



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# C/C++ Program Design

## CS205

**Prof. Shiqi Yu (于仕琪)**

yusq@sustech.edu.cn

<http://faculty.sustech.edu.cn/yusq/>



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Integer Numbers



# int

- **int** is the most frequently used integer type

```
int i; //declare a variable  
int j = 10; //declare and initialize  
int k;  
k = 20; //assign a value
```

- Remember to initialize a variable!
- Will the compiler give an error?

```
int i;  
cout << i; //what is i's value?
```



# Variable Initialization

init.cpp

```
#include <iostream>
using namespace std;
int main()
{
    int num1; //bad: uninitialized variable
    int num2; //bad: uninitialized variable
    cout << "num1 = " << num1 << endl;
    cout << "num2 = " << num2 << endl;
}
```

```
yushiqi: examples $ g++ init.cpp
yushiqi: examples $ file a.out
a.out: Mach-O 64-bit executable x86_64
yushiqi: examples $ ./a.out
num1 = 2
num2 = 84402213
```

```
yushiqi: examples $ g++ init.cpp
yushiqi: examples $ file a.out
a.out: Mach-O 64-bit executable arm64
yushiqi: examples $ ./a.out
num1 = 0
num2 = 0
```

- Uninitialized variables may have random values
- The behavior depends on the compiler. Clang (x86\_64) and Clang (arm64) in the demo.
- Please initialize variables **EXPLICITLY!**



# How to initialize

```
int num;
```

```
num = 10; //do not forget this line
```

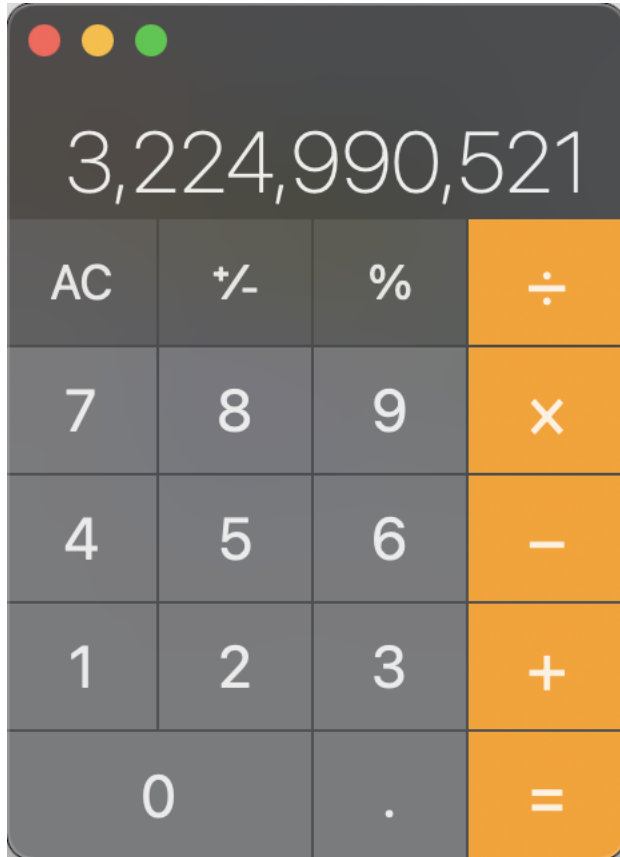
```
int num = 10;
```

```
int num (10);
```

```
int num {10};
```



# Overflow



overflow.cpp

```
int a = 56789;  
int b = 56789;  
int c = a * b;  
cout << c << endl;
```

The output is a **negative** number!

**-1069976775**

Because 56789 is 0xDDD5, 16 bits

The correct result is  
3,224,990,521 (0x C0 39 73 39).

The sign bit is 1!



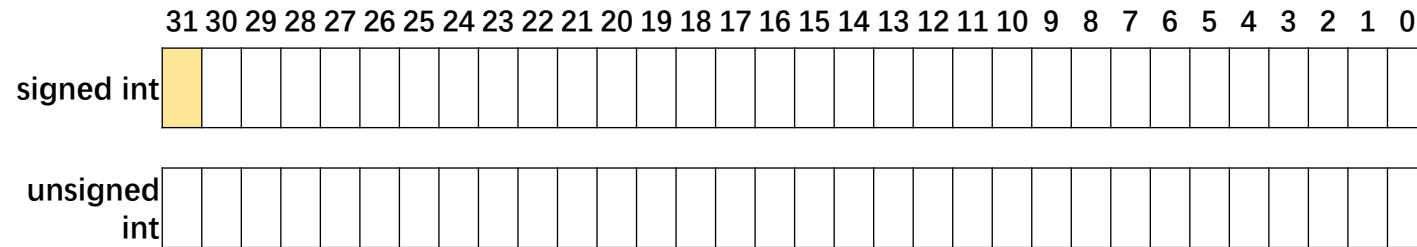
# signed and unsigned

- The following code can give the correct answer.

```
unsigned int a = 56789;
```

```
unsigned int b = 56789;
```

```
unsigned int c = a * b;
```



- `signed int` can be shorten as `int`. Its range is  $[-2^{31}, 2^{31}-1]$  if it's 32-bit.
- `unsigned int`: Its range is  $[0, 2^{32}-1]$  if it's 32-bit.
- 32 bits for most modern systems, 16 for some old ones.



# Different Data Types for Integer

- use `long int` for longer integers.
- use `short int` for shorter integers.
- and `long long`

But

- C and C++ standards do not fix the widths of them



Type specifier	Equivalent type	Width in bits by data model									
		C++ standard	LP32	ILP32	LLP64	LP64					
<code>short</code>	<code>short int</code>	at least <b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>	<b>16</b>					
<code>short int</code>											
<code>signed short</code>											
<code>signed short int</code>											
<code>unsigned short</code>	<code>unsigned short int</code>										
<code>unsigned short int</code>											
<code>int</code>	<code>int</code>	at least <b>16</b>	<b>16</b>	<b>32</b>	<b>32</b>	<b>32</b>					
<code>signed</code>											
<code>signed int</code>											
<code>unsigned</code>	<code>unsigned int</code>										
<code>unsigned int</code>											
<code>long</code>	<code>long int</code>	at least <b>32</b>	<b>32</b>	<b>32</b>	<b>32</b>	<b>64</b>					
<code>long int</code>											
<code>signed long</code>											
<code>signed long int</code>											
<code>unsigned long</code>	<code>unsigned long int</code>										
<code>unsigned long int</code>											
<code>long long</code>	<code>long long int</code> (C++11)	at least <b>64</b>	<b>64</b>	<b>64</b>	<b>64</b>	<b>64</b>					
<code>long long int</code>											
<code>signed long long</code>											
<code>signed long long int</code>											
<code>unsigned long long</code>	<code>unsigned long long int</code> (C++11)										
<code>unsigned long long int</code>											

- Width in bits of different data models
- `sizeof` operator can return the width in bytes.



# sizeof

- It is an operator, not a function!

size.cpp

```
int i = 0;  
short s = 0;  
cout << "sizeof(int)=" << sizeof(int) << endl;  
cout << "sizeof(i)=" << sizeof(i) << endl;  
cout << "sizeof(short)=" << sizeof(s) << endl;  
cout << "sizeof(long)=" << sizeof(long) << endl;  
cout << "sizeof(size_t)=" << sizeof(size_t) << endl;
```



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# More Integer Types



# char

- `char`: type for character, 8-bit integer indeed!
- `signed char`: signed 8-bit integer
- `unsigned char`: unsigned 8-bit integer
- `char`: **either** `signed char` **or** `unsigned char`



# Integers and characters

- How we represent a character?

➤ Use an 8-bit integer

char.cpp

```
char c1 = 'C';
```

```
char c2 = 80;
```

```
char c3 = 0x50;
```

- Chinese characters?

```
char16_t c = u'于'; //c++11
```

```
char32_t c = U'于'; //c++11
```



# bool

- A C++ keyword, but not a C keyword
- bool width: 1 byte (8 bits), **NOT** 1 bit!
- Value: **true** (1) or **false** (0)

What is the output?

`bool.cpp`

```
bool b = true;
```

```
int i = b;
```

```
cout << "i=" << i << endl;
```

```
cout << "b=" << b << endl;
```



# bool

- Boolean data conversion

```
bool b = true;  
int i = b; // the value of i is 1.
```

```
bool b = -256; // unrecommended conversion. the value of b is  
true
```

```
bool b = (-256 != 0); // better choice
```



# Boolean in C

- Use `typedef` to create a type

```
typedef char bool;  
#define true 1  
#define false 0
```

- Defined in `stdbool.h` since C99

```
#include <stdbool.h>
```





# size\_t

- Computer memory keeps increasing
- 32-bit int was enough in the past to for data length
- But now it is not.

`size_t`:

- Unsigned integer
- Type of the result of `sizeof` operator
- Can store the maximum size of a theoretically possible object of any type
- 32-bit, or 64-bit



# Fixed width integer types (since C++11)

Defined in <cstdint>

int8\_t

int16\_t

int32\_t

int64\_t

uint8\_t

uint16\_t

uint32\_t

uint64\_t

...

Some useful macros

INT8\_MIN

INT16\_MIN

INT32\_MIN

INT64\_MIN

INT8\_MAX

INT16\_MAX

INT32\_MAX

INT64\_MAX

...

intmax.cpp

```
#include <iostream>
#include <cstdint>
using namespace std;
int main()
{
    cout << "INT8_MAX=" << INT8_MAX << endl;
}
```



# Choose appropriate integer types

- Wider integers consume more memory, and slower sometimes
- `char(byte)` is widely used for image pixels
- Choose a data type carefully, and consider all possibilities (short for wide dynamic range images)

$$6720 \times 3780 \times 3 = 76,204,800 = 76\text{M Bytes}$$





南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Floating-point Numbers



# What's the output?

float.cpp

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    float f1 = 1.2f;
```

```
    float f2 = f1 * 10000000000000000.0f; //1.0e15
```

```
    cout << std::fixed << std::setprecision(15) << f1 << endl;
```

```
    cout << std::fixed << std::setprecision(15) << f2 << endl;
```

```
    return 0;
```

```
}
```



- How many numbers in range  $[0, 1]$ ?

**Infinite!**

- How many numbers can 32 bits represent?

**$2^{32}$**

- You want 1.2, but `float` can only provide you 1.2000000047683716...



# Understanding Computing

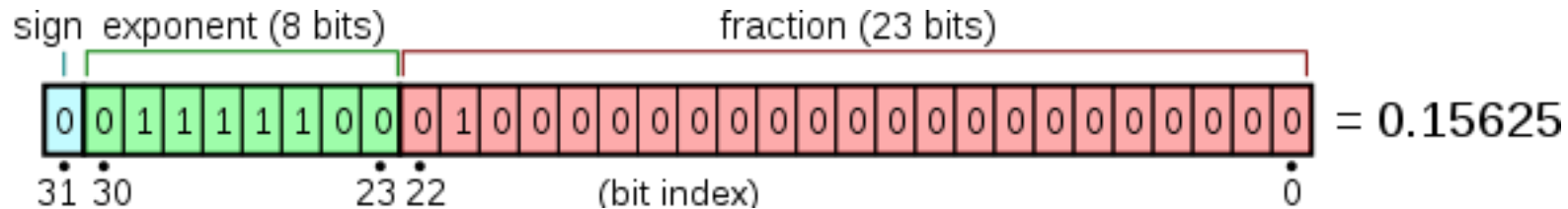
- Are computers always accurate?
- Floating-point operations always bring some tiny errors.
- Those errors cannot be eliminated.
- What we can do: to manage them not to cause a problem.





# Floating-point types

- `float`: single precision floating-point type, 32 bits



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\dots b_{23})_2 - 127} \times (1.b_{22}b_{21}\dots b_0)_2$$

- `double`: double precision floating-point type, 64 bits
- `long double`: extended precision floating-point type
  - 128 bits if supported
  - 64 bits otherwise
- half precision floating-point, 16 bits  
(popular in deep learning, but not a C++ standard)





# Floating-point VS integers

- Represent values between integers
- A much greater range of values
- Floating-point operations are slower than integer operations
- Lose precision
- `double` operations is slower than `float`



# Precision

- Will f2 be greater than f1?

precision.cpp

```
float f1 = 2.34E+10f;
```

```
float f2 = f1 + 10; // but f2 = f1
```

- Why?
- Can we use == operator to compare two floating point numbers?

```
if (f1 == f2) //bad
```

```
if (fabs(f1 - f2) < FLT_EPSILON) // good
```



# inf and nan

- What will f1 and f2 be?

nan.cpp

```
float f1 = 2.0f / 0.0f;
```

```
float f2 = 0.0f / 0.0f;
```

- $\pm$ inf: infinity (Exponent=11111111, fraction=0)
- nan: not a number (Exponent=11111111, fraction!=0)



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Arithmetic Operators



# Constant numbers

95 // decimal

0137 // octal

0x5F // hexadecimal

3.14159 // 3.14159

6.02e23 //  $6.02 \times 10^{23}$

1.6e-19 //  $1.6 \times 10^{-19}$

3.0 // 3.0

95 // int

95u // unsigned int

95l // long

95ul // unsigned long

95lu // unsigned long

6.02e23L // long double

6.02e23f // float

6.02e23 // double



# const type qualifier

```
const float pi = 3.1415926f;
```

```
pi += 1; //error!
```

- If a variable/object is const-qualified, it cannot be modified.
- It must be initialized when you define it.



# auto (since C++11)

**auto** is placeholder type specifier.

The type of the variable will be deduced from its initializer.

```
auto a = 2; // type of a is int
auto bc = 2.3; // type of b is double
auto c ; //valid in C, error in C++
auto d = a * 1.2;
```

- Question:

```
auto a = 2; // type of a is int
```

```
// will a be converted to a
// double type variable?
```

```
a = 2.3;
```

**No!** 2.3 will be converted to a `int` 2, then assigned to `a`



# Arithmetic operators

Operator name	Syntax
unary plus	+a
unary minus	-a
addition	a + b
subtraction	a - b
multiplication	a * b
division	a / b
modulo	a % b
bitwise NOT	~a
bitwise AND	a & b
bitwise OR	a   b
bitwise XOR	a ^ b
bitwise left shift	a << b
bitwise right shift	a >> b

- Operator Precedence

If you cannot remember the precedence, use parentheses!

➤ a++

➤ ++a

➤ \* /

➤ + -

➤ << >>





# Other operators

## Assignment Operators

`a = b`

`a += b`

`a -= b`

`a *= b`

`a /= b`

`a %= b`

`a &= b`

`a |= b`

`a ^= b`

`a <<= b`

`a >>= b`

## Increment/decrement

`a++`

`++a`

`a--`

`--a`

```
int a = 3;  
int b = a++; // What's the value of b?  
int c = ++a; // What's the value of c?
```



# Data type conversions

## conversion.cpp

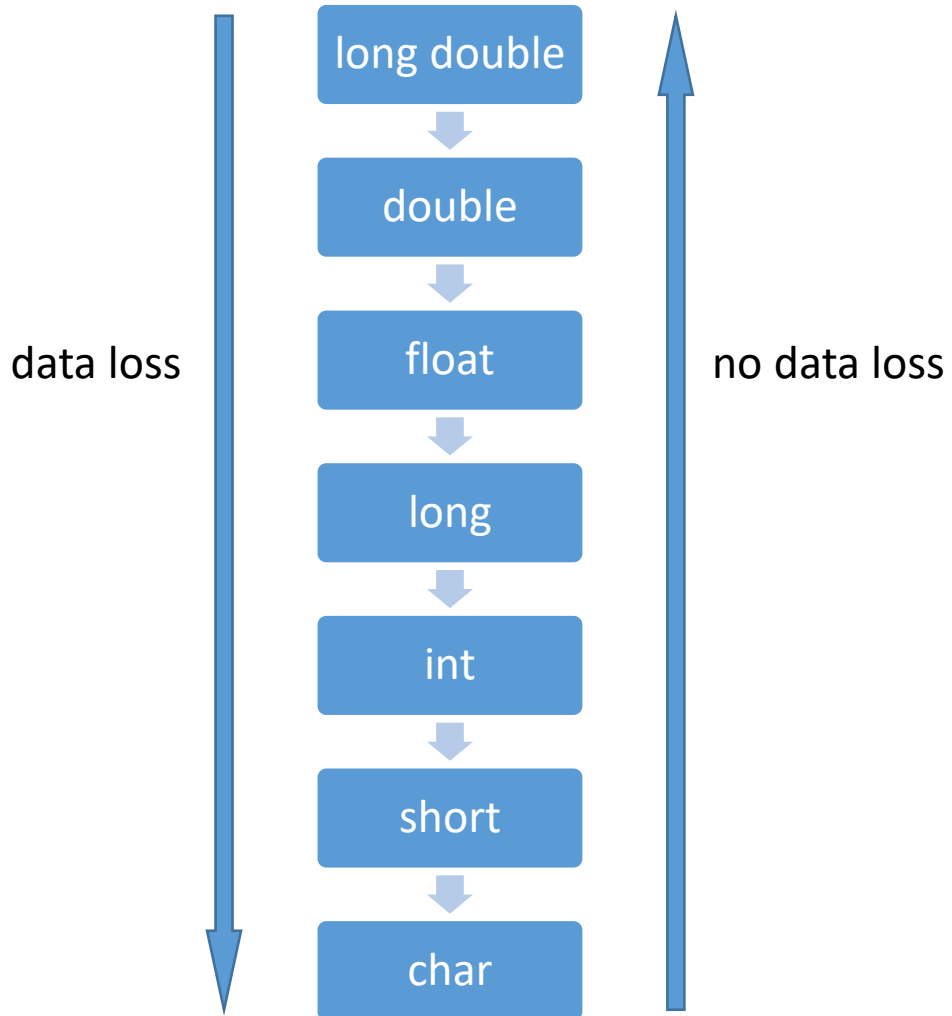
```
int num_int1 = 9; // initializing an int value to num_int1
int num_int2 = 'C'; // implicit conversion
int num_int3 = (int)'C'; // explicit conversion, C-style
int num_int4 = int('C'); // explicit conversion, function style
int num_int5 = 2.8; //implicit conversion
float num_float = 2.3; //implicit conversion from double to float
short num_short = 650000;
```

## DANGER:

- The source code can be compiled successfully (even with warning messages) when the data types do not match.
- Please use explicit conversion if possible



# Data loss



But

`conversion.cpp`

```
int num_int1 = 100000004;  
float num_int_float = num_int1;  
int num_int2 = (int)(num_int_float);
```

Will `num_int2` be the same with `num_int1`?



# Operator Associativity

- Left-to-right associativity or a right-to-left associativity
  - Ref: [https://en.cppreference.com/book/operator\\_precedence](https://en.cppreference.com/book/operator_precedence)
- The following two lines are equivalent.

```
int i = 17 / 5 * 5;
```

```
int i = (17 / 5) * 5;
```



# Divisions

- Both operands are integers
  - Perform **integer division**
  - Any fractional part of the answer is discarded to make the result an integer  
`float f = 17 / 5; // f will be 3.f, not 3.4f.`
- One or both operands are floating-point numbers
  - Perform floating-point division  
`float f = 17 / 5.f; // f will be 3.4f.`



# Distinct Operations for Different Types

- `int, long, float, double`: four kinds of operations
- If the operands are not the four types, automatic convert their types  
`unsigned char a = 255;`  
`unsigned char b = 1;`  
`int c = a + b; // c = ?`
- The operands will be converted to one of the four types without losing data: `int, long, float, double`
  - Ref: [https://en.cppreference.com/w/cpp/language/implicit\\_conversion](https://en.cppreference.com/w/cpp/language/implicit_conversion)



# C/C++ Supposes

- You (the programmer) are smart enough!
- You know what exactly the source code means!