CIS 4360 Secure Computer Systems

Integers

Professor Qiang Zeng

Spring 2017

TEMPLE UNIVERSITY

Some slides are courtesy of Dr. Robert Seacord

Previous Class

- Buffer overflows can be devastating
  - C/C++/Objective-C vulnerable to them
  - Most other languages not natively vulnerable, but many components/languages in C/C++
- The system has done a lot for us (but it is insufficient)
  - Canary
  - DEP
  - ASLR
- What you should do
  - Always enforce bounds checking in your code
  - Use snprintf/strlcpy/strlcat in your code (Do not use strncpy)

Previous Class

Compare strcpy, strncpy, and snprintf

(1) strcpy(dst, src): regardless of the size of the dst buffer, copy the src (including the ending null) to dst

(2) strncpy(dst, src, n): copy at most n characters from src to dst (the ending null is not guaranteed); extra space will be padded with 0’s (so inefficient)

(3) snprintf(dst, n, "%s", src): copy at most n-1 characters from src to dst and the ending null is guaranteed

Previous Class

What concrete attacks can be launched by exploiting a buffer overflow vulnerability? List at least four.

1. Stack buffer overflow can conveniently overwrites the return address on the stack to hijack the control flow
2. Stack buffer overflow can overwrite local variables
3. Heap buffer overflow can modify buffers adjacent to the vulnerable buffer
4. A sub-object buffer overflow can modify the adjacent fields of the containing object

* Thanks to some compilation option (e.g., SSP in gcc), 1 and 2 have been largely mitigated

The System Has Done the Following…

- Canary
  - Insert a secret value (cookie) between the return address and the buffer on the stack at call entry, and check the canary integrity at the call exit. If the canary is corrupted, a buffer overflow is detected
  - Very effective against stack based overflows
- DEP: Data Execution Prevention
  - Data on stack and heap cannot be executed
  - Prevent injected code from being executed
- ASLR (Address Space Layout Randomization)
  - Scramble the locations of libraries and executables
  - So attackers do not know how to reuse their code

Outline

- Integer Representation and Ranges
- Rank
- Integer Conversion
  – a = b
  – a op b // op is +, -, *, /, <, >, …

Something that may shock you!

- int x = -2;
- unsigned int y =1000;
- x > y ?
- Yes!

- short a = -2;
- unsigned short b = 1000;
- return a > b ? 1 : 0;
- No!

- unsigned int g = -2; // what is the value of g?
- unsigned int h = g+1; // what is the value of h;
- h == UINT_MAX (i.e., 0xffffffff)

Another motivating example

- In binary search, m = (low + high) /2
  - This contains an integer overflow vulnerability
  - E.g., low = 8; high = INT_MAX – 4
  - You expect m = (INT_MAX + 4) / 2
  - But the real value you get is m = (INT_MIN + 3) / 2
  - You will find out the reason and the solution

Integer Representation: Two’s Complement

The two’s complement form of a negative integer is created by adding one to the one’s complement representation.

$$\begin{tikzcd}
0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & & & & 0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & + & 1 & = & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1
\end{tikzcd}$$

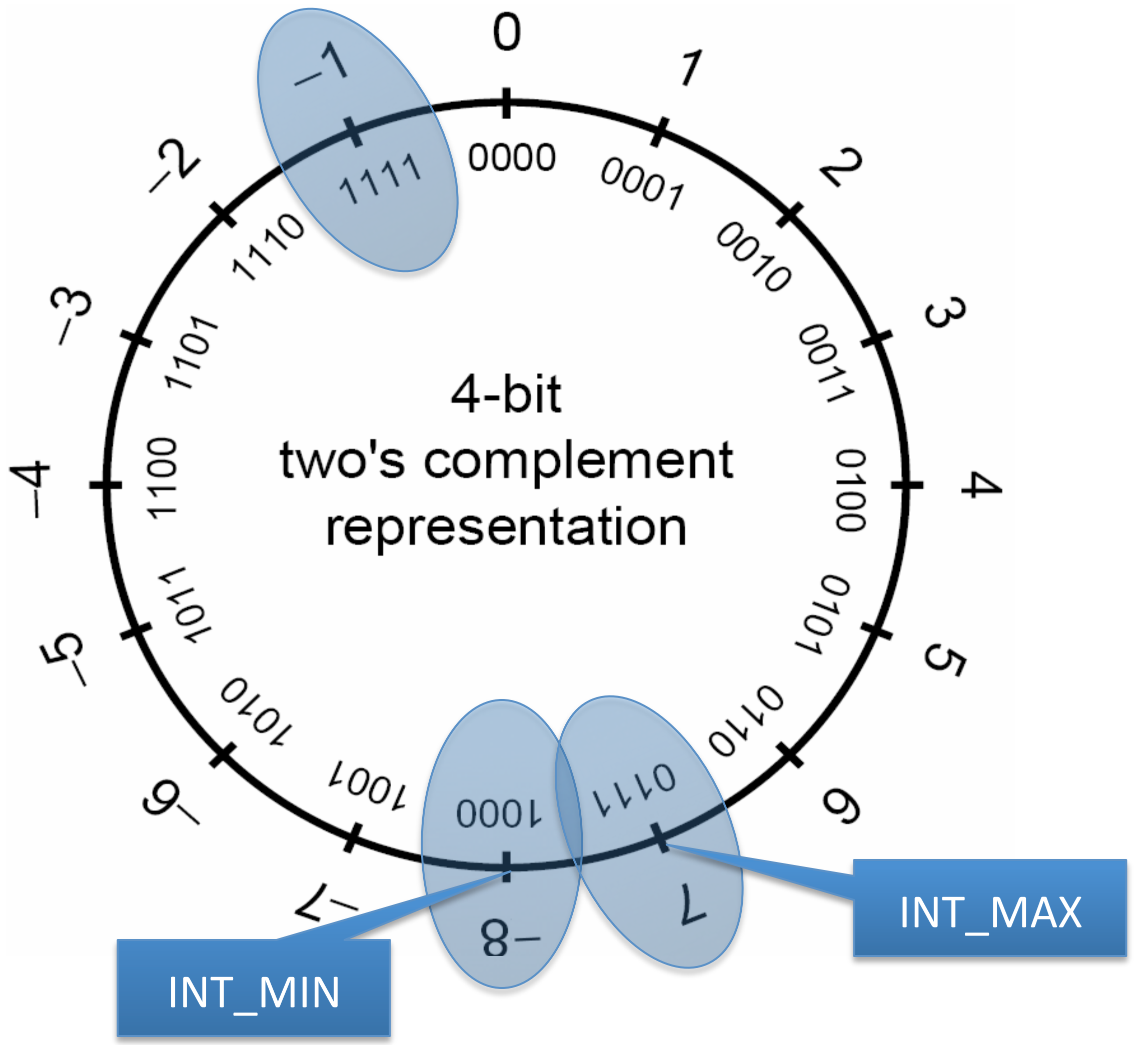
$$\begin{tikzcd}
\boxed{\text{Write the binary representation of its absolute value}} \arrow[r] & \boxed{\text{Flip its bits}} \arrow[r] & \boxed{\text{Add one}}
\end{tikzcd}$$

Signed Integers

Signed integers are used to represent positive and negative values.

On a computer using two’s complement arithmetic, a signed integer ranges from -2^{n-1} through $2^{n-1}-1$.

Signed Integer Representation



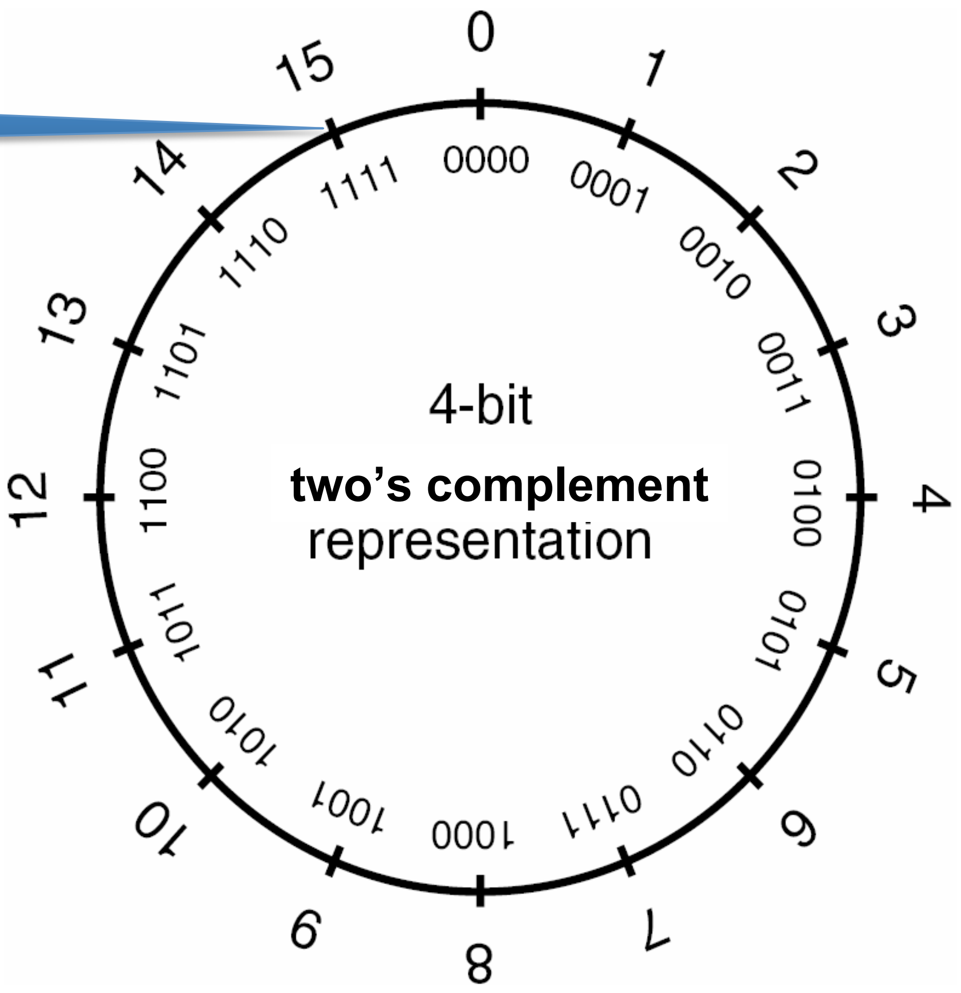
Unsigned Integers

- Unsigned integer values range from zero to a maximum that depends on the size of the type

This maximum value can be calculated as 2^n-1, where n is the number of bits used to represent the unsigned type.

Unsigned Integer Representation

UINT_MAX



Overflow Examples

```c
int i;
unsigned int j;

i = INT_MAX;  // 2,147,483,647
i++;
printf("i = %d\n", i);

j = UINT_MAX; // 4,294,967,295;
j++;
printf("j = %u\n", j);
```

```
i=-2,147,483,648
```

```
j = 0
```

Standard Integer Types

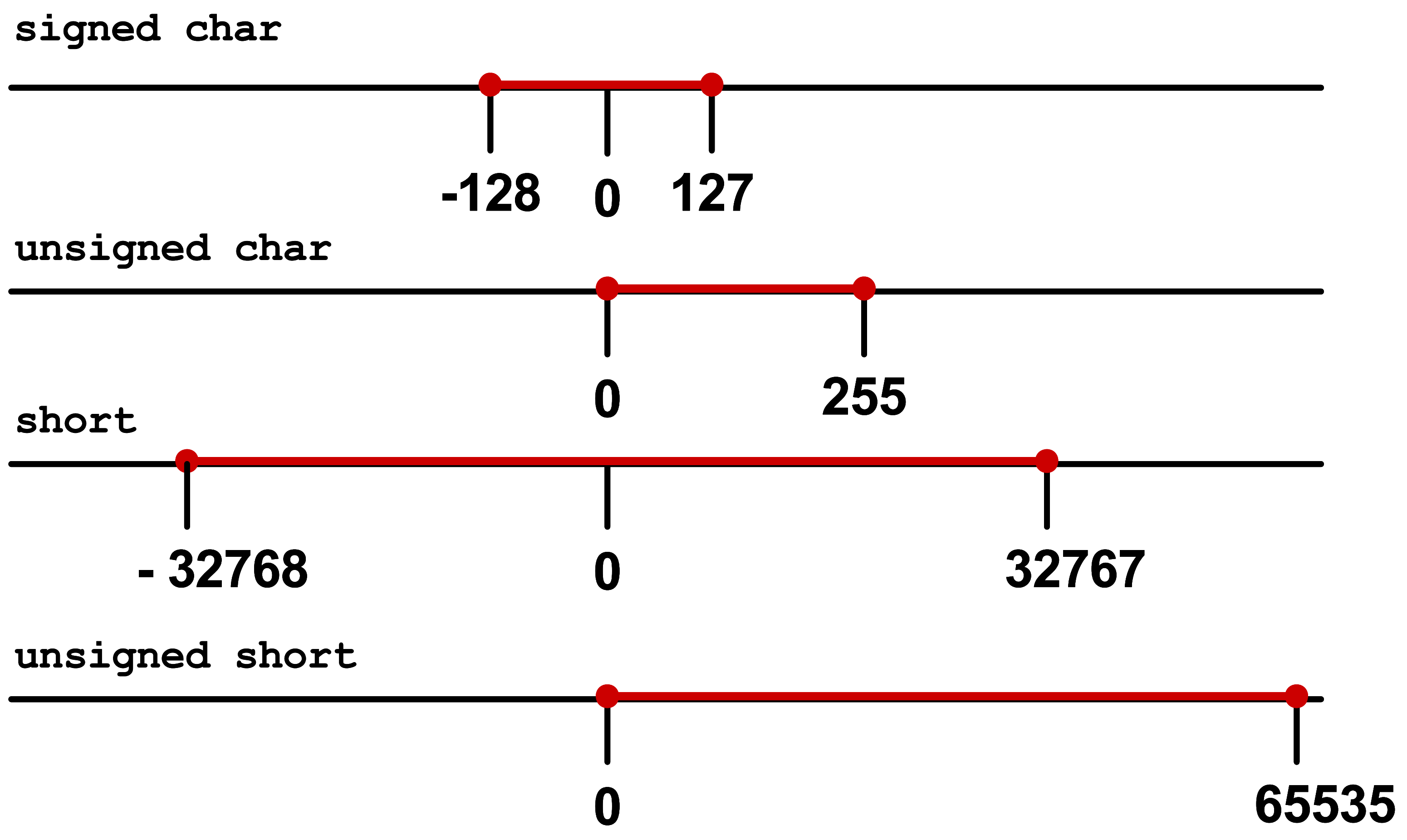
Standard integers include the following types, in non-decreasing length order:

- `signed char`
- `short int`
- `int`
- `long int`
- `long long int`

size_t is the (unsigned) result of the **sizeof** operator

char can be singed or unsigned depending on implementation

Example Integer Ranges



Term: Rank

No two signed integer types have the same rank, even if they have the same representation.

The rank of **`long long int`** is > the rank of **`long int`**, which is > the rank of **`int`**, which is > the rank of **`short int`**, which is > the rank of **`signed char`**.

The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type.

```
a = b; // seems simple but really?
```

Signed Converted to Signed

- When a signed integer is converted to a signed integer with equal or greater size
  - the value will not change (sign-extended)
- When a signed integer is converted to a signed integer with smaller size
  - The result is implementation-defined
  - But in most implementations, the high-order bits will be truncated and the remaining bits are re-interpreted

Example

```
// Note this is implementation dependent, though
        long x3= ~0L & (~(1<<15));
        short y3 = x3;
        cout << hex;
        cout << "x3 = 0x" << x3 <<endl;
        cout << "y3 = 0x" << y3 <<endl;
        cout << dec;
        cout << "x3 = " << x3 <<endl;
        cout << "y3 = " << y3 <<endl;
// result
        x3 = 0xffffffffffff7fff
        y3 = 0x7fff
        x3 = -32769
        y3 = 32767
```

Signed Converted to Unsigned

- When a signed integer x is converted to an unsigned integer y with equal or greater size
  - y = x % (dst_type_max + 1) // sign-extended and then re-interpreted
  - E.g., short x = -2; // 0xfffe

    unsigned long y = x; //0xfffffffffffffffe
- When a signed integer x is converted to an unsigned integer y of smaller size
  - y = x % (dst_type_max + 1) // truncation and re-interpreted
  - E.g., long x = -2; //0xfffffffffffffffe

    unsigned short y = x; // 0xfffe

Unsigned Converted to Unsigned

- When an unsigned integer x is converted to an unsigned integer y with equal or greater size
  - The value is not changed // zero-extended
- When an unsigned integer x is converted to an unsigned integer y with smaller size
  - y = x % (dst_type_max + 1); // truncation
  - E.g., unsigned long x = 0xfffffffe;

    unsigned short y = x;

    // 0xfffffffe % 0x10000 = 0xfffe

Unsigned Converted to Signed

- When an unsigned integer x is converted to a signed integer y with lager size
  - More precisely, all instances of x's type can be represented by y's type
  - The value does not change
- When an unsigned integer x is converted to a signed integer y with equal or smaller size
  - The conversion will be implemented dependent
  - But in many systems, higher-order bits will be truncated and the remaining bits are re-interpreted
  - E.g., unsigned long x = ~0UL & (~(1<<15));

    ```
    short y = x;
    // x = 0xffffffffffff7fff
    // y5 = 7fff
    ```

Re-cap about Conversion

- A conversion to a type that can represent the value being converted is always well-defined
  - the value simply stays unchanged
  - E.g., signed -> larger signed; unsigned -> larger unsigned; unsigned short -> int
- A conversion to an unsigned type is always well-defined
  - y = x % (dst_type_max +1)
- A conversion to a signed type that cannot represent the value being converted results is implementation dependent
  - But in many systems: truncate the higher-order bits and re-interpret the remaining

a + b; // seems simple but really?

Similarly, a – b, a > b, a < b, …

Integer Conversions – Integer Promotion

Integer types smaller than **`int`** are promoted when an operation is performed on them.

If all values of the original type can be represented as an **`int`**

- the value of the smaller type is converted to **`int`**
- otherwise, it is converted to **`unsigned int`**

Integer promotions are applied as part of the usual arithmetic conversions.

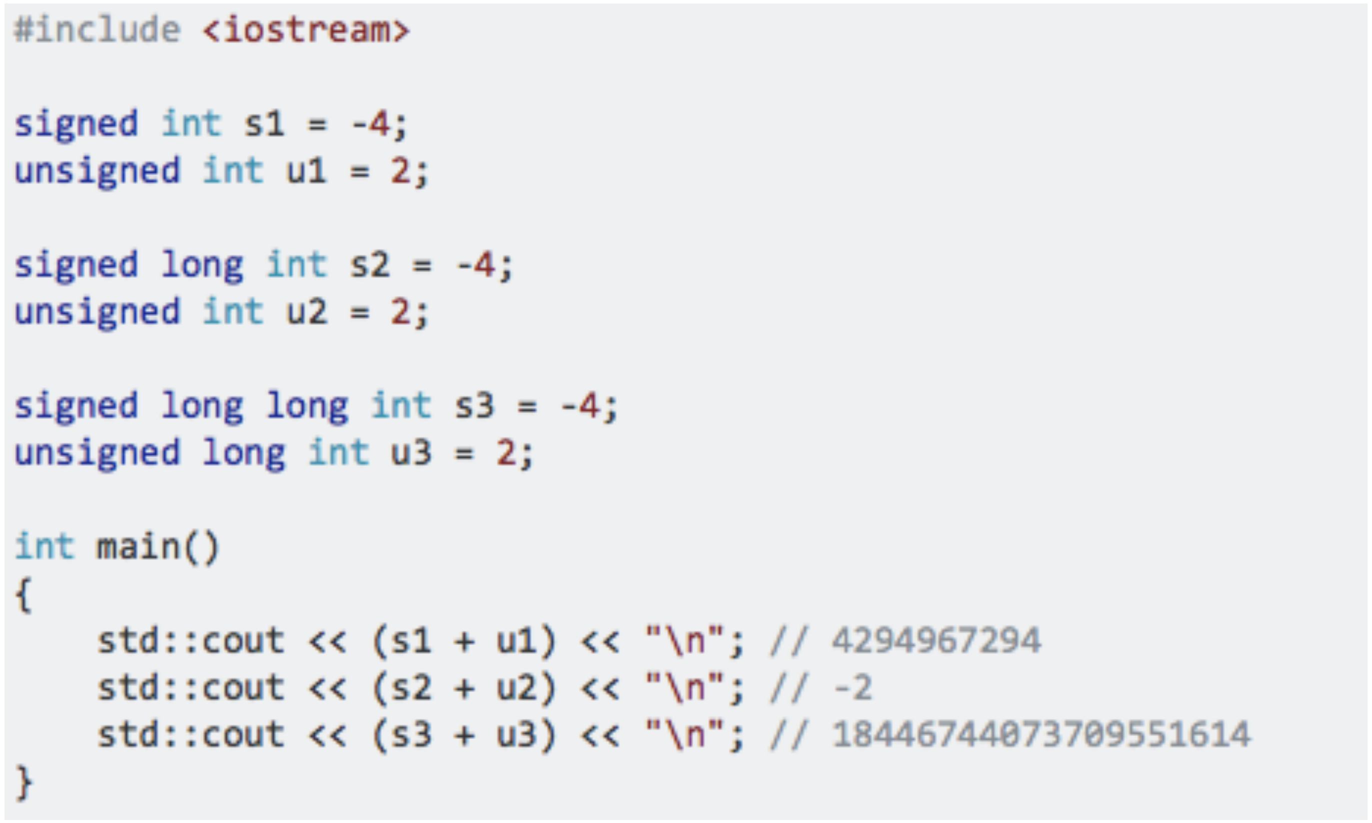
Why is Integer Promotion Beneficial?

- Consider the following example

```
unsigned short x = USHORT_MAX; // 0xffff
unsigned short y = 1;
int z = x + y; // 0x0 or 0x10000?
```

- If there is no integer promotion, z would be 0 due to the overflow; as both x and y have been promoted to “int”, z will get a correct result 0x10000

Integer Conversion Examples



```
#include <iostream>

signed int s1 = -4;
unsigned int u1 = 2;

signed long int s2 = -4;
unsigned int u2 = 2;

signed long long int s3 = -4;
unsigned long int u3 = 2;

int main()
{
    std::cout << (s1 + u1) << "\n"; // 4294967294
    std::cout << (s2 + u2) << "\n"; // -2
    std::cout << (s3 + u3) << "\n"; // 18446744073709551614
}
```

- Why “-4 + 2” in the three statements gives a different result each?

Integer Conversion Rules for “a op b”

0. Integer Promotion always occur for char and short
1. If both operands have the same type and sign, no conversion is needed
2. If both operands have the same sign, the operand with the type of lesser rank is converted to the type of the operand with greater rank.
3. Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.
4. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.
5. Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

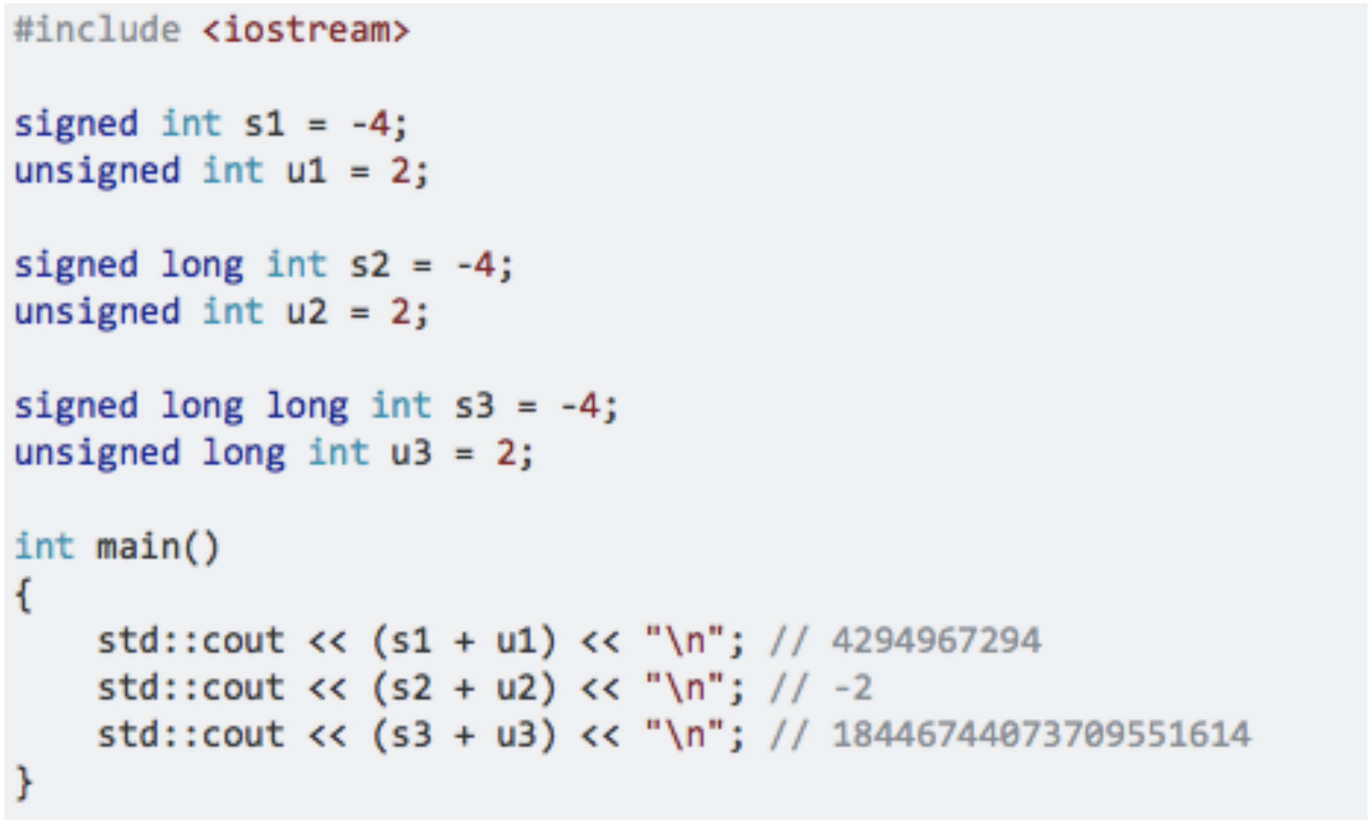
unsigned a; signed b;

Rank(a) >= Rank (b), apply rule 3

Rank(a) < Rank(b) and b’s type has more bits, apply rule 4

Rank(a) < Rank(b) and b’s type does not have more bits, apply rule 5

Integer Conversion Examples



```
#include <iostream>

signed int s1 = -4;
unsigned int u1 = 2;

signed long int s2 = -4;
unsigned int u2 = 2;

signed long long int s3 = -4;
unsigned long int u3 = 2;

int main()
{
    std::cout << (s1 + u1) << "\n"; // 4294967294
    std::cout << (s2 + u2) << "\n"; // -2
    std::cout << (s3 + u3) << "\n"; // 18446744073709551614
}
```

- 64-bit system: int 32 bits, long 64, long long 64
- The three outputs apply the rules (3), (4), (5) respectively

References

- Secure Coding in C and C++. 2nd edition Seacord. 2013.
  - http://ptgmedia.pearsoncmg.com/images/0321335724/samplechapter/seacord_ch05.pdf
- C++ implicit Conversion
  - http://stackoverflow.com/a/19274544/577165
  - http://stackoverflow.com/a/17833338