[C++学习网 – 世界上最好的中文C++学习网站](https://www.studycpp.cn/)

## 程序调试

std::cerr << "getUserInput() called\n"; //用于调试代码

## 基本数据类型

无符号整数：当不需要负数（如物品的数量，数组索引）时，有双倍的表示范围

**unsigned** **short** us;

**unsigned** **int** ui;

**unsigned** **long** ul;

**unsigned** **long** **long** ull;

如果数学运算（例如算术或比较）有一个有符号整数和一个无符号整数，则有符号整数通常会转换为无符号整数。因此，结果将是无符号的。->应避免使用无符号数

固定宽度的整数：

***#include*** ***<cstdint>***

std::int#\_t

在存储整数值时，通常最好避免使用std::int8\_t和std::uint8\_t（以及相关的快速和最少类型），并改用std::int16\_t或std::uint16\_t。（因为8bit整形数通常被视作字符）

当输出浮点数时，std::cout的默认精度为6，使用std::setprecision() 重新设置精度。

ASCII字符：

‘A’=65,‘Z’=90,’a’=97,’z’=122

char ch{ 97};//使用 数字97，即字符 ‘a’ 初始化字符

char ch{‘5’};//使用字符5初始化

使用std::cin允许输入多个字符，但是ch只能容纳一个字符，多余的被暂存在缓冲区中

列表初始化可以在发生隐式类型转换时报错

显式类型转换：static\_cast<int>(5.5);

## 常变量，字符串

定义常变量时必须对其进行初始化，之后不能通过赋值更改该值

constexpr（“常量表达式”的缩写）变量只能是编译时常量。std::string、std::vector和其他使用动态内存分配的类型与constexpr不兼容

在编程时应拒绝使用意义不明的魔数，相反将其定义为一个常变量可维护性更强。

浮点字面值的默认类型为double而非float，若是想强制令其为float在末尾加上字段f，如**float** f { 4.1f };

**double** avogadro { 6.02e23 }; *// 6.02 x 10^23 是科学计数法写法*

所有的C字符串，都有一个隐式的null结尾符。例如"hello"，看起来只有五个字符，但实际是6个 ‘h’, ’e’, ’l‘, ’l’, ‘o’, and ‘\0’ (ASCII 码 0)。结尾的’\0’是一个特殊的字符，即是所说的null结尾符，用来标记字符串结束位置。

**int** x{ 012 }; *// 0 开头的数据在C++中代表八进制，应避免使用八进制数*

**int** x{ 0xF }; *// 0x 开头的数据在C++中代表十六进制*

可以使用一对（即两个）十六进制数字来精确表示整个字节。

**int** x{ 0b1 }; *// 0b 开头的数据在C++中代表二进制（c++14以后）*

**long** value { 2'132'673'462 }; *// 阅读起来更加容易，数字分隔符’只影响阅读体验，不会以任何方式影响对应的值。*

条件运算符（? :）是c++中唯一的三元运算符，可以帮助压缩代码，而不会丢失可读性。条件运算符是表达式而不是语句，因此可用于判断：

**constexpr** **int** classSize { inBigClassroom **?** 30 **:** 20 };

通常条件运算符用于二选一

inline关键字：内联函数（慎用）

C++20引入了关键字consteval，用于指示函数必须在编译时求值，否则将导致编译错误。这种函数称为即时函数（immediate functions）。

对string进行输入时，std::cin读取到 空白字符，就会停止，并将剩余内容存在缓冲区，规避使用getline结合ws使用。

std::string的返回长度不包括隐式null终止字符。

std::string::length() 返回无符号整数值（很可能是size\_t类型）。如果要将长度分配给int变量，则应将其static\_cast<int>

注意：初始化std::string的开销很大， 应尽量避免字符串的复制

using namespace std::string\_literals//方便使用s后缀

std**::**cout **<<** "goo\n"s; *//在双引号字符串后面使用s后缀来创建类型为std::string的字符串字面值。 s 后缀意味着 std::string 字面值*

std::string\_view（C++ 17）提供对现有字符串（C样式字符串、std::string或另一个std::string\_view）的只读访问，而不制作副本。只读意味着我们可以访问和使用正在查看的值，但不能修改它，大大减少了额外开销，像是看string的一个窗户

注：std::string\_view不会隐式转换为std::string，实在想用强制类型转换或用string\_view初始化一个新的string

将新字符串分配给std::string\_view会导致std::string\_view替换成新的字符串。但不会以任何方式修改先前的字符串。

与std::string不同，std::string\_view完全支持constexpr

视图查看的结果取决于正在查看的对象。如果正在查看的对象在视图在使用时被修改或销毁，将导致意外或未定义的行为。

不要使用std::string字面值来初始化std::string\_view。（字面值会在语句结尾自动销毁从而产生错误）

使用c样式字符串变量或字面值，std::string变量，std::string\_view变量或字面值均可。

修改std::string会使该std::string中的所有视图无效。

因为C样式的字符串在整个程序执行期间均有效（所以即使在某函数中，函数调用结尾依然不会销毁该串），因此可以从返回类型为std::string\_view的函数中返回C样式的串文本。

查看子字符串的能力带来了一个值得注意的后果，std::string\_view可以以null结尾，也可以不以null结束。注意不要编写任何假设std::string\_view以null结尾的代码。

## 运算符

C++不包含进行求幂的运算符（运算符^在C++代表按位XOR）。

余数运算符（%）也可以处理负操作数。 x % y总是返回带x符号的结果。

因此，如果要比较余数运算的结果，最好与0进行比较，因为0没有正数/负数问题。

后缀自增比前缀自增的性能更差，因为它多了个拷贝的操作

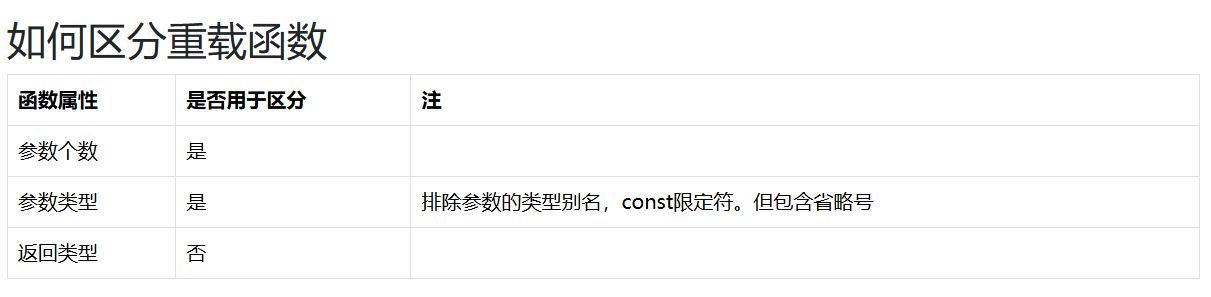
避免使用运算符==和运算符!=比较浮点值，因为浮点可能发生舍入

注意逻辑运算符的短路求值

## 函数重载与函数模板

函数重载允许我们创建具有相同名称的多个函数，只要每个同名函数具有不同的参数类型

当对已重载的函数进行函数调用时，编译器将基于函数调用中使用的参数，尝试将函数调用与适当的重载函数相匹配。这称为重载决议。



在区分重载函数时，不考虑函数的返回类型，因为返回类型不在函数的类型签名中。

在我们有一个显式不希望可调用的函数的情况下，可以使用=delete说明符将该函数定义为删除。如果编译器将函数调用与已删除函数相匹配，则将因编译错误而停止编译。

默认参数：进行函数调用时。如果调用方提供了对应值，则以传递的值为准。如果调用方不提供对应的值，则使用默认参数的值。请注意，必须使用等号来指定默认参数。使用括号或大括号不符合语法

多个默认参数：

如果为参数给定默认值，则所有后续参数（右侧）也必须给定默认值。即默认参数只能放在参数的最右端，无论是构造还是使用

如果函数具有前向声明（特别是头文件中的声明），请在那里放置默认参数。否则，再将默认参数放在函数定义中。

多个文件中需要的模板应该在头文件中定义，然后在需要的地方使用include引用。

## 复合类型：引用与指针

左值：可识别对象或函数（或位字段）的表达式，可以通过标识符、引用或指针访问。如果 “&表达式;“编译通过，表达式一定是左值

右值：包括字面值以及按值返回的函数和运算符的返回值，即表达式的计算结果，并且在表达式结束后丢弃。

引用绑定：左值引用必须在初始化时绑定到可修改的左值

int& invalidRef; // error: 引用必须被初始化

int& ref { x }; // okay: 有初始化值

引用不是C++中的对象。引用不需要存在或占用存储。

常量引用：通常情况下，引用不可作用于常量，但我们加入const 字段后则可以，但需注意到我们仍然不能通过常量引用修改对象

const int x { 5 }; // x 是不可修改的左值

const int& ref { x }; // okay: ref 可以引用到x

ref = 6; // error: 不能通过常量引用修改对象

对常量的左值引用也可以绑定到右值：

const int& ref { 5 }; // okay: 5 是一个 右值

用法：  
在调用函数时，避免生成参数的昂贵副本的一种方法是按引用传递参数，而不是按值传递。绑定引用总是代价很小，不需要复制x。因为引用充当被引用对象的别名，所以当printValue()使用引用y时，它访问的是实际的x（而不是x的副本），因此对引用参数所做的任何更改都将影响原始对象。

然而这样的问题是只能用于接受可修改左值，如果我们使引用参数为const，则它将能够绑定到任何类型的参数。通过常量引用传递与通过引用传递参数提供相同的好处（避免复制参数），同时还保证函数不能更改被引用的值，即不允许函数修改参数的值。

void printValue(const int& y) // y 是常量引用

最佳实践：

通过值传递的基本类型与枚举类型和std::string\_view。

通过（const）引用传递的其他常见类型:std::string、std::array和std::vector。

在大多数情况下，我们更推荐使用string\_view 而不是不 const string&

**指针(pointer):**

指针类型使用星号（\*）声明:

int\*; // 指向int的指针 (保存对应的内存地址)

int\* ptr; // 未初始化的指针 (包含一个垃圾地址)

int\* ptr2{}; // 空指针

int\* ptr3{ &x }; // 指向变量 x 地址的指针

std::cout << \*ptr << '\n'; // 使用解引用操作获取ptr保存的地址上的值 (ptr保存的是x的地址)

note:不允许使用字面值初始化指针

int\* ptr{ 5 }; // not okay

int\* ptr{ 0x0012FF7C }; // not okay, 0x0012FF7C 被认为是int字面值

指针和左值引用的行为类似。指针和引用都提供了间接访问另一个对象的方法。主要的区别是，对于指针，需要显式地获取指向的地址，并且必须显式地解引用指针才能获得值。对于引用，获取地址和解引用隐式发生。引用更安全！可以时通过引用传递参数，必须时才通过地址传递参数

指针的大小始终相同。这是因为指针只是内存地址，并且访问内存地址所需的位数是恒定的。

常量指针是指向常量值的指针。常量指针实际可以指向非const变量。但常量指针将所指向的值视为常量，因此不可修改常量指针指向的内容。

const int x{ 5 };

const int\* ptr { &x }; // okay: ptr 现在指向 "const int"

\*ptr = 6; // 不被允许: 不能修改常量

指针常量：指针常量是其指向地址在初始化后不能更改的指针。

指针常量必须在定义时初始化，并且不能通过赋值更改该值。然而，由于所指向的值是非常量的，因此可以通过解引用const指针来更改所指向的数值。

int x{ 5 };

int\* const ptr { &x }; // 星号后面的 const 代表这是一个 const pointer

通过引用来传递指针，则可以修改指针，而不是对它的副本做修改

void nullify(int\*& refptr) // refptr 现在是指针的引用

类型std::nullptr\_t只能保存一个值：nullptr，如果希望编写只接受nullptr参数的函数，可以将该参数类型设置为std::nullptr\_t

类型自动推导auto:

默认情况下，类型推导将删除const（和constexpr）限定符以及引用属性, 通过在定义中添加const（或constexpr）限定符，可以（重新）应用const（或contexpr）

const double cd{ 7.8 };

auto x{ cd }; // double (const 被丢弃)

constexpr auto y{ cd }; // constexpr double

## 复合类型：枚举与结构体

按照惯例，程序定义的类型以大写字母开头命名，并且不使用后缀.

声明程序定义的类型，定义语句必须以分号结尾。

struct Fraction

{

int numerator {};

int denominator {};

};

在代码中使用魔数是有害的，可以通过使用constexpr变量来去除魔数，并且使用类型别名使程序更加清晰：

using Color = int; // 定义一个类型别名 Color

// 下面是Color可能的值

constexpr Color red{ 0 };

constexpr Color green{ 1 };

constexpr Color blue{ 2 };

int main()

{

Color appleColor{ red };

Color shirtColor{ green };

return 0;

}

但这仍可能产生语义外的问题，如我们定义Color appleColor{ 7},但是7并没有对应的int含义——> 解决方法：枚举，枚举元素隐式为constexpr。枚举类型的名称以大写字母开头，以小写字母开头命名枚举元素。

非限定作用域枚举：

enum Color// 定义名为 Color 的非限定作用域枚举

{

// 下面是枚举的元素

// 这些符号常量，标识了所有可能出现值的集合

// 这些元素以逗号分隔

red,

green,

blue, // 最后一个元素的结尾逗号是可选的，推荐带上

}; // 枚举定义必须以分号结尾

int main()

{

// 定义Color类型的一些变量

Color apple { red }; // 苹果是红色

Color shirt { green }; // 衬衫是绿色

Color cup { blue }; // 杯子是蓝色

Color socks { white }; // 错误: white 不是 Color 定义的枚举元素

Color hat { 2 }; // 错误: 2 不是 Color 定义的枚举元素

return 0;

}

当用户需要在两个或多个选项之间进行选择时，枚举类型也可以成为有用的函数参数。

enum SortOrder

{

alphabetical,

alphabeticalReverse,

numerical,

};

void sortData(SortOrder order)

{

switch (order)

{

case **alphabetical**:

// 使用正向的字母表排序

break;

case **alphabeticalReverse**:

// 使用反向的字母表排序

break;

case **numerical**:

// 使用数字顺序排序

break;

}

}

然而，我们需要避免非限定作用域枚举元素命名冲突，一个选项是用枚举本身的名称作为每个枚举元素的前缀：

enum Color

{

color\_red,

color\_blue,

color\_green,

};

enum Feeling

{

feeling\_happy,

feeling\_tired,

feeling\_blue, // 不在与 color\_blue 命名冲突

};

更好的选择是将枚举类型放在提供单独作用域的其它东西里面，例如命名空间：

namespace Color

{

// Color, red, blue, 和 green 在命名空间 Color 中

enum Color

{

red,

green,

blue,

};

}

namespace Feeling

{

enum Feeling

{

happy,

tired,

blue, // Feeling::blue 不与 Color::blue 冲突

};

}

int main()

{

Color::Color paint{ Color::blue };

Feeling::Feeling me{ Feeling::blue };

return 0;

}

这意味着现在必须用命名空间作为枚举和枚举元素名称的前缀。

可以使用相等运算符（operator==和operator!=）来测试对象是否具有特定枚举元素的值。

枚举类型实际上持有整数值。重载 << 以打印枚举数（打印名称而不是int）

#include <iostream>

enum Color

{

black,

red,

blue,

};

std::ostream& operator<<(std::ostream& out, Color color)

{

switch (color)

{

case **black**: return out << "black";

case **red**: return out << "red";

case **blue**: return out << "blue";

default: return out << "???";

}

}

int main()

{

Color shirt{ blue };

std::cout << "Your shirt is " << shirt << '\n'; // it works!

return 0;

}

在必要时可以指定枚举的底层类型。

enum Color : std::int8\_t

{

black,

red,

blue,

};

非限定作用域枚举的输入：

std::istream& operator>> (std::istream& in, Pet& pet)

{

int input{};

in >> input; // 读取一个整数

pet = static\_cast<Pet>(input);//强制类型转换

return in;

}

限定作用域枚举（枚举类），用关键字 enum class，限定作用域枚举充当其枚举元素的命名空间。限定作用域枚举不会隐式转换为整数，如果转换有用使用static\_cast将它显式转换为整数。

#include <iostream>

int main()

{

enum class Color // "enum class" 定义了 限定作用域枚举

{

red, // red 在 Color 的作用域中

blue,

};

enum class Fruit

{

banana, // banana 在 Fruit 的作用域中

apple,

};

Color color { Color::red }; // 注: red 不能直接使用, 需要使用 Color::red

Fruit fruit { Fruit::banana }; // 注: banana 不能直接使用, 需要使用 Fruit::banana

if (color == fruit) // 编译失败: 编译器不知道如何比较 Color 与 Fruit 类型

std::cout << "color and fruit are equal\n";

else

std::cout << "color and fruit are not equal\n";

return 0;

}

在C++20中引入了using enum语句，该语句将枚举中的所有枚举元素导入到当前作用域中。这允许我们访问枚举元素，而不必添加前缀。

初始化聚合类型时，首选列表初始化。

struct Employee

{

int id {};

int age {};

double wage {};

};

Employee joe { 2, 28, 45000.0 }; // 列表初始化 (推荐)

joe = { joe.id, 33, 66000.0 }; //更新， Joe 大了一岁 也加薪了，不想更改joe.id，所以需要在列表中提供joe.id的当前值作为占位符

joe = { .id = joe.id, .age = 33, .wage = 66000.0 };

如果初始化列表中，初始化值的数量小于成员变量的数量，则所有剩余的成员都将被值初始化。将新成员添加到聚合时，最安全的做法是将其添加到定义列表的底部，以便其他成员的初始值设定项不会移动。

结构体类型的变量可以是const（或constexpr），就像所有const变量一样，它们必须被初始化。

非静态成员初始化，列表初始化中的显式值始终优先于成员变量默认的初始化值。

struct Something

{

int x; // 无初始化值 (bad)

int y {}; // 值初始化

int z { 2 }; // 显示的默认值

};

为了避免未初始化成员的可能性，只需确保每个成员都有一个默认值（显式默认值或空的大括号对）。这样，无论是否提供初始值设定项列表，成员都将使用某些值进行初始化。

对于聚合类型，首选值初始化（有空大括号），而不是默认初始化（没有大括号）。

大多数情况下，结构体应该作为数据的所有者（不是查看者，指针或引用）

通过按大小的递减顺序定义成员，可以最小化填充。

C++编译器不允许对成员重新排序，因此必须手动完成。

指针运算符（->）（有时也称为箭头运算符），可以用于从指向对象的指针中选择成员

(\*ptr).id 等价于 ptr->id

类模板

与函数不同，类型定义不能重载。

template <typename T>

struct Pair

{

T first{};

T second{};

};

该定义等价于：

// Pair 类模版定义

// (模版实例化后这个定义不再被需要了)

template <typename T>

struct Pair;

// 显示定义 Pair<int>

template <> // 告诉编译器这是一个没有类型参数的模版

struct Pair<int>

{

int first{};

int second{};

};

// 显示定义 Pair<double>

template <> // 告诉编译器这是一个没有类型参数的模版

struct Pair<double>

{

double first{};

double second{};

};

类模板参数推导（CTAD）：C++ 17

std::pair<int, int> p1{ 1, 2 }; // 显示声明 std::pair<int, int> (C++11)

std::pair p2{ 1, 2 }; // CTAD，从初始值列表推导 std::pair<int, int> (C++17)

仅当不存在模板参数列表时才执行CTAD。因此，以下两项都是错误：

std::pair<> p1 { 1, 2 }; // error: 模版参数太少, 2个模版参数都缺少了

std::pair<int> p2 { 3, 4 }; // error: 模版参数太少, 第二个模版参数缺少

为非静态成员设置默认值时，CTAD在此上下文中不起作用；

CTAD不能用于函数参数

## 函数

程序使用的内存通常分为几个不同的区域，称为段：

1. 代码段（也称为文本段），编译后的程序二进制位于内存中。代码段通常是只读的。
2. bss段（也称为未初始化的数据段），其中存储零初始化的全局变量和静态变量。
3. 数据段（也称为初始化数据段），其中存储初始化的全局变量和静态变量。
4. 堆，存储分配动态分配的变量。
5. 调用栈，其中存储函数参数、局部变量和其他函数相关信息。

在C++中，当使用new操作符分配内存时，该内存分配在应用程序的堆段中。连续内存分配请求可能不会分配连续的内存地址，删除指针不会删除变量，它只是将相关地址的内存返回给操作系统。

栈的大小有限，因此只能保存有限的信息量。在Visual Studio for Windows上，默认栈大小为1MB。如果程序试图在栈上放置太多信息，则会导致栈溢出。当栈中的所有内存都已分配时，就会发生栈溢出——在这种情况下，进一步的分配会导致栈溢出到内存的其他部分。

因为栈相对较小，所以做任何占用大量栈空间的事情通常不是一个好主意。这包括分配或复制大型数组或其他内存密集型结构。