

C/C++ Programming Language

CS205 Fall

Feng Zheng

2023.09.21



南方科技大学

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



Content

- Brief Review
- Dealing With Data
 - Fundamental types
 - ① Integral Variables
 - ② Floating-Point Numbers
 - ③ C++ Arithmetic Operators

Brief Review



Components of Program

- Files created by **yourself**
 - Header file
 - Code file
- C/C++ **standard** file included
- **Static library** file included
 - Header file
 - Compiled **code** in object file
- **Dynamic** link file (dynamic lib)





Components of Function

- Function prototype
- Function header
 - Arguments
 - Name
 - Return type
- Function body
 - Statements
 - A return statement
- Pair of braces





Others

- Preprocessor directives
- Compilers
- Declaration of variable
- Identifiers

Dealing with Data



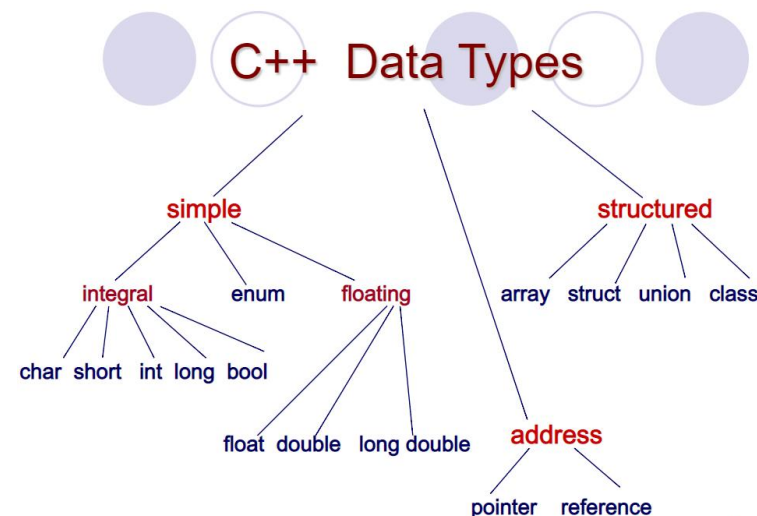
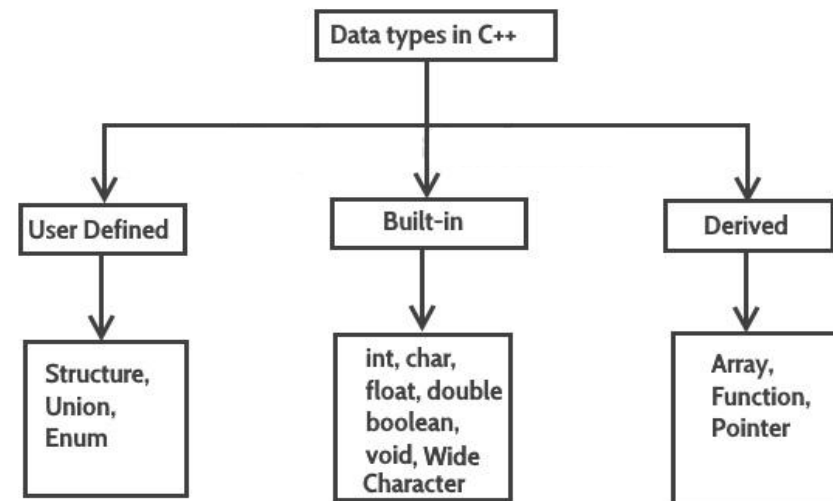
Dealing with Data (Content)

- **Rules** for naming C++ variables (identifiers)
- C++'s built-in **integer** types (no fractional part)
 - The **climits** file, which represents system limits for various integer types
 - Numeric literals (**constants**) of various integer types
- C++'s built-in **floating-point** types: float, double, and long double
 - The **cfloat** file, which represents system limits for various floating-point types
- C++'s **arithmetic** operators
- Automatic type **conversions**
- Forced type **conversions** (type casts)



Data Types in C++

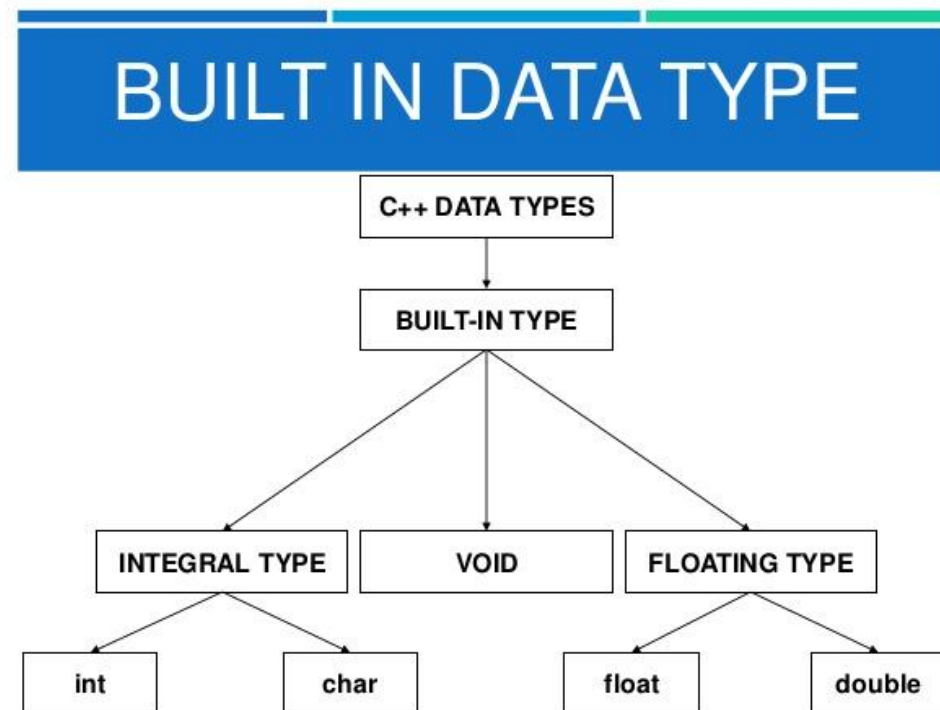
- The essence of OOP is designing and extending your own **data types**
- Built-in types will be your building **blocks**
- Be aware of
 - **Address** in C/C++ is also a variable
 - **Void** is a type (function)





Built-in C++ Types

- Fundamental types
 - Integers
 - Floating-point numbers
- Compound types
 - Arrays
 - Strings
 - Pointers
 - Structures

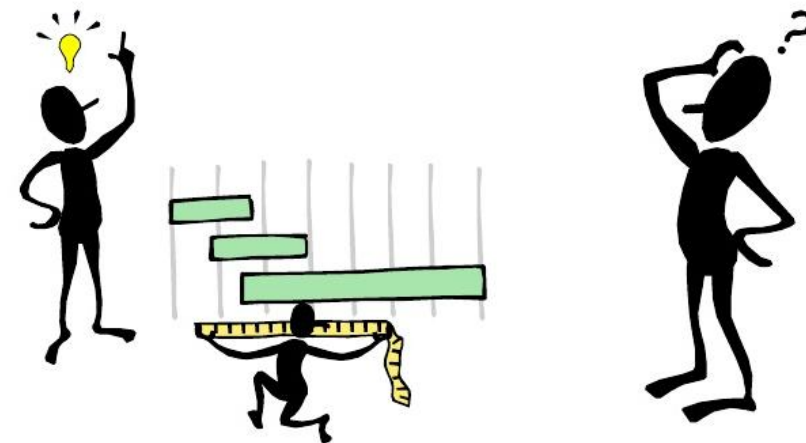




Simple Variables

- **Where** the information is stored?
- **What kind** of information is stored?
- **What value** is kept there?
- The strategy is to declare a **variable**
 - The **type** describes the information
 - The variable **name** represents the value
 - The program **locates** a chunk of memory large enough to hold an integer, **notes** the location, and **copies** the value into the location

What is a Variable?





Names for Variables

- Can't use a C++ **keyword** for a name
- **No limits** on the length of a name
- It is **case** sensitive
- Use **meaningful** names for variables
 - Alphabetic character
 - Underscore (`_`) character
 - Numeric digits
- The beginning of a name
 - **Cannot** be a numeric **digit**
 - **Two underscore** characters are **reserved**
 - An **underscore** character followed by an **uppercase** letter are **reserved**



Integers



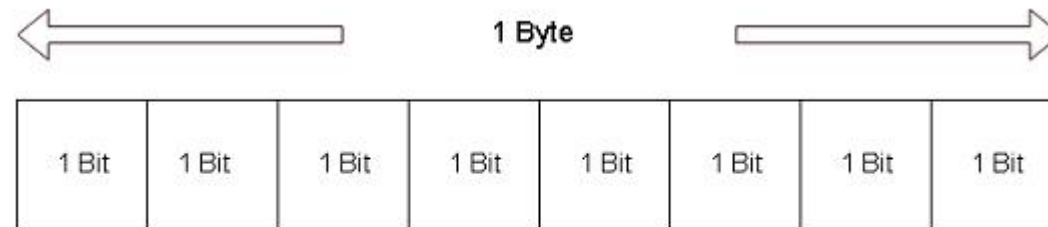
Integer Types

- What is the integer?
 - Integers are numbers with **no fractional** part
- C++ provides several choices
 - char
 - short, int, long, long long
 - C++ integer types differ in the **amount** of memory
 - **Width** is used to describe the differences
 - Both **signed** and **unsigned** versions

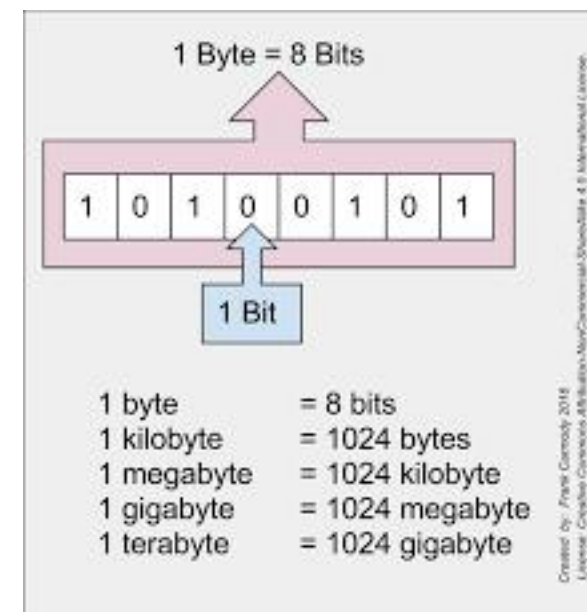


Bits and Bytes

- How to describe the width?
- Bits
 - Unit of computer memory is the bit
 - A bit is an **electronic switch**
 - Off means the value **0**, and on mean the value **1**
 - An **8-bit** can be set to **256** different values
- Bytes
 - A byte **usually** means an 8-bit unit of memory
 - The basic **character set**
 - A **kilobyte** equals to 1,024 bytes
 - A **megabyte** equals to 1,024 kilobytes



8-Bits





Program Example

- Integer Types: width
 - int is 16 bits (the same as short) for older **IBM** PC implementations
 - int is **32 bits** (the same as long) for Windows XP, Windows Vista, Windows 7, Macintosh OS X, VAX, and many other minicomputer implementations
 - The width depends on the **platforms (CPU+OS+Compiler)**
- Run program example 1
 - www.cpp.sh
 - <https://www.onlinegdb.com/>
 - // limits.cpp -- some integer limits



In Example: sizeof Operator

- A important operator: how to use it?

- A **type** name
- A **variable** name

```
cout << "int is " << sizeof (int) << " bytes.\n";  
cout << "short is " << sizeof n_short << " bytes.\n";
```

- What is it used for?

- To **allocate** block of memory dynamically
- To find out **number** of elements in a array

- May give **different** output according to machine
- It is a **keyword** in C Programming



In Example: Header File-climits

- The climits header file defines symbolic constants
- The **compiler** manufacturer provides a climits file
- Could you please remember how the preprocessor directives **#include** and **#define** work?

```
#define INT_MAX 32767
```

Symbolic Constant	Represents
CHAR_BIT	Number of bits in a char
CHAR_MAX	Maximum char value
CHAR_MIN	Minimum char value
SCHAR_MAX	Maximum signed char value
SCHAR_MIN	Minimum signed char value
UCHAR_MAX	Maximum unsigned char value
SHRT_MAX	Maximum short value
SHRT_MIN	Minimum short value
USHRT_MAX	Maximum unsigned short value
INT_MAX	Maximum int value
INT_MIN	Minimum int value
UINT_MAX	Maximum unsigned int value
LONG_MAX	Maximum long value
LONG_MIN	Minimum long value
ULONG_MAX	Maximum unsigned long value
LLONG_MAX	Maximum long long value
LLONG_MIN	Minimum long long value
ULLONG_MAX	Maximum unsigned long long value



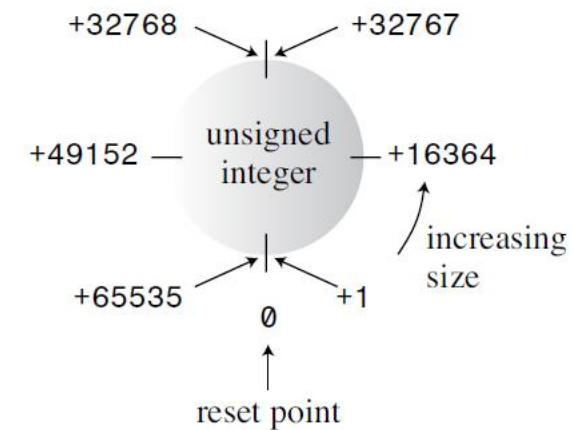
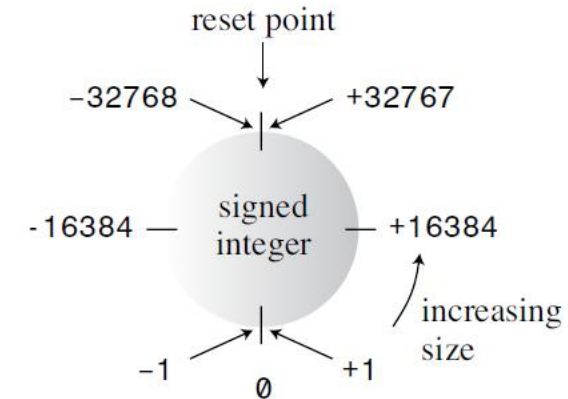
In Example: Initialization

- **Initialization** combines assignment with declaration
 - Use **literal** constants: 11101
 - Use **macros**: INT_MAX
 - Use another **variable**
- Could you please remember how and why **declaration** works?
- Initializing the variable when you declare
- Initialization with C++11
 - Using a **braced** initializer: `int a{1}; int a = {1};`
 - The braces can be left **empty**
- **Run program example 2**
 - `// Initialization.cpp-- with C++11`



Unsigned Types

- Use unsigned types only for quantities that are **never negative**
- Increasing the **largest** value
- **Run program example 3**
 - Go beyond the limits for integer types
 - `//exceed.cpp -- exceeding some integer limits`





Choosing an Integer Type

- The most "natural" integer size: **int**
- Unsigned int
 - Something that is never **negative**
 - Integer values need to be **too great**
- Using **short** can **conserve** memory
- If you need only a **single** byte, you can use **char**

```
// myprofit.cpp
...
int receipts = 560334;
long also = 560334;
cout << receipts << "\n";
cout << also << "\n";
...
```



560334
560334

Type int worked on this computer.

```
// myprofit.cpp
...
int receipts = 560334;
long also = 560334;
cout << receipts << "\n";
cout << also << "\n";
...
```



-29490
560334

Type int failed on this computer.



Integer **Literals**

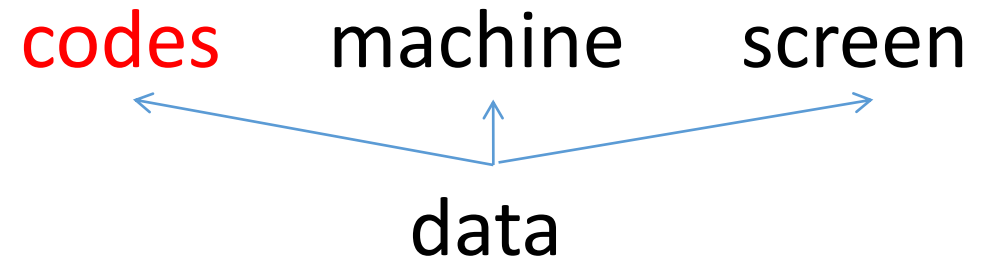
- Number Bases

- **Base 10** (Decimal: the public favorite)
- **Base 8** (Octal: the old Unix favorite)
- **Base 16** (Hexadecimal: the hardware hacker's favorite)

- Uses the **first** digit or **two** (Prefix)

- The first digit is in the range **1–9**, the number is **base 10**
- The first digit is **0** and the second digit is in the range **1–7**, the number is **base 8** (octal)
- The first two characters are **0x** or **0X**, the number is **base 16** (hexadecimal)

- These notations are merely notational **conveniences**





Examples of Integer Literals

- Run program example 4
 - //hexoct1.cpp -- shows hex and octal literals
 - cout displays integers in **decimal** form
- Run program example 5
 - It provides the dec, hex, and oct **manipulators** to give cout the messages
 - //hexoct2.cpp -- display values in hex and octal
- For different integral types
 - **Suffixes** of integer constant

Types allowed for integer literals		
suffix	decimal bases	hexadecimal or octal bases
no suffix	int long int long long int (since C++ 11)	int unsigned int long int unsigned long int long long int (since C++ 11) unsigned long long int (since C++ 11)
u or U	unsigned int unsigned long int unsigned long long int (since C++ 11)	unsigned int unsigned long int unsigned long long int (since C++ 11)
l or L	long int unsigned long int (until C++ 11) long long int (since C++ 11)	long int unsigned long int long long int (since C++ 11) unsigned long long int (since C++ 11)
both l/L and u/U	unsigned long int unsigned long long int (since C++ 11)	unsigned long int (since C++ 11) unsigned long long int (since C++ 11)
ll or LL	long long int (since C++ 11)	long long int (since C++ 11) unsigned long long int (since C++ 11)
both ll/LL and u/U	unsigned long long int (since C++ 11)	unsigned long long int (since C++ 11)



The char Type: Characters and Small Integers

- **char** type is designed to store **characters**, such as letters, punctuation and numeric digits
 - Using **number** codes for letters
 - The char type is another **integer** type
 - **ASCII** character set
 - International **Unicode** character set
- Run program example 6
 - // chartype.cpp -- the char type
- Run program example 7
 - // morechar.cpp -- the char type and int type contrasted

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1_16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2_32	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
3_48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
4_64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
5_80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
6_96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
7_112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F

Legend: Letter (light blue), Number (light purple), Punctuation (light green), Symbol (light orange), Other (light yellow), undefined (grey), Changed from 1963 version (light pink)



char Literals

- Enclose the character in **single quotation marks**
- There are some characters that you **can't enter** into a program directly
- Run program example 8
 - `// bondini.cpp -- using escape sequences`

Character Name	ASCII Symbol	C++ Code	ASCII Decimal Code	ASCII Hex Code
Newline	NL (LF)	<code>\n</code>	10	0xA
Horizontal tab	HT	<code>\t</code>	9	0x9
Vertical tab	VT	<code>\v</code>	11	0xB
Backspace	BS	<code>\b</code>	8	0x8
Carriage return	CR	<code>\r</code>	13	0xD
Alert	BEL	<code>\a</code>	7	0x7
Backslash	<code>\</code>	<code>\\</code>	92	0x5C
Question mark	<code>?</code>	<code>\?</code>	63	0x3F
Single quote	<code>'</code>	<code>\'</code>	39	0x27
Double quote	<code>"</code>	<code>\"</code>	34	0x22



More About char

- signed char $[-128, 127]$ and unsigned char $[0, 255]$
 - Allow the compiler developer to **best fit** the type to the hardware properties
 - Can use signed char or unsigned char **explicitly**
- wchar_t, char16_t and char32_t

- wchar_t for **wide character** type is an integer type

```
wchar_t bob = L'P';           // a wide-character constant
wcout << L"tall" << endl;    // outputting a wide-character string
```

- **New C++11** Types: char16_t and char32_t

```
char16_t ch1 = u'q';           // basic character in 16-bit form
char32_t ch2 = U'\U0000222B';  // universal character name in 32-bit form
```



The bool Type

- The predefined literals **true** and **false**
- The literals **true** and **false** can be converted to type **int** by **promotion**

```
int ans = true;           // ans assigned 1
int promise = false;      // promise assigned 0
```

- Any nonzero value converts to **true**, whereas a zero value converts to **false**

```
bool start = -100;        // start assigned true
bool stop = 0;            // stop assigned false
```





The const Qualifier

- Could you please remember **#define** directives?
- Use the **const** keyword to modify a variable declaration and initialization

```
const int Months = 12;  // Months is symbolic constant for 12
```

- The keyword **const** is termed a qualifier
- Note that you **initialize** a const in the declaration
- Allow you to specify the **type** explicitly

```
const int toes;    // value of toes undefined at this point  
toes = 10;         // too late!
```

Floating-Point Numbers



Floating-Point Numbers

- How to represent fractional numbers?
- Computer stores numbers with fractional parts in **two parts**
 - One part represents a **value**
 - The other part **scales** that value up or down
 - ✓ The scaling factor serves to **move** the **decimal point**
- **Benefits**
 - Floating-point numbers enable to represent **fractional**, very **large**, and very **small** values
 - C++ is based on binary numbers, so the **scaling** is by factors of **2**



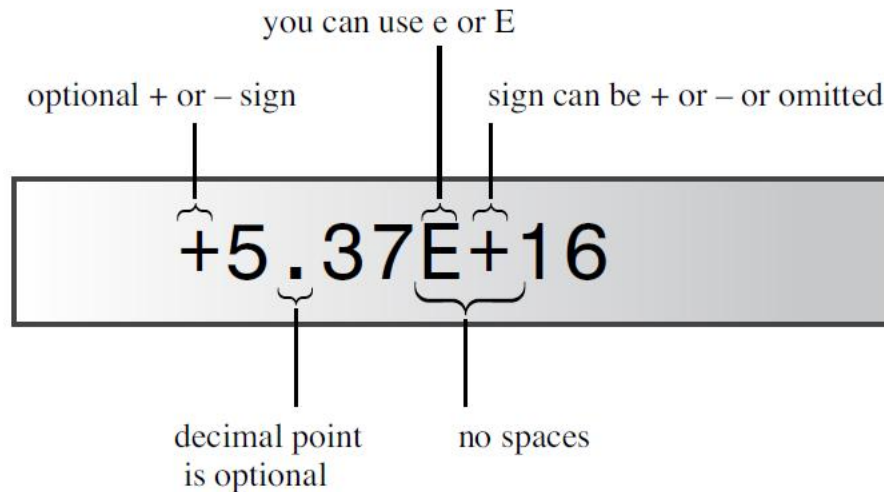
Writing Floating-Point Numbers

- C++ has **two** ways of **writing** floating-point numbers

- Standard decimal-point notation

```
12.34           // floating-point
939001.32       // floating-point
0.00023         // floating-point
8.0             // still floating-point
```

- E notation (*mantissa and exponent*)



```
2.52e+8         // can use E or e, + is optional
8.33E-4         // exponent can be negative
7E5             // same as 7.0E+05
-18.32e13       // can have + or - sign in front
1.69e12         // 2010 Brazilian public debt in reais
5.98E24         // mass of earth in kilograms
9.11e-31        // mass of an electron in kilograms
```




Floating-Point Types

- Three floating-point types: **float**, **double**, and **long double**
- Look in the **cfloat** or **float.h** header files to find the **limits** for your system

```
// the following are the minimum number of significant digits
#define DBL_DIG 15          // double
#define FLT_DIG 6           // float
#define LDBL_DIG 18        // long double
```

Why?

```
// the following are the number of bits used to represent the mantissa
#define DBL_MANT_DIG 53
#define FLT_MANT_DIG 24
#define LDBL_MANT_DIG 64
```

```
// the following are the maximum and minimum exponent values
#define DBL_MAX_10_EXP +308
#define FLT_MAX_10_EXP +38
#define LDBL_MAX_10_EXP +4932
```

```
#define DBL_MIN_10_EXP -307
#define FLT_MIN_10_EXP -37
#define LDBL_MIN_10_EXP -4931
```




Precision of Floating-Point Types

- Run program example 9

- `// floatnum.cpp -- floating-point types`
- `setf()` forces output to stay in **fixed-point** notation
- `ios_base::fixed` and `ios_base::floatfield` are **constants**
- `cout` print **six figures** of digits to the right of decimal point
- `float (pow(2,23) : 7 figures)` and `double (pow(2,52) : 15 figures)`

- Floating-Point Constants

- By default, floating-point constants is **double** type

```
1.234f           // a float constant
2.45E20F         // a float constant
2.345324E28      // a double constant
2.2L             // a long double constant
```



Advantages and Disadvantages of Floating-Point Types

- Advantages

- Represent **values between** integers
- Represent a **much greater range** of values

- Disadvantages

- Slightly **slower** than integer operations
- **Lose** precision

- Run program example 10

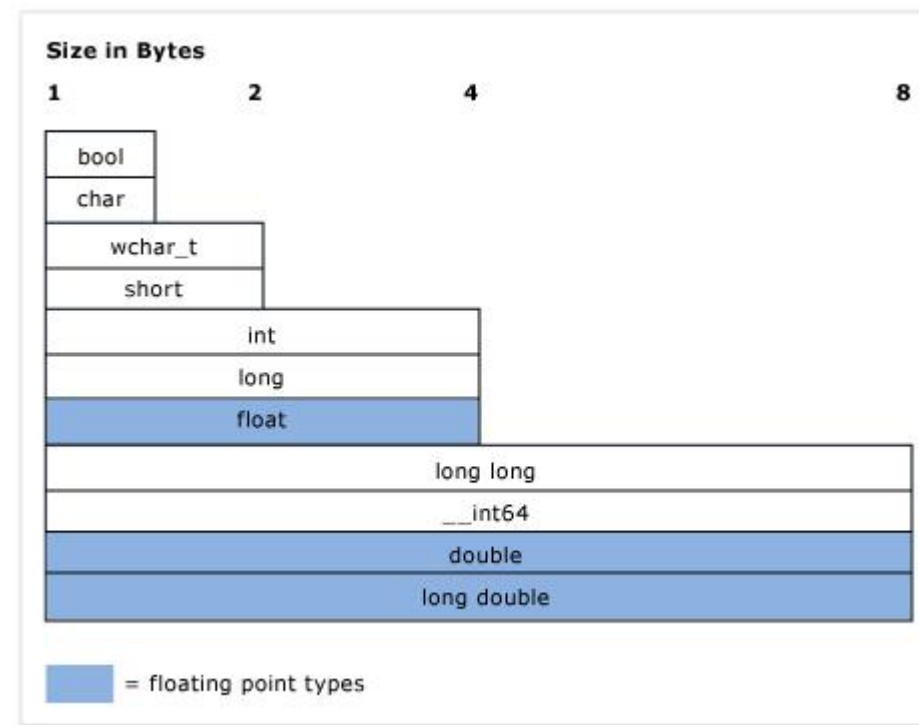
- `// fltadd.cpp -- precision problems with float`



Summary of Float and Integer Types

- Arithmetic types
 - Integer types: signed and unsigned
 - Floating-point types

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	Double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	Long double precision floating point number.	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character



C++ Arithmetic Operators



C++ Arithmetic Operators

- C++ uses **operators** to do arithmetic
- Operators include **five basic** arithmetic calculations: addition, subtraction, multiplication, division, and taking the modulus: **+, -, *, /, %**.
- Operators use **two values** (operands)
- The operator and its operands constitute an **expression**



Order of Operation: Operator Precedence and Associativity

- Use **precedence** rules to decide which operator is used first
 - Usual **algebraic** precedence
 - Use **parentheses** to enforce your own priorities
 - Left-to-right **associativity** or a right-to-left associativity

```
float logs = 120 / 4 * 5;    // 150 or 6?
```



Four Divisions

- Both operands are **integers**
 - Perform integer division.
 - Any fractional part of the answer is **discarded**,
 - Making the result an **integer**
- One or both operands are **floating-point** values
 - The fractional part is **kept**
 - Making the result **floating-point**
- Four distinct operations
 - int, long, float, and double

type int /type int
9 / 5
operator performs
int division

type long /type long
9L / 5L
operator performs
long division

type double /type double
9.0 / 5.0
operator performs
double division

type float /type float
9.0f / 5.0f
operator performs
float division



The Modulus Operator

- Return the **remainder** of an integer **division**
 - Symbol **%** is used
 - **Integer**
- Run program **example 11**
 - `// modulus.cpp` -- uses % operator to convert lbs to stone



Type Conversions

- C++ converts values, in cases:
 - Assign a value of **one arithmetic** type to a variable of **another** arithmetic type
 - Combine **mixed types** in expressions
 - Pass **arguments** to functions

- Result in the **loss** of some precision

- double -> float
- floating-> integer
- long-> short

Conversion Type

Bigger floating-point type to smaller floating-point type, such as `double` to `float`

Floating-point type to integer type

Bigger integer type to smaller integer type, such as `long` to `short`

Potential Problems

Loss of precision (significant figures); value might be out of range for target type, in which case result is undefined.

Loss of fractional part; original value might be out of range for target type, in which case result is undefined.

Original value might be out of range for target type; typically just the low-order bytes are copied.



Type Conversions in Initialization and Assignment

- C++ uses **truncation** (discarding the fractional part) and **not rounding** (finding the closest integer value) when converting **floating-point** types to **integer** types
- Run program example 12
 - // init.cpp -- type changes on initialization
- Initialization conversions when **{}** are used (C++11)

```
const int code = 66;
int x = 66;
char c1 {31325}; // narrowing, not allowed
char c2 = {66};  // allowed because char can hold 66
char c3 {code};  // ditto
char c4 = {x};   // not allowed, x is not constant
x = 31325;
char c5 = x;     // allowed by this form of initialization
```



Automatic Conversions in Expressions

- 1. If either operand is type **long double**, the other operand is converted to **long double**.
- 2. Otherwise, if either operand is **double**, the other operand is converted to **double**.
- 3. Otherwise, if either operand is **float**, the other operand is converted to **float**.
- 4. Otherwise, the operands are **integer** types and the integral promotions are made.
- 5. In that case, if both operands are **signed** or if both are **unsigned**, and one is of lower rank than the other, it is converted to the higher rank.
- 6. Otherwise, one operand is **signed** and one is **unsigned**. If the **unsigned** operand is of higher rank than the **signed** operand, the latter is converted to the type of the **unsigned** operand.
- 7. Otherwise, if the **signed** type can represent all values of the **unsigned** type, the **unsigned** operand is converted to the type of the **signed** type.
- 8. Otherwise, both operands are converted to the **unsigned** version of the **signed** type.



Other Conversions

- Conversions in **passing** arguments for functions

- C++ **promotes** float arguments to double

- Type **Casts**

- **Force** type conversions explicitly via the type cast mechanism

```
(long) thorn    // returns a type long conversion of thorn  
long (thorn)    // returns a type long conversion of thorn
```

- Run program example 13

- // typecast.cpp -- forcing type changes



auto Declarations in C++11

- Allow the compiler to **deduce** a type from the type of an **initialization** value
 - Compiler assigns the variable the **same type** as that of the initializer

```
auto n = 100;      // n is int
auto x = 1.5;      // x is double
auto y = 1.3e12L; // y is long double
```

- STL (**Standard Template Library**)

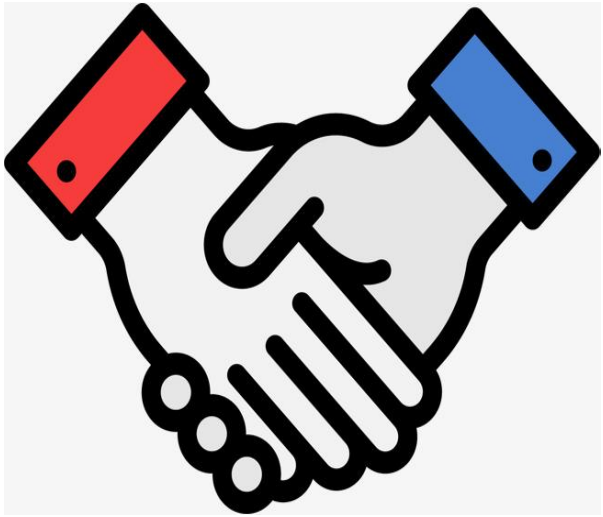
```
std::vector<double> scores;
std::vector<double>::iterator pv = scores.begin();

std::vector<double> scores;
auto pv = scores.begin();
```



Summary

- Integral types
 - Char is integral type
- Floating-point types
 - Loss information
- Arithmetic operators
 - Precedence
 - Conversions



Thanks



zhengf@sustech.edu.cn