2. 什么是“auto”关键字?

2.1 定义和基本概念

C++11中，标准委员会赋予了auto全新的含义即：auto不再是一个存储类型指示符，而是作为一个新的类型指示符来指示编译器，auto声明的变量必须由编译器在编译时期推导而得，auto关键字被引入以简化变量的类型声明。使用auto关键字，编译器会根据初始化表达式的类型自动推导出变量的类型。这可以使得代码更加简洁，尤其是在处理复杂或冗长的类型时。

2.2 “auto”关键字的基本用法

1. **自动类型推导**： 当使用“auto”关键字声明变量时，编译器会根据变量的初始值来推导其类型。例如：

```
auto x = 10; // x 被推导为 int
auto y = 3.14; // y 被推导为 double
auto str = "Hello, World!"; // str 被推导为 const char*
```

2. **简化代码**： 使用“auto”关键字可以简化代码，特别是在处理复杂类型时。例如：

```
std::vector<std::pair<int, std::string>> vec;
auto it = vec.begin(); // it 被推导为 std::vector<std::pair<int, std::str
```

3. **提高代码可读性**： 通过使用“auto”关键字，可以使代码更加简洁和易读，减少冗长的类型声明。例如：

```
auto result = someFunction(); // result 的类型由 someFunction() 的返回值决定
```

4. **与C++11及更高版本的特性结合使用**： “auto”关键字可以与其他C++11及更高版本的特性结合使用，如lambda表达式和范围for循环。例如：

```
auto lambda =  { return a + b; };
std::vector<int> vec = {1, 2, 3};
for (auto& elem : vec) {
    std::cout << elem << std::endl;
}
```

2.3 “auto”关键字的限制和注意事项

1. 不能用于函数参数：

> “auto”关键字不能用于 函数 参数声明。这是因为函数参数的类型必须在函数声明时明确指定，而“auto”关键字只能用于变量的类型推导。例如，以下代码是错误的：

```
void func(auto x); // 错误，auto 不能用于函数参数
```

2. auto不能直接用来声明数组：

限制说明：

“auto”关键字在C++中不能用于直接声明数组类型。这是因为数组的大小必须在编译时确定，而“auto”关键字用于类型推导时，无法推导出数组的大小。例如，以下代码是错误的：

```
auto arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // 错误，auto 不能用于数组声明
```

正确的做法：

如果需要使用自动类型推导来声明数组，可以考虑使用std::array或std::vector等标准库容器，这些容器可以与“auto”关键字一起使用。例如：

```
#include <array>
#include <vector>

int main() {
    // 使用 std::array
    std::array<int, 10> arr = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto arrCopy = arr; // 自动推导类型为 std::array<int, 10>

    // 使用 std::vector
    std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto vecCopy = vec; // 自动推导类型为 std::vector<int>

    return 0;
}
```

通过使用std::array或std::vector，可以享受自动类型推导的便利，同时避免数组声明

的限制。

3. 可能导致代码可读性下降的情况：

> 虽然“auto”关键字可以简化代码，但在某些情况下，它可能会导致代码的可读性下降，特别是当类型推导不明显时。例如：

```
auto result = someFunction(); // result 的类型不明确
```

> 在这种情况下，读者需要知道someFunction()的返回类型才能理解result的类型。如果函数返回类型复杂或不常见，可能会增加理解代码的难度。

2.4 “auto”关键字高级用法

2.4.1 与范围for循环结合使用

1 语法

使用“auto”关键字与范围for循环（range-based for loop）结合，可以简化遍历容器的代码，使其更加简洁和易读。以下是一些具体的示例：

示例1：遍历std::vector

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用 auto 关键字和范围for循环
    for (auto& elem : vec) {
        std::cout << elem << std::endl; // 自动推导 elem 的类型为 int&
    }

    return 0;
}
```

在这个示例中，auto& elem会自动推导elem的类型为int&，即vec中元素的引用。使用引

用可以避免不必要的拷贝，提高效率。

示例2：遍历`std::map`

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> myMap = {{"apple", 1}, {"orange", 2}, {"pear",

    // 使用 auto 关键字和范围for循环
    for (auto& pair : myMap) {
        std::cout << pair.first << ": " << pair.second << std::endl; // 自动推
    }

    return 0;
}
```

在这个示例中，`auto& pair`会自动推导`pair`的类型为`std::pair<const std::string, int>&`，即`myMap`中元素的引用。

示例3：遍历`std::array`

```
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> arr = {1, 2, 3, 4, 5};

    // 使用 auto 关键字和范围for循环
    for (auto& elem : arr) {
        std::cout << elem << std::endl; // 自动推导 elem 的类型为 int&
    }

    return 0;
}
```

2 范围for的使用条件

1. for循环迭代的范围必须是确定的

对于数组而言，就是数组中第一个元素和最后一个元素的范围；对于类而言，应该提供

begin和end的方法，begin和end就是for循环迭代的范围。

注意：以下代码就有问题，因为for的范围不确定

```
void TestFor(int array[])
{
    for(auto& e : array)
        cout<< e <<endl;
}
```

2. 迭代的对象要实现++和==操作

迭代的对象必须支持递增（++）和比较（==）操作。这是因为范围for循环在内部使用这些操作来遍历容器。

范围for循环在C++中是一种简化遍历容器的语法糖。为了理解为什么迭代的对象必须支持递增（++）和比较（==）操作，我们需要了解范围for循环的工作原理。

范围for循环的工作原理

当编译器遇到范围for循环时，它会将其转换为等价的传统for循环代码。这个转换过程依赖于迭代器的递增和比较操作。以下是一个示例：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 范围for循环
    for (auto& elem : vec) {
        std::cout << elem << " ";
    }

    return 0;
}
```

编译器会将上述范围for循环转换为类似以下的代码：

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

```
int main() {        std::vector<int> vec = {1, 2, 3, 4, 5};

    // 等价的传统for循环
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        auto& elem = *it;
        std::cout << elem << " ";
    }

    return 0;
}
```

递增（++）和比较（==）操作的作用

1. **递增操作（++）**： 递增操作用于移动迭代器到下一个元素。在传统for循环中，`++it`将迭代器`it`移动到容器中的下一个元素。这是遍历容器的关键步骤。
2. **比较操作（==）**： 比较操作用于检查迭代器是否到达容器的末尾。在传统for循环中，`it != vec.end()`用于判断迭代器是否已经遍历完所有元素。如果迭代器等于容器的`end()`，则循环终止。

为什么需要这些操作

- **递增操作**：没有递增操作，迭代器无法移动到下一个元素，循环将无法遍历整个容器。
- **比较操作**：没有比较操作，循环无法判断何时停止，可能会导致无限循环或访问越界。

2.4.2 与Lambda表达式结合使用

使用“auto”关键字与Lambda表达式结合，可以使代码更加简洁和灵活。Lambda表达式是C++11引入的一种匿名函数，可以在函数内部定义并使用。以下是一些具体的示例：

示例1：基本Lambda表达式

```
#include <iostream>

int main() {
    auto add =  { return a + b; };
    int result = add(3, 4); // result 被推导为 int，值为 7
    std::cout << "3 + 4 = " << result << std::endl;

    return 0;
}
```

在这个示例中，auto关键字用于推导Lambda表达式的类型，使代码更加简洁。

示例2：捕获外部变量

```
#include <iostream>

int main() {
    int factor = 2;
    auto multiply = factor { return a * factor; };
    int result = multiply(5); // result 被推导为 int，值为 10
    std::cout << "5 * 2 = " << result << std::endl;

    return 0;
}
```

在这个示例中，Lambda表达式捕获了外部变量factor，并在表达式内部使用。

示例3：与STL算法结合使用

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用 Lambda 表达式和 auto 关键字
    std::for_each(vec.begin(), vec.end(),  { n *= 2; });
```

在这个示例中，Lambda表达式与STL算法std::for_each结合使用，简化了代码。

示例4：返回Lambda表达式

```
#include <iostream>
#include <functional>

auto createLambda() {
    return  { return a + b; };
```

```
}
int main() {
    auto add = createLambda();
    int result = add(3, 4); // result 被推导为 int，值为 7
    std::cout << "3 + 4 = " << result << std::endl;

    return 0;
```