 0x7fadedface30: 8 bytes inserted

Note:

- none of the variables have been defined (no values assigned)
- changing *SIZEBYTES* will not change the output above
    - *Question: do you know why?*

Compile '''padding.c''' with ''dcc''

The output is:

```
sizeof(a.c)=1, sizeof(a.i)=4, sizeof(a.s)=8
sizeof(b.c)=1, sizeof(b.s)=8, sizeof(b.i)=4
sizeof(a) = 16, sizeof(b) = 24

&a.c = 0x7fadedface60
&a.i = 0x7fadedface64
&a.s = 0x7fadedface68

&b.c = 0x7fadedface80
&b.s = 0x7fadedface88
&b.i = 0x7fadedface90
```

Note:

- appears the same as with *gcc* <u>but</u> ...
    - the memory order of *a* and *b* is in declaration order (not reversed)
        - *Question: what's the last memory address used by variable a?*
    - the padding between *a* and *b* is 16 bytes

Language definition says nothing about memory allocation

The C language definition does not define where variables are placed in memory

- memory allocation is dependent on compiler and underlying architecture (i.e. instruction set)

- different compilers generate different code for the same machine ('gcc' vs 'dcc' above)
- often a *space vs time* trade-off
    - every variable occupies 1 word optimises speed → poor space utilisation
    - 'packing' data in memory optimises space → poor performance (extra time cost to 'unpack' the data)

Lec01Sizeof (2019-06-14 11:02:32由AlbertNymeyer编辑)