

# C++\_benning\_1

From SHTU LYM

## "Better C"

1. `bool`, `true`, `false` 是内嵌的类型不再是 `int` 类型的了
2. 逻辑判断返回的也是 `bool` 类型而不是 `int` 类型
3. "`hello`" 这个字面量在 C++ 中是 `const char [N+1]` 而不是 `char [N+1]`。
4. 字符 '`a`' 不再是 `int` 类型, 而是 `char` 类型。

## 输入输出

- 特殊输出方式

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    cout << setiosflags(ios::left | ios::showpoint); // 设左对齐, 以一般实数方式显示, 会一直影响后面的内容
    cout.precision(5); // 设置除小数点外有五位有效数字, 没有规定ios::fixed; or; ios::scientific就会保留三位有效数字
    cout << 123.456789 << endl;
    cout.width(10); // 设置显示域宽10
    cout.fill('*'); // 在显示区域空白处用*填充
    cout << resetiosflags(ios::left); // 清除状态左对齐
    cout << setiosflags(ios::right); // 设置右对齐
    cout << 123.456789 << endl;
    cout << setiosflags(ios::left | ios::fixed); // 设左对齐, 以固定小数位显示
    cout.precision(3); // 设置实数显示三位小数
    cout << 999.123456 << endl;
    cout << resetiosflags(ios::left | ios::fixed); // 清除状态左对齐和定点格式
    cout << setiosflags(ios::left | ios::scientific); // 设置左对齐, 以科学技术法显示
    cout.precision(3); // 设置保留三位小数
    cout << 123.45678 << endl;
    return 0;
}
```

- 和进制相关的输入输出的函数
- 输入

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int hexNumber, decNumber, octNumber;
    cin >> std::hex >> hexNumber;//以16进制输入然后存贮为10进制
    cout << hexNumber << endl;
    cin >> std::dec >> decNumber;//记得要转换成为十进制
    cout << decNumber << endl;
    cin >> std::oct >> octNumber;
    cout << octNumber << endl;
    return 0;
}
```

- 输出

```
#include <iostream>
using namespace std;
int main()
{
    //将十进制的数转换成为其他的进制输出
    int hexNumber = 10, decNumber = 10, octNumber = 10;
    cout << "Octal: " << std::oct << octNumber << endl;
    cout << "Decimal: " << std::dec << decNumber << endl;
    cout << "Hexadecimal: " << std::hex << hexNumber << endl;
    return 0;
}
```

```
output:
Octal: 12
Decimal: 10
Hexadecimal: a
```

## getchar()与getline()

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string a;
    getline(cin, a);
    cout << "The string you entered is :" << a;
    return 0;
}
```

在遇到\n停止读取，内部的空格，制表符都能够读取。

- getline的用法

- 读取C++类型的字符串 `istream& getline(istream&is, string& str, char delim):` `is` 表示的是输入流对象，`str` 表示用于存储读取到的行内容的字符串对象，`delim` 表示读取到什么内容就停止读取本字符串
- 读取C风格的字符串 `getline(char *s, streamsize n, char delim):` `s` 表示的是存储的字符串的位置，`n` 表示最多读取字符数，`delim` 表示的是指定结束符

```
getline(std::cin, str, '\n');
```

## 数据类型

### 常见的数据类型的类型字节大小以及范围

```
#include <iostream>
#include <limits>

using namespace std;

int main()
{
    cout << "type: \t\t" << "*****size*****" << endl;
    cout << "bool: \t\t" << "size:" << sizeof(bool);
    cout << "\tmax:" << (numeric_limits<bool>::max)();
    cout << "\tmin:" << (numeric_limits<bool>::min)() << endl;
    cout << "char: \t\t" << "size:" << sizeof(char);
    cout << "\tmax:" << (numeric_limits<char>::max)();
    cout << "\tmin:" << (numeric_limits<char>::min)() << endl;
    cout << "signed char: \t" << "size:" << sizeof(signed char);
    cout << "\tmax:" << (numeric_limits<signed char>::max)();
    cout << "\tmin:" << (numeric_limits<signed char>::min)() << endl;
    cout << "unsigned char: \t" << "size:" << sizeof(unsigned char);
    cout << "\tmax:" << (numeric_limits<unsigned char>::max)();
    cout << "\tmin:" << (numeric_limits<unsigned char>::min)() << endl;
    cout << "wchar_t: \t" << "size:" << sizeof(wchar_t); // 其实是由short类型别名得到的
    cout << "\tmax:" << (numeric_limits<wchar_t>::max)();
    cout << "\tmin:" << (numeric_limits<wchar_t>::min)() << endl;
    cout << "short: \t\t" << "size:" << sizeof(short);
    cout << "\tmax:" << (numeric_limits<short>::max)();
    cout << "\tmin:" << (numeric_limits<short>::min)() << endl;
    cout << "int: \t\t" << "size:" << sizeof(int);
    cout << "\tmax:" << (numeric_limits<int>::max)();
    cout << "\tmin:" << (numeric_limits<int>::min)() << endl;
    cout << "unsigned: \t" << "size:" << sizeof(unsigned);
    cout << "\tmax:" << (numeric_limits<unsigned>::max)();
    cout << "\tmin:" << (numeric_limits<unsigned>::min)() << endl;
    cout << "long: \t\t" << "size:" << sizeof(long);
    cout << "\tmax:" << (numeric_limits<long>::max)();
    cout << "\tmin:" << (numeric_limits<long>::min)() << endl;
    cout << "unsigned long: \t" << "size:" << sizeof(unsigned long);
```

```

cout << "\tmax:" << (numeric_limits<unsigned long>::max)();
cout << "\tmin:" << (numeric_limits<unsigned long>::min)() << endl;
cout << "double: \t" << "size:" << sizeof(double);
cout << "\tmax:" << (numeric_limits<double>::max)();
cout << "\tmin:" << (numeric_limits<double>::min)() << endl;
cout << "long double: \t" << "size:" << sizeof(long double);
cout << "\tmax:" << (numeric_limits<long double>::max)();
cout << "\tmin:" << (numeric_limits<long double>::min)() << endl;
cout << "float: \t\t" << "size:" << sizeof(float);
cout << "\tmax:" << (numeric_limits<float>::max)();
cout << "\tmin:" << (numeric_limits<float>::min)() << endl;
cout << "size_t: \t" << "size:" << sizeof(size_t);
cout << "\tmax:" << (numeric_limits<size_t>::max)();
cout << "\tmin:" << (numeric_limits<size_t>::min)() << endl;
cout << "string: \t" << "size:(SSO)" << sizeof(string) << endl;
// << "\t最大值: " << (numeric_limits<string>::max)() << "\t最小值: " <<
(numeric_limits<string>::min)() << endl;
cout << "type: \t\t" << "*****size*****" << endl;
return 0;
}

```

type:	*****size*****		
bool:	size:1	max:1	min:0
char:	size:1	max:	min:€
signed char:	size:1	max:	min:€
unsigned char:	size:1	max:	min:
wchar_t:	size:2	max:65535	min:0
short:	size:2	max:32767	min:-32768
int:	size:4	max:2147483647	min:-2147483648
unsigned:	size:4	max:4294967295	min:0
long:	size:4	max:2147483647	min:-2147483648
unsigned long:	size:4	max:4294967295	min:0
double:	size:8	max:1.79769e+308	min:2.22507e-308
long double:	size:16	max:1.18973e+4932	min:3.3621e-4932
float:	size:4	max:3.40282e+38	min:1.17549e-38
size_t:	size:8	max:18446744073709551615	min:0
string:	size:(SSO)32		
type:	*****size*****		

## typedef声明类型

`typedef type newname;`例如`typedef int new_int`则会告诉编译器`new_int`是`int`的一个别名则会有`new_int distance;`这样的声明式合法的。

注意`typedef`存在一定的作用域

## 类型转换

类型转换是将一个数据类型的值转换成为另一个数据类型的值，C++中有四种类型转换：静态转换、动态转换、常量转换和重新解释转换。

## Static Cast

静态转换是将一种数据类型的值强制转换成为另一种数据类型的值。静态类型转换同煮成用于比较类型相似的对象之间的转换，例如将int类型转换成为float静态转换的时候不会有任何的类型检查，所以可能会导致运行时候的错误。

```
int i = 10;
float f = static_cast<float>(i);
```

## const\_cast

```
const int a = 10;
int *ptr = &a;//这里会导致错误因为使用ptr可以改变const的值
int *ptr2 = const_cast<int*>(&a);//强制转换成为int*
*ptr2++;//UB本质上不能够改变const的数值
```

```
const int a = 10;
int &b = a;//Error!
int &b = const_cast<int &>(a);
b++;//UB
```

## reinterpret\_cast

```
int ival = 42;
char *pc = reinterpret_cast<char *>(&ival);//Dangerous
```

## 继承自C的强制转换说明

```
int i = 10;
cout << (double)i << endl;
```

## C++ 中也存在隐式转换

- 在表达式中比int类型还小的整数类型（char,short）会被提升为int类型。如果int类型无法表示该值，会被提升为unsigned int类型
- 混合类型运算：在包含不同算术类型的表达式中，编译器会将操作数转换为一种共同的类型，以便进行运算。通常，类型转换的方向是从表示范围小的类型向表示范围大的类型转换，例如从int到double。
- 赋值转换：在将一个值赋给不同类型的变量时，会发生隐式转换。赋值转换会将右侧表达式的值转换为左侧变量的类型。（注意有可能丢失精度）

- 函数中调用的隐式转换：在调用函数时，如果实参的类型与形参的类型不匹配，编译器会尝试进行隐式转换。
- 函数中返回值的类型转换：如果函数的返回值类型与实际返回表达式的类型不匹配，编译器会进行隐式转换。
- 空指针和整数0可以隐式转换为任何指针类型：

```
#include <iostream>
int main() {
    int* ptr = nullptr; // nullptr 被隐式转换为 int* 类型
    return 0;
}
```

- 但是注意在C++中不能够通过void\*将两个不同类别的指针进行转换。同时还有const int\*这样类型的指针不能够转换成为void\*类型的指针，如果要将一个const int\*类型的指针转换成为void\*类型的指针：const\_cast<void\*>(static\_cast<const void\*>(p))要转换两次
- bool类型的转换：在条件判断语句中，非布尔类型的值会被隐式转换为布尔类型。通常，零值会被转换为false，非零值会被转换为true。
- 所有的浮点计算都是双精度进行的，也就是说所有的float都会转换成为double

## C++中的常量

### 整数常量和小数常量

整数常量可以是十进制或者是八进制或者是16禁止的常量。前缀指定基数：0x或者是0X表示十六进制，0表示八进制，不带前缀则默认表示十进制

整数常量也可以带一个后缀，后缀是U和L的组合，U表示无符号的整数L表示长整数。后缀可以大写也可以小写。(U和L的顺序随意)

```
212      //correct
215u     //correct
0xFee    //correct
078      //error:there are no 8 in oct
032      //error:You Can Not repeat the U
3.14159   // 合法的
314159E-5L // 合法的
510E      // 非法的：不完整的指数
210f      // 非法的：没有小数或指数
.e55      // 非法的：缺少整数或分数
```

### 字符常量

在C++中某一些字符\?等具有一些特殊的含义，如果想要直接输出这一些字符，就需要使用转义字符\所以有\?表示的就是?字符。

字符串字面量或者是常量通常是引用在栓引号中但是如果这个字符串太长，我们往往将他使用\来进行分行

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string greeting = "hello, runoob";
    cout << greeting;
    cout << "\n";
    string greeting2 = "hello, \
                        runoob";
    cout << greeting2;
    return 0;
}
```

## #define和const变量

内容和C相同，详情查看[C\\_beinning](#)

## 运算符

基本与C一致，有几个需要注意一下：

- 整数除法或者是取余的过程中的正负号的取法
  - 整除：两者的符号相同则为正，如果两者的符号相反则为负
  - 取余：余数的正负号和被除数相同

## auto与decltype

```
auto str = "hello"//`const char*
const int x = 10;
auto y = x//`int`Not`const int`!!
auto&py = x;//`const int&`!!
auto *ptr = &x;//`const int`
```

- decltype(x) y = 10;如果x是int则会有int y = 10;如果x是double则会有double y = 10;但是如果decltype(foo(x))这里的foo函数并不会被调用，这里完全是计算机基于代码推测的。

## C++'s string vs C's string

1. 内存自动管理，`std::string`会自动处理内存分配和释放，创建`std::string`对象的时候，内部会根据储存需求自动分配内存；对象生命周期结束，会自动释放内存，无需手动处理。
2. 动态调整储存内存，对`std::string`执行操作插入，如`insert`方法或者是删除方法，内部的内存会自动调整。
3. 无需先是终止符`\0`，与C风格的字符串不同，`std::string`内部通过记录字符串长度而不是`\0`来识别字符串结尾。这使得`std::string`可以安全储存`'\0'`的字符

## Length of string

- Member function `s.size()` and `s.empty()`

```
std::string str{"hello world"};
std::cout << str.size() << std::endl;
```

Not `strlen`, not `sizeof`.

- `if(str.empty()){...}`

## Concatenation of strings

Use `+` and `+=` directly!

- Not need to care about the memory alloction
- No awkward functions like `strcat()`

```
std::string s1 = "Hello";
std::string s2 = "world";
std::string s3 = s1 + ' ' + s2; // "Hello world"
s1 += s2; // s1 becomes "Helloworld"
s2 += "C++string"; // s2 becomes "worldC++string"
```

如果是将原来的C风格的string和C++风格的字符串相比较的结果是什么？

```
const char *old_bad_ugly_C_style_string = "hello";
std::string good_beautiful_Cpp_string = "hello";
std::string s1 = good_beautiful_Cpp_string + "aaaaa"; // OK.
std::string s2 = "aaaaa" + good_beautiful_Cpp_string; // OK.
std::string s3 = old_bad_ugly_C_style_string + "aaaaa"; // Error
```

至少需要1个是`std::string`这样才能够运行，但是如果存在C风格的字符串，程序会warnning.

Question: Is this ok?

```
std::string hello = "Hello";
std::string s = hello+"World"+"C++";
```

**Yes! `+` is left-associated**

Use `+=`

- 在C++中`a+=b`是直接在字符串后面append新的字符串
- 但是如果`a = a+b`那么表示的是首先拷贝一遍`a`再将`b` append`到后面最后再拷贝回`a`（速度很慢）

## Deep copy in c++ string

```
std::string s1{"Hello"};
std::string s2{"world"};
s2 = s1; // s2 is a copy of s1
s1 += 'a'; // s2 is still "Hello"
```

这里的copy很明显是深拷贝

## string IO

- `std::cin>>s` 的行为: `std::cin>>s` 会忽略开头的空白字符 (如空格、制表符) , 但是它不会读取整行的内容, 只会读取连续的非空字符串
- `std::getline(std::cin,s)` 从输入流当前位置开始读取, 遇到第一个 `\n` 的时候结束, 将读取内容 (不包含 `\n`) 存入 `s`

Traversing a string :Use range-based `for` loops

```
#include <iostream>
#include <string>
#include <cctype>
int main(void)
{
    std::string s = {"Hello"};
    for (std::size_t i = 0; i != s.size(); ++i)
        if (std::isupper(s[i]))
            std::cout << s[i];
    std::cout << std::endl;
    return 0;
}
```

Equivalent way:

```
#include <iostream>
#include <string>
#include <cctype>
int main(void)
{
    std::string s = {"Hello"};
    for(char c : s)
        if(std::isupper(c))
            std::cout<<c;
    std::cout << std::endl;
    return 0;
}
```

Use range-based `for` loops. They are modern ,clear and hence more recommended.

## Conversion between strings and arithmetic numbers

```
#include <iostream>
#include <string>
int main(void)
{
    int ival = 42;
    double dval = 3.14;
    std::string s = std::to_string(ival) + std::to_string(dval);
    std::cout << s << '\n';
    return 0;
}
```

## Conversion from string to int or long long

```
#include <iostream>
#include <string>

int main() {
    std::string num_str = "12345";

    // 使用 std::stoi 转换
    int int_val = std::stoi(num_str);
    std::cout << "std::stoi result: " << int_val << std::endl;

    // 使用 std::stol 转换
    long long_val = std::stol(num_str);
    std::cout << "std::stol result: " << long_val << std::endl;

    // 带进制和 pos 的示例
    std::string bin_str = "1010";
    std::size_t pos;
    int bin_val = std::stoi(bin_str, &pos, 2);
    std::cout << "Binary to int: " << bin_val
        << ", end position: " << pos << std::endl; // 输出: 10, 4 pos反应的是
    // 二进制数的长度，同时如果不需要这个数可以使用nullptr代替
    return 0;
}
```

## stringstream

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

int main()
{
```

```

    std::string str;
    std::string input;
    std::cin >> input;
    std::stringstream ss(input); // 将字符串视为一个流，常见用于将数值转换为字符串或者是
    将字符串进行处理
    std::vector<std::string> arr_str;
    // 读取以逗号分隔的字符串
    while (getline(ss, str, ',')) {
        std::cout << "Read: " << str << std::endl; // 调试输出
        arr_str.push_back(str);
    }
    // 输出结果
    for (auto i : arr_str)
        std::cout << i << std::endl;

    return 0;
}

```

## Reference

- A reference defines an *alternative name* for an object ("refers to" that object).
  - **ReferredType** is the type of the object that refers to
  - **&** is the symbol indicating that is a reference.

```

int ival = 42;
int &ri = ival;

std::cout << ri << '\n';
++ri; // Same as `++ival`

```

This option is different from `int x = ival`. They are two different variables.

### A Reference must be initialized

Reference must be bound to existing objects(左值)

```

int &r1 = 42;
// Error: binding a reference to a literal
int &r2 = 2 + 3; // Error: binding a reference to a temporary object
int a = 10, b = 15;
int &r3 = a + b; // Error: binding a reference to a temporary object

```

引用没有“引用的引用”也不存在指针指向的引用，但是存在指针的引用

```
int ival = 42;
int &ri = ival;
int & &rr = ri;//Error
int &*pr = &ri;//Error
```

```
void modify(int *&ptrRef){//如果没有传递的是指针的引用，那么这里的修改是无效的（指针传入的也是指针的副本）
    static int value = 10;
    ptrRef = &value;
}
int main(){
    int a = 10;
    int *ptr = &a;
    int *&ptrRef = ptr;
    modify(ptrRef);
    std::cout<<*ptrRef<<std::endl;
}
```

## 应用例子

- 使用引用来进行修改

```
for(char c:string)
    if(std::islower(c))
        c = std::toupper(c);//There is no effect
```

```
for(char &c:string)
    if(std::islower(c))
        c = std::toupper(c);
```

- 传递引用加快速度

```
int count_lowercase(std::string str) {
    int cnt = 0;
    for (char c : str)
        if (std::islower(c))
            ++cnt;
    return cnt;
}
```

这里很慢：因为每一次传递进入都会重新复制开辟一个新的字符串地址

```
int count_lowercase(std::string &str) {
    int cnt = 0;
    for (char c : str)
        if (std::islower(c))
            ++cnt;
    return cnt;
}
```

但是这样也存在一定的问题，因为引用只能引用左值

```
std::string s1 = something(), s2 = som_other_thing();
int result = count_lowercase(s1+s2)//Error
```

## reference-to-const

There is an exception to this rule: Reference-to-const can be bound to anything.

```
const int &rci = 42;//OK
const std::string &rcs = a+b;//OK
std::string temp = a+b;
const std::string &rcs = temp;
```

那我们再看刚才的传递方式

```
int count_lowercase(const std::string &str) {
    int cnt = 0;
    for (char c : str)
        if (std::islower(c))
            ++cnt;
    return cnt;
}
```

同时这里的str也是只读类型，通过str来修改字符串。

## 对于数组的引用

```
int arr[10];
int (&arrRef)[10] = arr;
int arr_2[3][5];
int (&two_dimiension_arr)[3][5] = arr_2;
```

和指针不同数组的引用传递的大小一定是和原来数组的引用大小相同，但是指针的可以随意改动大小

## <vector> in C++

Define and Basic use

```

std::vector<int> v;
std::vector v;           // Error: missing template argument.
std::vector<int> vi;    // An empty vector of `int`s.
std::vector<std::string> vs; // An empty vector of strings.
std::vector<double> vd;   // An empty vector of `double`s.
std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s.// "2-d" vector
std::vector<int> v{2,3,5,7};
std::vector v{2,3,5,7}; //自动推测
std::vector<std::string> vs{"Hello","World"};
std::vector<int> v3(10); //A vector of ten `int`s ,all initialized to 0;
std::vector<int> v4(10,42); //Initialized to 42
std::vector v5(10); //Error!
std::vector v6(10,42); //OK

std::vector<int> v{2,3,5,7};
std::vector<int> v2 = v; //`v2` is a copy of `v` (deep copy)
std::vector<int> v3(v); //Equivalent
std::vector<int> v4{v}; //Equivalent

std::vector<int> v1, std::vector<int> v2(10,42);
v1 = v2; //copy

```

### .size and .empty()

- `v.size()` returns the size of the vector.
- `v.empty()` returns 1 whether the vector is empty.
- `v.clear()`: Remove all the elements

### push\_back(), v.back(), v.front(), v.popback(), v.at()

- `v.back()` and `v.front()` return the **Reference** to the last element
- `v.at(Index)` returns the **Reference** element of the index. (`.at()` function has border check)

## 函数

- 但是在C++中允许函数重名，但是需要能够通过传入值的不同（传递参数的个数/传递参数的类型）来区别，**注意返回值的不同不能够作为判断标准**
- 同时和Python一样C++中的函数允许存在默认值，但是不能够像Python一样通过关键字传参，因此，如果存在默认值的情况，只能够在参数的最后使用，在前面参数使用默认值而后面参数没有默认值的情况是没有效率的。

## Lambda函数与表达式

C++11中提供了对匿名函数的支持，称为Lambda函数，Lambda将函数看成对象，比如可以将他们赋值给变量作为参数传递，还可以像函数一样对他求值。[capture list] (parameter list) mutable(可选) exception(可选) -> return type(可选) { function body }, 其中capture list用于指定lambda函数中可以访问那些外部的变量，以及如何访问这一些变量（按值捕捉或者是按照引用捕捉）mutable表示的意思是按值捕捉的变量内容可以被修改（注意只是能够修改副本，如果没有这个mutable就不能够修改传递进来的副本的数值）return type表示的是如果存在返回值那么返回值的类型是什么。

- 几个注意事项
  1. 如果在括号中没有参数的传递，那么就没有办法对外部的参数进行引用
  2. 如果要将所有的外部的副本全部传递进来，应该使用[=]{}
  3. 如果要将外部的所有的变量都引用调入则应该使用[&]

## Class