

# CS100 Recitation 2

GKxx

# 目录

- 字面值
- 运算符
- 控制流

# 字面值 (literals)

## 字面值 (literals)

像 `42` 这样把值写在脸上的就是字面值。

C 语言的每一个表达式都有一个明确的、编译期确定的类型。所以，`42` 是什么类型？

3.14 又是什么类型？

## 字面值 (literals)

像 `42` 这样把值写在脸上的就是字面值。

C 语言的每一个表达式都有一个明确的、编译期确定的类型。所以，`42` 是什么类型？  
`3.14` 又是什么类型？

- `42` 是 `int` ， `3.14` 是 `double` ， 尽管它们也许可以用其它类型表示。

如何写出一个其它类型的字面值？

# 字面值 (literals)

像 `42` 这样把值写在脸上的就是字面值。

- 整型字面值 (integer literals) : `42` , `100L` , `011` , `405ul`
  - 不写后缀，默认是 `int` ，如果 `int` 不够大就是 `long` ，还不够大就是 `long long` 。还不够大的话：
    - 如果编译器支持 `__int128` 并且它够大，那就是 `__int128`
    - 否则报错 (ill-formed)。
  - 不存在负字面值： `-42` 是将一元负号 `-` 作用在字面值 `42` 上形成的表达式。
  - 后缀有 `u` ( `unsigned` ), `l` ( `long` ), `ll` ( `long long` )。大小写不敏感，但是不可以是 `lL` 或 `Ll` 。
  - 后缀可以任意组合。

# 字面值 (literals)

像 `42` 这样把值写在脸上的就是字面值。

- 整型字面值 (integer literal) : `42` , `100L` , `011` , `405ul`
  - 还可以有十六进制字面值 : `0xBAADF00D`
  - 以及八进制字面值 : `052`
  - 以及 C23 的二进制字面值 : `0b101010`
    - 实际上编译器早就支持了, 隔壁 C++14 就有二进制字面值了。
  - 这里所有的字母的大小写都随意。

## 字面值 (literals)

浮点数字面值： `3.14` , `3.14f` , `3.14l` , `1e8` , `3e-8`

- 不写后缀，默认是 `double` 。 `f` 是 `float` ， `l` 是 `long double` ，大小写不敏感。
- `1e8` 表示  $10^8$  ，但它是 `double` 而非整数。

字符字面值： `'a'`

- 类型是 `int` ？—— C++ 里它就是 `char` 了。



## 字符？一个小整数罢了

```
char c = 'a';
```

`c` 所存储的内容**就是**整数 `97`，即 `'a'` 的 [ASCII](#) 码，而非任何神秘的图像。

**不存在“字符和它对应的 ASCII 码之间的转换”！它们本就是同一个东西**

假如字符 `c` 是一个数字字符，如何获得它所表示的数值？

## 字符？一个小整数罢了

```
char c = 'a';
```

`c` 所存储的内容**就是**整数 `97`，即 `'a'` 的 [ASCII](#) 码，而非任何神秘的图像。

**不存在“字符和它对应的 ASCII 码之间的转换”！它们本就是同一个东西**

假如字符 `c` 是一个数字字符，`c - '0'` 就是它所表示的数值。

练习：假如你只能使用 `getchar()` 读取一个字符，如何读入一个非负整数？

# 读入整数

暂时假设输入不会出错。

```
int read(void) {  
    int x = 0;  
    char c = getchar();  
    while (isspace(c)) // 跳过空白  
        c = getchar();  
    while (isdigit(c)) {  
        x = x * 10 + c - '0';  
        c = getchar();  
    }  
    return x;  
}
```

- 使用声明在标准库头文件 `<ctype.h>` 中的函数
- `isupper(c)` 判断 `c` 是不是大写字母
- `islower(c)` 判断 `c` 是不是小写字母
- `isdigit(c)` 判断 `c` 是不是数字字符
- `isspace(c)` 判断 `c` 是不是空白字符
- .....

# 运算符

# 优先级，结合性，求值顺序

## 优先级和结合性表

- 优先级：`f() + g() * h()` 被解析为 `f() + (g() * h())` 而非 `(f() + g()) * h()`
- 结合性：`f() - g() + h()` 被解析为 `(f() - g()) + h()` 而非 `f() - (g() + h())`

但是 `f()`，`g()`，`h()` 三者的调用顺序是 unspecified 的！

类似的还有：`func(f(), g(), h())` 这里的 `f()`，`g()`，`h()` 三个函数的调用顺序是 unspecified 的。

我的亲身经历：`add_edge(read(), read(), read());`

## 优先级，结合性，求值顺序

如果两个表达式 `A` 和 `B` 的求值顺序是 unspecified 的，而它们

- 都修改了一个变量的值，或者
- 一个修改了某个变量的值，另一个读取了那个变量的值，

那么这就是 undefined behavior。

例：`i = i++ + 2` 的结果是什么？

## 优先级，结合性，求值顺序

如果两个表达式 `A` 和 `B` 的求值顺序是 unspecified 的，而它们

- 都修改了一个变量的值，或者
- 一个修改了某个变量的值，另一个读取了那个变量的值，

那么这就是 undefined behavior。

例：`i = i++ + 2` 的结果是什么？

- 一旦你开始分析它是 `+1` 还是 `+2`，你就掉进了 undefined behavior 的陷阱里。
- `++` 对 `i` 进行了修改，赋值也对 `i` 进行了修改，两个修改的顺序 unspecified，所以就是 undefined behavior。

## 求值顺序确定的运算符

常见的运算符中，其运算对象的求值顺序确定的只有四个：`&&`，`||`，`?:`，`,`

- `&&` 和 `||`：短路求值，先求左边，非必要不求右边。
- `cond ? t : f`：先求 `cond`，根据它的真假性选择求 `t` 还是求 `f`。
- `,`：一种存在感极低的运算符。



# 位运算符

`~` , `&` , `^` , `|` , `<<` , `>>` , 以及复合赋值运算符 `&=` , `|=` , `^=` , `<<=` , `>>=`

Operator	Operator name	Example	Result
<code>~</code>	bitwise NOT	<code>~a</code>	the bitwise NOT of <code>a</code>
<code>&amp;</code>	bitwise AND	<code>a &amp; b</code>	the bitwise AND of <code>a</code> and <code>b</code>
<code> </code>	bitwise OR	<code>a   b</code>	the bitwise OR of <code>a</code> and <code>b</code>
<code>^</code>	bitwise XOR	<code>a ^ b</code>	the bitwise XOR of <code>a</code> and <code>b</code>
<code>&lt;&lt;</code>	bitwise left shift	<code>a &lt;&lt; i</code>	<code>a</code> left shifted by <code>i</code>
<code>&gt;&gt;</code>	bitwise right shift	<code>a &gt;&gt; i</code>	<code>a</code> right shifted by <code>i</code>

## 位运算符

- $\sim a$  : 返回  $a$  的每个二进制位都取反后的结果。例如  $10010110_{\text{two}}$  求反后等于  $1101001_{\text{two}}$
- $a \& b$  的第  $i$  位是 1 当且仅当  $a$  和  $b$  的第  $i$  位都是 1。
- $a | b$  的第  $i$  位是 1 当且仅当  $a$  和  $b$  的第  $i$  位至少有一个是 1。
- $a \wedge b$  的第  $i$  位是 1 当且仅当  $a$  和  $b$  的第  $i$  位不同。

例：  $a \& 1$  就是拿  $a$  和  $\underbrace{0 \dots 001}_{n-1 \text{ 个}}_{\text{two}}$  求 bitwise AND，就相当于  $a \% 2$ 。

# 位运算符

假设 `a` 是无符号整数。

- `a << i` 返回将 `a` 的二进制位集体左移 `i` 位的结果。
  - 例如，`a << 1` 就是 `a * 2`，`a << i` 就是 `a` 乘以  $2^i$ 。
  - 左边超出的部分丢弃。
- `a >> i` 返回将 `a` 的二进制位集体右移 `i` 位的结果。
  - 例如，`a >> 1` 就是 `a / 2`。右边超出的部分被丢弃。

例：如何得到  $2^n$ ？

# 位运算符

假设 `a` 是无符号整数。

- `a << i` 返回将 `a` 的二进制位集体左移 `i` 位的结果。
  - 例如，`a << 1` 就是 `a * 2`，`a << i` 就是 `a` 乘以  $2^i$ 。
  - 左边超出的部分丢弃。
- `a >> i` 返回将 `a` 的二进制位集体右移 `i` 位的结果。
  - 例如，`a >> 1` 就是 `a / 2`。右边超出的部分被丢弃。

例：如何得到  $2^n$ ？ `1 << n`

- 但是 `1` 是 `int` 类型的字面值。
- 如果 `n` 比较大，你可能需要 `1u << n`，`1ll << n` 等等。

# 位运算符

如何获得一个无符号整数 `x` 的第 `i` 位？

- 我们约定 `i`  $\in [0, N)$ ，其中  $N$  是 `x` 的类型的总位数。第 `0` 位是最右边的位 (least significant bit)。

```
int test_bit(unsigned x, unsigned i) {  
  
}
```

# 位运算符

如何获得一个无符号整数 `x` 的第 `i` 位？

```
int test_bit(unsigned x, unsigned i) {  
    return (x >> i) & 1u;  
}
```

或者

```
int test_bit(unsigned x, unsigned i) {  
    return (x & (1u << i)) >> i;  
}
```

# 位运算符

如何翻转一个无符号整数 `x` 的第 `i` 位？

```
unsigned bit_flip(unsigned x, unsigned i) {  
  
}
```

# 位运算符

如何翻转一个无符号整数 `x` 的第 `i` 位？

```
unsigned bit_flip(unsigned x, unsigned i) {  
    return x ^ (1u << i);  
}
```



## 位运算符

如何截取一个无符号整数 `x` 的第  $[low, high)$  位？

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {  
    }  
}
```

## 位运算符

如何截取一个无符号整数 `x` 的第  $[low, high)$  位？

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {  
    x >>= low; // 先右移 low 位  
    // 然后我们需要和一个 0...00011111 这样的数 & 一下  
    // 如何获得一个这样的数？  
}
```

## 位运算符

如何截取一个无符号整数 `x` 的第  $[low, high)$  位？

- 先右移 `low` 位，然后和一个末 `high - low` 位全是 `1`、其它全是 `0` 的数 `&` 一下

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {  
    return (x >> low) & ((1u << (high - low)) - 1);  
}
```

还有没有别的办法？

# 位运算符

如何截取一个无符号整数 `x` 的第  $[low, high)$  位？

- 先右移 `low` 位，然后和一个末 `high - low` 位全是 `1`、其它全是 `0` 的数 `&` 一下

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {  
    return (x >> low) & ((1u << (high - low)) - 1);  
}
```

或者，先和一个末 `high` 位全是 `1`、其它全是 `0` 的数 `&` 一下，再右移 `low` 位

```
unsigned bit_slice(unsigned x, unsigned low, unsigned high) {  
    return (x & ((1u << high) - 1)) >> low;  
}
```

## 神奇的异或 (XOR)

输入  $n$  个 32 位非负整数，其中有一个数出现了奇数次，其它数都出现了偶数次。求那个出现了奇数次的数。

# 神奇的异或 (XOR)

输入  $n$  个 32 位非负整数，其中有一个数出现了奇数次，其它数都出现了偶数次。求那个出现了奇数次的数。

答案： $x_1 \oplus x_2 \oplus \cdots \oplus x_n$

```
int main(void) {
    int n; scanf("%d", &n);
    unsigned result = 0;
    while (n--) {
        unsigned x; scanf("%u", &x);
        result ^= x; // 啊？
    }
    printf("%u\n", result);
    return 0;
}
```

## 神奇的异或 (XOR)

输入  $n$  个 32 位非负整数，其中有一个数出现了奇数次，其它数都出现了偶数次。求那个出现了奇数次的数。

首先证明，异或具有交换律和结合律：

- $a \oplus b = b \oplus a$
- $(a \oplus b) \oplus c = a \oplus (b \oplus c)$

并且  $a \oplus a = 0$ ,  $a \oplus 0 = a$ 。

那么对于  $x_1 \oplus \cdots \oplus x_n$ ，任意交换它们的顺序，把相同的数碰到一起，它们就会消失，剩下的就是那个出现了奇数次的。

## 神奇的异或 (XOR)

输入  $n$  个 32 位非负整数，其中有一个数出现了奇数次，其它数都出现了偶数次。求那个出现了奇数次的数。

另一种想法：考虑那个出现了奇数次的数  $r$

- 如果  $r$  的第  $i$  位是 1，意味着所有的数中，第  $i$  位一共有奇数个 1。
- 如果  $r$  的第  $i$  位是 0，意味着所有的数中，第  $i$  位一共有偶数个 1。

所以我们需要一种神奇的操作，让偶数个 1 变成 0，让奇数个 1 变成 1。

- 或者说，这是二进制意义下的不进位加法。



## 神奇的异或 (XOR)

输入  $n$  个 32 位非负整数，其中有一个数出现了奇数次，其它数都出现了偶数次。求那个出现了奇数次的数。

另一种想法：考虑那个出现了奇数次的数  $r$

- 如果  $r$  的第  $i$  位是 1，意味着所有的数中，第  $i$  位一共有奇数个 1。
- 如果  $r$  的第  $i$  位是 0，意味着所有的数中，第  $i$  位一共有偶数个 1。

所以我们需要一种神奇的操作，让偶数个 1 变成 0，让奇数个 1 变成 1。

- 或者说，这是二进制意义下的不进位加法。——这就是异或！

# 控制流

主要讲两个作用域问题

## do-while 的作用域问题

每次输入一个整数，做一些处理，如果输入的是 0 那么处理完毕后停止。你会怎么写？

## do-while 的作用域问题

每次输入一个整数，做一些处理，如果输入的是 0 那么处理完毕后停止。你会怎么写？

```
do {  
    int x;  
    scanf("%d", &x);  
    do_something(x);  
} while (x != 0);
```

```
a.c:9:12: error: 'x' undeclared (first use in this function)  
    9 |     } while (x != 0);
```

## do - while 的作用域问题

do - while 的循环体是一个内层的作用域，它以 { 开始、} 结束，不包含 while (cond) 的部分。

在 do - while 循环体内声明的变量，无法在 cond 部分使用。

```
do {  
    int x;  
    scanf("%d", &x);  
    do_something(x);  
} while (x != 0); // Error: `x` undeclared.
```

## switch - case 的作用域问题

始终牢记，控制流跳转到一个 `case` 标签对应的语句后会一直往下执行，直到碰到 `break;` 或者到达末尾。

```
switch (expr) {  
    case 1:  
        int x = 42;  
        do_something(x, expr);  
    case 2:  
        // 如果这里使用了 `x`，怎么办？  
        printf("%d\n", x);  
}
```

如果 `expr == 2`，控制流根本就没有经过 `int x = 42;` 这条语句，但是根据名字查找的规则却能找到 `x`。

## switch - case 的作用域问题

```
switch (expr) {  
    case 1: { // 用 {} 将 `x` 限定在内层作用域中。  
        int x = 42;  
        do_something(x, expr);  
    }  
    case 2:  
        printf("%d\n", x); // Error: `x` was not declared in this scope.  
}
```

如果 `expr == 2`，控制流根本就没有经过 `int x = 42;` 这条语句，但是根据名字查找的规则却能找到 `x`。

为了解决这个问题，如果在某个 `case` 内部声明了变量，这个变量必须存在于一个内层作用域中。

- 简单来说就是要加 `{}`。