Computer System: A Programmer's Perspective

CMU15-213 AKA. CMU zip code

Chapter 1 计算机系统漫游

- 课程主题Course theme
- 五大现实Five realities
- 该课程如何融入 CS/ECE 课程How the course fits into the CS/ECE curriculum
- 学术诚信Academic integrity

课程主题 Course Theme

Abstraction is Good But Don't Forget Reality

Chapter 2 信息的表示和处理

数值系统

原码:

反码:

补码:

浮点数:

```
# inital
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;

x < 0   =>  ((x*2)<0)
ux > -1
x > y   =>  -x < -y
x * x >= 0
```

```
define flip2(a)
    (((a) & 0xFF) << 8 + ((a) & 0xFF00) >> 8)

define flip2(a)
    ((a) << 8 | ((a) >> 8) & 0xFF)
```

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
    size_t i;
    for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
    printf("\n");
}

int main(){
    int a = 0x01234567;
    show_bytes((pointer) &a, sizeof(int));
}
```

当进行0.1 + 0.2这样的浮点数运算时，由于0.1和0.2无法在二进制中精确表示，计算结果会受到舍入误差的影响。这是因为0.1和0.2在二进制表示中都是无限循环小数。

```
PS C:\Users\ChenYifan> ipython
Python 3.9.12 (main, Apr  4 2022, 05:22:27) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.31.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 0.1 + 0.2
Out[1]: 0.30000000000000004
```

```
#include <assert.h>

int main(){
    assert(+0. == -0.);          // 断言成功
    assert(1.0/+0. == 1.0/-0.); // 断言失败
    return 0;
}
```

带符号数产生意外结果的例子。这个例子会造成无限循环，因为sizeof会返回unsigned int 类型，由此带来的结果是，i - sizeof(char)这个表达式的求值结果将会是 unsigned int (隐式转换 !!)，无符号数 0 减 1 会变成 0xFFFFFFFF，从而产生无限循环，有时候你需要特别留心这种不经意的错误！

```
int n = 10, i;
for (i = n - 1 ; i - sizeof(char) >= 0; i--)
    printf("i: 0x%x\n",i);

if (-1 > 0U)
    printf("You Surprised me!\n");
```

二分法

```
int binary_search(int a[], int len, int key){
    int low = 0;
    int high = len - 1;

    while ( low <= high ) {
        int mid = (low + high)/2;   // 提示：这里有溢出Bug!
```

```
        if (a[mid] == key) {
            return mid;
        }
        if (key < a[mid]) {
            high = mid - 1;
        }else{
            low = mid + 1;
        }
    }
    return -1;
}
```

```
int binary_search(int a[], int len, int key){
    int low = 0;
    int high = len - 1;

    while ( low <= high ) {
        int mid = low + (high - low) / 2;    // 在原始范围内不溢出
        if (a[mid] == key) {
            return mid;
        }
        if (key < a[mid]) {
            high = mid - 1;
        }else{
            low = mid + 1;
        }
    }
    return -1;
}
```

当 `low` 和 `high` 的值很大时，它们的和可能会超过整型的最大表示范围，从而导致溢出。溢出后的结果会被截断为一个错误的值，从而导致 mid 的计算错误。

通过修改为 `low + (high - low) / 2` 的方式，我们先计算 `high` 和 `low` 之间的差值，再除以 2 并加上 `low`，这样可以避免溢出。因为差值不会超过原始范围，所以不会引发溢出问题。

```
// 初学者版本
void remove_list_node(List *list, Node *target)
{
    Node *prev = NULL;
    Node *cur = list->head;
    // Walk the list
    while (cur != target) {
        prev = cur;
        cur = cur->next;
    }
    // Remove the target by updating the head or the previous node.
    if (!prev)
        list->head = target->next;
    else
        prev->next = target->next;
}
```

```
// 有品位的版本，消除特例，简单优雅的代码
void remove_list_node(List *list, Node *target)
{
    // The "indirect" pointer points to the *address*
    // of the thing we'll update.
    Node **indirect = &list->head;
    // Walk the list, looking for the thing that
    // points to the node we want to remove.
    while (*indirect != target)
        indirect = &(*indirect)->next;
    *indirect = target->next;
}
```

通过 `indirect` 指针追踪前一个节点的 `next` 指针的地址。通过这种方式，我们可以在不使用特例情况（如头节点的处理）的情况下，简洁地删除链表中的目标节点