Secure Coding

2 INTEGER

- 2.1 整数
- 2.2 整数数据类型
- 2.3 整数转换
- 2.4 整数操作

2.1 整数

- -1,-2,-3,……
- 0
- 1,2,3,……

- 1.6　1/2

2.1 整数

程序仅对于一个给定的范围内的期望输入运行良好

缺少系统的整数范围检查

2.2 整数数据类型

- 整数类型：

整数数学集合的一个有限子集模型

一个整数对象代表一个值

整数对象值的表示方式，是为该对象分配存储空间中该值的特定位模式编码

2.2 整数数据类型

<limits.h>

每个整数对象需要一个固定的存储字节空间

CHAR_BIT 至少8

可能需要填充字节padding

2.2 整数数据类型

- **2.2.1** 无符号整数类型

无偏移纯二进制表示$\sum_{i=0}^{N} 2^i$

最右侧位权数2^0，其左侧临位权数2^1

全零位始终表示0，范围0~ $2^{w(type)}$-1

2.2 整数数据类型

- 按照长度非递减排序

unsigned char	8
unsigned short int	16
unsigned int	32
unsigned long int	32
unsigned long long int	64

2.2 整数数据类型

“最小值”：

要求实现的至少要达到的一个最大值

常数表达式	最小值	X86-32	对象
UCHAR_MAX	255(2^8-1)	255	unsigned char
UCHAR_TMAX	65535(2^{16}-1)	65535	unsigned short char
UINT_MAX	2^{16}-1	2^{16}-1	unsigned int
ULONG_MAX	2^{32}-1	2^{32}-1	unsigned long int
ULLONG_MAX	2^{64}-1	2^{64}-1	unsigned long long int

2.2 整数数据类型

- 2.2.1 回绕

对于一个无符号的整数，在其最高值的递增结果是该类型的最小值。

0 1 2 3 4 5 6 7 8

000 001 010 011 100 101 110 111 000

2.2 整数数据类型

```
//2.2-1
#include <windows.h>
#include <limits.h>
#include <iostream>
using namespace std;
int main(   )
{
      unsigned int ui;
      ui=UINT_MAX;
      ui++;
      cout<<ui<<endl;
      ui=0;
      ui--;
      cout<<ui<<endl;
      system("pause");
      return 0;
}
```

2.2 整数数据类型

```
unsigned int i;
for( i=100; i>=0; --i)
```

循环何时停止？

2.2 整数数据类型

```
//2.2-2 检查最大值回绕
unsigned int sum=0;
if(sum>UINT_MAX)
{
        cout<<"too big"<<endl;
 }
else
{
        sum+=1000;
        cout<<sum<<endl;
}
```

2.2 整数数据类型

```
//2.2-2 检查最大值回绕
unsigned int sum=0;
if(sum>UINT_MAX-100)
{
        cout<<"too big"<<endl;
 }
else
{
        sum+=1000;
        cout<<sum<<endl;
}
```

2.2 整数数据类型

//2.2-3 检查最小值0回绕

```
unsigned int j=10,sum=100;
if(sum-j<0)
{
        cout<<"too small"<<endl;
}
else
{
        sum-=j;
}
```

2.2 整数数据类型

//2.2-3 检查最小值0回绕

```
unsigned int j=10,sum=100;
if(j>sum)
{
        cout<<"too small"<<endl;
}
else
{
        sum-=j;
}
```

2.2 整数数据类型

2.2.3 有符号整数类型

有符号整数类型用于表示正值和负值，其值的范围取决于该类型分配的位数以及其表示方式。

每一种无符号类型，都有一种对应的占用相同存储空间的有符号类型，bool除外。

2.2 整数数据类型

signed char	8
short int	16
int	32
long int	32
long long int	64

2.2 整数数据类型

负数表示有3中方法，原码表示(sign and magnitude)，反码表示(one’s complement)，补码表示(two’s complement)。

2.2 整数数据类型

- 原码表示法

符号位0表示正数，符号位1表示负数，其他值（非填充）表示以纯二进制表示法表示其绝对值。

43 0000101011

-43 1000101011

若要取一个原码的相反数，只要改变符号位。

2.2 整数数据类型

- 反码表示法

符号位具有权数-(2^{N-1}-1)，其他值位的权数与无符号类型相同。

43 0000101011

-43 1111010100

若要取一个反码的相反数，要改变每一位（包括符号位）。

2.2 整数数据类型

- 补码表示法

符号位具有权数-(2^{N-1})，其他值位的权数与无符号类型相同。

43 0000101011

-43 1111010101

若要取一个补码的相反数，首先构造反码的相反数，然后再加1。

2.2 整数数据类型

0	1	2	3	-4	-3	-2	-1
000	001	010	011	100	101	110	111

2.2 整数数据类型

- 2.2.4 有符号整数的取值范围

“最小值”确定每个标准有符号类型整数保证的可移植范围。

2.2 整数数据类型

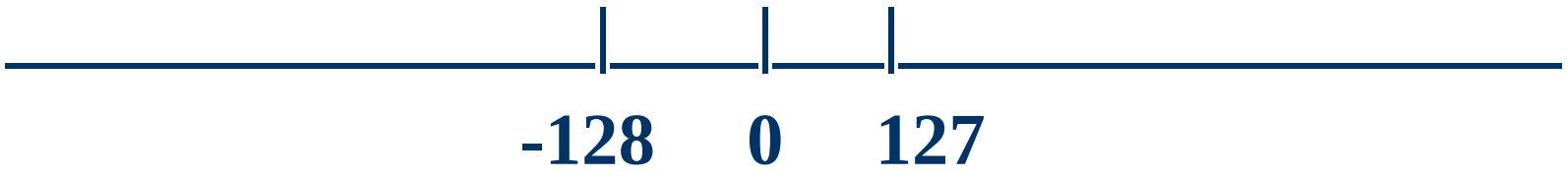
常数表达式	最小值	X86-32	对象
CHAR_MIN	-127//-(2^7-1)	-128	signed char
CHAR_MAX	+127//2^7-1	+127	signed char
SHRT_MIN	-32767//-(2^{15}-1)	-32768	short int
SHRT_MAX	+32767//2^{15}-1	+32767	short int
INT_MIN	-32767//-(2^{15}-1)	-2147483648	int
INT_MAX	+32767//2^{15}-1	+2147483647	int
LONG_MIN	-2147483648//-(2^{31}-1)	-2147483648	long int
LONG_MAX	+2147483647//2^{31}-1	+2147483647	long int
LLONG_MIN	-(2^{63}-1)	-9223372036854750808	long long int
LLONG_MAX	2^{63}-1	+9223372036854750807	long long int

2.2 整数数据类型

最小宽度	
signed char	8
short	16
int	16
long	32
long long	64

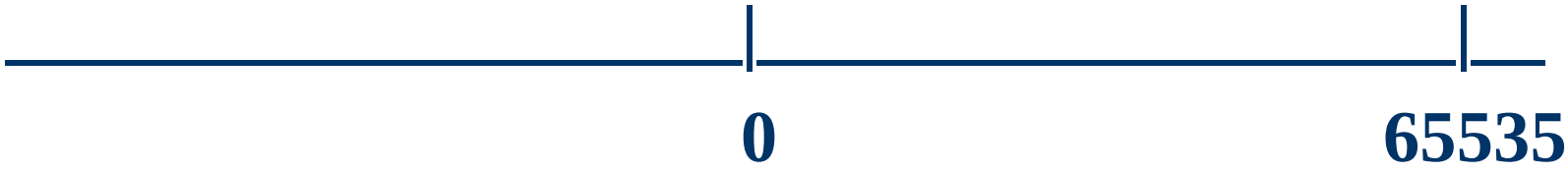
2.2 整数数据类型

signed char



unsigned char

short



2.2 整数数据类型

- 2.2.5 整数溢出

```
//2.2-4整数溢出演示
#include <windows.h>
#include <limits.h>
#include <iostream>
using namespace std;
int main()
{
        int i;
        i=INT_MAX;
        i++;
        cout<<i<<endl;
        i=INT_MIN;
        i--;
        cout<<i<<endl;
        system("pause");
        return 0;
}
```

2.2 整数数据类型

#define abs(n) ((n)<0 ? -(n):(n))

例： 2.2-5 字符串转换整数演示

2.3 整数转换

- 转换是一种用于表示赋值、类型强制转换或者计算的结果值的底层数据类型的改变。

小宽度->大宽度　安全

大宽度->小宽度　危险

2.3 整数转换

例：

unsigned char-> signed char->short int

8　　8　　16

2.3 整数转换

- 2.3.2 整数转换级别

有符号整数级别：

long long int > long int > int

> short int > signed char

无符号整数级别=有符号整数级别

2.3 整数转换

- 2.3.3 整数类型提升

如果一个整数类型具有低于或等于int 或unsigned int 的整数转换级别，那么它的对象表达式在用于一个需要int 或unsigned int 的表达式时，就会被提升。

2.3 整数转换

- 整数提升保留值，包括符号。

如果在所有的原始值中，较小的类型可以被表示为一个int，那么

原始值较小的类型会被转换成int

否则，它被转换成unsigned int

2.3 整数转换

```
int sum;
char c1,c2;
sum=c1+c2;
```

c1 和 c2 均被提升到 int

2.3 整数转换

```
signed char result, c1,c2,c3;
c1=100;                    //signed char 最大值127
c2=3;
c3=4;
result=c1*c2/c3;
```

c1*c2溢出

提升c1c2c3到int

2.4 整数操作

- 整数操作可能会导致异常：溢出、回绕、截断。

运算符	异常	运算符	异常	运算符	异常	运算符	异常
+	溢出 回绕	-=	溢出 回绕 截断	<<	溢出 回绕	<	无
-	溢出 回绕	*=	溢出 回绕 截断	>>	无	>	无
*	溢出 回绕	/=	溢出 截断	&	无	>=	无
%	溢出	<<=	溢出 回绕 截断	^	无	==	无
++	溢出 回绕	>>=	截断	~	无	!=	无
--	溢出 回绕	&=	截断	!	无	&&	无
=	截断	|=	截断	一元+	无 ||		无
+=	溢出 回绕 截断	^=	截断	一元-	溢出 回绕	?:	无

2.4 整数操作

- 2.4.1赋值

a = b

右操作数被转换为赋值表达式类型并替换存储在左操作数指定的对象的值。

2.4 整数操作

例

```
int f( );
char c;
……
if( (c=f( ))== -1)
……
```

f()的返回值int在赋值给c的时候可能会被截断

在char 具有unsigned char 相同范围的系统中，
转换结果永远不为负

2.4 整数操作

```
int f( );
int c;
……
if( (c=f( ))== -1)
……
```

2.4 整数操作

例：

```
char c;
int i;
long l;
if ( l==(c=i))
```

c=i过程中，i被转换为char，如果i不在char的范围内，则l==i永远不为真。

2.4 整数操作

例：

```
unsigned char sum, c1, c2;
c1=200;
c2=90;
sum=c1+c2;
```

c1+c2 被转换成int，赋值给sum时产生截断

2.4 整数操作

```
//2.4-1赋值符号扩展
signed int si=SCHAR_MAX+1;  //128----0x00000080
                            //00000000 00000000 00000000 10000000
signed char sc=si;          //si被截断 10000000

si=sc;   //补码显示-128---0x80
         //扩展sc 11111111 11111111 11111111 10000000
```

2.4 整数操作

```
//2.4-1赋值符号扩展
signed int si=SCHAR_MAX+1;  //128----0x00000080
                    //00000000 00000000 00000000 10000000
signed char sc=si;              //si被截断 10000000

si=(unsigned char)sc;
```

2.4 整数操作

- 2.4.2加法

加法可以以用来将两个操作数或者一个指针与一个整数相加。

如果将一个整数类型加到一个指针上，那么结果将是一个指针。

两个整数相加的结果总能够用比两个操作数中较大者的宽度大1位的数来表示。

2.4 整数操作

- 先验条件测试，补码表示

在操作之前测试操作数的值，防止溢出。

2.4 整数操作

```
//2.4-2先验条件测试1
signed int si1,si2,sum;
si1=2147483647;
si2=10;
unsigned int usum=(unsigned int)(si1+si2);  //预计算
if( (usum^si1)&(usum^si2)&INT_MIN )  //异或测试符号位，只对补码有效
{ /*ERROR*/}
else
{
        sum=si1+si2;
}
```

2.4 整数操作

```
//2.4-3 先验条件测试2
unsigned int ui1,ui2,usum;
ui1=2000;
ui2=10;
If ( UINT_MAX - ui1<ui2 )
{ /*ERROR*/ }
else
{
        usum=ui1+ui2;
}
```

2.4 整数操作

- 后验条件测试

在操作被执行后进行，测试所得到的的值，以确定它是否在有效范围内。

2.4 整数操作

```
//2.4-4 加法后验条件测试
unsigned int ui1,ui2,usum;
ui1=UINT_MAX-1;
ui2=10;
usum=ui1+ui2;
If (    usum  <  ui1    ) //后验条件测试
{
        /*ERROR*/
}
```

2.4 整数操作

■ 2.4.3 减法

减法也是一种加法操作，两个操作数必须是算数类型或指向兼容对象类型的指针。

2.4 整数操作

如果两个操作数异号，并且结果与第一个操作数不同，则发生减法溢出。

2.4 整数操作

```
//2.4-5减法先验条件测试1
signed int si1,si2,result;
si1=INT_MIN;
si2=10;
if(    (si1^si2)&(((unsigned int)(si1-si2))^si1)&INT_MIN    ) //检测溢出
{  /*ERROR*/}
else
{
        result=si1-si2;
}
```

2.4 整数操作

如果两个操作数之差是负数，则无符号减法产生回绕。

2.4 整数操作

```
//2.4-6减法先验条件测试2
unsigned int ui1,ui2,udiff;
ui1=5;
ui2=10;
if( ui1 < ui2 ) //检测回绕
{ /*ERROR*/}
else
{
        udiff=ui1-ui2;
}
```

2.4 整数操作

```
//2.4-7减法后验条件测试
unsigned int ui1,ui2,udiff;
ui1=5;
ui2=10;
udiff=ui1-ui2;
if(   udiff  >  ui1 )   //检测回绕
{
        /*ERROR*/
}
```

Summary

unsigned char	8	signed char	8
unsigned short int	16	short int	16
unsigned int	32	int	32
unsigned long int	32	long int	32
unsigned long long int	64	long long int	64