

我要打RM我要进决赛

1. C++ 程序可以定义为对象的集合

- **对象** - 对象具有状态和行为。例如：一只狗的状态 对象是类的实例。
- **类** - 类可以定义为描述对象行为/状态的模板/蓝图。
- **方法** - 从基本上说，一个方法表示一种行为。一个类可以包含多个方法。可以在方法中写入逻辑、操作数据以及执行所有的动作。
- **即时变量** - 每个对象都有其独特的即时变量。对象的状态是由这些即时变量的值创建的。

2. 程序结构

- C++ 语言定义了一些头文件，这些头文件包含了程序中必需的或有用的信息。上面这段程序中，包含了头文件 `<iostream>`。
- `using namespace std;` 告诉编译器使用 std 命名空间。命名空间是 C++ 中一个相对新的概念。
- 单行注释以 `//` 开头，在行末结束。
- 下一行 `int main()` 是主函数，程序从这里开始执行。。
- 下一行 `return 0;` 终止 `main()` 函数，并向调用进程返回值 0。

3. 编译 & 执行 C++ 程序

Vscode自帶了，太好了

4. C++ 中的分号 & 语句块

在 C++ 中，分号是语句结束符。也就是说，每个语句必须以分号结束。它表明一个逻辑实体的结束。

语句块是一组使用大括号括起来的按逻辑连接的语句。

C++ 不以行末作为结束符的标识，因此，可以在一行上放置多个语句。

5. C++ 标识符

C++ 标识符是用来标识变量、函数、类、模块，或任何其他用户自定义项目的名称。一个标识符以字母 A-Z 或 a-z 或下划线 `_` 开始，后跟零个或多个字母、下划线和数字（0-9）。

C++ 标识符内不允许出现标点字符，比如 @、& 和 %

C++ 是区分大小写的编程语言。

6. C++ 关键字

主要靠积累了

7. 三字符组

不用

8. C++ 中的空格

可读性

只包含空格的行，被称为空白行，可能带有注释，C++ 编译器会完全忽略它。

在 C++ 中，空格用于描述空白符、制表符、换行符和注释。空格分隔语句的各个部分，让编译器能识别语句中的某个元素（比如 int）在哪里结束，下一个元素在哪里开始。因此，在下面的语句中：

```
int age;
```

在这里，int 和 age 之间必须至少有一个空格字符（通常是一个空白符），这样编译器才能够区分它们。另一方面，在下面的语句中：

```
fruit = apples + oranges; // 获取水果的总数
```

fruit 和 =，或者 = 和 apples 之间的空格字符不是必需的，但是为了增强可读性，可以根据需要适当增加一些空格。

9. 数据类型

C++ 为程序员提供了种类丰富的内置数据类型和用户自定义的数据类型。下表列出了七种基本的 C++ 数据类型：

类型	关键字
布尔型	Bool
字符型	char
整型	int
浮点型	float

双浮点型	double
无类型	void
宽字符型	wchar_t

Wchar_t和short是一样的

还有好多好多数据类型

派生数据类型

数组 相同类型元素的集合 `int arr[5] = {1, 2, 3, 4, 5};`

指针 存储变量内存地址的类型 `int* ptr = &x;`

引用 变量的别名 `int& ref = x;`

函数 函数类型，表示函数的签名 `int func(int a, int b);`

结构体 用户定义的数据类型，可以包含多个不同类型的成员 `struct Point { int x; int y; };`

类 用户定义的数据类型，支持封装、继承和多态 `class MyClass { ... };`

联合体 多个成员共享同一块内存 `union Data { int i; float f; };`

枚举 用户定义的整数常量集合 `enum Color { RED, GREEN, BLUE };` typedef 声明

10. typedef type newname;

下面的语句会告诉编译器，feet 是 int 的另一个名称：

```
typedef int feet;
```

创建一个整型变量 distance：

```
feet distance;
```

11. 枚举类型

枚举类型(enumeration)是C++中的一种派生数据类型，它是由用户定义的若干枚举常量的集合。

如果一个变量只有几种可能的值，可以定义为枚举(enumeration)类型。所谓"枚举"是指将变量的值一一列举出来，变量的值只能在列举出来的值的范围内。

默认情况下，第一个名称的值为 0，第二个名称的值为 1，第三个名称的值为 2，以此类推。但是，您也可以给名称赋予一个特殊的值，只需要添加一个初始值即可。例如，在下面的枚举中，**green** 的值为 5。

```
enum color { red, green=5, blue };
```

在这里，**blue** 的值为 6，因为默认情况下，每个名称都会比它前面一个名称大 1，但 red 的值依然为 0。

在C++中，枚举（enum）是一种用户自定义的数据类型，用于定义一组命名的整数常量，从而提高代码的可读性和可维护性。下面通过几个具体的例子来说明如何使用枚举。

示例1：基本枚举定义

```
enum Day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};
```

```
Day today = Mon;
```

```
cout << "Today is " << today << endl; // 输出 Today is 1
```

在这个例子中，Day 是一个枚举类型，包含了从 Sun 到 Sat 的七个常量。默认情况下，这些常量的值从0开始依次递增。因此，Mon 的值为1。

示例2：自定义初始值

```
enum Month {Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

```
Month currentMonth = Mar;
```

```
cout << "Current month is " << currentMonth << endl; // 输出 Current month is 3
```

在这个例子中，我们将 Jan 的值设为1，因此 Feb 的值为2，Mar 的值为3，以此类推。

示例3：混合自定义值

```
enum Fruit {Apple=3, Orange, Banana=4, Bear};
```

```
Fruit myFruit = Orange;
```

```
cout << "My fruit is " << myFruit << endl; // 输出 My fruit is 4
```

在这个例子中，我们为 Apple 和 Banana 分别指定了值3和4。由于 Orange 没有指定值，它会自动继承前一个常量的值加1，即4。而 Bear 也没有指定值，因此它的值为5。

示例5：枚举类型的应用场景

cpp

复制

```
enum Status {OK = 200, NOT_FOUND = 404, ERROR = 500};

Status response = OK;

if(response == OK) {
    cout << "Request successful!" << endl; // 输出 Request successful!
}
```

在这个例子中，我们定义了一个 Status 枚举类型来表示HTTP响应的状态码。通过这种方式，我们可以更直观地理解代码中的状态值。

通过以上几个例子，可以看出枚举在C++编程中的重要作用。它不仅能够提高代码的可读性，还能减少因使用硬编码数字而导致的错误。

总结：感觉没什么很厉害的地方。

12. 类型转换

类型转换是将一个数据类型的值转换为另一种数据类型的值。

C++ 中有四种类型转换：静态转换、动态转换、常量转换和重新解释转换。

静态转换（Static Cast）

静态转换是将一种数据类型的值强制转换为另一种数据类型

静态转换通常用于比较类型相似的对象之间的转换，例如将 int 类型转换为 float 类型。

静态转换不进行任何运行时类型检查，因此可能会导致运行时错误。

实例

```
int i = 10; float f = static_cast<float>(i); // 静态将int类型转换为float类型
```

```

myselftrials > A3.cpp > main()
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int num1 = 12;
7      long long num2 = static_cast <long long> (num1) ;
8      cout << num1 <<" " << num2<<endl;
9      cout << sizeof num1 << " " << sizeof num2<<endl;
10     system("pause");
11     return 0;
12
13 }

```

动态转换（Dynamic Cast）（看不懂哈哈）

动态转换（dynamic_cast）是 C++ 中用于在继承层次结构中进行向下转换（downcasting）的一种机制。

动态转换通常用于将一个基类指针或引用转换为派生类指针或引用。

动态转换在运行时进行类型检查。如果转换失败，对于指针类型会返回 nullptr，对于引用类型则会抛出 std::bad_cast 异常。

语法：

dynamic_cast<目标类型>(表达式)

- **目标类型：**必须是指针或引用类型。
- **表达式：**需要转换的基类指针或引用。

实例：指针类型的动态转换

```
#include <iostream>
```

```
class Base {
```

```
public:
```

```
    virtual ~Base() = default; // 基类必须具有虚函数
```

```
};
```

```
class Derived : public Base {
```

```
public:
```

```

void show() {
    std::cout << "Derived class method" << std::endl;
}

};

int main() {
    Base* ptr_base = new Derived; // 基类指针指向派生类对象

    // 将基类指针转换为派生类指针
    Derived* ptr_derived = dynamic_cast<Derived*>(ptr_base);

    if (ptr_derived) {
        ptr_derived->show(); // 成功转换，调用派生类方法
    } else {
        std::cout << "Dynamic cast failed!" << std::endl;
    }

    delete ptr_base;
    return 0;
}

```

输出：

Derived class method

实例：引用类型的动态转换

```
#include <iostream>
```

```
#include <typeinfo>
```

```
class Base {
```

```
public:
```

```
    virtual ~Base() = default; // 基类必须具有虚函数
```

```
};
```

```
class Derived : public Base {
public:
    void show() {
        std::cout << "Derived class method" << std::endl;
    }
};

int main() {
    Derived derived_obj;
    Base& ref_base = derived_obj; // 基类引用绑定到派生类对象

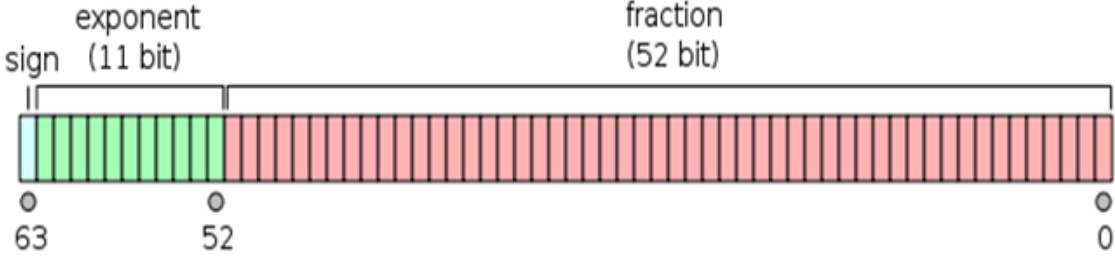
    try {
        // 将基类引用转换为派生类引用
        Derived& ref_derived = dynamic_cast<Derived&>(ref_base);
        ref_derived.show(); // 成功转换，调用派生类方法
    } catch (const std::bad_cast& e) {
        std::cout << "Dynamic cast failed: " << e.what() << std::endl;
    }

    return 0;
}
```

输出：

Derived class method

特性	指针类型	引用类型
转换失败返回值	返回 nullptr	抛出 std::bad_cast 异常
适用场景	向下转换、运行时类型检查	向下转换、运行时类型检查

	
void	表示类型的缺失。
wchar_t	宽字符类型，用于存储更大范围的字符，通常占用 2 个或 4 个字节。

1. 注释

Note的功能最方便，可以一行多行，但是基本上用//
还有一种 /* */感觉基本不会用

2. main函数

输出hello world

3. 常量

Define的用法是 #define 一个东西 一个数字（只能是数字 否则会报错）
“一个东西”可以是任意字母 汉字 但是不能出现特殊符号,否则在cout输出时会报错
Const的用法是 const int 一个东西 = 一个数字； 条件同上
但是，const必须先标注数据类型 并且在一个数字后面要有；
并且const好像只能写在{ }之中
但是define既可以在{ }外面也可以在里面

还可以直接用int定义一个量的值并且可以随时修改，上面两个不能随时修改
值得注意的是{ }里面基本上写一行加一个；
少加一个； 程序无法运行

4. int

就是上面提到过的int

5. 关键字

“② 在定义变量或者常量的时候，不要用关键字。” 在上面我以亲身经历明白了这个道理

Sizeof

接下来我打算先看黑马的教程，上面的是看各位学长编辑的文档的

标识符命名规则

数字不能当第一个 只能出现字母 数字 _

6. 整型

看了这个我才知道那个作业里的那个两个整数相加为什么还有一个范围限制以及为什么我只有70分一开始

短整型Short -2^{15} $2^{15}-1$ 2byte 32768

整形Int -2^{31} $2^{31}-1$ 4byte

长整型Long -2^{31} $2^{31}-1$ windows and linux (32-bit) 4byte(1byte=8bit)

Linux(64-bit) 8byte

长长整形Long long -2^{63} $2^{63}-1$ 8 byte

所占用的空间不同

7. sizeof

求出数据类型占用的空间大小

语法 sizeof (数据类型/变量)

Short a1 =xxx

Int a2= xxx

Long a4= xxx

```
cout<<sizeof(a1,a2,a4)<<endl;
```

好像此时只会输出占用内存最大的

8. 浮点型

单精度 Float 4 byte 7 sig.fig. e.g. 3.14 has 3 sig.fig.

双精度 Double 8byte 14-15sig.fig.

3.14和3.14f

默认情况下输出小数，只能输出小数点后五位（vscode是这样的）

科学计数法 $3e2 = 300$ $3e-2=0.03$ 数据类型也是用浮点表示

warning: floating constant truncated to zero [-Woverflow]

```
26 | double b2 = 8e-2832;
```

```
| ^~~~~~
```

这里的woverflow 应该是溢出（虽然溢出的英文应该是overflow）

9. 字符型

Char 单引号 ‘ ’ 之间只能有一个字符

只占用 1 byte

储存对象不是直接的字符，而是对应的ASCII码

可以用(int) ch 查看 ch对应的ASCII码

空格也算一个字符

Int(ch) 和 (int) ch效果好像一样

9.1 摸索

1.我发现了想要让输出的多个数据之间有空格要这样干

比如说现在输出

```
Int a1 = 876;
```

```
Int a2 = 654;
```

```
Cout <<a1<<a2<< endl;
```

直接这样干会导致输出876654

我要呈现876 654 的话就要这样：

```
Cout << a1 <<" " <<a2 << endl;
```

1. 我现在想要完成金字塔作业，因此要学会换行

我知道了输出的 行 是和 endl 有关

```
cout << "这是第一行" << endl;
```

```
cout << "这是第二行" << endl;
```

2. 接下来便是如何以输入的字符构建金字塔了，即当用户指定一种材料时，他就要用相应的材料造房子。

我搜索了一下 用char

10. Return0有什么用

return 0 在C++中主要用于表示程序正常结束，并向操作系统报告程序执行状态。这种做法有助于提高代码的可读性、可维护性和稳定性，同时也符合C++语言的标准要求。

是一种习惯好像，本身没什么影响

11. int&r 和 int r 的区别

内存分配：

int r 声明了一个新的整型变量 r，它会在内存中开辟一个新的空间来存储整数值。这意味着 r 和任何其他变量都是独立的，修改 r 不会影响其他变量。

int&r 声明了一个引用 r，它并不开辟新的内存空间，而是成为已经存在的某个整型变量的别名。因此，通过 r 对该变量的操作实际上是对原变量的操作。

初始化要求：

引用必须在声明时初始化，并且初始化后不能重新绑定到其他变量。例如，int i = 1; int &r = i; 这里 r 必须绑定到 i，并且之后不能改变这种绑定关系。

普通变量则没有这样的限制，可以在任何时候被赋予新的值。

地址相同性：

当 int&r = i; 时，r 和 i 的地址是相同的，因为 r 只是 i 的另一个名字。

而 int r = i; 则不同，r 和 i 是两个独立的变量，拥有不同的内存地址。

修改影响：

修改引用 `r` 的值实际上是在修改它所绑定的原变量的值。例如，如果 `int i = 1; int &r = i; r = 2;`，那么 `i` 的值也会变成 2。

修改普通变量 `r` 的值不会影响其他变量。例如，如果 `int i = 1; int r = i; r = 2;`，那么 `i` 的值仍然是 1。

语法和用途：

引用通常用于函数参数传递，可以避免拷贝大对象，提高效率，并且可以直接修改调用者提供的数据。

普通变量则用于存储独立的数据，适用于不需要共享状态的场景。

12. C++中static cast 和 重新解释转换有什么区别

static_cast

用途：static_cast 主要用于执行较安全的类型转换，包括基本类型之间的转换、类层次结构中基类与派生类指针或引用的上行转换（安全）和下行转换（不安全）。它还可以用于将空指针转换为目标类型的空指针。

安全性：static_cast 进行编译时的类型检查，确保转换的有效性。它不能用于指针与整型之间的转换，也不能去除表达式的 `const`、`volatile` 或 `__unaligned` 属性。

值得注意的是：编译时类型检查：多个证据表明 static_cast 在编译时期进行类型检查，如果类型转换不合法，编译器会报错。强调了 static_cast 的编译时类型检查特性。

不进行运行时类型检查：尽管 static_cast 进行编译时的类型检查，但它并不进行运行时的类型检查。这意味着如果在运行时进行不安全的转换（例如将基类指针转换为派生类指针，而实际对象并不是该派生类类型），会导致未定义行为

示例：

基本类型转换：`int i = static_cast<int>(3.14);`

类层次结构中的转换：`Base* basePtr = new Derived(); Derived* derivedPtr = static_cast<Derived*>(basePtr);`

reinterpret_cast

用途：reinterpret_cast 用于底层、不安全的类型转换，主要用于指针类型之间的转换、引用类型或指针与整数类型之间的转换。它仅重新解释对象的位模式，不进行类型检查。

安全性：reinterpret_cast 不进行任何运行时检查，因此可能导致未定义行为。它常用于低层操作，如处理原始内存或设备I/O。

示例：

指针类型转换：`int* intPtr = new int(10); char* charPtr = reinterpret_cast<char*>(intPtr);`

指针与整数之间的转换：`int* ptr = new int(42); uintptr_t val = reinterpret_cast<uintptr_t>(ptr);`

区别总结

类型检查：`static_cast` 进行编译时的类型检查，而 `reinterpret_cast` 不进行任何类型检查。

安全性：`static_cast` 更安全，因为它确保了转换的有效性；而 `reinterpret_cast` 更灵活但可能不安全，因为它可能导致未定义行为。

用途：`static_cast` 适用于具有继承关系或密切相关的类型之间的转换；而 `reinterpret_cast` 适用于低层次的指针操作和任意类型的转换。

1

2

3

。