# 目录

# 1 简介

这是一份 pzr 在 cpp quiz 网站上的刷题笔记, 其中的题目形式如下。

- 给出一段代码，请说出代码在 ISO C++17 标准下的运行结果。

给出的代码不一定是完全正确的，在代码不正确的情况，你需要指出它是以下哪种情况：

- 代码中无法通过编译。
- 代码存在未指定行为 (unspecified behavior) 或实现定义 (implementation defined) 行为。
- 代码中存在未定义行为 (undefined behavior)。

每道题目将附有解析, 解析中对于不是显见的结论, 将给出在 N4659 中的引用。

习题部分, 一部分是刷题过程中零散的感悟, 另一部分来自于两本有趣的书。分别是 *How Not to Program in C++ 111 Broken Programs and 3 Working Ones, or Why Does 2+2=5986* 和 *Exceptional C++ 47 Engineering Puzzles, Programming Problems, and Solutions*。

需要指出的是, 这篇笔记并不是针对 C++ 学习的, 这些题目对于 C++ 的学习也并不一定都是有意义的。但是, 刷题的过程还是相当有意思的。感兴趣的同学, 可以在 cpp quiz 网站上进行尝试。

注:

- (1) 编译错误: 程序中存在 C++ 标准所不允许的行为, 不应该通过编译。例如使用 C++ 关键字 (keyword) 作为变量名。
- (2) 未定义行为: 标准没有做任何要求的行为, 结果是不可预测的。例如除以 0。标准中常常出现的 is ill-formed, no diagnostic required, 也归为此类。
- (3) 未指定行为: 标准通常给出了可能的实现范围, 具体的选择由实现决定。例如函数参数的评估顺序。
- (4) 实现定义行为: 结果取决于具体实现和实现文档, 例如 sizeof(int) 的值。

# 2 C++ 基本问题

## 2.1 C++ 基本类型

**Question 151** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <type_traits>
using std::cout;
int main()
{
    cout << std::is_signed<char>::value;
}
```

**答案和解析** 程序存在未指定行为

[basic.fundamental#1]

> It is implementation-defined whether a char object can hold negative values.

char 并不一定是有符号的。

在标准中我们还可以看到, char 也不一定是 8 位的, 这些都是由实现决定的。

**Question 179** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main()
{
    std::cout << (sizeof(long) > sizeof(int));
}
```

**答案和解析** 程序存在未指定行为

依据标准, long 的大小应当大于等于 int, 具体由实现决定。

因此, 输出 1 或 0 都是可能的。

[basic.fundamental#2]

> There are five standard signed integer types : “signed char”, “short int”, “int”, “long int”, and “long long int”. In this list, each type provides at least as much storage as those preceding it in the list.

## 2.2 C++ 基本语法

**Question 6** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 3; i++)
    std::cout << i;
    for (int i = 0; i < 3; ++i)
    std::cout << i;
}
```

**答案和解析** 程序完全正确，输出为：012012

[stmt.for#1] 指出,

> ```cpp
> for(init_statement; condition; expression)
>     statement;
> ```
>
> is equivant to
>
> ```cpp
> init_statement;
> while(condition){
>     statement;
>     expression;
> }
> ```

因此, 无论第三句 (expression) 是 i++ 还是 ++i, 作用都是将 $i$ 增加 1。两者的运行结果是相同的。

**Question 111** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    int i=1;
    do {
        std::cout << i;
        i++;
        if(i < 3) continue;
    } while(false);
    return 0;
}
```

**答案和解析** 程序完全正确, 输出为: 1

这里的问题在于, 执行 continue 语句后, 是否再进行 while 条件的判断。根据[stmt.cont#1],

> The continue statement shall occur only in an iteration-statement and causes control to pass to the loop-continuation portion of the smallest enclosing iteration-statement, that is, **to the end of the loop**.

所以, continue 语句的实际作用是跳跃到循环的末尾, 所以依然会进行循环条件的判断。

**Question 333** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
int main()
```

```
{
    int x = 3;
    while (x --> 0) // x goes to 0
    {
        std::cout << x;
    }
}
```

**答案和解析** 程序完全正确, 输出为: 210

ok fine, 这是一个经典笑话了。C/C++ 语言中根本没有 '快速趋向于'。它是由 - - 符号和 > 符号拼起来的。[lex.token] 指出, 空格 (以及换行, 制表符等) 在 C++ 语言中都会被忽略, 除非它们是用于分隔标记。

> There are five kinds of tokens: identifiers, keywords, literals,19 operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, 'white space'), as described below, **are ignored except as they serve to separate tokens.** [Note: Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — end note]

因此, 循环将执行三次, 分别输出 2 1 0。

**Question 161** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

int main() {
    int n = 3;
    int i = 0;

    switch (n % 2) {
        case 0:
```

```
9           do {
10              ++i;
11              case 1: ++i;
12          } while (--n > 0);
13      }
14
15      std::cout << i;
16  }
```

**答案和解析** 程序完全正确, 输出为 5

这个程序看上去太奇怪了, 似乎无法通过编译。

然而, 这却是完全合法的, 执行方法如下所述。根据 [stmt.switch#5], switch 语句执行时, 条件值将与每个 case 标签后跟随的常量进行对比, 如果存在合适的标签, 将跳转到对应的标签位置; 如果没有合适的标签, 在包含 default 标签的情况下将跳转到 default 标签, 否则跳出 switch。

> When the switch statement is executed, its condition is evaluated and compared with each case constant. **If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label.** If no case constant matches the condition, and if there is a default label, control passes to the statement labeled by the default label. If no case matches and if there is no default then none of the statements in the switch is executed.

于是, 程序跳转到 case 1 处, i 自增到 1, 执行 while, n 减少到 2。

接下来, i 自增到 3, 执行 while, n 减少到 1。

接下来, i 自增到 5, 不再符合 while 的条件。

## 2.3 字面量的类型

**Question 115** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
```

```
3 void f(int) { std::cout << "i"; }
4 void f(double) { std::cout << "d"; }
5 void f(float) { std::cout << "f"; }
6
7 int main() {
8     f(1.0);
9 }
```

**答案和解析** 程序完全正确, 输出 d

[lex.fcon#1] 指出, 浮点字面量的类型是 double。

> The type of a floating literal is double unless explicitly specified by a suffix.

**Question 4** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2 void f(float) { std::cout << 1; }
3 void f(double) { std::cout << 2; }
4
5 int main() {
6     f(2.5);
7     f(2.5f);
8 }
```

**答案和解析** 程序完全正确, 输出为: 21

上一题我们说到, 字面量浮点数 2.5 的类型是 double, 但是留了一个例外:unless explicitly specified by a suffix.

那么, 根据[lex.fcon#1], 字面量 2.5f 的类型是 float, 因此调用第 1 个函数。

> The type of a floating literal is double unless explicitly specified by a suffix. **The suffixes f and F specify float**, the suffixes l and L specify long double

**Question 335** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <cstddef>
#include <iostream>

void f(void*) {
    std::cout << 1;
}

void f(std::nullptr_t) {
    std::cout << 2;
}

int main() {
    int* a{};

    f(a);
    f(nullptr);
    f(NULL);
}
```

**答案和解析** 程序存在实现定义行为

[support.types.nullptr#2]

> The macro NULL is an implementation-defined null pointer constant.

[conv.ptr#1]

> A null pointer constant is an integer literal with value zero or a prvalue of type std::nullptr_t.

根据 C++ 标准, NULL 可以实现定义作为 std::nullptr_t 类型的纯右值, 因此是完美匹配。这里将输出 122, 然而这是实现定义行为, 主流的编译器并没有这样实现, 可能会产生二义性问题。

**Question 277** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    std::cout << sizeof("");
}
```

**答案和解析** 程序正常运行, 输出为: 1

字符串字面量"" 的类型为 const char[1], 注意需要包含尾部的 '\0'。

事实上, sizeof(char) 是有规定的, 值为 1。expr.sizeof#1

> sizeof(char), sizeof(signed char) and sizeof(unsigned char) are 1. The result of sizeof applied to any other fundamental type is implementation-defined.

那么, 对数组施以 sizeof, 得到的答案是数组元素个数 * sizeof(元素类型), 即 1 * 1 = 1

expr.sizeof#2

> the size of an array of n elements is n times the size of an element.

## 2.4 变量的作用域和生命周期

**Question 197** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
```

```
2
3 int j = 1;
4
5 int main() {
6     int& i = j, j;
7     j = 2;
8     std::cout << i << j;
9 }
```

**答案和解析** 程序完全正确, 输出 12

[basic.scope.declarative#1]指出, 当内部作用域定义了一个与外部作用域相同的名称时, 内部作用域的范围将被排除在外部声明的范围之外 (也就是, 内部作用域中, 内部的声明有效而外部的声明无效。)

> The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region **is excluded from the scope of the declaration in the outer (containing) declarative region.**

但是, 第 6 行的 i 声明时, 内部的 j 还没有被定义, 因此 int&i = j 一句引用的 j 是外部的 j。而内部的 j 定义后, 外部声明的 j 在内部作用域就不再生效, 因此第七行则是对内部的 j 赋值为 2。

因此, 执行完成后, i = 1, j = 2。

**Question 105** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 using namespace std;
4
5 class A {
6     public:
```

```
7       A() { cout << "a"; }
8       ~A() { cout << "A"; }
9   };
10
11  int i = 1;
12
13  int main() {
14      label:
15      A a;
16      if (i--)
17      goto label;
18  }
```

**答案和解析** 程序完全正确, 输出 aAaA

采用 goto 回到 label 处, 会再次遇到 A a 的定义语句, 此时会发生什么?

根据 [stmt.jump#2], 若跳转回到一个已经初始化的具有自动存储周期 (automatic storage duration) 的变量尚未定义的时间点，将销毁该对象。

> Transfer out of a loop, out of a block, **or back past an initialized variable with automatic storage duration involves the destruction of objects with automatic storage duration that are in scope at the point transferred from but not at the point transferred to.**

根据[basic.stc.auto#1], 块内部的变量如果没有显式以 static, thread_local 或 extern 声明, 都拥有自动存储周期。在退出这个程序块的时候它们的生命周期即结束。

> **Block-scope variables not explicitly declared static, thread_local, or extern have automatic storage duration.** The storage for these entities lasts until the block in which they are created exits.

变量 a 是局部变量, 且没有被 static 等修饰, 所以拥有自动存储周期。

这里从一处已经定义了 a 的时间点跳跃到 a 尚未定义的时间点, 符合上面的定义, 因此将执行 a 的析构函数。那么, 再次执行语句 `A a` 的时候, 就会重新定义这个变量, 执行它的构造函数。

作为补充, 我们将叙述静态存储周期 (static storage duration) 的定义。

[basic.stc.static#1] 指出, 如果变量不具有动态、线程存储周期, 也不是局部变量, 就具有静态存储周期。它们直到程序结束才会销毁。[basic.stc.static#3] 告诉我们, 使用 static 语句也能定义一个拥有静态存储周期的变量。

> **All variables which do not have dynamic storage duration, do not have thread storage duration, and are not local have static storage duration.** The storage for these entities shall last for the duration of the program.
>
> The keyword 'static' can be used to declare a local variable with static storage duration.

**Question 295** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

char a[2] = "0";

struct a_string {
    a_string() { *a='1'; }
    ~a_string() { *a='0'; }
    const char* c_str() const { return a; }
};

void print(const char* s) { std::cout << s; }
a_string make_string() { return a_string{}; }

int main() {
    a_string s1 = make_string();
    print(s1.c_str());

```

```
18      const char* s2 = make_string().c_str();
19      print(s2);
20
21      print(make_string().c_str());
22  }
```

**答案和解析** 程序完全正确，输出为：101

虽然不影响本题的答案, 但依然有必要指出, `a_string s1 = make_string();` 一句, **只调用了一次构造函数, 并没有产生任何复制构造**。这被称为 copy/-move elision。 [class.copy.elision#1.1]说, 当表达式与 return 语句的返回值 (忽略 cv-qualifier) 是同一类型时, 若表达式不是 volitale 的, 则通过直接构造表达式的方法来省略复制构造。

同时, 在 `a_string s1 = make_string();` 中, 返回值 make_string() 也不会通过复制构造来初始化 s1。[dcl.init#17.6.1] 说到, 在初始化器是纯右值, 且与目标类型相同时, 将直接使用它来构造目标。

> If the initializer expression is a prvalue and the cv-unqualified version of the source type is the same class as the class of the destination, the initializer expression is used to initialize the destination object. [ Example: T x = T(T()); calls the T default constructor to initialize x. — end example ]

执行 `print(s1.c_str())` 时, a 是字符串 `"1"`。

接下来 s2 呢? 这里执行完 18 行的语句后, 临时的 a_string 对象就析构了。第二次输出将输出 0。

s3 则是在问, 临时对象确切的析构时间点。事实上, 只有执行完整个表达式 `print(make_string().c_str());`, 临时变量才会析构。

[class.temporary#4]

> Temporary objects are destroyed as the last step in evaluating the **full-expression that (lexically) contains the point** where they were created.

因此, 最后应该输出 1。

## 2.5 变量的声明和定义

**Question 197** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int _global = 1;

int main() {
    std::cout << _global;
}
```

**答案和解析** 程序存在未定义行为

[lex.name#3]指出, 以双下划线开头的标识符以及单下划线与大写字母开头的标识符应当保留作任意用途; 单下划线开头的标识符则用作全局命名空间的名称。它们不应该被使用。

> (3)In addition, some identifiers are reserved for use by C++ implementations and **shall not be used otherwise**; no diagnostic is required.
>
> (3.1)Each identifier that contains a double underscore __ or begins with an underscore followed by an uppercase letter is reserved to the implementation for any use.
>
> (3.2)Each identifier that begins with an underscore is reserved to the implementation for use as a name in the global namespace.

**Question 119** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    void * p = &p;
    std::cout << bool(p);
}
```

**答案和解析** 程序完全正确，输出为：1

[basic.scope.pdecl#1] 指出, 变量名定义的时间点在完整声明者 (declarator) 之后, 在初始化者 (initializer) 之前。

> The point of declaration for a name is immediately after its complete declarator and before its initializer (if any)

这里的完整声明者是第一个 p, 初始化者是 &p, 在两者之间完成了变量名的定义, 因此初始化者可以使用 p 并且取得它的地址。第 4 行语句结束后, p 是一个指向自己的指针, 它一定不是空指针, 因此答案是 1。

这里再给出几个练习。请回答, 在 C++17 标准下, 以下程序的答案。

```
1 #include <iostream>
2 int main()
3 {
4     const int x = 42;
5     {int x = x ; std::cout << x; }
6 }
```

A: 是 42。

B: 有可能不是 42。

C: 无法编译。

D: ub, 取决于具体实现。

```
1 #include <iostream>
2 int main()
3 {
4     const int x = 42;
5     {int x[x] ; std::cout << sizeof x; }
6 }
```

在这个问题中, 假设 sizeof(int) = 4

A: 是 168。

B: 有可能不是 168。

C: 无法编译。

D: ub, 取决于具体实现。

参考答案:

第一个程序中, x 在初始化之前已经完成了声明, 因此初始化将使用它本身 (尚未确定) 的值, 所以可能不是 42。

第二个程序中, 直到 x[x] 处声明者才完全结束, 声明前还没有隐藏外部变量 x, 因此这里数组大小 x 是外部的 x, 这里等效于声明 int x[42], 答案是 168。

**Question 198** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1
2 #include <iostream>
3
4 namespace A {
5     extern "C" { int x; }
6 };
7
8 namespace B {
9     extern "C" { int x; }
10 };
11
12 int A::x = 0;
13
14 int main() {
15     std::cout << B::x;
16     A::x=1;
17     std::cout << B::x;
18 }
```

**答案和解析** 程序会引发编译错误

使用 `extern 字符串字面量 语段` 的语法可以在 C++ 程序中链接非 C++ 代码。

在链接 C 语言代码时, 不能多次定义同一个变量, 即使它们处在不同的 namespace 中。

[dcl.link#6]

> [ Note: Only one definition for an entity with a given name with C language linkage may appear in the program (see [basic.def.odr]); **this implies that such an entity must not be defined in more than one namespace scope.** — end note ]

因此, 该程序两次定义 int x, 将引发编译错误。

为什么这里的 int x 是定义而不是声明? `extern int x;` 确实是变量的声明, 但这里的 extern 是存储类别说明符 (storage class specifier), 与上面所说的链接 extern 含义不同。根据 dcl.link#7, 直接接在 `extern` 字符串字面量后的声明定义语句将隐式地含有 extern 存储类别说明符, 并以此判断它是定义还是声明。

> A declaration **directly contained** in a linkage-specification is treated **as if it contains the extern specifier** for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class.
>
> Example:

```
extern "C" double f();
static double f();          // error
extern "C" int i;           // declaration
extern "C" {
    int i;                  // definition
}
extern "C" static void g();// error
```

正如标准中给出的例子, 这里 x 没有直接跟在 extern "C" 后面, 因此不会含有 extern 说明符, 所以按照一般的方式, 视为 x 的定义。

**Question 106** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

```

```
3 extern "C" int x;
4 extern "C" { int y; }
5
6 int main() {
7
8     std::cout << x << y;
9
10    return 0;
11 }
```

**答案和解析** 程序存在未定义行为

**extern "C" int x;** 是一次定义, 而 **extern "C" int y** 是一次声明。

根据 C++ 标准, 变量必须恰好被定义一次 (One-definition rule)。由于 y 没有被定义, 所以程序存在未定义行为。

## 2.6 变量的初始化

**Question 11** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 int a;
4
5 int main () {
6     std::cout << a;
7 }
```

**答案和解析** 程序完全正确，输出为：0

本题的关键在于未显式初始化的全局变量的初值。

[basic.start.static#2] 指出了具有静态存储周期 (static storage duration) 的变量的初始化方法。

> If constant initialization is not performed, a variable with static storage duration (6.7.1) or thread storage duration (6.7.2) is zero-initialized.

具有静态存储周期的变量若不被常量初始化 (constant initialization), 则会被零初始化 (zero initialization)。

这里, a 是一个全局变量, 根据上一节的知识, a 具有静态存储周期, 将被零初始化。零初始化的方法由 [dcl.init#6.1] 给出。

> (6) To zero-initialize an object or reference of type T means:
>
> (6.1) if T is a scalar type, the object is initialized to the value obtained by converting the integer literal 0 (zero) to T;

这里 a 是一个标量类型, 因此 a 被初始化为字面量 0。

**Question 12** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    static int a;
    std::cout << a;
}
```

**答案和解析** 程序完全正确，输出为：0

与 Question 11 相同。由于 a 具有静态存储周期且未被常量初始化, 所以被零初始化。

因此 a 被初始化为字面量 0。

**Question 221** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct S {
```

```
4       int one;
5       int two;
6       int three;
7   };
8
9   int main() {
10      S s{1,2};
11      std::cout << s.one;
12  }
```

**答案和解析** 程序完全正确，输出为：1

此处, 我们正在使用初始化列表 (initializer list) 对聚合体 (Aggregate) 进行初始化。聚合体可以是一个数组, 也可以是一个简单的类 ([dcl.init.aggr#1])。这个类不能有用户定义的或继承的构造函数, 不能有 protected 或 private 的非静态成员, 不能有虚函数, 也不能以 virtual, protected, private 继承其他类。这里 S 显然是一个聚合体, 那么, 我们就会按照定义顺序, 一个个地将元素进行初始化。

如果初始化列表中的元素个数多于聚合体中的元素个数, 程序是 ill-formed([dcl.init.aggr#7])。如果元素个数比聚合体中要少呢?

[dcl.init.aggr#8] 指出, 没有被初始化的元素将:

- 如果该元素有默认初始化器 (例如 `struct S{int x=1;}` 中 x 的默认初始化器是 1), 将使用默认初始化器初始化它。
- 若不然, 且该元素不是引用类型, 则尝试使用空列表对该元素进行 copy-initialization。
- 否则, 程序是 ill-formed。

这里, three 没有默认初始化器, 所以尝试用空列表进行 copy-intialization, three 被初始化为 int, 也就是 0。

作为练习, 请说出以下程序的运行结果。

```
1  #include <iostream>
2
3  struct S{
```

```
    struct A{
        A(int x = 1) : data(x){};
        int data;
    }a = A(2);
    struct B{
        B(int x = 3) : data(x){};
        int data;
    }b = B(4);
};

int main() {
    S s{5};
    std::cout << s.a.data << "␣" << s.b.data;
    return 0;
}
```

参考答案: 5 4, 因为 b 有默认初始化器, 所以不会选择空列表初始化。

**Question 107** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <vector>

int f() { std::cout << "f"; return 0;}
int g() { std::cout << "g"; return 0;}

void h(std::vector<int> v) {}

int main() {
    h({f(), g()});
}
```

**答案和解析** 程序完全正确，输出为：fg 这需要讨论到, 使用大括号初始化时, 各元素的评估顺序。事实上这一点已经在[dcl.init.list#4]中明确。

> Within the initializer-list of a braced-init-list, the initializer-clauses, including any that result from pack expansions, **are evaluated in the order in which they appear.**

所以, 将依次输出 f, g。

**Question 221** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct C {
    int& i;
};

int main() {
    int x = 1;
    int y = 2;

    C c{x};
    c.i = y;

    std::cout << x << y << c.i;
}
```

**答案和解析** 程序完全正确，输出为：222

C c{x} 是聚合体初始化, 这里, c.i 是 x 的引用, 因此 c.i = y 相当于说 x = y, 因此操作后, x, y, c.i 的值均为 2。

## 2.7 cv 限定符

**Question 15** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <memory>
#include <vector>

class C {
    public:
    void foo()       { std::cout << "A"; }
    void foo() const { std::cout << "B"; }
};

struct S {
    std::vector<C> v;
    std::unique_ptr<C> u;
    C *const p;

    S()
    : v(1)
    , u(new C())
    , p(u.get())
    {}
};

int main() {
    S s;
    const S &r = s;

    s.v[0].foo();
    s.u->foo();
    s.p->foo();

    r.v[0].foo();
    r.u->foo();
```

```
33      r.p->foo();
34  }
```

**答案和解析** 程序完全正确, 输出为: AAABAA

容易知道, 前两次调用都将输出 A。调用 s.p->foo() 时需要注意, 虽然 p 是 const 的, 但它所指向的对象依然是 non-const 的。即依然会输出 A。

调用 r.v[0].foo() 时, 由于 r 是一个常引用, 所以 r 的成员 r.v 是 const 的。

expr.ref#4.2

> If E2 is a non-static data member and the type of E1 is “cq1 vq1 X”, and the type of E2 is “cq2 vq2 T”, the expression designates the named member of the object designated by the first expression. If E1 is an lvalue, then E1.E2 is an lvalue; otherwise E1.E2 is an xvalue. Let the notation vq12 stand for the “union” of vq1 and vq2; that is, if vq1 or vq2 is volatile, then vq12 is volatile. Similarly, let the notation cq12 stand for the “union” of cq1 and cq2; that is, if cq1 or cq2 is const, then cq12 is const. If E2 is declared to be a mutable member, then the type of E1.E2 is “vq12 T”. If E2 is not declared to be a mutable member, **then the type of E1.E2 is “cq12 vq12 T”**.

由 sequence.reqmts#14, r.v[0] 返回的也是常引用, 因此 r.v[0]->foo() 输出 B。

> expression: a[n]
> return type: reference; const_reference for constant a

接下来两次调用时, 虽然指针本身是 const 的, 但指向的对象依然是 non-const 的, 依然输出 A。

**Question 148** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
```

```cpp
volatile int a;

int main() {
    std::cout << (a + a);
}
```

**答案和解析** 程序存在未定义行为

volatile 修饰变量, 表示该变量值可能随时改变, 要求每次访问都必须重新读值。

intro.execution#14 指出读取 volatile 左值或将亡值 (lvalue/xvalue) 会带来副作用。

intro.execution#17 中表明, 如果存在多个含有副作用的内存访问, 且没有规定它们的顺序, 则会引发未定义行为。

> If a side effect on a memory location is unsequenced relative to either another side effect on the same memory location, the behavior is undefined.

在这里, 两个 a 的评估顺序是不确定的, 且 a 的访问带有副作用, 因此程序存在未定义行为。

## 2.8 习题

**Exercise 2.1** 请说出 C++ 17 标准下, 程序的输出。

```cpp
#include <iostream>

int main() {
    int a[3] = {1};
    for(int i = 0; i < 3; i++)
    {
        std::cout << a[i];
    }
    return 0;
}
```

**Exercise 2.2** 不考虑任何对复制构造/移动构造的省略 (-fno-elide-constructors), 说出以下程序在 C++ 14 标准下的输出。

```
#include<iostream>
using std::cout;
struct A{
    int u = 0;
    A(int x = 1) {cout << 1;}
    A(const A&) {cout << 2;}
    A(A&&) {cout << 3;}
    ~A() {cout << 4;}
};
A fun() {A a; return a;}
int main()
{
    A a = fun();
    return 0;
}
```

**Exercise 2.3** 解读 `int *const** const* u = nullptr;` 中 u 的类型。

**Exercise 2.4** (Question 347, M) 指明以下程序的结果。

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

int main() {
    std::cout << std::numeric_limits<unsigned
        char>::digits;
}
```

**Exercise 2.5** (Question 152, M) 指明以下程序的结果。

```
#include <iostream>
#include <type_traits>
```

```
3
4 int main() {
5     if(std::is_signed<char>::value){
6         std::cout << std::is_same<char, signed
              char>::value;
7     }else{
8         std::cout << std::is_same<char,
              unsigned char>::value;
9     }
10 }
```

**Exercise 2.6** 指明以下程序的结果。

```
1         #include <iostream>
2         int main() {
3             std::cout << sizeof(char);
4             return 0;
5         }
```

# 3 C++ 表达式和类型转换

## 3.1 运算和运算符

**Question 26** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 int main() {
4     int i = 42;
5     int j = 1;
6     std::cout << i / --j;
7 }
```

**答案和解析** 程序存在未定义行为

[expr.mul#4] 指出, 除以 0 是未定义行为。

> If the second operand of / or % is zero the behavior is undefined.

**Question 220** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

bool f() { std::cout << 'f'; return false; }
char g() { std::cout << 'g'; return 'g'; }
char h() { std::cout << 'h'; return 'h'; }

int main() {
    char result = f() ? g() : h();
    std::cout << result;
}
```

**答案和解析** 程序完全正确, 输出为: fhh

这称为条件表达式 (conditional-expression),[expr.cond#1] 指出, 如果第一个表达式 (转换为 bool) 的值为 true, 则答案是第二个表达式, 否则是第三个表达式。第二个和第三个中, 只有一个表达式会被计算。

> Conditional expressions group right-to-left. The first expression is contextually converted to bool. It is evaluated and if it is true, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. **Only one of the second and third expressions is evaluated.**

**Question 120** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

```

```
int main() {
    int a = 10;
    int b = 20;
    int x;
    x = a, b;
    std::cout << x;
}
```

**答案和解析** 程序完全正确, 输出为: 10

逗号具有最低优先级, 所以解读为 x = a 和 b 两个表达式。

**Question 265** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

void f(char*&&) { std::cout << 1; }
void f(char*&) { std::cout << 2; }

int main() {
    char c = 'a';
    f(&c);
}
```

**答案和解析** 程序完全正确, 输出 1

虽然 c 是一个左值, 但根据[expr.unary.op#2, #3] &c 将返回一个指向 c 的指针, 且是一个纯右值。所以, 它和第一个函数完美匹配, 输出 1。

> (2) The result of each of the following unary operators is a prvalue.
> (3) The result of the unary & operator is a pointer to its operand.

**Question 286** Difficulty: Easy

**假设 int 是 32 位整数类型, 而 unsigned short 是无符号 16 位整数类型。**

According to the C++17 standard, what is the output of this program?

```cpp
#include<iostream>

int main()
{
    unsigned short x=0xFFFF;
    unsigned short y=0xFFFF;
    auto z=x*y;
    std::cout << (z > 0);
}
```

**答案和解析** 程序存在未定义行为

x * y 会导致 Integral Promotion, 所以会转化为 int 类型, 从而引发有符号整数的溢出。根据 [expr#4], 有符号整数的溢出是未定义行为。

> If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

无符号整数呢? 不过, 其实从概念上来说, 无符号整数是不可能溢出的, 因为它的所有运算都会对最大值取模。

**Question 259** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

void f(unsigned int) { std::cout << "u"; }
void f(int)          { std::cout << "i"; }
void f(char)         { std::cout << "c"; }

int main() {
```

```
8       char x = 1;
9       char y = 2;
10      f(x + y);
11  }
```

**答案和解析** 程序存在未指定行为

这里确实应当发生 Integral promotion, 从而输出 i ————如果 char 是 8 位、int 是 32 位的话。

反之, 结果可能是未指定的。

**Question 25** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  #include <limits>
3
4  int main() {
5      int i = std::numeric_limits<int>::max();
6      std::cout << ++i;
7  }
```

**答案和解析** 程序会导致未定义行为

i 是 int 类型的最大值, 则 i + 1 不能被 int 表示, 会发生有符号整数溢出, 这是未定义行为。

[expr#4]

> If during the evaluation of an expression, the result is not mathematically defined or **not in the range of representable values for its type**, the behavior is undefined.

**Question 144** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <limits>
using std::numeric_limits, std::cout;
int main()
{
    int N[] = {0,0,0};

    if ( numeric_limits<long int>::digits==63
        &&
    numeric_limits<int>::digits==31 &&
    numeric_limits<unsigned int>::digits==32 )
    {
        for (long int i = -0xffffffff; i ; --i)
        {
            N[i] = 1;
        }
    }
    else
    {
        N[1]=1;
    }

    cout << N[0] << N[1] << N[2];
}
```

**答案和解析** 程序完全正确, 输出为: 010

如果执行 else 分支, 则结果是显然的。所以在这里我们认为 long long 是 64 位有符号整数, int 是 32 位有符号整数, unsigned int 是 32 位无符号整数。

[lex.icon#2] 的表格指出, 对于不含后缀的十六进制字面量, 会在以下列表中找到第一个能够表示它的类型:{int, unsigned int, long int, unsigned long int, long long int, unsigned long long int}

0xffffffff 可以被 unsigned int 表示, 但不能被 int 表示, 所以该字面量的类型是 unsigned int。那么, 无符号数的' 相反数' 是怎么计算的呢? 其实在前面我们提到, 无符号数是不可能溢出的, 它的所有运算都是对最大值取模的。[expr.unary.op#8] 中也有提到, 无符号整数的相反数的计算方法是用 $2^n$ 减去它。

> The negative of an unsigned quantity is computed by subtracting its value from $2^n$, where n is the number of bits in the promoted operand.

因此, i 的初值为 1, 与 else 分支的效果相同。

**Question 24** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

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

int main() {
    unsigned int i = std::numeric_limits<
        unsigned int>::max();
    std::cout << ++i;
}
```

**答案和解析** 程序完全正确，输出为：0 无符号数的运算总是对最大值取模, 因此输出 0。

**Question 277** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    int I = 1, J = 1, K = 1;
    std::cout << (++I || ++J && ++K);
    std::cout << I << J << K;
```

```
}
```

**答案和解析** 程序正常运行, 输出为: 1211

首先, 分析表达式的执行顺序。&& 的优先级大于 ||, 所以被解析为 (++I || (++J && ++K))

由于 ++I 是 true, 所以右边被" 短路"。

expr.log.or#1

> || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to true.

因此, I 自增为 2, 其余变量依然是 1。

**Question 217** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    int i = 1;
    int const& a = i > 0 ? i : 1;
    i = 2;
    std::cout << i << a;
}
```

**答案和解析** 程序完全正确，输出为：21

这需要明确, a 此时是否是 i 的引用。换言之, 三元运算符的结果是左值还是右值。

事实上可以猜出, 结果是右值是合理的。因为 i 有可能转换为右值, 但 1 不可能转换为左值。一一比对[expr.cond#2 ~ 6] 中的规则, 答案确实是纯右值。

因此, a 只是引用了一个常量, 而不是引用了 i。

**Question 138** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    std::cout << +!!"";
}
```

**答案和解析** 程序完全正确, 输出为: 1

这个表达式的操作数只有一个: 字符串字面量 ""。它的类型是 const char[1]。根据 [over.built#24], ！ 运算只能用于 bool 类型。因此会先进行 array-to-pointer conversion([conv.array#1]) 转换为 const char*, 再通过 boolean conversion([conv.bool]) 转化为 bool 类型。由于它不是一个空指针, 转换的结果将是 true。经过两次运算, 答案依然保持 true。

接下来就是 operator+, 它只有一个操作数。根据[expr.unary.op#7]], 它的结果是参数值本身, 操作会引发 Integral promotion。因此表达式的结果是 int 类型的 1。

> The operand of the unary + operator shall have arithmetic, unscoped enumeration, or pointer type and the result is the value of the argument. **Integral promotion is performed on integral or enumeration operands.**

**Question 179** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    int a = 5,b = 2;
    std::cout << a+++++b;
}
```

**答案和解析** 程序会引发编译错误

事实上， `a++++++b` 有多种解析方法, 除了 `a++ + ++ b` 之外, `a++ ++ +b` 和 `a+ ++ ++ b` 都是可能的。从词法分析的角度, 我们采用最大咀嚼原理 (maximal munch principle), 每次有多个可能的 token 匹配时, 选择最长的那个。

因此, `a+++++b` 前两次都与 `++` 匹配, 即 `a++ ++ +b`, 使得右值`a++` 再次进行自增, 引发编译错误。

[lex.pptoken#3]

> the next preprocessing token is **the longest sequence of characters** that could constitute a preprocessing token, **even if that would cause further lexical analysis to fail**

## 3.2 类型转换

**Question 153** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

int main() {
    char* str = "X";
    std::cout << str;
}
```

**答案和解析** 程序会导致编译错误

[lex.string#8] 指出, 字面量'X' 的类型是 const char[1]

> A narrow string literal has type 'array of n const char'

此处, 我们期望将 const char[1] 类型转化为 char* 类型, 这里首先会发生 array-to-pointer conversion, 将 const char[1] 退化为 const char* , 此后再尝试转化为 char*。根据 [conv.qual#4]

> a prvalue of type 'pointer to cv1 T' can be converted to a prvalue of type 'pointer to cv2 T' if 'cv2 T' is more cv-qualified than 'cv1 T'.

它表明指向 cv1 T 类型的纯右值指针转换为 cv2 T 类型的纯右值指针时，这一转换只有当 cv2 T 比 cv1 T **更符合 cv 条件 (cv-qualified)** 时才能进行。

[basic.type.qualifier#4] 指出, 可以说一个类型比另一个类型更 cv-qualified。

> There is a partial ordering on cv-qualifiers, so that a type can be said to be more cv-qualified than another.
>
> no cv-qualifier < const

因此, const char* 比 char* 更 cv-qualified, 由 const char* 向 char* 的转换是不允许的。

Note1: 在 C 语言中, 程序是完全合法的。因为 C 语言中字符串字面量的类型为 char[N]。(但是修改它依然是不允许的)

Note2: 虽然大部分编译器只会报 warning, 但实际上这种行为是被禁止的。

**Question 249** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

using namespace std;

int main() {
    int a = '0';
    char const &b = a;
    cout << b;
    a++;
    cout << b;
}
```

**答案和解析** 程序完全正确, 输出为: 00

待补充。。

[dcl.type#4]

ww

**Question 190** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct A {
    A(int i) : m_i(i) {}
    operator bool() const { return m_i > 0; }
    int m_i;
};

int main() {
    A a1(1), a2(2);
    std::cout << a1 + a2 << (a1 == a2);
}
```

**答案和解析** 程序完全正确，输出为：21

类型 A 没有重载 + 运算符, 因此将调用内置的加法运算符。[over.built#10]

> For every promoted arithmetic type T, there exist candidate operator functions of the form: T operator+(T)

bool 不是能通过 promotion 得到的类型 (int, double 等), 所以不存在针对 bool 的 + 运算符。我们只能采用 int operator+(int)。

我们有办法进行以上函数转换。A 先转换为 bool, 再通过 integral promotion 转为 int。因此调用 a1 + a2 时, 实际上两个参数均为 1, 所以输出 2。a1 == a2 同理, 输出 1。

**Question 205** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    constexpr unsigned int id = 100;
    unsigned char array[] = { id % 3, id % 5 };
    std::cout
    << static_cast<unsigned int>(array[0])
    << static_cast<unsigned int>(array[1]) ;
}
```

**答案和解析** 程序完全正确，输出为：10

本题的关键点在于 `unsigned char array[] = {id % 3, id % 5}` 一句是否合法。按理说使用初始化列表对聚合体进行初始化时, 不允许发生窄化转换 (narrowing conversion)([dcl.init.aggr#3])。

然而, 对于这个例子而言, 实际上没有发生窄化转换。这需要说到它的定义。[dcl.init.list#7.4]

> from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, **except where the source is a constant expression whose value after integral promotions will fit into the target type.**

从范围更大的类型到范围更小的类型, 但是有一个例外————变换前, 类型是一个常量表达式, 其值在目标类型的可行范围内。

这里, 1 和 0 都在 unsigned char 范围内, 且 id 是一个常量, 所以程序没有问题。

**Question 236** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

```

```cpp
struct Foo {
    operator auto() {
        std::cout << "A";
        return 1;
    }
};

int main() {
    int a = Foo();
    std::cout << a;
}
```

**答案和解析** 程序完全正确，输出为：A1

operator auto() 即类型转换函数模板 (conversion function template), 它不需要指明返回值。

在执行主函数第一行时, 调用 operator auto 将 Foo 类型变量转换为 int 类型。输出 A, 同时返回 1。由于 a = 1, 第二行将输出 1。

## 3.3 cast 类型转换

**Question 179** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main() {
    const int i = 0;
    int& r = const_cast<int&>(i);
    r = 1;
    std::cout << r;
}
```

**答案和解析** 程序存在未定义行为

任何试图修改 const 变量的行为都是未定义行为。此处 r 是 i 的引用，所以 r = 1 一步在试图修改 i 的值。

[dcl.type#4]

> Except that any class member declared mutable can be modified, **any attempt to modify a const object during its lifetime** results in undefined behavior.

**Question 188** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

int main() {
    char* a = const_cast<char*>("Hello");
    a[4] = '\0';
    std::cout << a;
}
```

**答案和解析** 程序中存在未定义行为

即使我们采用 const_cast<char*> 去除了 const 属性，修改字符串字面量依然是不允许的。观察汇编可以发现，字符串字面量会保存在只读区域，允许修改它们并不是一个好主意。

[lex.string#16]

> [ Note: The effect of attempting to modify a string literal is undefined. — end note ]

## 3.4 习题

**Exercise 3.1** 假设 $i$ 是初值为 0 的 int 类型变量，在 C++ 17 标准下，解读下面的表达式。

(1) `i = i++;`

(2) `i = i++ + i;`

(3) `++i = i++ + 1;`

**Exercise 3.2** 叙述以下语句的执行过程和结果。

```
std::string s = 0;
```

# 4 C++ 函数

## 4.1 函数声明与函数调用

**Question 30** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
struct X {
    X() { std::cout << "X"; }
};

int main() { X x(); }
```

**答案和解析** 程序完全正确, 输出为空

实际上, `X x();` 是一个函数声明。就如同 `int max(int, int);` 一样自然。而它不可能是一个对象定义, 这一点在 [dcl.init#11] 中明确。

> Note: Since () is not permitted by the syntax for initializer,
> X a();
> is not the declaration of an object of class X, but the declaration of a function taking no argument and returning an X。

在更复杂的情况中, 可能产生二义性。例如 `int f(int(a))`。[dcl.ambig.res] 给出了这种二义性的判别规则。它指出, 在面对函数式转换 (function-style cast) 和定义的二义性时, 优先认为是定义。

`int(a)` 是一种函数式转换 (将 a 转换为 int 类型), 也可以认为是定义了一个 int 类型的变量 a(参考[stmt.ambig]), 因此这里认为是一次定义, 等同于 int f(int a)。所以这是一个函数声明。

**Question 31** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct X {
    X() { std::cout << "X"; }
};

struct Y {
    Y(const X &x) { std::cout << "Y"; }
    void f() { std::cout << "f"; }
};

int main() {
    Y y(X());
    y.f();
}
```

**答案和解析** 程序会引发编译错误

正如上一题所说, `Y y(X())` 是一个函数定义, y.f() 的调用是错误的。

**Question 9** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int f(int &a, int &b) {
    a = 3;
    b = 4;
    return a + b;
}

int main() {
    int a = 1;
    int b = 2;
```

```
12      int c = f(a, a);
13      std::cout << a << b << c;
14  }
```

**答案和解析** 程序完全正确, 输出为: 428

函数的形参 a, b 都是外部实参 a 的引用, 所以 a 的值被修改为 4, 返回值为 8。

**Question 250** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  #include <typeinfo>
3
4  void takes_pointer(int* pointer) {
5      if (typeid(pointer) == typeid(int[])) std::
           cout << 'a';
6      if (typeid(pointer) == typeid(int*)) std::
           cout << 'p';
7  }
8
9  void takes_array(int array[]) {
10     if (typeid(array) == typeid(int[])) std::
           cout << 'a';
11     if (typeid(array) == typeid(int*)) std::
           cout << 'p';
12 }
13
14 int main() {
15     int* pointer = nullptr;
16     int array[1];
17
18     takes_pointer(array);
19     takes_array(pointer);
```

```
20
21      std::cout << (typeid(int*) == typeid(int[])
            );
22  }
```

**答案和解析** 程序完全正确, 输出为: pp0

函数的形参实际上不可能是数组类型。

[dcl.fct#5]

> After determining the type of each parameter, any parameter of type 'array of T' or of function type T is adjusted to be 'pointer to T'

所以, 两个函数实际上是相同的, 形参的类型都是 int*。即使输入的是数组, 也会由 array-to-pointer conversion 转换为指针。因此前两次都会输出 p。

**Question 132** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  using namespace std;
3
4  int foo() {
5      cout << 1;
6      return 1;
7  }
8
9  void bar(int i = foo()) {}
10
11 int main() {
12     bar();
13     bar();
14 }
```

**答案和解析** 程序完全正确, 输出为: 11

C++ 标准并没有规定函数默认值必须是 constexpr, 也就是说, 必须在每次调用时, 都重新计算一遍参数默认值。[dcl.fct.default#9]中也有规定

> A default argument is evaluated each time the function is called with no argument for the corresponding parameter.

因此, foo 被调用 2 次。

**Question 289** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

void f(int a = []() { static int b = 1; return
    b++; }())
{
    std::cout << a;
}

int main()
{
    f();
    f();
}
```

**答案和解析** 程序完全正确，输出为：12

函数默认值将在每次调用时重新计算。那么, 唯一的疑问就是, 这两次调用是否是对同一个 lambda 表达式的调用?

[expr.prim.lambda.closure#1]

> The type of a lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type, called the closure type

因此, 这两次调用都是对相同的函数, `static int b` 也是同一个函数中的静态变量。输出为 12。

**Question 140** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
using namespace std;

size_t get_size_1(int* arr)
{
    return sizeof arr;
}

size_t get_size_2(int arr[])
{
    return sizeof arr;
}

size_t get_size_3(int (&arr)[10])
{
    return sizeof arr;
}

int main()
{
    int array[10];
    //Assume sizeof(int*) != sizeof(int[10])
    cout << (sizeof(array) == get_size_1(array)
        );
    cout << (sizeof(array) == get_size_2(array)
        );
    cout << (sizeof(array) == get_size_3(array)
        );
}
```

**答案和解析** 程序完全正确，输出为：001

在前两次调用中, array 都退化为指针。而第三次调用是对数组的引用, 因此函数内 sizeof arr 依然保持为 10。

**Question 305** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

void print(int x, int y)
{
    std::cout << x << y;
}

int main() {
    int i = 0;
    print(++i, ++i);
    return 0;
}
```

**答案和解析** 程序存在未指定行为

这一题的主要问题在于, 函数的两个参数评估的顺序。

[expr.call#5]

> The initialization of a parameter, including every associated value computation and side effect, is **indeterminately sequenced** with respect to that of any other parameter.

所以, 参数评估的顺序不确定。indeterminately sequenced 指的是要么 A 比 B 先评估, 要么 B 比 A 先评估, 这是不确定 (unspecified) 的。答案可能是 12, 也可能是 21。

有趣的是, GCC 对此输出 22, 这显然是不符合标准的。

**Question 192** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <vector>
#include <iostream>

std::vector<int> v;

int f1() {
    v.push_back(1);
    return 0;
}

int f2() {
    v.push_back(2);
    return 0;
}

void g(int, int) {}

void h() {
    g(f1(), f2());
}

int main() {
    h();
    h();
    std::cout << (v[0] == v[2]);
}
```

**答案和解析** 程序存在未指定行为

与上一题一样, f1() 和 f2() 被调用的次序是未指定的。

**Question 159** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int i;

void f(int x) {
    std::cout << x << i;
}

int main() {
    i = 3;
    f(i++);
}
```

**答案和解析** 程序完全正确, 输出为: 34

[intro.execution#18]

> every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, **is sequenced before execution of every expression or statement in the body** of the called function

所以在评估 i 的值时, 按 3 传入。但在 std::cout 时, i 的值已经变为 4。

**Question 227** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

using Func = int();

struct S {
    Func f;
};

```

```
int S::f() { return 1; }

int main() {
    S s;
    std::cout << s.f();
}
```

**答案和解析** 程序完全正确, 输出为: 1

`using Func = int();` 一句等同于 `typedef int Func();`, 可以用于函数的声明。声明的函数将不接收任何参数, 返回值类型为 int。

[dcl.fct#12]

> A typedef of function type may be used to declare a function but shall not be used to define a function. [ Example:
>
> ```
> typedef void F();
> F  fv;              // OK: equivalent to void
>    fv();
> F  fv { }           // ill-formed
> void fv() { }       // OK: definition of fv
> ```
>
> — end example ]

## 4.2 名称查找

**Question 126** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include<iostream>

int foo()
{
    return 10;
}

```

```cpp
struct foobar
{
    static int x;
    static int foo()
    {
        return 11;
    }
};

int foobar::x = foo();

int main()
{
    std::cout << foobar::x;
}
```

**答案和解析** 程序完全正确, 输出为: 11

这个问题的关键在于, foo 进行名称查找 (name lookup) 的时候, 采用全局的 foo() 还是类内的 foobar::foo()。

[basic.lookup.unqual#13] 指出, 对于类内的静态成员定义时使用的名称进行名称查找, 等同于在类的成员函数中进行名称查找。

> A name used in the definition of a static data member of class X (after the qualified-id of the static member) is looked up **as if the name was used in a member function of X.**

在进行未限定作用域的时, 将先查找类内的 foobar::foo()。找到后, 名称查找直接结束。若没有找到, 将在更外层的作用域寻找 foo。

因此, 答案为 11。

**Question 184** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

```

```cpp
struct Base {
    void f(int) { std::cout << "i"; }
};

struct Derived : Base {
    void f(double) { std::cout << "d"; }
};

int main() {
    Derived d;
    int i = 0;
    d.f(i);
}
```

**答案和解析** 程序完全正确, 输出为: d

本题有一个陷阱, 看上去 d.f(i) 与 Base 类中的 f(int) 完全匹配, 而 Derived 类中的 f(double) 需要转换。实际上, Base 类的 f(int) 根本不在考虑的范围之内。[class.member.lookup] 详细阐述了在类 C 中查找成员 f 的过程。

我们称名称查找的结果集合是 S(f, C), 它包含声明集 (declaration set) 和子对象集 (subobject set), 它的计算方法如下:

- 如果 C 中包含 f 的声明, 则声明集包含所有在 C 中声明的成员 f, 子对象集是 C 本身, 计算完成。
- 若不然, 即 C 中不包含 f 的定义, 则遍历 C 的所有直接基类 Bi, 按以上方法计算 S(f, Bi), 则 S(f, C) 是所有 S(f, Bi) 的合并。合并的具体方法可以在上述链接中找到。

因此, 在第一步计算中, S(f, C) 就找到了符合的函数 Derived::f, 不再计算子类, 声明集内仅包含 Derived::f, 故输出 d。

**Question 196** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

namespace x {
    class C {};
    void f(const C& i) {
        std::cout << "1";
    }
}

namespace y {
    void f(const x::C& i) {
        std::cout << "2";
    }
}

int main() {
    f(x::C());
}
```

**答案和解析** 程序完全正确, 输出为: 1

【待补充】

## 4.3 函数重载

**Question 118** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

void print(char const *str) { std::cout << str;
    }
void print(short num) { std::cout << num; }

```

```
6  int main() {
7      print("abc");
8      print(0);
9      print('A');
10 }
```

**答案和解析** 程序会导致编译错误

根据 [over.match.best], 可以说一个函数 F1 比另一个函数 F2 更好 ( a viable function F1 is defined to be a better function than another viable function F2), 按照以下方法判定。

- 定义 $ICS_i(F)$ 是一种转换, 它将 F **接收到的参数列表**的第 i 个变量转化为 F **定义时的第 i 个参数**的对应类型。各种转换之间有排名, [over.ics.scs#3] 给出了**标准转换**的排名规则。等级 1 是排名最高的, 等级 3 是排名最低的。
  - 等级 1 Exact Match
    - Identity Transformation
      - 无需转换
    - Lvalue Transformation
      - 左值到右值的转换
      - 数组到指针的转换
      - 函数到函数指针的转换
    - Qualification Adjustment
      - Qualification conversions, 即由 T 到 const T/ voliatile T
      - 函数指针到函数指针的转换
  - 等级 2 Promotion
    - 整型提升 (Integral promotions)
    - 浮点提升 (Floating-point promotion)
  - 等级 3 Conversion
    - 整型之间的转换

    - 浮点型之间的转换
    - 整型和浮点型之间的转换
    - 指针之间的转换 (例如 T* 转为 void*, 或转为 std::nullptr_t)
    - 指向成员之间的指针的转换 (例如 D 是 B 的父类, 由 B* 转为 D*)

  判定法 ：若满足以下条件, 则称 F1 比 F2 好。

- 对于任意的 $i$, $ICS_i(F1)$ 的排名都不比 $ICS_i(F2)$ 的排名低
- 对于某些 $i$, $ICS_i(F1)$ 的排名比 $ICS_i(F2)$ 的排名高。判定法 ：若满足以下条件, 则称 F1 比 F2 好。
- F1 和 F2 对应的转换序列是标准转换序列, 且不考虑 Lvalue Transformation(左值到右值的转换, 数组到指针的转换, 函数到函数指针的转换) 的情况下, F1 对应的转换序列是 F2 对应转换序列的真子集。

除了标准转换外, 还存在其他隐式转换。[over.ics.rank]给出了转换之间的排名规则。标准转换比用户定义的转换更好，用户定义的转换比省略号的转换更好。

> a standard conversion sequence is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and a user-defined conversion sequence is a better conversion sequence than an ellipsis conversion sequence.

对于函数重载二义性的处理, 还有许多复杂的规则, 可以在上述链接中找到更多规则和例子。

根据以上判定方法, 'abc' 是 const char* 类型, 与第一个函数完美匹配, 没有办法转化为第二个函数的参数。

对于字面量 0, 它可能是空指针, 也可能是 int 类型的 0。考察第一个函数, 解释为空指针可以转化使得参数匹配, 这一转化的等级为 5。考察第二个函数, 解释为整型的 0 可以转化为 short, 这一转化的等级为 5。因此两种转化的等级相同, 不能说一个函数比另一个函数更好。根据[over.match.best#2], 会引发二义性问题。

> If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed.

'A' 没有办法转化为第一个函数的参数, 转化为第二个函数的参数的转化等级为 5。

给出两个额外练习。在 C++17 标准下, 下列程序的结果是:

```
#include <iostream>

void print(const char *str) { std::cout << 1; }
void print(int num) { std::cout << 2; }
void print(unsigned int num) {std::cout << 3; }
int main() {
    print(0);
    print((short) 0);
    print((unsigned short) 0);
}
```

参考答案: 222

第一次调用, 转化为 int 是完美匹配, 因此选用第二个函数。后两次调用, 转化为 int 只需要 integral promotion, 等级为 2, 比第三个函数要好。

在 C++ 17 标准下, 以下程序的结果是:

```
#include <iostream>
void foo(int x, int y, const char* c)
{
    std::cout << 1;
}

void foo(int x, short y, std::string s)
{
    std::cout << 2;
}
int main() {
    short x = 1, y = 2;
```

```
13      foo(x, y, "hello");
14  }
```

参考答案: 程序会导致编译错误。

注意, 在第二个参数上, 第一个函数比第二个函数更差; 在第三个参数上, 第一个函数比第二个函数更好。所以, 没有办法分辨它们的优劣, 即使第一个函数无需用户转换。

**Question 2** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  void f(int) { std::cout << 1; }
4  void f(unsigned) { std::cout << 2; }
5
6  int main() {
7      f(-2.5);
8  }
```

**答案和解析** 程序会引发编译错误

参数的类型为 double, 向 int 和 unsigned int 转换的级别都是 Conversion, 因此具有二义性。

**Question 2** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  #include <string>
3
4  void f(const std::string &) { std::cout << 1; }
5
6  void f(const void *) { std::cout << 2; }
7
8  int main() {
```

```
9       f("foo");
10      const char *bar = "bar";
11      f(bar);
12  }
```

**答案和解析** 程序完全正确, 输出为: 22

字符串字面量的类型是 const char[n], 因此调用 `f("foo")` 只需要采用标准转换 (先进行一次 array-to-pointer conversion, 再由 const char* 转为 const void* 需要一次 pointer conversion) 而另一函数则需要一次用户定义的转换。调用 `foo(bar)` 的情况类似。

## 4.4 习题

**Exercise 4.1** 对以下程序的解读是否有误? 如果有, 请指出其中的错误。

```
1  #include<iostream>
2
3  void f(int x) { std::cout << 1; }
4
5  void f(const int x) {std::cout << 2;}
6
7  int main() {
8       int x = 10;
9       f(x);
10 }
```

**解读** 程序完全正确, 输出为: 1

第 9 行调用 f(x) 时, void f(int x) 是完美匹配, 而 void f(const int x) 需要一次 Qualification conversion, 根据[over.ics.rank#3.2.1], 如果在不考虑 Lvalue Transformation 的情况下, 标准转换序列 S1 是 S2 的真子集, 则 S1 比 S2 更好; Identity conversion 序列视为任何非 Identity conversion 序列的真子集, 因此选择第一个函数, 输出为: 1。

> Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if:
>
> **S1 is a proper subsequence of S2** (comparing the conversion sequences in the canonical form defined by [over.ics.scs], excluding any Lvalue Transformation; **the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence**)

**Exercise 4.2** 对以下程序的解读是否有误? 如果有, 请指出其中的错误。

```
#include<iostream>

void f(int& x) { std::cout << 1; }

void f(const int& x) { std::cout << 2; }

int main() {
    short x = 10;
    f(x);
}
```

**解读** 程序完全正确, 输出为: 1

这里 x 本身是一个 short 类型的左值。因为从 short 转为 int 需要一次 Integral Promotion, 而 short 转为 const int 则需要一次 Integral Promotion 和一次 Qualification conversion, 虽然两种转换序列都是 Promotion 等级的, 但第一个函数的转换序列是第二个函数的转换序列的真子集, 所以将采用第一个函数, 不存在二义性问题。

# 5 C++ 重要关键字

## 5.1 sizeof

## 5.2 auto

**Question 276** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

auto sum(int i)
{
    if (i == 1)
        return i;
    else
        return sum(i-1)+i;
}

int main()
{
    std::cout << sum(2);
}
```

**答案和解析** 程序完全正确，输出为：3

从 C++14 开始, auto 可以推导函数的返回值类型。[dcl.spec.auto#8] 指出, 我们将对每一个 return 表达式进行类型推导, 如果每次的推导结果不同, 程序将是 ill-formed。

在 `return i;` 一句时, 我们能推导出返回值类型为 int, 但是 `return sum(i-1) + i;` 一句呢? 我们需要用 `sum(i-1) + i` 的类型来推导 sum(i) 的返回值类型吗?

[dcl.spec.auto#10] 表明, 如果在类型推导时, 其类型需要通过某未被推导的表达式的类型来确定, 则程序是 ill-formed。**但是, 如果已经能从之前的 return 表达式中推导出函数的返回值, 则它将可以应用于函数的其余部分。** 那么, sum(i-1) 的类型将被暂定为 int, 因此 sum(i-1) + i 的类型依然是 int。两个 return 语句的结果一致, 所以函数的返回值类型是 int。

> If the type of an entity with an undeduced placeholder type is needed to determine the type of an expression, the program is ill-formed. **Once a non-discarded return statement has been seen in a function, however, the return type deduced from that statement can be used in the rest of the function, including in other return statements.**

作为补充, 如果将函数改为以下形式, 能否运行?

```
auto sum(int i)
{
    if (i != 1)
        return sum(i-1)+i;
    else
        return 1;
}
```

另一种修改如下。

```
auto sum(short i)
{
    if (i == 1)
        return (short)1;
    else
        return sum(i) + i;
}
```

参考答案: 均不能。

**Question 163** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

class A {
    int foo = 0;

```

```
6       public:
7       int& getFoo() { return foo; }
8       void printFoo() { std::cout << foo; }
9   };
10
11  int main() {
12      A a;
13
14      auto bar = a.getFoo();
15      ++bar;
16
17      a.printFoo();
18  }
```

**答案和解析** 程序完全正确，输出为：0

这题的关键在于, auto 推导的类型是 int 还是 int& ?

【待补充】

## 5.3 typeid

## 5.4 typedef

## 5.5 decltype

**Question 37** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  int main() {
4      int a = 0;
5      decltype(a) b = a;
6      b++;
7      std::cout << a << b;
8  }
```

**答案和解析** 程序完全正确，输出为：01

[dcl.type.simple#4] 给出了 e 是表达式时, decltype(e) 的推导方法。简单来说, 设 T 是 e 的类型, 如果 e 是**不加括号的 id-expression**(一般可以理解为标识符), 将直接返回 T。

> otherwise, if e is an unparenthesized id-expression or an unparenthesized class member access, decltype(e) is the type of the entity named by e.

这里 a 是 id-expression, 所以, 这里 decltype(a) 是它的类型, 即 int。

**Question 38** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

int main() {
    int a = 0;
    decltype((a)) b = a;
    b++;
    std::cout << a << b;
}
```

**答案和解析** 程序完全正确，输出为：11

[dcl.type.simple#4]

> otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;

根据上述规则, 这里是加括号的 id 表达式, 且 a 是一个左值, 所以 decltype((a)) 是 `int&`

**Question 337** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <type_traits>

int main() {
    auto a = "Hello, World!";
    std::cout << std::is_same_v<decltype("Hello, World!"), decltype(a)>;

    return 0;
}
```

**答案和解析** 程序完全正确，输出为：0

”Hello, World” 的类型为 const char[14]。

【待补充】[dcl.type.simple#4]

> otherwise, if e is an lvalue, decltype(e) is T&, where T is the type of e;

所以 decltype(”Hello World!”) 得到的结果是 const char(&)[14]

# 6 C++ 类和对象

## 6.1 构造函数和初始化

**Question 28** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct A {
    A() { std::cout << "A"; }
    A(const A &a) { std::cout << "B"; }
```

```
    virtual void f() { std::cout << "C"; }
};

int main() {
    A a[2];
    for (auto x : a) {
        x.f();
    }
}
```

**答案和解析** 程序完全正确, 输出 AABCBC

当定义数组时, 调用 A 类的默认构造函数构造这些对象。

接下来采用范围 for 循环进行枚举, auto 将被推导为 A 类型。【待补充】

**Question 32** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

struct X {
    X() { std::cout << "a"; }
    X(const X &x) { std::cout << "b"; }
    const X &operator=(const X &x) {
        std::cout << "c";
        return *this;
    }
};

int main() {
    X x;
    X y(x);
    X z = y;
    z = x;
```

```
17 }
```

**答案和解析** 程序完全正确, 输出为: abbc

【待补充】

**Question 264** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 struct C {
4     C() = default;
5     int i;
6 };
7
8 int main() {
9     const C c;
10    std::cout << c.i;
11 }
```

**答案和解析** 程序会导致编译错误

【待补充】

**Question 187** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 struct C {
4     C() { std::cout << "1"; }
5     C(const C& other) { std::cout << "2"; }
6     C& operator=(const C& other) {
7         std::cout << "3"; return *this;
8     }
```

```
};

int main() {
    C c1;
    C c2 = c1;
}
```

**答案和解析** 程序完全正确, 输出为: 12

显然, 第一次调用将输出 1。而在执行 `c2 = c1` 时, 由于对象 c2 尚未初始化, 所以显然是需要使用 (复制) 构造函数进行初始化, 而不是采用 operator=。

[class.expl.init#1]]

> [ Note: Overloading of the assignment operator has no effect on initialization. — end note ]

**Question 226** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <utility>

struct X {
    X() { std::cout << "1"; }
    X(X &) { std::cout << "2"; }
    X(const X &) { std::cout << "3"; }
    X(X &&) { std::cout << "4"; }
    ~X() { std::cout << "5"; }
};

struct Y {
    mutable X x;
    Y() = default;
    Y(const Y &) = default;
```

```
16 };
17
18 int main() {
19     Y y1;
20     Y y2 = std::move(y1);
21 }
```

**答案和解析** 程序完全正确，输出为：1255

主函数的第一行定义了 Y 类型的变量 y1，因此调用 Y 的构造函数。

类型 Y 拥有一个 X 类型的成员 x，因此调用 x 的默认构造函数，输出 1。

第二行中，std::move 将 y1 变换为右值，但是 Y 并没有移动构造函数，只能退而求其次，使用复制构造函数。因此, 其成员 x 也将使用 X 类的复制构造函数进行构造。

接下来的问题就是，我们将采用 X 类的哪一个构造函数。此时，x 本应该是 const X& 类型的，但有 mutable 修饰，根据 dcl.stc#9

> The mutable specifier on a class data member nullifies a const specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is const.

因此, x 并不是 const 的, 在两个复制构造函数中，选择第六行的 X(X &) 较为合适, 故输出 2。

在 main 函数结束后, y1 和 y2 执行析构函数。输出两次 5。

**Question 281** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 class C
4 {
5     public:
6     C(){}
```

```
7       C(const C&){} //User-declared, disables
            move constructor
8   };
9
10  int main()
11  {
12      C c;
13      C c2(std::move(c));
14      std::cout << "ok";
15  }
```

**答案和解析** 程序完全正确，输出为：ok

【待补充】

## 6.2 构造函数、析构函数的调用顺序

**Question 5** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1   #include <iostream>
2
3   struct A {
4       A() { std::cout << "A"; }
5   };
6   struct B {
7       B() { std::cout << "B"; }
8   };
9
10  class C {
11      public:
12      C() : a(), b() {}
13
14      private:
15      B b;
```

```
16      A a;
17  };
18
19  int main()
20  {
21      C();
22  }
```

**答案和解析** 程序完全正确，输出为：BA

[class.base.init#13.3] 指出, 成员的初始化顺序按照声明顺序。

> (13) In a non-delegating constructor, initialization proceeds in the following order:
>
> (13.3) Then, non-static data members are initialized in the order they were declared in the class definition (again regardless of the order of the mem-initializers).

b 优先声明, 因此较先调用构造函数。

**Question 283** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  class show_id
4  {
5      public:
6      ~show_id() { std::cout << id; }
7      int id;
8  };
9
10 int main()
11 {
12     delete[] new show_id[3]{ {0}, {1}, {2} };
```

```
13 }
```

**答案和解析** 程序完全正确, 输出为: 210

在主函数中, 使用初始化列表对聚合体进行初始化。在前面的章节我们已经知道, 我们会采用从左到右的顺序进行评估和初始化。问题在于, delete 将采用怎样的顺序进行析构? [expr.delete#6] 指出, delete 将会以与构造时相反的顺序进行析构。

> If the value of the operand of the delete-expression is not a null pointer value, **the delete-expression will invoke the destructor** (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, **in reverse order of the completion of their constructor**;

**Question 13** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

class A {
    public:
    A() { std::cout << "a"; }
    ~A() { std::cout << "A"; }
};

class B {
    public:
    B() { std::cout << "b"; }
    ~B() { std::cout << "B"; }
};

class C {
    public:
    C() { std::cout << "c"; }
```

```
18      ~C() { std::cout << "C"; }
19  };
20
21  A a;
22  int main() {
23      C c;
24      B b;
25  }
```

**答案和解析** 程序完全正确，输出为：acbBCA

全局变量 a 将在 b 和 c 之前进行动态初始化。在 main 函数结束后, c 和 b 按相反顺序析构, 然后 a 析构。

[basic.start.term#1]

> Destructors for initialized objects (that is, objects whose lifetime has begun) with **static storage duration**, and functions registered with std::atexit, **are called as part of a call to std::exit.** The call to std::exit is sequenced before the invocations of the destructors and the registered functions. [ Note: **Returning from main invokes std::exit** ([basic.start.main]). — end note ]

**Question 16** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  class A {
4      public:
5      A() { std::cout << 'a'; }
6      ~A() { std::cout << 'A'; }
7  };
8
9  class B {
10     public:
```

```
11      B() { std::cout << 'b'; }
12      ~B() { std::cout << 'B'; }
13      A a;
14  };
15
16  int main() { B b; }
```

**答案和解析** 程序完全正确, 输出 abBA

按照 [class.base.init#13] 中的规则, 构造完所有成员后, 才会调用构造函数。析构函数的调用顺序与构造函数的调用顺序相反。

**Question 145** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  struct E
4  {
5      E() { std::cout << "1"; }
6      E(const E&) { std::cout << "2"; }
7      ~E() { std::cout << "3"; }
8  };
9
10 E f()
11 {
12     return E();
13 }
14
15 int main()
16 {
17     f();
18 }
```

**答案和解析** 程序完全正确, 输出 13

这一篇解析尚待完善。

**Question 14** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

class A {
    public:
    A() { std::cout << "a"; }
    ~A() { std::cout << "A"; }
};

class B {
    public:
    B() { std::cout << "b"; }
    ~B() { std::cout << "B"; }
};

class C {
    public:
    C() { std::cout << "c"; }
    ~C() { std::cout << "C"; }
};

A a;

void foo() { static C c; }
int main() {
    B b;
    foo();
}
```

**答案和解析** 程序完全正确，输出为：abcBCA

【待补充】

**Question 17** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

class A {
    public:
    A() { std::cout << 'a'; }
    ~A() { std::cout << 'A'; }
};

class B : public A {
    public:
    B() { std::cout << 'b'; }
    ~B() { std::cout << 'B'; }
};

int main() { B b; }
```

**答案和解析** 程序完全正确, 输出为: abBA

【待补充】

## 6.3 类的继承和多态

‘

**Question 15** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

class A {};
```

```
4
5  class B {
6      public:
7      int x = 0;
8  };
9
10 class C : public A, B {};
11
12 struct D : private A, B {};
13
14
15 int main()
16 {
17     C c;
18     c.x = 3;
19
20     D d;
21     d.x = 3;
22
23     std::cout << c.x << d.x;
24 }
```

**答案和解析** 程序将引发编译错误

我们知道 C 以 public 方式继承 A, 但是以怎样的方式继承 B 呢? [class.access.base#2]

> n the absence of an access-specifier for a base class, **public** is assumed when the derived class is defined with the class-key **struct** and **private** is assumed when the class is defined with the class-key **class**

因此, C 将以 private 方式继承 B, 因此在类外不能访问 c.x。相对地, d.x 则不会出现问题。

**Question 29** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct A {
    A() { foo(); }
    virtual ~A() { foo(); }
    virtual void foo() { std::cout << "1"; }
    void bar() { foo(); }
};

struct B : public A {
    virtual void foo() { std::cout << "2"; }
};

int main() {
    B b;
    b.bar();
}
```

**答案和解析** 程序完全正确，输出为：121

虽然 A::foo 确实是虚函数, 但在 A 构造和析构的过程, 依然不会考虑它。我们可以不太严格地去理解这一点————在 A 构造时, B 还没有开始构造, 也就不存在 B::foo 这个函数。在析构时, 先要调用 B 的析构函数, 这可能导致 B::foo 的结果与预期不符, 所以 A 的析构函数去调用它也是不合适的。

这一点也在[class.cdtor#3] 中明确, 虚函数在构造和析构期间可以被调用, 但是在构造和析构期间调用的虚函数就是当前类的函数, 而不能在派生类中重写它。

> Member functions, including virtual functions, can be called during construction or destruction ([class.base.init]). When a virtual function is called directly or indirectly from a constructor or from a destructor, including during the construction or destruction of the class's non-static

> data members, and the object to which the call applies is the object (call it x) under construction or destruction, **the function called is the final overrider in the constructor's or destructor's class and not one overriding it in a more-derived class.**

**Question 7** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

class A {
    public:
    void f() { std::cout << "A"; }
};

class B : public A {
    public:
    void f() { std::cout << "B"; }
};

void g(A &a) { a.f(); }

int main() {
    B b;
    g(b);
}
```

**答案和解析** 程序完全正确，输出为：A

[class.virtual#9]

> Note: The interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), **whereas the interpretation of a call of a non-virtual member function**

> **depends only on the type of the pointer or reference denoting that object**

简而言之, 虚函数看实际类型, 普通函数看对象 (或指针/引用) 的类型。

**Question 8** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

class A {
    public:
    virtual void f() { std::cout << "A"; }
};

class B : public A {
    public:
    void f() { std::cout << "B"; }
};

void g(A a) { a.f(); }

int main() {
    B b;
    g(b);
}
```

**答案和解析** 程序完全正确，输出为：A

这里 a 只是 A 类的普通对象, 所以会调用 a::f()。

**Question 18** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

```

```cpp
class A {
    public:
    virtual void f() { std::cout << "A"; }
};

class B : public A {
    private:
    void f() { std::cout << "B"; }
};

void g(A &a) { a.f(); }

int main() {
    B b;
    g(b);
}
```

**答案和解析** 程序完全正确，输出为：B

即使 B::f() 是私有函数, 也可以被访问。[class.access.virt#1]指出, 虚函数的访问控制仅由定义决定, 不受后续重载的影响。

> The access rules (Clause [class.access]) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it.

**Question 33** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct GeneralException {
    virtual void print() { std::cout << "G"; }
};

```

```
7  struct SpecialException : public
       GeneralException {
8      void print() override { std::cout << "S"; }
9  };
10
11 void f() { throw SpecialException(); }
12
13 int main() {
14     try {
15         f();
16     }
17     catch (GeneralException e) {
18         e.print();
19     }
20 }
```

**答案和解析** 程序完全正确，输出为：G

由于是值传递, 所以 e 的类型是 GeneralException, 因此输出 G。

如果采用引用传递, 则 e 的动态类型实则是 SpecialException, 此时将输出 S。

**Question 27** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2
3  struct A {
4      virtual std::ostream &put(std::ostream &o)
           const {
5          return o << 'A';
6      }
7  };
8
```

```
9  struct B : A {
10     virtual std::ostream &put(std::ostream &o)
           const {
11          return o << 'B';
12     }
13 };
14
15 std::ostream &operator<<(std::ostream &o, const
       A &a) {
16     return a.put(o);
17 }
18
19 int main() {
20     B b;
21     std::cout << b;
22 }
```

**答案和解析** 程序完全正确，输出为：B

由于函数中 a 的动态类型是 B, 将执行 B::put。

这是一种使得 operator << 也能展现出” 多态” 的方法。

**Question 44** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  struct X {
3      virtual void f() const { std::cout << "X";
           }
4  };
5
6  struct Y : public X {
7      void f() const { std::cout << "Y"; }
8  };
```

```
9
10 void print(const X &x) { x.f(); }
11
12 int main() {
13     X arr[1];
14     Y y1;
15     arr[0] = y1;
16     print(y1);
17     print(arr[0]);
18 }
```

**答案和解析** 程序会引发编译错误

y1 的类型为 Y, print 函数在引用时, 它拥有静态类型 X 和动态类型 Y, 因此调用 f() 将输出 Y。

arr[0] 本身的类型就是 X, 在 arr[0] = y1 的时候就已经进行了类型的转换, 因此 print 函数调用时输出 X。

**Question 160** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 struct A {
4     virtual void foo (int a = 1) {
5         std::cout << "A" << a;
6     }
7 };
8
9 struct B : A {
10     virtual void foo (int a = 2) {
11         std::cout << "B" << a;
12     }
13 };
```

```
14
15 int main () {
16     A *b = new B;
17     b->foo();
18 }
```

**答案和解析** 程序完全正确，输出为：B1

b 具有静态类型 A, 动态类型 B。由于 A::foo() 是虚函数, `b->foo();` 将调用 B::foo() 这个函数。

然而, 这一函数将采用 A::foo() 的默认值。

[dcl.fct.default#10]

> A virtual function call uses the default arguments in the declaration of the virtual function **determined by the static type** of the pointer or reference denoting the object.

所以, 输出的结果是 B1。

## 6.4 友元函数

**Question 52** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 class A;
4
5 class B {
6     public:
7     B() { std::cout << "B"; }
8     friend B A::createB();
9 };
10
11 class A {
12     public:
```

```
13      A() { std::cout << "A"; }
14
15      B createB() { return B(); }
16  };
17
18  int main() {
19      A a;
20      B b = a.createB();
21  }
```

**答案和解析** 程序将引发编译错误

在将 A::createB() 声明为友元函数时, 该函数本身还未声明。

## 6.5 习题

**Exercise 6.1** 如何修改 Question 52 的代码, 使得程序正常运行?

```
1       \begin{lstlisting}[language=C++]
2   #include <iostream>
3
4   class A;
5
6   class B {
7       public:
8       B() { std::cout << "B"; }
9       friend B A::createB();
10  };
11
12  class A {
13      public:
14      A() { std::cout << "A"; }
15
16      B createB() { return B(); }
17  };
```

```
18
19 int main() {
20     A a;
21     B b = a.createB();
22 }
```

# 7 C++ 模板

## 7.1 模板参数推导

**Question 184** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2
3 template <typename T> void f() {
4     static int stat = 0;
5     std::cout << stat++;
6 }
7
8 int main() {
9     f<int>();
10    f<int>();
11    f<const int>();
12 }
```

**答案和解析** 程序完全正确, 输出为: 010

本题的关键在于, f<const int>() 和 f<int>() 是不是同一个函数?

[temp.deduct#4]

> [Note: f<int>(1) and f<const int>(1) call distinct functions even though both of the functions called have the same function type. — end note]

**Question 125** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

using namespace std;

template <class T> void f(T) {
    static int i = 0;
    cout << ++i;
}

int main() {
    f(1);
    f(1.0);
    f(1);
}
```

**答案和解析** 程序完全正确，输出为：112

[temp.fct.spec#2]

> Each function template specialization instantiated from a template **has its own copy of any static variable.**

**Question 347** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <type_traits>

template <typename T>
void foo(T& x)
{
    std::cout << std::is_same_v<const int, T>;
}
```

```
9
10 template <typename T>
11 void bar(const T& x)
12 {
13     std::cout << std::is_same_v<const int, T>;
14 }
15
16 int main()
17 {
18     const int i{};
19     int j{};
20
21     foo(i);
22     foo(j);
23     bar(i);
24     bar(j);
25
26 }
```

**答案和解析** 程序正常运行, 输出为: 1000

在进行函数模板的参数推导时, 可以参照 temp.deduct.call#3 中的规则。

> If P is a cv-qualified type, the top-level cv-qualifiers of P's type are ignored for type deduction. If P is a reference type, the type referred to by P is used for type deduction.

即, 进行模板参数推导时, 不考虑函数形参的顶级 cv 限定符及引用。

foo 函数的参数类型可表为 T, 而 i 的类型是 const int, 因此 foo(i) 中, T 被推导为 const int, 输出 1。

j 的类型是 int, 因此 foo(j) 中 T = int, 输出 0。

bar 函数的参数类型可表为 const T, 而 i 的类型是 const int, 因此 bar(i) 中, T 被推导为 int, 输出 0。

j 的类型是 int, 无法与 const T 匹配。但此时根据 temp.deduct.call#4 中的规则, 模板匹配一般要求类型完全一致, 但有例外。

> If the original P is a reference type, the deduced A (i.e., the type referred to by the reference) can be more cv-qualified than the transformed A.

即, 若形参是引用类型, 则允许通过推导, 使得实参类型转化为更加符合 cv 条件 (cv-qualified) 的类型。这里将 T 推导为 int, 则实参类型 const int 比原先类型 int 更加符合 cv 条件, 这一推导是允许的。

因此, T 被推导为 int, 输出 0。

## 7.2 模板的定义和初始化

**Question 243** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

template <typename T>
struct A {
    static_assert(T::value);
};

struct B {
    static constexpr bool value = false;
};

int main() {
    A<B>* a;
    std::cout << 1;
}
```

**答案和解析** 程序完全正确, 输出为: 1

待完善。

**Question 1** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

template <class T>
void f(T &i) { std::cout << 1; }

template <>
void f(const int &i) { std::cout << 2; }

int main() {
    int i = 42;
    f(i);
}
```

**答案和解析** 程序完全正确, 输出为: 1

f(const int&) 是对 f(T&) 的模板特化。但是在模板类型推导中, T 被推导为 int, 而特化是针对 T = const int 的, 因此不选择模板的特化, 输出 1。

**Question 113** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

using namespace std;

template<typename T>
void f(T) {
    cout << 1;
}

template<>
void f(int) {
```

```
12      cout << 2;
13  }
14
15  void f(int) {
16      cout << 3;
17  }
18
19  int main() {
20      f(0.0);
21      f(0);
22      f<>(0);
23  }
```

**答案和解析** 程序完全正确，输出为：132

此处有三个不同的 f 函数。第一个是函数模板, 第二个是模板的特化, 第三个是普通的函数。

第一次调用时, 对于函数模板, T 被推导为 double, 因此不适用模板特化。而函数模板 f(double) 是一次完美匹配 (Exact Match), 因此选用第一个函数。第二次调用时, 对于函数模板, T 被推导为 int, 使用模板特化。此时有特化的 f<>(int) 和普通函数 f(int) 进入重载决议。两者都是完美匹配级别, 但根据 [over.match.best#1.6], 我们优先选择非模板特化的函数。

第三次调用中, [temp.arg.explicit#4] 告诉我们, 在使用 f<> 进行调用时, 将优先使用模板特化的函数。

> Note: An empty template argument list can be used to indicate that a given use refers to a specialization of a function template even when a non-template function ([dcl.fct]) is visible that would otherwise be used.

**Question 338** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
1  #include <type_traits>
2  #include <iostream>
3  #include <string>
```

```
4
5  template<typename T>
6  int f()
7  {
8      if constexpr (std::is_same_v<T, int>) {
           return 0; }
9      else { return std::string{}; }
10 }
11
12 int main()
13 {
14     std::cout << f<int>();
15 }
```

**答案和解析** 程序存在未定义行为

**Question 250** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
1  #include<iostream>
2
3  template<typename T>
4  void foo(T...) {std::cout << 'A';}
5
6  template<typename... T>
7  void foo(T...) {std::cout << 'B';}
8
9  int main(){
10     foo(1);
11     foo(1,2);
12 }
```

**答案和解析** 程序完全正确, 输出为: AB

【待补充】

**Question 251** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

template<class T>
void f(T) { std::cout << 1; }

template<>
void f<>(int*) { std::cout << 2; }

template<class T>
void f(T*) { std::cout << 3; }

int main() {
    int *p = nullptr;
    f( p );
}
```

**答案和解析** 程序完全正确，输出为：bcad

【待补充】

## 7.3 模板与名称查找

**Question 162** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

void f()
{
    std::cout << "1";
}
```

```
7
8  template<typename T>
9  struct B
10 {
11     void f()
12     {
13         std::cout << "2";
14     }
15 };
16
17 template<typename T>
18 struct D : B<T>
19 {
20     void g()
21     {
22         f();
23     }
24 };
25
26 int main()
27 {
28     D<int> d;
29     d.g();
30 }
```

**答案和解析** 程序完全正确, 输出为: 1

本题的关键在于, D<int> 类型的成员函数中调用 f(), 执行父类的 f() 还是全局函数 f()。

按照一般的名称查找流程, B<T>::f() 被优先查找到。然而在模板的情况下有所不同, 根据 [temp.dep#3], 在执行非限定名称查找时, **如果基类的确定依赖于模板参数**, 则不考虑该基类作用域。

> In the definition of a class or class template, the scope of a dependent base class is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member.

应用到本题, D 类的基类的确定依赖于传入的参数 T, 因此在非限定名称查找时不考虑。

标准中给出了另一例子, 作为补充。

```
struct A {
    struct B { /* ... */ };
    int a;
    int Y;
};

int a;

template<class T> struct Y : T {
    struct B { /* ... */ };
    B b;                            // The B
        defined in Y
    void f(int i) { a = i; }        //  : : a
    Y* p;                           // Y<T>
};

Y<A> ya;
```

在这一例子中, Y 的基类依赖于传入的模板参数 T, 因此不考虑 A 的作用域, A::B, A::a, A::Y 对 Y<A> 中的成员没有影响。

当然, 如果采用限定名称查找, 定义 T::B b; 则会考虑 A::B。

# 8 C++ 容器

## 8.1 string

**Question 287** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

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

int main() {
    using namespace std::string_literals;
    std::string s1("hello world",5);
    std::string s2("hello world"s,5);

    std::cout << s1 << s2;
}
```

**答案和解析** 程序正常运行, 输出为: hello world

查询 std::string 的构造函数。第一句话传入"const char(&)[12]", 对应的构造函数为:

string.cons#9

> basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
>
> Requires: s points to an array of at least n elements of charT.
>
> Effects: Constructs an object of class basic_string and determines its initial string value from the array of charT of length n whose first element is designated by s, as indicated in Table 58.

即, s1 是"hello world" 的前 5 个字符, 即 hello

"hello world"s 是 basic_string 类型的字符串字面量。

basic.string.literals#1

> string operator""s(const char* str, size_t len);
>
> Returns: stringstr, len.

因此, s2 调用另一个 string 的构造函数:

string.cons#2

> basic_string(const basic_string& str, size_type pos, const Allocator& a = Allocator());
>
> Effects: Constructs an object of class basic_string and determines the effective length rlen of the initial string value as str.size() - pos, as indicated in Table 57.

因此, 这里的 5 表示 pos, 即从下标 5 开始的字符串, 即" world"。

最终, 输出"hello world"。

**Question 284** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

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

auto main() -> int {
    std::string out{"Hello world"};
    std::cout << (out[out.size()] == '\0');
}
```

**答案和解析** 程序完全正确, 输出为: 1

在做这道题的时候会产生两个疑问, std::string 是否和 char[] 一样, 以'0' 标识结尾呢? 即使是, 访问 out[out.size()] 是否合法呢?

[string.access]

> const_reference operator[](size_type pos) const; reference operator[](size_type pos);
>
> Requires: pos <= size().
>
> Returns: *(begin() + pos) if pos < size(). Otherwise, returns a reference to an object of type charT with value charT()

这表明, 允许使用 [] 访问字符串的末尾的后一个位置。charT 指的是字符串的字符类型, 这里即 char。那么, char() 表示一个被值初始化的变量, 它的值是 0 (由于 char 是基本类型, 因此转为零初始化, 而 char 是又标量类型, 因此是字面量 0)。

C++ 标准中实际上并没有非常明确地指出这一点, 但一般认为'\0' 就指代空字符 (null character), [lex.charset#3]表明空字符的值为 0。因此, 这里的判断是正确的, 输出 1。

## 8.2 initializer_list

**Question 235** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <initializer_list>
#include <iostream>

class C {
    public:
    C() = default;
    C(const C&) { std::cout << 1; }
};

void f(std::initializer_list<C> i) {}

int main() {
    C c;
    std::initializer_list<C> i{c};
    f(i);
    f(i);
}
```

**答案和解析** 程序完全正确，输出为：1

[dcl.init.list#5] 提到, 当使用初始化列表构造 std::initializer_list 时, 每个元素都将采用复制初始化 (copy-initialization) **构造一个临时数组对象**, 而 std::initializer_list 则 **引用这个数组**。因此这一过程调用一次复制构造函数。

> An object of type std::initializer_list<E> is constructed from an initializer list as if the implementation generated and materialized a prvalue

> of type 'array of N const E', where N is the number of elements in the initializer list. Each element of that array is **copy-initialized with the corresponding element** of the initializer list, and the std::initializer_list<E> object is constructed to **refer to that array**.

std::initializer_list 只是一个数组的引用, 所以调用 f 的过程并不会发生数组元素的复制。

## 8.3 vector

**Question 35** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v1(1, 2);
    std::vector<int> v2{ 1, 2 };
    std::cout << v1.size() << v2.size();
}
```

**答案和解析** 程序完全正确, 输出 12

这涉及到 std::vector 的构造函数。

[vector.cons#6]

> vector(size_type n, const T& value, const Allocator& = Allocator());
>
> Effects: Constructs a vector with n copies of value, using the specified allocator.

因此 v1 中包含一个 int 类型元素 2。

[over.match.list] 指明, 使用初始化列表构造非聚合体 X 时, 如果 X 含有以 std::initializer_list 为参数的构造函数, 则优先调用它。这里 std::vector 是 sequence container, 可以用 std::initializer_list 构造 (见[vector.overview#2]中的构造函数列表), 因此构造完成后 v2 包含两个元素, 分别是 1 和 2。

## 8.4 tuple

**Question 278** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <tuple>

int main()
{
    const auto t = std::make_tuple(42, 3.14,
        1337);
    std::cout << std::get<int>(t);
}
```

**答案和解析** 程序会导致编译错误

[tuple.elem#5] 给出了语法 `get<T>(x)` 的使用条件。

> Requires: The type T occurs exactly once in Types.... Otherwise, the program is ill-formed.

## 8.5 map

**Question 208** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <map>
using namespace std;

bool default_constructed = false;
bool constructed = false;
bool assigned = false;

class C {
    public:
```

```
11      C() { default_constructed = true; }
12      C(int) { constructed = true; }
13      C& operator=(const C&) { assigned = true;
            return *this;}
14  };
15
16  int main() {
17      map<int, C> m;
18      m[7] = C(1);
19
20      cout << default_constructed << constructed
            << assigned;
21  }
```

**答案和解析** 程序完全正确, 输出为: 111

关于这个问题, 我们需要先了解 map 的 [ ] 运算。[map.access] 指出, map 的 [ ] 运算符将执行以下操作。

> T& operator[](const key_type& x);
> Effects: Equivalent to: return try_emplace(x).first->second;
> T& operator[](key_type&& x);
> **Effects: Equivalent to: return try_emplace(move(x)).first->second;**

这里的 7 是右值, 将优先匹配第二个函数。

try_emplace 的作用由 [map.modifiers#8] 给出。如果已经包含待插入的 k, 则无效。否则相当于插入一个类型为 value_type 的元素, 即一个 key_type, mapped_type 的 pair。

> template <class ··· Args>
> pair<iterator, bool> try_emplace(key_type&& k, Args&& ··· args);
> Effects: If the map already contains an element whose key is equivalent to k, there is no effect. **Otherwise inserts**

> **an object of type value_type constructed with piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...)**.

[map.overview#2]

> For a map<Key,T> the key_type is Key and the value_type is pair<const Key,T>.

该元素按照以下方法构造: piecewise_construct, forward_as_tuple(std::move(k)), forward_as_tuple(std::forward<Args>(args)...). piecewise_construct 是 pair 类型的一个构造函数 ([pairs.pair#14]), 其参数是一个空类 piecewise_construct_t 和两个 tuple, 然后用两个 tuple 中的参数构造这两个对象。由于这里我们只传入了一个参数, 所以第二个元素将是空的, 即会调用默认构造函数 C()。

mp[7] 的返回值是 try_emplace(move(x)).first → second;。其中 try_emplace 返回值的 first 表示对应的 pair(=<const Key, T>) 的迭代器。这个 pair 对象的 second 已经被构造完成, 因此 mp[7] = C(1) 不是一次初始化, 它将调用 operator=, 而 C(1) 则调用构造函数 C(int x)。

根据以上, 三个函数都将被调用 (顺序是 2 1 3)。

如果希望避免 operator= 和 C() 的构造, 我们可以采用以下方法:

```
1 m.try_emplace(7, C(1));
```

这样, 只会调用一次 C(1)。

**Question 135** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int main()
6 {
7     map<bool,int> mb = {{1,2},{3,4},{5,0}};
8     cout << mb.size();
9     map<int,int> mi = {{1,2},{3,4},{5,0}};
```

```
10      cout << mi.size();
11  }
```

**答案和解析** 程序完全正确，输出为：13

map 属于关联容器 (associative container)。[associative.reqmts#8] 的表格列出了对关联容器使用初始化列表来构造它会发生的事情————将每个元素插入容器中。对于 mb 而言, 由于 map 要求键值唯一, 所以只会有一个元素保留下来。

# 9 C++ 异常处理

**Question 239** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <stdexcept>
2  #include <iostream>
3
4  int main() {
5      try {
6          throw std::out_of_range("");
7      } catch (std::exception& e) {
8          std::cout << 1;
9      } catch (std::out_of_range& e) {
10         std::cout << 2;
11     }
12 }
```

**答案和解析** 程序完全正确，输出为：1

现在有两个 catch 都能捕获 std::out_of_range, 会选择哪个呢? [expect.handle#4] 告诉我们, 按顺序匹配。

The handlers for a try block are tried in order of appearance.

**Question 15** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
#include <iostream>
#include <exception>

int x = 0;

class A {
    public:
    A() {
        std::cout << 'a';
        if (x++ == 0) {
            throw std::exception();
        }
    }
    ~A() { std::cout << 'A'; }
};

class B {
    public:
    B() { std::cout << 'b'; }
    ~B() { std::cout << 'B'; }
    A a;
};

void foo() { static B b; }

int main() {
    try {
        foo();
    }
```

```
30      catch (std::exception &) {
31          std::cout << 'c';
32          foo();
33      }
34  }
```

**答案和解析** 程序完全正确, 输出为: acabBA

这是一道综合题, 我们来分析它的流程。

在主函数中, 调用 foo 后, 我们尝试初始化 b。在调用构造函数 B() 之前, 还要先构造成员函数 a。

在调用 A() 的过程中, 输出 a 并引发异常。注意, 此时 a 的构造还没有完成, 因此之后也不会调用 a 的析构函数 (相对地, 如果 a 有一些已经构造的其他成员, 退出时依然会析构它们)。

catch(std::exception &) 这里捕获了异常, 输出 c 后再次调用 foo()。这一次将顺利构造, 按顺序输出 ab, 程序结束后按相反顺序输出 BA。

**Question 323** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  #include <stdexcept>
3
4  struct A {
5      A(char c) : c_(c) {}
6      ~A() { std::cout << c_; }
7      char c_;
8  };
9
10 struct Y { ~Y() noexcept(false) { throw std::
       runtime_error(""); } };
11
12 A f() {
13     try {
```

```
14          A a('a');
15          Y y;
16          A b('b');
17          return {'c'};
18      } catch (...) {
19      }
20      return {'d'};
21  }
22
23  int main()
24  {
25      f();
26  }
```

**答案和解析** 程序正常运行, 输出为: bcad

...

# 10 C++ 新特性

## 10.1 lambda 表达式

**Question 229** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  int a = 1;
3  int main() {
4      auto f = [](int b) { return a + b; };
5      std::cout << f(4);
6  }
```

**答案和解析** 程序完全正确，输出为：5

这需要说到 lambda 表达式的隐式捕获规则。【待补充】

## 10.2 折叠表达式

**Question 219** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

template<typename T>
T sum(T arg) {
    return arg;
}

template<typename T, typename ...Args>
T sum(T arg, Args... args) {
    return arg + sum<T>(args...);
}

int main() {
    auto n1 = sum(0.5, 1, 0.5, 1);
    auto n2 = sum(1, 0.5, 1, 0.5);
    std::cout << n1 << n2;
}
```

**答案和解析** 程序完全正确，输出为：32

【待补充】

**Question 323** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

void f(float &&) { std::cout << "f"; }
void f(int &&) { std::cout << "i"; }

template <typename... T>
```

```
7  void g(T &&... v)
8  {
9      (f(v), ...);
10 }
11
12 int main()
13 {
14     g(1.0f, 2);
15 }
```

**答案和解析** 程序正常运行, 输出为: bcad

...

## 10.3 std::variant

**Question 279** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
1  #include <iostream>
2  #include <variant>
3
4  struct C
5  {
6      C() : i(1){}
7      int i;
8  };
9
10 struct D
11 {
12     D() : i(2){}
13     int i;
14 };
15
16 int main()
```

```
{
    const std::variant<C,D> v;
    std::visit([](const auto& val){ std::cout
        << val.i; }, v);
}
```

**答案和解析** 程序完全正确，输出为：1

为了解决这一问题, 我们需要了解 std::variant 和 std::visit 在每一句话中究竟做了什么。首先是 std::variant 定义的时候, 它调用了每个类的构造函数吗?

[variant.ctor#2]

> constexpr variant() noexcept(see below);
>
> Effects: Constructs a variant holding a value-initialized value of type T0.

联合体的值将是一个类型为 T0 的、被值初始化的变量值。这里将调用 C 类的构造函数, 联合体的值是 1。

[variant.visit#2]

> constexpr see below visit(Visitor&& vis, Variants&&... vars);
>
> Effects: Let is... be vars.index()....
>
> Returns INVOKE(forward<Visitor>(vis), get<is>( forward<Variants>(vars))...);.

在该程序的情况中, INVOKE(f, x1, x2, ...) 可以理解为 f(x1, x2, ...), 在这里也就是调用 f(get<is>(v)), 也就是将 v 此刻的值 (C 类型, i = 1) 传入 lambda 表达式中, 将输出 1。

**Question 222** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```
#include <variant>
#include <iostream>

```

```
using namespace std;

int main() {
    variant<int, double, char> v;
    cout << v.index();
}
```

**答案和解析** 程序完全正确，输出为：0

容易猜测 index() 是用于指示 v 此刻的类型的, 实际上也正是如此。[variant.status#3]

> constexpr size_t index() const noexcept;
>
> Effects: If valueless_by_exception() is true, returns variant_npos. Otherwise, returns the zero-based index of the alternative of the contained value.

在上一题我们知道, 如果不做初始化, variant 的值将是**第一个类型的**、被值初始化的变量的值, 因此 index 为 0。

## 10.4 promise/future

**Question 340** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <future>
#include <iostream>

int main()
{
    std::promise<int> p;
    std::future<int> f = p.get_future();
    p.set_value(1);
    std::cout << f.get();
    std::cout << f.get();
}
```

**答案和解析** 程序存在未定义行为

在一对 promise/future 变量中, promise 变量只能执行一次 set_value(), future 变量也只能执行一次 get()。

以下是更详细的分析。

主函数第一行创建了 std::promise 对象, 并在第二行获取了与其共享状态的 std::future 对象。

future.promise#12

> future<R> get_future();
> Returns: A future<R> object with the same shared state as *this.

接下来执行 set_value(), 将值存入共享的状态。该函数只能执行一次, 否则抛出异常。

futures.promise#15

> void promise::set_value(const R& r);
> void promise::set_value(R&& r);
> void promise<R&>::set_value(R& r);
> void promise<void>::set_value();
> Effects: Atomically stores the value r in the shared state and makes that state ready.
> Throws:
> **future_error if its shared state already has a stored value or exception.**

get() 函数则获取并且释放共享的状态。第一次 get() 时将得到 1, 第二次 get() 时, 由于 valid() == false, 不满足前置条件, 引发未定义行为。

futures.unique_future#16

> R future::get();
> R& future<R&>::get();
> Effects:
> (15.1)
> wait()s until the shared state is ready, then retrieves the value stored in the shared state;
> (15.2)

**releases any shared state**.

Returns:

(16.1)

future::get() returns the value v stored in the object's shared state as std::move(v).

(16.2)

future<R&>::get() returns the reference stored as value in the object's shared state.

Throws: the stored exception, if an exception was stored in the shared state.

**Postconditions: valid() == false.**

bool valid() const noexcept;

**Returns: true only if *this refers to a shared state.**

**Question 340** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```
#include <future>
#include <iostream>

int main()
{
    try {
        std::promise<int> p;
        std::future<int> f1 = p.get_future();
        std::future<int> f2 = p.get_future();
        p.set_value(1);
        std::cout << f1.get() << f2.get();
    } catch(const std::exception& e)
    {
        std::cout << 2;
    }
}
```

**答案和解析** 程序正常运行，输出为：2

futures.promise#12

> future<R> get_future();
>
> Returns: A future<R> object with the same shared state as *this.
>
> Throws: future_error if *this has no shared state or if **get_future has already been called on a promise with the same shared state as *this.**

# 11 杂项

**Question 193** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```
#include <iostream>

int main() {
    int a[] = <%1%>;
    std::cout << a<:0:>;
}
```

**答案和解析** 程序完全正确，输出为：1

What the f***? 这些运算符是什么?

原来, 一些运算符存在别名, 详细的对应表可以在[lex.digraph#2]中查到。这里, <% 是 { 的别名, 而%> 是 } 的别名, <: 是 [ 的别名, :> 是] 的别名。

大家可能比较熟悉以下几个别名。or 是 | | 的别名, xor 是ˆ 的别名, not 是! 的别名, 等等。

因此, 这段程序实际上是想说

```
int a[] = {1};
std::cout << a[0];
```

答案是 1。

**Question 195** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
#include <cstddef>
#include <type_traits>

int main() {
    std::cout << std::is_pointer_v<decltype(
        nullptr)>;
}
```

**答案和解析** 程序完全正确，输出为：1

decltype(nullptr) 应该直接返回它的类型, 即 std::nullptr_t。不过,[lex.nullptr#1] 指出, 它并不是指针类型。

> The pointer literal is the keyword nullptr. It is a prvalue of type std::nullptr_t. [Note: std::nullptr_t is a distinct type that is neither a pointer type nor a pointer to member type;

**Question 16** Difficulty: Easy

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << (argv[argc] == nullptr);
}
```

**答案和解析** 程序完全正确, 输出 1

根据 [basic.start.main#2], 答案是 1。

> The value of argv[argc] shall be 0.

**Question 224** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct Base {
    virtual int f() = 0;
};

int Base::f() { return 1; }

struct Derived : Base {
    int f() override;
};

int Derived::f() { return 2; }

int main() {
    Derived object;
    std::cout << object.f();
    std::cout << ((Base&)object).f();
}
```

**答案和解析** 程序完全正确，输出为：22

程序的输出并没有疑问。稍微有些反直觉的是, 纯虚函数也是可以被定义的。

[class.abstract#2] 暗示了这一点。

> A pure virtual function need be defined only if called with, or as if with, the qualified-id syntax.

**Question 232** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>

struct S {
    template <typename Callable>
    void operator[](Callable f) {
        f();
    }
};

int main() {
    auto caller = S{};
    caller[ []{ std::cout << "C";} ];
}
```

**答案和解析** 程序将引发编译错误

事实上, 程序逻辑没有任何问题, 但是如果尝试执行, 编译器 (gcc) 会提示

error: two consecutive '[' shall only introduce an attribute before '[' token.

查阅标准[dcl.attr.grammar#7]

> Two consecutive left square bracket tokens shall appear only when introducing an attribute-specifier or within the balanced-token-seq of an attribute-argument-clause.

这是在说, 一般只有类似 `[[nodiscard]] node* f(int x) {return new node(x);}` 之类的情况, 才用到两个连续的左中括号。对于这题而言, 我们不去深究 balanced-token-seq of an attribute-argument-clause 等概念的具体含义, 因为标准给出了一个相似的例子。

```cpp
int p[10];
int(p[[x] { return x; }()]);  // error: invalid
    attribute on a nested declarator-id and not
    a function-style cast of an element of p.
```

**Question 147** Difficulty: Hard

According to the C++17 standard, what is the output of this program?

```cpp
#include<iostream>

int main(){
    int x=0; //What is wrong here??/
    x=1;
    std::cout<<x;
}
```

**答案和解析** 程序完全正确, 输出为: 1

??/ 是一个三字符序列 (trigraph)。在某些早期计算机系统中，某些字符无法直接输入，因此三字符序列被用于表示那些无法在键盘上输入的字符。??/ 用于表示反斜杠‘\’

三字符序列在 C++17 中已经被弃用, 因此程序将输出 1。如果打开 -trigraph 编译选项 (gcc), 注释将延伸到 x=1 这一行, 输出 0。

**Question 41** Difficulty: Medium

According to the C++17 standard, what is the output of this program?

```cpp
#include <iostream>
int main() {
    std::cout << 1["ABC"];
}
```

**答案和解析** 程序完全正确，输出为：B

[expr.sub#1]

> The expression E1[E2] is identical (by definition) to *((E1)+(E2))

# 12 习题参考答案

## 12.1 C++ 基本问题

**Exercise 2.1** 程序完全正确，输出为：100

根据聚合体初始化规则, 此处初始化列表的元素个数少于聚合体的元素数量, 所以 a[0] 被初始化为 1, 而 a[1] 和 a[2] 则选择采用空列表初始化, 所以均被初始化为 0。

因此, 平时我们写的

```
const int N = 100;
int a[N] = {0};
```

确实能够保证将 a 数组的所有元素都初始化为 0, 但其实列表中的 0 只赋值了第一个元素, 其他的元素则是被空列表初始化的。采用 `int a[N] = {}` 能达到同样的效果。

**Exercise 2.2** 待补充

【待补充】

**Exercise 2.3** 我们可以逐步分析, 参考以下程序。

```
int main() {
    int *const** const* u = nullptr;
    int x = 1; // x 是一个 int 类型的常量。
    int *const p = &x; // p 是一个指针常量, 指向 int 类型的 x。
    int *const* p1 = &p; // p1 是一个指针, 指向 p
    int *const* *const p2 = &p1; // p2 是一个指针常量, 指向 p1
    int *const* *const* p3 = &p2; // p3 是一个指针, 指向 p2
    u = p3; // ok
}
```

所以, u 是一个指针, 指向一个指向指针的指针常量, 被指向的指针又指向一个指向 int 类型对象的指针。

## 12.2 C++ 函数

**Exercise 4.1** 对于该程序的解读有误。事实上, 第二个函数根本就不能定义。[over.load#3.4] 指出, 如果两个函数只在 (最外层的) cv 限定符上不同, 则是一种重复定义。

> Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called.

不过, 解读中提到的规则是正确的。如果你写:

```
#include <iostream>

void f(const char*) { std::cout << 1;}

void f(const char(&)[14]) { std::cout << 3;}

int main() {
    char s[] = "Hello, World!";
    f(s);
}
```

则选择第二个函数, 虽然两者的 Rank 都是 Exact Match, 但是第二个函数的标准转换序列 (Identity conversion) 是第一个函数的转换 (Qualification conversion) 的真子集。

而如果改为 `const char s[] = "Hello, World!";` 则不同, 虽然第一个函数确实做了一次 Array-to-pointer conversion, 但在真子集比较中并不考虑这一等级的变换, 因此导致二义性。

**Exercise 4.2** 对于该程序的解读有误。事实上, 解读有意淡化了引用。Integral Promotion 得到的临时常量是一个右值, 也就是 int 类型的右值。那么, int& 不能绑定它, 所以第一个函数根本不在重载的考虑范围之内。

虽然本题到此就可得出结论, 但这里还是要提出一点。第二个函数的转换仅仅只是一次 Integral Promotion, 而没有 Qualification conversion。根据[over.ics.ref#1], **如果形参可以直接绑定实参, 则这一转换是一次 Identity conversion**。那么, const int& 是可以直接绑定 int 类型的右值的 (当然, 它也可以直接绑定 int 类型的左值), 因此不需要 Qualification conversion。

> When a parameter of reference type binds directly to an argument expression, the implicit conversion sequence is the identity conversion.

如果你尝试写

```
#include<iostream>

void f(int x) { std::cout << 1; }

void f(const int& x) { std::cout << 2; }

int main()
{
    f(10);
}
```

会引发二义性错误, 因为两个函数转换级别都是 Identity, 无法分出好坏。而如果采用

```
void f(const int& x) { std::cout << 1; }
void f(const int&& x) { std::cout << 2; }
```

答案就会是 2。这是因为, 根据[over.ics.rank#3.2.3]], 如果标准转换 S1 通过右值引用 && 绑定了一个右值, 而 S2 绑定了一个左值, 则 S1 更优。

## 12.3 C++ 类和对象

**Exercise 6.1** 交换两者的定义顺序, 确保将 A::createB() 为友元函数时, 该函数已经被定义。

```
#include <iostream>
```

```
2
3 class B;  // Forward declaration of B
4
5 class A {
6     public:
7     A() { std::cout << "A"; }
8
9     B createB();  // Declare createB() before
          defining class B
10 };
11
12 class B {
13     public:
14     B() { std::cout << "B"; }
15     friend B A::createB();  // Now the friend
          declaration can reference A::createB()
16 };
17
18 B A::createB() {
19     return B();
20 }
21
22 int main() {
23     A a;
24     B b = a.createB();
25 }
```

# 13 后记

不足之处, 请各位读者批评指正。【待补充】