# 课堂上不教的 C++ 的基本特性

C++ 作为一个历史久远，功能丰(yong)富(zhong)而且标准与时俱进的语言，理应什么都能做，什么都用得起来。不过日常使用中我们初学者真的好像只学到了其中的一部分，对于一些另类的特性都不怎了解。这篇文章列举一些 C++ 的用到的或多或少，但是学习中几乎都会忽视的语言特(lou)性(dong)，希望读者看完能有收获。如果你没有收获，建议去做一名语言律师~

以下一部分都是从 SO 的高票问题找出的，还有一些是业余的收集，如果有纰漏敬请指出。当然过于高(zhuang)级(bi)的奇技淫巧就没有必要介绍了。以下例子纯手打，在 clang 测试过。

**交换数组变量和下标 (c)**

如果你想获取数组元素，可以写 A[i] 或者是 i[A]：

**int** a[] { 1,2,3 };

cout **<<** a[1] **<<** endl; *// 2*

cout **<<** 1[a] **<<** endl; *// 2*

因为数组取下标相当于计算指针地址与偏移量，而 \*(A+i) 和 \*(i+A) 意思是相同的。

**合并字符串 (c)**

**const** **char** **\***s **=**

"welcome to my\n"

" home!\n"

" enjoy life!\n";

最后 s 的值是 "welcome to my\n home!\n enjoy life!\n"，即以上三行合起来。这个语法源于 C，在 C 标准库宏中广泛出现。大多数常见语言也都有这个特性。

**逻辑运算关键字 (c++98)**

对于布尔值运算，C++ 也提供 and，or 这样的关键字，与 &&，|| 等作用相同：

**bool** b **=** not (false or true and false);

**int** i **=** 8 xor 18;

cout **<<** b **<<** endl; *// 1*

cout **<<** i **<<** endl; *// 26*

**双字符组 Digraph / 三字符组 Trigraph (c)**

过往部分地区的人们的键盘不方便打出大括弧之类的特殊符号，所以类似字符串的转义字符，这些语法符号也可以被另外常见的符号所代表

**%:**include **<**iostream**>**

**using** **namespace** std;

**int** **main**() **<%**

**int** a**<::>** **=** **<%** 1,2,3 **%>**;

cout **<<** a**<:**1**:>** **<<** endl; *// 2*

*// trigraph 写法，必须开启 -trigraphs 选项*

cout **<<** a**??**(1**??**) **<<** endl; *// 2*

**%>**

是可以在现代编译器上编译的。 尽管不再有用，Digraph 仍然存在，不过它的兄弟 Trigraph 则很早就已经被废弃了。

**变量类型修饰符的顺序 (c)**

我们知道像 const，static 等变量修饰符可以随意交换顺序，不过你有没有想过这种情况呢？

**long** **const** **int** **static** **long** value **=** 2;

它可以通过编译，而且类型是 const long long。

**uniform/aggregate initialization (c++11)**

C++98 中对象和内建类型的初始化方式五花八门，为了解决这个问题，C++11 引入了所谓的集合初始化：

**int** a {}; *// 默认初始化，对于整形，初始化为 0，相当于 bzero*

**int** b {10}; *// 初始化为 10*

**int** f **=** {}, g **=** {50}; *// !*

**double** d {10};

**const** **char** **\***s {"hello"};

cout **<<** f **<<** ' '**<<** g **<<** endl; *// 0 50*

这旨在使内建非对象的类型（基础类型和数组）都能像用户定义的对象一样以相同的语法被初始化。这样就相当于“像 int 这类的类型也有了 initializer\_list 构造函数”，这给模板函数带来了很大的方便：

**struct** **int\_wrapper** {

**int** value;

int\_wrapper(**int** value\_)**:** value{value\_} {} *// <-*

};

**template<class** **T>** T factory() {

**return** T{114514}; *// <-*

}

**int** **main**() {

**int** i **=** factory**<int>**();

int\_wrapper w **=** factory**<**int\_wrapper**>**();

}

本文的第一个例子就用到了这个特性。统一和集合初始化的门道很多，建议大家查阅相关专业资料。

**int** **z**(10.1); *// OK. 强转为 10*

*// int x{10.1}; // 报错*

**函数返回 void (c)**

如果一个函数的返回类型是 void，那么你是可以用 return 返回它的：

**void** **print\_nums**(**int** i) {

**if** (i **==** 0)

**return**;

**else** {

cout **<<** i **<<** endl;

**return** print\_nums(i **-** 1); *// <- return*

}

}

这个特性我觉得大家应该都会知道，而且它在动态语言里也很常见。和上一条一样，其在模板中有应用。但是这个例子可以写得更激进：

**void** **print\_nums**(**int** i) {

**return** i **==** 0 **?** **void**() **:** (cout **<<** i **<<** endl, print\_nums(i **-** 1));

}

**表达式返回左值 (c++98)**

像函数或者表达式返回一个左值 (lvalue) 是 C++ 中最基础的操作，不过有时候也能玩出花儿来：

**int** divisibleby3 **=** 0, others **=** 0;

**for** (**int** i **=** 0; i **<** 1000; **++**i)

(i **%** 3 **==** 0 **?** divisibleby3 : others) **+=** 1;

cout **<<** divisibleby3 **<<** ' ' **<<** others **<<** endl; *// 334 666*

只要等号左边是左值，什么东西都可以放。哪怕你是有逗号运算符，还是 lambda。

**int** total **=** 0;

("loudly get the total count",

([**&**]() **->** **int&** {

cout **<<** "assigning total count!\n";

**return** total;

})())

**=** divisibleby3 **+** others; *// total = 1000*

当心被同事打死。

**整形字面量分隔符 (c++14)**

C++14 不仅加入了二进制字面量的支持 (0bxxxyyyzzz)还加入了分隔符，再也不用担心数字太多眼睛看花的问题了：

**int** opcode **=** 0b0001'0011'1010;

**double** e **=** 2.7'1828'1828'459045;

**函数域 try/catch (c++98)**

很少有人知道函数，构造函数，析构函数可以声明全局的异常捕获，就像这样：

**int** **bad\_func**() { **throw** runtime\_error("hahaha I crashed"); **return** 0; }

**struct** **try\_struct** {

**int** x, y;

try\_struct() **try:** x{0}, y{bad\_func()} {}

**catch** (...) {

cerr **<<** "it is crashing! I can't stop it." **<<** endl;

}

};

**int** **main**() **try** {

try\_struct t;

} **catch** (exception **&**e) {

cerr **<<** "program crashed. reason: " **<<** e.what() **<<** endl;

**return** 1;

}

*// 输出：*

*// it is crashing! I can't stop.*

*// program crashed. reason: hahaha I crashed*

对于函数，在这里其作用就相当于在 main 下多写一层花括号。构造函数类似，但是在 catch 块中如果用户不抛出异常，编译器会一定隐式抛出原异常。 ([https://en.cppreference.com/w/cpp/language/function-try-block](https://link.zhihu.com/?target=https%3A//en.cppreference.com/w/cpp/language/function-try-block))

**匿名类 (c++98)**

C 就支持匿名 (untagged) 类的定义，不过 C++ 更进一步，你可以在函数的任何地方定义匿名类，甚至循环变量的声明中，如以下反转数组的函数：

**void** **reverse**(**int** **\***arr, **int** size) {

**for** (**struct** { **int** l, r; } i **=** { 0, size**-**1 }; i.l **<=** i.r; **++**i.l, **--**i.r) {

swap(arr[i.l], arr[i.r]);

}

}

匿名类也可以出现在 using 后（自行尝试）。有了 C++ 的 auto (c++14)，用户甚至可以返回真~匿名类：

**auto** **divide**(**int** x, **int** y) {

**struct** /\* **unnamed** \*/ {

**int** q, r;

*// cannot define friend function*

ostream **&**print(ostream **&**os) **const** {

os **<<** "quotient: " **<<** q **<<** " remainder: " **<<** r;

**return** os;

}

} s { x **/** y, x **%** y };

**return** s;

}

**int** **main**() {

divide(11,2).print(cout) **<<** endl;

}

*// 输出*

*// quotient: 5 remainder: 1*

除此还可以定义虚函数和构造析构函数。如果仔细想想，其原理和 lambda 差不多。

**if 语句中声明变量 (c++98)**

如下所示：

**bool** **get\_result**() { **return** false; }

**int** **main**() {

**if** (**bool** i **=** get\_result())

cout **<<** "success!" **<<** endl;

**else**

cout **<<** "fail" **<<** endl;

*// i 不可用*

}

*// 输出*

*// fail*

在 while 语句中也可以使用这个结构。例子中 i 的生命周期限于这个 if/else 块中。其就相当于

**int** **main**() {

{

**bool** i **=** get\_result();

**if** (i)

cout **<<** "success!" **<<** endl;

**else**

cout **<<** "fail" **<<** endl;

}

}

能在作用域中声明变量，这和 C 程序中的类似技巧很不同。

在 C++17 中，这个特性被加强了。不仅可以声明，还可以像 for 循环一样附加条件，和 golang 很像

**int** **get\_result**() { **return** 17; }

**int** **main**() {

**if** (**int** i **=** get\_result(); i **>=** 18)

cout **<<** "ok." **<<** endl;

**else**

cout **<<** "fail. your age is too low: " **<<** i **<<** endl;

}

*// 输出*

*// fail. your age is too low: 17*

**结构化绑定 (c++17)**

C++17 的新语法使得我们可以在一个语句里解包变量，例如解包一个长度为 3 的数组：

**int** arr[3] { 1,2,9 };

*// 相当于把数组的值都拷贝到这三个变量里*

**auto** [cpy0, cpy1, cpy2] **=** arr;

cpy0 **=** 2;

cout **<<** cpy0 **<<** ' ' **<<** arr[0] **<<** endl; *// 2 1*

*// 相当于把数组的值起了三个别名*

**auto** **&**[ref0, ref1, ref2] **=** arr;

ref0 **=** 2;

cout **<<** ref0 **<<** ' ' **<<** arr[0] **<<** endl; *// 2 2*

这个特性可以解包标准库中的 std::tuple，也可以以成员声明顺序解包一个结构体：

**auto** **get\_data**() {

**struct** { **int** code; string header, body; }

r { 200, "200 OK\r\nContent-Type: application/json", "{}" };

**return** r;

}

**int** **main**() {

*// std::ignore 用来丢弃一个不需要的数据*

**if** (**auto** [code, ignore, json] **=** get\_data(); code **==** 200)

cout **<<** "success: " **<<** json **<<** endl;

**else**

**throw** runtime\_error{"request failed"};

}

在 C++17 之前，使用 [std::tie](https://link.zhihu.com/?target=https%3A//en.cppreference.com/w/cpp/utility/tuple/tie) 也可以实现相类似的效果：

**int** a **=** 1, b **=** 2, c **=** 3;

*// std::make\_tuple 封包，std::tie 解包*

tie(a, b, c) **=** make\_tuple(b, c, a);

cout **<<** a **<<** b **<<** c **<<** endl; *// 231*

**placement new (c++98)**

当对象被 new 的时候，大家都知道发生的过程是先调用 operator new 来分配内存，再调用构造函数。不过 new 拥有另一个重载，我们可以跳过分配内存的一步，也就是在我们指定的内存区域直接初始化对象。

**struct** **vector2d** {

string name;

**long** x, y;

vector2d(**long** x\_, **long** y\_)**:** x(x\_), y(y\_), name("unnamed") {}

};

**int** **main**() {

**char** buff[1 **<<** 10];

*// 在 buffer 上创建对象*

vector2d **\***p **=** **new** (buff) vector2d(3, 4);

cout **<<** p**->**name **<<** "("

**<<** p**->**x **<<** ", " **<<** p**->**y **<<** ")" **<<** endl;

cout **<<** **\*reinterpret\_cast<**string **\*>**(buff)

**<<** "("

**<<** **\*reinterpret\_cast<long** **\*>**(buff **+** **sizeof**(string))

**<<** ", "

**<<** **\*reinterpret\_cast<long** **\*>**(buff **+** **sizeof**(string) **+** **sizeof**(**long**))

**<<** ")" **<<** endl;

*// 析构对象*

p**->~**vector2d();

*// 不保证原内存一定会被清零！*

}

*// 输出*

*// unnamed(3, 4)*

*// unnamed(3, 4)*

**全局命名空间操作符 (c++98)**

可以使用 :: 来显式表示当前所表示的符号来自于全局命名空间，从而消除歧义：

**namespace** un {

**void** **func**() { cout **<<** "from namespace un\n"; }

}

**void** **func**() { cout **<<** "from global\n"; }

**using** **namespace** un;

**int** **main**() {

**::**func();

}

*// 输出*

*// from global*

**匿名命名空间 (c++98)**

使用匿名的命名空间可以限制符号的可见性。当你写下这段代码时：

*// test.cpp*

**namespace** {

**void** **local\_function**() {}

}

它相当于这段代码：

*// test.cpp*

**namespace** \_\_\_some\_unique\_name\_test\_cpp\_XADDdadh876Sxb {}

**using** **namespace** \_\_\_some\_unique\_name\_test\_cpp\_XADDdadh876Sxb;

**namespace** \_\_\_some\_unique\_name\_test\_cpp\_XADDdadh876Sxb {

**void** **local\_function**() {}

}

也就是一个独一无二，对于当前编译单元的命名空间被创建。因为这个命名空间只在这个文件中引用，故而用户只可以在当前文件 (test.cpp) 中引用 local\_function。这样做就避免了不同文件中可能出现的名称冲突。其效果与

*// test.cpp*

**static** **void** **local\_function**() {}

相同。对于匿名命名空间和 static，可以参考这个 SO 问题： [https://stackoverflow.com/questions/154469/unnamed-anonymous-namespaces-vs-static-functions](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/154469/unnamed-anonymous-namespaces-vs-static-functions)

**“内联” (inline) 命名空间 (c++11)**

如果不是库作者，可能会对这个特性十分惊讶，内联这个关键字的含义已经和 static 一样要起飞了。如果一个命名空间被内联，那么它会被给予优先级。

**namespace** un {

**inline** **namespace** v2 {

**void** **func**() { cout **<<** "good func.\n"; }

}

**namespace** v1 {

**void** **func**() { cout **<<** "old func. use v2 instead.\n"; }

}

}

**int** **main**() {

un**::**func();

un**::**v2**::**func();

un**::**v1**::**func();

}

*// 输出*

*// good func.*

*// good func.*

*// old func. use v2 instead.*

**自定义字面量 (c++11)**

自从 C++11，用户可以自己重载 operator"" 来以字面量的形式初始化对象：

**long** **double** **constexpr** **operator**""\_deg (**long** **double** deg) {

**return** deg **\*** 3.14159265358979323846264L **/** 180;

}

**int** **main**() {

**long** **double** r **=** 270.0\_deg; *// r = 4.71239...*

}

**重载 , 运算符 (c++98)**

没想到吧，逗号也能重载！这个例子受 boost/assign/std/vector.hpp 的启发，可以简便地向标准库的 vector 中一个个地插入元素：

**template<class** **VT>**

**struct** **vector\_inserter** {

VT **&**vec;

vector\_inserter(VT **&**vec\_)**:** vec(vec\_) {}

**template<class** **TT>**

vector\_inserter**<**VT**>** **&operator**,(TT **&&**v) {

vec.emplace\_back(forward**<**TT**>**(v));

**return** **\*this**;

}

};

**template<class** **T**, **class** **Alloc**, **class** **TT>**

vector\_inserter**<**vector**<**T, Alloc**>>**

**operator+=**(vector**<**T, Alloc**>** **&**vec, TT **&&**v) {

vec.emplace\_back(forward**<**TT**>**(v));

**return** {vec};

}

**int** **main**() {

vector**<int>** v;

*// 使用 emplace\_back 赋值*

v **+=** 1,2,3,4,5,6,7;

cout **<<** v.size() **<<** endl; *// 7*

}

逗号操作符还有一处另类的地方是当操作数的类型为 void 时，没有重载可以改变它的行为。这可以（在c++98）用来检测一个表达式是否为 void: [https://stackoverflow.com/questions/5602112/when-to-overload-the-comma-operator#answer-7033345](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/5602112/when-to-overload-the-comma-operator%23answer-7033345)

除此之外大多数重载 , 的行为都不是什么好行为，知道能重载就好，想用的话最好三思！

**类成员声明顺序 (c++98)**

类成员在使用时不需要关心它们的声明顺序，所以我们可以先使用变量再定义。所以有时为了方便完全可以把所有的代码写在一个类里。

**struct** **Program** {

vector**<**string**>** args;

**void** **Main**() {

cout **<<** args[0] **<<** ": " **<<** a **+** foo() **<<** endl;

}

**int** a **=** 3;

**int** **foo**() { **return** 7; }

};

**int** **main**(**int** argc, **char** **\*\***argv) {

Program{vector**<**string**>**(argv, argv**+**argc)}.Main();

}

**类函数引用修饰符 (c++11)**

不同于 const，noexcept 这样仅修饰类方法本身行为的修饰符，& 和 && 修饰符会根据 \*this 是左值还是右值引用来选择合适的重载。

**struct** **echoer** {

**void** **echo**() **const** **&** {

cout **<<** "I have long live!\n";

}

**void** **echo**() **const** **&&** {

cout **<<** "I am dying!\n";

}

};

**int** **main**() {

echoer e;

e.echo();

echoer().echo();

}

*// 输出*

*// I have long live!*

*// I am dying!*

**类命名空间操作符 (c++98)**

只有子类型的指针，如何调用父类型的方法？用命名空间。

**struct** **Father** {

**virtual** **void** **say**() {

cout **<<** "hello from father\n";

}

};

**struct** **Mother** {

**virtual** **void** **say**() {

cout **<<** "hello from mother\n";

}

};

**struct** **Derived1:** Father, Mother {

**void** **say**() {

Father**::**say(); Mother**::**say();

cout **<<** "hello from derived1\n";

}

};

**int** **main**() {

Derived1 **\***p **=** **new** Derived1{};

p**->**say();

p**->**Father**::**say();

}

*// 输出*

*// hello from father*

*// hello from mother*

*// hello from derived1*

*// hello from father*

**构造器委托 (c++11)**

可以使用和继承类相同的语法在一个构造函数中调用另外一个构造函数：

**struct** **CitizenRecord** {

string first, middle, last;

CitizenRecord(string first\_, string middle\_, string last\_)

**:** first{move(first\_)}, middle{move(middle\_)}, last{move(last\_)} {

**if** (first **==** "chao") cout **<<** "important person inited\n";

}

*// 默认参数委托*

CitizenRecord(string first\_, string last\_)

**:** CitizenRecord{move(first\_), "", move(last\_)} {}

*// 拷贝构造函数委托*

CitizenRecord(**const** CitizenRecord **&**o)

**:** CitizenRecord{o.first, o.middle, o.last} {}

*// 移动构造函数委托*

CitizenRecord(CitizenRecord **&&**o)

**:** CitizenRecord{move(o.first), move(o.middle), move(o.last)} {}

};

这个特性有时可以适量减少代码重复，或者转发默认参数。

**自定义枚举存储类型 (c++11)**

C 时代的枚举中定义中的整形都必须是 int，在 C++ 中可以自定义这些类型：

**enum** **class** **sizes** **:** size\_t {

ZERO **=** 0,

LITTLE **=** 1ULL **<<** 10,

MEDIUM **=** 1ULL **<<** 20,

MANY **=** 1ULL **<<** 30,

JUMBO **=** 1ULL **<<** 40,

};

**int** **main**() {

cout **<<** **sizeof** sizes**::**JUMBO **<<** endl; *// 8*

}

其中 sizes::ZERO 这些常量都持有 size\_t 类型。顺便一提 enum 和 enum class 的区别是前者的作用域是全局，后者则需要加上 sizes::.

**模板联合 (c++98)**

联合也是 C++ 的类，也支持方法，构造和析构，同样还有模板参数。

**template<class** **T**, **class** **U>**

**union** **one\_of** {

**private:**

T left\_value;

U right\_value;

**public:**

T **&**left\_cast() { **return** left\_value; }

U **&**right\_cast() { **return** right\_value; }

};

**int** **main**() {

one\_of**<long** **double**, **long** **long>** u;

u.left\_cast() **=** 3.3;

cout **<<** u.left\_cast() **<<** endl; *// 3.3*

u.right\_cast() **=** 1LL **<<** 32;

cout **<<** u.right\_cast() **<<** endl; *// 4294967296*

}

一点需要注意的是，union 的成员必须是 "trivial" 的，也就是无需显式构造函数，否则轻则编译失败，重则 UB (未定义行为)。

**模板位域 (c++98)，模板 align (c++11)**

你可能很熟悉 C/C++ 的位域特性，但是你知道位域可以写进模板吗？

**template<**size\_t I, size\_t J**>**

**struct** **some\_bits** {

**int32\_t** a : I, b : I;

**int32\_t** c : J;

};

**int** **main**() {

some\_bits**<**8, 16**>** s;

s.a **=** 127;

s.b **=** 128;

s.c **=** 65535;

cout **<<** "a: " **<<** s.a **<<** "\nb: " **<<** s.b **<<** "\nc: " **<<** s.c

**<<** "\ntotal size: " **<<** **sizeof** s **<<** endl;

}

*// 输出*

*// a: 127*

*// b: -128*

*// c: -1*

*// total size: 4*

与此相似，C++11 的 alignas 也可以参与模板：

**template<**size\_t Size**>**

**struct** **alignas**(Size) empty\_space {};

**int** **main**() {

empty\_space**<**64**>** pad;

cout **<<** **sizeof** pad **<<** endl; *// 64*

}

**类成员指针 (c++98)**

正如 int(\*)(int, int) 代表一个接受两个整形返回一个整形的函数指针，对于一个类型 T，int(T::\*)(int, int) 表示一个非静态的类成员函数的类型。这是因为这种函数总会有一个隐式的 this 指针作为第一个参数，所以我们需要使用不同的语法来区分它们。

**struct** **echoer** {

string name;

**void** **echo1**(string **&**c) **const** {

cout **<<** "I'm " **<<** name **<<** ". hello " **<<** c **<<** "!\n";

}

**static** **void** **echo2**(string **&**c) {

cout **<<** c **<<** "! I have long/short live!\n";

}

};

**int** **main**() {

echoer me {"mia"};

string you {"alice"};

string echoer**::\***aptr **=** **&**echoer**::**name; *// 类成员变量指针*

**void** (echoer**::\***mptr)(string**&**) **const** **=** **&**echoer**::**echo1; *// 类方法指针*

**void** (**\***smptr)(string**&**) **=** **&**echoer**::**echo2; *// 类静态方法指针 (就是普通函数指针)*

**void** (**&**smref)(string**&**) **=** echoer**::**echo2; *// 类静态方法引用*

(me.**\***mptr)(you);

smptr(you);

smref(you);

echoer **\***another **=** **new** echoer{"valencia"};

(another**->\***mptr)(me.**\***aptr);

**delete** another;

}

*// 输出*

*// I'm mia. hello alice!*

*// alice! I have long/short live!*

*// alice! I have long/short live!*

*// I'm valencia. hello mia!*

成员指针和函数指针是实现 traits 必不可少的。

**返回值后置 (c++11)**

R (Args) 等同于 auto (Args) -> R。 返回值后置可以用在很多地方，如常见的 lambda：

**const** **auto** add **=** [](**int** a, **int** b) **->** **int** { *// 返回类型为 int*

**return** a **+** b;

};

以及在一般的函数中表示返回类型依赖于参数：

**template<class** **T>**

**auto** serialize\_object(T **&**obj) **->** **decltype**(obj.serialize()) {

*// 返回类型为 obj.serialize() 的类型*

**return** obj.serialize();

}

一般不为人知的是，后置返回值还可以简化函数指针的写法。

**int** **add**(**int** a, **int** b) { **return** a **+** b; }

**int** **mul**(**int** a, **int** b) { **return** a **\*** b; }

**auto** **produce**(**int** op) **->** **auto** (**\***)(**int**, **int**) **->** **int** { *// 声明后置*

**if** (op **==** 0) **return** **&**add;

**else** **return** **&**mul;

}

*// 不后置的话这个函数的声明是这样的：*

*// int (\*(produce(int op)))(int, int);*

**int** **main**() {

**using** producer\_t **=** **auto** (**\***)(**int**) **->** **auto** (**\***)(**int**, **int**) **->** **int**; *// 指针后置*

*// 不后置的话这个类型的声明是这样的*

*// using producer\_t = int (\*((\*)(int)))(int, int);*

producer\_t ptr {**&**produce};

cout **<<** ptr(0)(1, 2) **<<** endl; *// 3*

}

类成员函数也类似地可以这样写。在 C++14 中函数声明的返回值可以只写 auto 来让编译器自动推导返回类型了。

因为函数声明 = 后置返回值的函数声明，我们可以在模板参数里也使用这个语法，例如 std::function：

**const** function**<auto** (**int**, **int**) **->** **int>** factorial

**=** [**&**](**int** n, **int** acc) {

**return** n **==** 1 **?** acc : factorial(n**-**1, acc **\*** n);

};

cout **<<** factorial(5, 1) **<<** endl; *// 120*

**不求值表达式和编译期常量 (c++98)**

在 C 时代就有仅存在于编译期的表达式。在程序编译运行后，这些代码就被删除，替换成常量了。比如常见的 sizeof 运算符

**int** count **=** 0;

cout **<<** **sizeof**(count**++**) **<<** endl; *// 4*

cout **<<** count **<<** endl; *// 0*

因为 sizeof(count++) 直接在编译期被替换成了 4，其所原本应该具有的副作用也会没有作用。  
在 C++11 中引入了 decltype，它和 noexcept, sizeof 操作符相同，仅存在于编译时期。而这个操作符的作用是获得一个表达式的类型，譬如如此：

**decltype**(**new** **int**[10]) ptr {}; *// 等同于 int \*ptr = nullptr; 根本没有内存被分配！*

**decltype**(**int**{} **\*** **double**{}) value {}; *// 只是声明一个变量，这个变量的类型是 int 乘 double 的类型*

*// 具体是什么类型我自己不知道*

与其一同来临的还有 declval 等。同时还有 constexpr 表达式，可以保证表达式一定会在编译期内计算完毕，不拖延到运行时。

**int** **constexpr** **fib**(**int** n) {

**return** n **==** 0 **?** 0 **:** n **==** 1 **?** 1 **:** fib(n**-**1) **+** fib(n**-**2);

}

**int** **main**() {

**int** **constexpr** x **=** fib(20); *// 完全等同于 x = 6765！运行程序时根本不会再去计算*

}

实际上于此我们可以解释凭什么有的函数没有定义就能跑：

**template<int** v**>** **struct** **i32** { **static** **constexpr** **int** value **=** v; };

**template<int** v**>** i32**<**v **\*** 2 **+** 1**>** weird\_func(i32**<**v**>**);

i32**<**0**>** weird\_func(...);

**int** **main**() {

cout **<<** **decltype**(weird\_func(i32**<**7**>**{}))**::**value **<<** endl; *// 15*

cout **<<** **decltype**(weird\_func(7))**::**value **<<** endl; *// 0*

}

因为它们根本没跑。

**decltype(auto) (c++14)**

之前讲过 auto 可以作为函数的返回值来自动推导，不过对于 auto 和 decltype(auto)，虽然大多数情况下后者是累赘，也存在两者意义不同情况：

**auto** **incr1**(**int** **&**i) { **++**i; **return** i; } *// 返回: int (拷贝)*

**decltype**(**auto**) incr2(**int** **&**i) { **++**i; **return** i; } *// 返回: int&*

**int** **main**() {

**int** a **=** 0, b **=** 0;

*// cout << incr1(incr1(a)) << endl; // 报错*

cout **<<** incr2(incr2(b)) **<<** endl; *// 输出 2*

}

auto 会看所要推导的变量原生类型 T，而 decltype(auto) 会推导出变量的实际类型 T& 或是 T&&。

**引用折叠和万能引用 (universal reference) (c++11)**

T&& 不一定是 T 的右值引用，它既有可能是左值引用，也有可能是右值引用，但一定不是拷贝原值。

**int** val **=** 0;

**int** **&**ref **=** val;

**const** **int** **&**cref **=** val;

**auto** s1 **=** ref; *// 拷贝了一个 int！*

**auto** **&&**t1 **=** ref; *// int& && = int&*

**auto** **&&**t2 **=** cref; *// const int& && = const int&*

**auto** **&&**t3 **=** 0; *// int&& && = int&&*

当遇到需要推导类型的情况，被推导的 auto 类型会与 && 相结合，按照以上的规则得出总的类型。因为这个特性可以不用拷贝且可以保持变量的原有引用类型，它常和移动构造函数配合进行所谓的“完美转发”：

**struct** **CitizenRecord** {

string first, middle, last;

**template<class** **F**, **class** **M**, **class** **L>**

CitizenRecord(F **&&**first\_, M **&&**middle\_, L **&&**last\_) *// perfect forwarding!*

**:** first{forward**<**F**>**(first\_)}, middle{forward**<**M**>**(middle\_)}, last{forward**<**L**>**(last\_)} {}

**template<class** **F**, **class** **L>**

CitizenRecord(F **&&**first\_, L **&&**last\_)

**:** CitizenRecord{forward**<**F**>**(first\_), "", forward**<**L**>**(last\_)} {}

};

以上就相当于一次性把 &, const &, && 的重载都写了。

**显式模板初始化 (c++98)**

大家都知道模板只在编译时存在。如果一个模板定义从来没有被使用过的话，那么它就没有实例，相当于从来没有定义过模板。所以我们不能把模板的实现和声明分别放在实现文件和头文件中。不过我们可以显式地告诉编译器实例化部分模板：

*// genlib.hpp*

**#pragma once**

**template<class** **T>** T my\_max(T a, T b);

*// genlib.cpp*

**template<class** **T>** T my\_max(T a, T b) {

**return** a **>** b **?** a : b;

}

**template** **int** my\_max**<int>**(**int**, **int**); *// <-*

**template** **double** my\_max**<double>**(**double**, **double**); *// <- 指定实例化*

*// user.cpp*

**#include** <iostream>

**#include** "genlib.hpp"

**using** **namespace** std;

**int** **main**() {

cout **<<** my\_max(4, 5) **<<** endl; *// 5*

cout **<<** my\_max(4.5, 5.5) **<<** endl; *// 5.5*

*// cout << my\_max(4, 5.5) << endl; // 错误：没有对应的重载*

}

在这里我在 genlip.cpp 定义了函数模板并且指定生成了两个实例，这样它们的符号就可见于外部，连接器就可以找到相应的定义。另外一个单元 user.cpp 即可引用相应的函数。

**模板模板参数 (template template) (c++98)**

以及它的兄弟姐妹模板x3参数，模板x4参数，......

初学者大都熟悉模板的类型参数 (template<class>) 和非类型参数 (template<auto>)，不过对于模板模板参数可能会很不熟悉，因为需要这个特性的地方很少。所谓的模板模板参数实际上就表示参数本身就是个模板，比如 std::vector 是一个模板类。如果要把它传入一个接受普通的模板参数的模板中，我们只能去实例化它，例如传入 std::vector<int>。对于模板模板参数，可以直接传入这个模板类 std::vector：

**template<template<class**...**>** **class** **GenericContainer>**

**struct** **Persons** {

GenericContainer**<int>** ids;

GenericContainer**<**Person**>** people;

};

**int** **main**() {

Persons**<**vector**>** ps;

ps.ids.emplace\_back(1);

ps.people.emplace\_back(Person{"alice"});

}

在这里 GenericContainer 就是模板模板，我们不仅可以使用 vector 来初始化 Persons，还可以使用任何 STL 的容器模板类。

因为参数可以套娃了，所以模板也有自己所谓的“高阶函数”

*// “变量”类型*

**template<int** v**>** **struct** **i32** {

**using** type **=** i32;

**static** **constexpr** **int** value **=** v;

};

*// 计算一个数加一*

**template<class** **I1>** **struct** **add1** **:** i32**<**I1**::**value **+** 1**>** {};

*// 组合两个函数*

**template<template<class>** **class** **F**, **template<class>** **class** **G>**

**struct** **compose** { *// 显式标注 typename 消除歧义*

**template<class** **I1>** **struct** **func** **:** F**<typename** G**<**I1**>::**type**>** {};

};

**int** **main**() {

*// 加一函数组合三次就是加三*

**using** add3 **=** compose**<**add1, compose**<**add1, add1**>::**func**>**;

*// “声明”一个“变量”*

**using** num **=** i32**<**7**>**;

cout **<<** add3**::**func**<**num**>::**value **<<** endl; *// 9*

}

**class template deduction guide (CTAD) (c++17)**

从 C++17 开始，在初始化模板类的时候可以不用标注模板参数，如这样：

vector vec1 {1,2,3,4,6}; *// 等同于 vector<int>*

vector vec2 {"hello", "alice"}; *// 等同于 vector<const char \*>*

相应的类型会被自动推导。除了使用编译器默认的设置，我们可以通过 deduction guide 定义自己的推导规则。之所以提及它，是因为这是一个大家可能会遇到的陌生语法：

**struct** **ptr\_wrapper** {

uintptr\_t ptr;

**template<class** **T>** ptr\_wrapper(T **\***p)**:** ptr{**reinterpret\_cast<**uintptr\_t**>**(p)} {}

};

**template<class** **T>** **struct** **bad\_wrapper** {

T thing;

};

*// 如果参数是 const char \*，则调用 bad\_wrapper<string> 的构造函数。这样写没问题因为*

*// string 可以由 const char \* 构造*

bad\_wrapper(**const** **char** **\***) **->** bad\_wrapper**<**string**>**;

*// 如果参数是指针，则调用 bad\_wrapper<ptr\_wrapper>，因为我自己定义了构造函数，所以 OK*

**template<class** **T>** bad\_wrapper(T **\***) **->** bad\_wrapper**<**ptr\_wrapper**>**;

**int** **main**() {

bad\_wrapper w {"alice"}; *// bad\_wrapper<string>*

cout **<<** w.thing.size() **<<** endl; *// 5*

bad\_wrapper p {**&**w}; *// bad\_wrapper<ptr\_wrapper>*

cout **<<** p.thing.ptr **<<** endl; *// 140723957134752*

}

当使用 {} 初始化对象时，编译器会查看大括号内参数的类型，如果参数类型符合推导规则的左侧，则编译器会按照箭头右侧的规则来实例化模板类。当然，这需要箭头右侧的表达式在被代入后有效，而且需要类型要可以被实际的参数构造。([https://en.cppreference.com/w/cpp/language/class\_template\_argument\_deduction](https://link.zhihu.com/?target=https%3A//en.cppreference.com/w/cpp/language/class_template_argument_deduction))

**递归模板 (c++98)**

模板在一定程度上就是编译期的函数，支持递归是理所应当的。在早期，这个高级学究的 zhuangbi 利器。利用类的继承和模板的特化就可以实现很多递归和匹配操作。下面是一个不用内建数组实现的数组功能：

*// 要先生成 长度为 Size 的数组，则要先在前面生成长度为 Size-1 的数组*

**template<class** **T**, size\_t Size**>** **struct** **flex\_array** **:** flex\_array**<**T, Size**-**1**>** {

**private:** T v;

};

**template<class** **T>** **struct** **flex\_array<**T, 1**>** {

T **&operator**[](size\_t i) { **return** **\***(**&**v **+** i); }

**private:** T v;

};

**int** **main**() {

flex\_array**<unsigned**, 4**>** arr;

cout **<<** **sizeof** arr **<<** endl; *// 16*

}

**动态类型信息 RTTI (c++98)**

依赖虚函数，dynamic\_cast 等工具我们可以在程序运行时获得类型的信息并且检查。当重载 = 算符的时候，我们要小心。

**class** **Base** {

**public:**

**virtual** Base **&operator=**(**const** Base**&**) { **return** **\*this**; }

};

**class** **Derived1:** **public** Base {

**public:**

Derived1 **&operator=**(**const** Base **&**o) **override**

**try** { *// 动态类型转换！*

**const** Derived1 **&**other **=** **dynamic\_cast<const** Derived1**&>**(o);

**if** (**this** **==** **&**other) **return** **\*this**;

Base**::operator=**(other);

**return** **\*this**;

} **catch** (bad\_cast**&**) { *// 类型信息*

**throw** runtime\_error{string{"type mixed! passed type: "} **+** **typeid**(o).name()};

}

};

**int** **main**() {

Base **\***p1 **=** **new** Base{};

Base **\***p2 **=** **new** Derived1{};

**\***p2 **=** **\***p1; *// exception: type mixed! passed type: 4Base*

}

**静态多态和静态内省 (c++98)**

动态类型信息和虚函数等方便使用，只不过也会造成运行时的损耗。通过递归的模板，我们有 CRTP 设计模式来实现静态的多态方法派发。

**template<class** **Derived>** **class** **IPersonaBase** {

**void** **speak\_impl\_**() **const** { cout **<<** "I'm unnamed." **<<** endl; }

**public:** *// 静态类型转换！*

**void** speak() **const** { **static\_cast<const** Derived **\*>**(**this**)**->**speak\_impl\_(); }

};

**class** **Alice** **:** **public** IPersonaBase**<**Alice**>** { *// <- curiously recurring*

**friend** IPersonaBase**<**Alice**>**;

**int** number **=** 665764;

**void** **speak\_impl\_**() **const** { cout **<<** "I'm alice bot #" **<<** number **<<** endl; }

};

**class** **Mian** **:** **public** IPersonaBase**<**Mian**>** {};

**int** **main**() {

Alice a; Mian m;

a.speak(); m.speak();

}

*// 输出*

*// I'm alice bot #665764*

*// I'm unnamed.*

CRTP 的一个特点是支持按子类的类型返回子类。这个 SO 答案描述了 CRTP 的用途 [https://stackoverflow.com/questions/4173254/what-is-the-curiously-recurring-template-pattern-crtp](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/4173254/what-is-the-curiously-recurring-template-pattern-crtp)

我们没有动态内省，但是静态的内省机制也已经足够，并且性能更强。通过 SFINAE (substitution failure is not an error)，用户在程序运行前就可以提前知道该使用什么重载来对付不同的类型了。比如，我们可以检测参数的类型有没有 serialize 方法

*// 辅助函数，用来检测 T 是不是有 serialize() 方法*

**template<class** **T>** **struct** **is\_serializable** {

**private:**

**template<class** **TT>**

**static** **decltype**(declval**<**TT**>**().serialize(), true\_type()) test(**int**);

**template<class** **TT>** **static** false\_type test(...);

**public:**

**static** **constexpr** **bool** value **=** is\_same\_v**<decltype**(test**<**T**>**(0)), true\_type**>**;

};

*// 如果 T x 有 serialize 方法，则打印它的 serialized，否则打印另外一条信息*

**template<typename** T**>** **void** print\_serialization(T **&**x) {

**if** **constexpr** (is\_serializable**<**T**>::**value)

cout **<<** x.serialize() **<<** endl;

**else**

cout **<<** "[not serializable]" **<<** endl;

}

**constraints & concepts (c++20)**

对于模板参数的约束语法是 C++ 最重大的升级之一。有了 concept，用户大多数情况已经可以摆托 SFINAE 那简直不可读的代码，来定义类似于其他语言，但是性能更优的鸭子类型“接口”。

**template<class** **T>** **concept** ISerializable **=** **requires**(T v) {

{v.serialize()} **->** same\_as**<**string**>**;

};

**template<class** **T>** **concept** IFileAlike **=** **requires**(T v) {

{v.open()} **->** same\_as**<void>**;

{v.close()} **noexcept** **->** same\_as**<void>**;

{v.write(declval**<**string**&>**())} **->** same\_as**<void>**;

};

*// 只有定义了上面这些函数的类型才可以 print\_file*

**template<class** **T>** **void** print\_file(T **&&**file) **requires** ISerializable**<**T**>** **&&** IFileAlike**<**T**>** {

file.open();

cout **<<** file.serialize() **<<** endl;

file.close();

}

我猜很快大家就会用到了！详细内容请参考 [https://en.cppreference.com/w/cpp/language/constraints](https://link.zhihu.com/?target=https%3A//en.cppreference.com/w/cpp/language/constraints)

**总结**

当我花了一整天边写边测完成这篇文章的时候，也获得了很大的收获，所以我觉得这篇文章应该会对读者有意义。文章里有的地方没有详细展开，如果你有兴趣，不妨就按着每一条的标题去搜索！

**其他引用/推荐**

* N4860, Programming Languages — C++, ISO, 2020
* [https://en.cppreference.com/w/](https://link.zhihu.com/?target=https%3A//en.cppreference.com/w/)
* [https://stackoverflow.com/questions/75538/hidden-features-of-c](https://link.zhihu.com/?target=https%3A//stackoverflow.com/questions/75538/hidden-features-of-c)

编辑于 2020-06-23