# 第二单元:

## 单元练习目的

学生在掌握计算机对于程序的数据基本表达方式后，能够比较深刻的了解计算机中比特的使用方式以及各种位操作符，理解整数类型，浮点类型和结构体的原理和具体构成方式。

### 练习1

建立一个整形变量，然后进行各种位操作，观察整形变量经过各种位操作后在内存中的结果是怎样的。

int oh\_my\_word;

if (oh\_my\_word & 0x00000004) cout << "third bit set" << endl;

// 0xFFFFFFFB, or 1111 1111 1111 1111 1111 1111 1111 1011

oh\_my\_word = oh\_my\_word & 0xFFFFFFFB;

// 0x00000004, or 0000 0000 0000 0000 0000 0000 0000 0100

oh\_my\_word = oh\_my\_word | 0x00000004;

// 0x00000004, or 0000 0000 0000 0000 0000 0000 0000 0100

oh\_my\_word = oh\_my\_word ^ 0x00000004;

// Get bit N from a word (right-most is bit 0).

// First, shift bit N to right-most place:

int temp = oh\_my\_word >> N;

// Second, mask the right-most bit:

temp = temp & 0x00000001;

### 练习2

在工程里加入以下代码，通过将x和y的数值设置为较大数值，观察overflow现象是否会发生。

unsigned long x, y, sum;

//在下面加入你想要设置的x,y的数值

sum = x + y;

// If overflow occurred, sum will be smaller

// than either x or y. Otherwise, sum will

// be greater than either x or y.

if (sum < x) handle\_overflow();

### 练习3

建立一个工程，在源文件中加入下列代码，观察输出的结果和预期的结果有何不同：

#include <iostream>

#include <iomanip>

using namespace std;

void main()

{

float x = 1.2F;

double y = x;

cout << setprecision(20) <<

x << ", " << y << endl;

cout << "1.2F == 1.2: " << (1.2F == 1.2) << endl;

}

建立一个工程，在源文件中加入下列代码，观察输出的结果和预期的结果有何不同：

#include <iostream>

#include <iomanip>

using namespace std;

void main()

{

double x = 1.3;

double y = 0.4;

if (x + y != 1.7)

cout << "addition failed?" << endl;

}

建立一个工程，在源文件中加入下列代码，观察输出的结果和预期的结果有何不同：

#include <iostream>

#include <iomanip>

using namespace std;

const double epsilon = 0.000001;

bool about\_equal(double x, double y)

{

return (x < y + epsilon) &&

(x > y - epsilon);

}

void main()

{

cout << "1.3 + 0.4 == 1.7: " <<

(1.3 + 0.4 == 1.7) << endl;

cout << "about\_equal(1.3 + 0.4, 1.7): " <<

about\_equal(1.3 + 0.4, 1.7) << endl;

}

### 练习4

下面是对一个浮点数值的操作，观察输出的结果：

double x = 1.0E160;

// note: largest double is about 1.8E308

// so x \* x will overflow

x = x \* x;

cout << x << endl;

### 练习5

建立一个整形数组，通过不同的方式进行对应地址的操作：

int myarray[5];

myarray + N = &(myarray[N])

myarray[N] = \*(myarray + N)

建立一个结构体，观察结构体内部的各个变量在内存中的分布方式，然后在工程属性里面，更改代码生成菜单里面的结构体对齐，选项再次编译，观察结构体的内存分配的改变：

struct {

char a, b;

double d;

int i;

} mystruct;