# C++是如何工作的

## 1.C++程序编译流程

C++程序由源文件组成，通过编译器转换成二进制文件（库或可执行程序）。

预处理器首先处理源文件中的预处理器语句（如#include），将头文件内容插入到源文件中。

## 2.编译器和链接器的作用

编译器将C++代码转换为机器代码，生成对象文件（.obj）。

链接器将所有对象文件合并成一个可执行文件（.exe）。

## 3.Visual Studio中的配置

可以通过修改项目属性来改变编译和链接的规则。

项目属性页允许设置编译器选项，如优化级别、预处理器定义等。

## 4.编译和调试

可以单独编译单个文件（如使用Ctrl + F7），也可以编译整个项目。

编译错误会在输出窗口中显示，可以点击错误信息跳转到源代码中的错误位置。

## 5.函数和头文件

函数需要声明和定义。声明告诉编译器函数的存在，定义提供函数的具体实现。

当函数定义在其他文件中时，需要在使用它的文件中提供声明（函数原型）。

## 6.链接错误

如果链接器找不到函数定义，会产生链接错误。这通常是因为函数声明和定义不匹配，或者函数定义不存在。

## 7.项目构建和运行

构建项目时，Visual Studio会将输出文件放在指定的目录（如“Debug”文件夹）。

运行可执行文件后，程序会执行并显示结果。

# C++编译器是如何工作的

## 1.C++编译器概述

C++编译器负责将文本形式的源代码转换成可执行的二进制文件。

这个过程主要分为两个步骤：编译和链接。

## 2.编译过程

预处理：编译器处理预处理器语句（如#include），将头文件内容插入到源文件中。

编译：将预处理后的代码转换成中间格式（目标文件）。这个过程包括标记化、解析和创建抽象语法树（AST）。

代码生成：编译器将AST转换成机器代码，生成目标文件。

## 3.翻译单元和目标文件

C++不关心文件，文件只是提供源代码给编译器的一种方式。

每个.cpp文件被视为一个翻译单元，编译后生成一个目标文件（.obj）。

## 4.预处理器的使用

预处理器语句（如宏定义#define）用于条件编译和包含头文件。

预处理器可以替换文本、包含文件内容等。

## 5.编译器的输出

编译器输出的目标文件包含机器代码和符号信息，但不包含完整的可执行程序。

通过链接器将多个目标文件合并成一个可执行文件。

## 6.优化和调试

### 优化

1.编译器优化：

编译器在生成机器代码时，会尝试优化代码以提高效率和性能。

优化可以包括减少不必要的指令、合并相似的代码段、使用寄存器代替内存访问等。

如果编译器设置为不优化（如在调试模式下），则生成的代码会包含额外的指令，以确保代码易于调试，但运行效率较低。

2.常量折叠（Constant Folding）：

编译器在编译时计算所有可以在编译时确定的常量表达式。

例如，如果代码中有5 \* 2这样的表达式，编译器会直接将其计算为10，而不是在运行时进行计算。

3.优化的副作用：

优化可能会导致代码行为的改变，尤其是在调试时。

如果编译器优化开启，一些看似无用的代码（如仅用于调试的打印语句）可能会被完全移除。

### 调试

1.调试信息：

编译器在调试模式下会生成额外的调试信息，这些信息有助于开发者理解程序的执行流程和变量状态。

这些信息通常包括变量名、行号等，使得在使用调试器时能够更容易地跟踪程序。

2.调试模式与发布模式：

调试模式（Debug）通常用于开发和测试阶段，它包含额外的调试信息，但运行速度较慢。

发布模式（Release）则移除调试信息，优化代码以获得最佳性能，适合最终用户使用。

3.调试器的使用：

视频中没有详细讨论调试器的使用，但强调了调试模式下编译器生成的调试信息对调试过程的重要性。

## 7.函数和链接

函数需要声明和定义。声明告诉编译器函数的存在，定义提供函数的具体实现。

链接器负责将所有目标文件中的符号（函数和变量）解析并链接成最终的可执行文件。

# C++链接器是如何工作的

## 链接过程

### 1.链接定义

链接是将编译后的目标文件（.obj）合并成一个可执行文件（.exe）的过程。

链接器的主要任务是找到每个符号（函数和变量）的位置，并将它们正确地连接在一起。

### 2.链接器的作用

确保程序中所有引用的函数和变量都能找到对应的定义。

确定程序的入口点（通常是main函数），并从那里开始执行程序。

处理外部库的链接，如C运行时库和C++标准库。

## 链接错误

### 链接错误类型

未解析的外部符号：这是最常见的链接错误之一，意味着链接器在尝试将程序中的函数或变量引用与实际定义匹配时失败了。例如，如果程序中调用了某个函数，但链接器找不到该函数的定义，就会出现这种错误。

重复的符号：当链接器发现多个具有相同名称和类型的符号定义时，会抛出重复符号的错误。这通常发生在同一个符号在多个文件中被定义时。

### 解决方法

确保符号定义唯一：每个函数和变量只能有一个定义。如果在多个文件中定义了相同的函数或变量，需要确保只有一个定义，并在其他文件中使用声明（即头文件中的函数原型）。

使用头文件声明：头文件中应包含函数和变量的声明，这样编译器在编译其他文件时能够知道这些符号的存在。实际的定义则应放在一个.cpp文件中。

使用static关键字：在函数或变量前加上static关键字可以限制其作用域仅限于当前文件，防止链接错误。

正确使用链接器选项：在某些情况下，可能需要使用特定的链接器选项来指定库文件的位置或处理其他链接问题。

## 链接与编译的区别

### 1.编译

将源代码文件（.cpp）转换为对象文件（.obj）。

包括预处理、编译（生成抽象语法树AST）、代码生成等步骤。

### 2.链接

将一个或多个对象文件合并成一个可执行文件。包括符号解析、地址分配等步骤。

## 链接器的高级功能

### 1.静态链接与动态链接

静态链接：将所有需要的库代码直接包含在最终的可执行文件中。

动态链接：可执行文件依赖于外部的共享库文件，这些文件在运行时被加载。

# C++变量

## 1.变量的定义和作用

变量是用于存储数据的命名位置，允许我们在程序中操纵和使用数据。

在编程中，我们经常需要存储和修改数据，比如游戏中的玩家位置，这需要使用变量来实现。

## 2.变量存储位置

变量存储在计算机内存中，具体位置可以是堆栈或堆。堆栈用于存储局部变量，而堆用于动态分配的内存。

## 3.C++中的数据类型

C++提供了多种原始数据类型，如整数（int）、字符（char）、浮点数（float和double）和布尔值（bool）。

每种数据类型占用的内存大小不同，影响了它们能存储的数据范围。

## 4.整数类型

int类型通常占用4个字节，可以存储从-2,147,483,648到2,147,483,647的整数。

无符号整数（unsigned int）可以存储更大的正整数，因为它们不存储负号。

## 5.浮点数类型

float和double用于存储十进制数，float通常占用4个字节，而double占用8个字节。

float类型在声明时需要加上f或F后缀。

## 6.布尔类型

bool类型用于表示真（true）或假（false），占用1个字节。

## 7.内存大小和数据类型

数据类型占用的内存大小取决于编译器，但通常有标准范围。

使用sizeof运算符可以查询特定数据类型的内存大小。

## 8.变量的声明和使用

变量通过指定类型、名称和可选的初始值来声明。

变量可以被重新赋值，其值可以在程序中改变。

# C++函数

## 1.函数的定义和作用

函数是代码块，用于执行特定任务，避免代码重复。函数有输入（参数）和输出（返回值）。

## 2.函数的组成

返回类型：函数执行后返回的数据类型。函数名：用于调用函数的标识符。

参数列表：函数接收的输入数据。函数体：包含执行任务的代码块。

## 3.函数的使用示例

函数可以不带参数，也可以返回特定类型的数据。

## 4.函数的优势

减少代码重复，提高代码的可维护性。使程序结构更清晰，易于理解和调试。

## 5.函数的调用

通过函数名和参数列表调用函数。函数调用可以多次发生，每次调用可以使用不同的参数。

## 6.函数的注意事项

不要过度拆分函数，以免造成代码混乱和性能下降。

主函数（main）是特殊的，可以不返回值，其他函数必须返回指定类型的数据。

## 7.函数的声明和定义

函数声明通常放在头文件中，定义则在源文件（.cpp）中。

函数声明告诉编译器函数的存在和如何调用它，而定义包含实际执行的代码。

## 8.调试和发布模式

在调试模式下，编译器会检查函数是否正确返回值。

在发布模式下，编译器可能不会强制要求返回值，但这可能导致未定义行为。

# C++头文件

## 1.函数声明的重要性

函数声明允许编译器知道函数的存在，即使函数的定义在其他文件中。

这对于在不同源文件中调用函数是必要的，因为C++编译器需要在编译时知道函数的签名。

## 2.头文件的作用

头文件通常用于存放函数声明、类定义和其他需要在多个源文件中共享的声明。

通过包含头文件（使用#include指令），可以在多个源文件中使用相同的声明，避免重复代码。

## 3.避免重复声明的问题

如果在多个文件中重复声明同一个函数，编译器会报错，因为C++要求每个函数只能有一个定义。使用头文件可以集中管理函数声明，避免重复。

## 4.头文件保护

为了避免头文件被多次包含导致的重复声明问题，通常会使用预处理指令（如#ifndef、#define和#endif）来保护头文件。更推荐使用#pragma once，它是一种更现代的头文件保护方式。

这种机制确保头文件内容只被编译一次，即使它被多个源文件包含。

## 5.头文件和源文件的组织

通常，函数定义放在源文件中，而函数声明放在头文件中。

## 6.头文件的包含方式

视频解释了使用尖括号（< >）和双引号（" "）来包含头文件的区别，尖括号用于标准库头文件，双引号用于用户自定义的头文件。

## 7.C++标准库头文件

视频提到了C++标准库头文件通常不带.h扩展名，例如<iostream>而不是<iostream.h>，这是C++与C标准库的一个区别。

# 如何在Visual Studio中调试代码

## 1.调试的重要性

调试是编程中不可或缺的一部分，它帮助开发者理解程序是如何工作的，以及计算机是如何运行代码的。调试可以揭示程序中的错误（bugs），并帮助开发者诊断和解决问题。

## 2.断点的使用

断点是调试过程中的一个关键概念，它允许开发者在程序的特定行暂停执行。在Visual Studio中，可以通过按F9或点击代码行旁边的侧边栏来设置断点。断点设置后，当程序运行到该行时会暂停，允许开发者检查程序状态。

## 3.内存查看

在断点处，开发者可以查看程序的内存状态，包括变量的值和内存地址。这有助于理解程序在运行时的状态，特别是对于未初始化或错误设置的变量。

## 4.调试器控制

调试器提供了多种控制程序执行的选项，包括：

Step Into (F11)：进入当前函数内部，逐行执行代码。

Step Over (F10)：执行当前行代码，如果当前行包含函数调用，则执行整个函数而不进入其内部。

Step Out (Shift + F11)：执行完当前函数的所有剩余代码，并返回到调用该函数的地方。

5.调试窗口

Visual Studio提供了多个窗口来帮助调试，包括：

Autos 和 Locals：显示当前作用域内的变量。

Memory：查看程序的内存内容，包括内存地址和对应的值。

## 6.调试模式与发布模式

调试模式下，编译器会添加额外的信息以帮助调试，如未初始化的变量会用特定值填充，这在发布模式下不会发生。发布模式优化了程序性能，但不利于调试。

# C++条件与分支（if语句）

## 1.条件语句的重要性

在编程中，条件语句允许程序根据特定条件执行不同的代码路径。if语句是实现条件分支的一种基本方式。

## 2.if语句的工作原理

if语句评估一个条件表达式，如果条件为真（true），则执行if块内的代码；如果条件为假（false），则跳过该块。条件表达式的结果通常是布尔值（true或false），但也可以是任何可以被解释为真或假的值。

## 3.比较运算符

在C++中，比较运算符（如==）用于比较两个值。例如，x == 5会检查变量x是否等于5。

## 4.布尔值的表示

在C++中，布尔值true和false实际上是以整数形式表示的，其中true通常表示为1，false表示为0。

## 5.编译器优化

编译器在编译时会对代码进行优化，有时甚至可以消除不必要的条件检查，特别是当条件是常量时。

## 6.if语句的简写形式

如果if语句只包含一行代码，可以省略花括号。此外，if语句可以与else和else if结合使用，以处理多个条件分支。

## 7.指针和if语句

if语句常用于检查指针是否为nullptr（或C++中的NULL），以避免空指针解引用导致的程序崩溃。

## 8.调试和反汇编

反汇编(Disassembly)是将编译后的机器代码转换回汇编语言的过程，它允许开发者查看程序底层的指令。这对于理解程序如何在硬件级别执行非常有用，尤其是在调试复杂问题时。

1. 查看汇编代码：

在调试模式下，可以查看程序的汇编代码。这有助于理解编译器如何将高级语言代码转换为机器代码，并且可以用来分析性能瓶颈或调试难以追踪的问题。

1. 理解CPU指令：

通过反汇编视图，开发者可以看到实际的CPU指令，包括加载、存储、算术运算等操作。这有助于理解程序在执行时到底做了什么。

1. 优化和性能分析：

反汇编可以揭示编译器优化的效果，有时开发者可以据此手动优化代码，以提高性能。

# Visual Studio的最佳设置

## 1.项目和解决方案的创建

在Visual Studio中创建一个新项目时，选择在"C"目录下存储项目，以便于跨平台工作和路径管理。创建一个空的Visual C++项目，并命名为"新项目"。

## 2.项目结构

Visual Studio会自动生成解决方案文件和项目文件。项目文件（.vcxproj）是一个XML文件，而解决方案文件（.sln）是一种特殊格式的文本文件。项目文件夹内包含vcxproj文件和过滤器文件，这些过滤器用于在解决方案资源管理器中虚拟地组织文件，但并不实际存在于磁盘上。

## 3.源代码组织

建议创建一个名为"source"或"src"的文件夹来存放所有的源代码和头文件，以保持项目结构的清晰和有序。

## 4.文件和文件夹的管理

使用Visual Studio的"显示所有文件"按钮可以查看和管理实际的磁盘文件结构。可以将文件拖动到不同的文件夹中，以在Visual Studio中重新组织它们。

## 5.输出和中间文件的配置

通过项目属性设置输出目录和中间目录，通常将输出目录设置在解决方案目录下，便于管理多个项目和不同平台的构建输出。中间目录用于存放编译过程中的中间文件，如OBJ文件。

输出目录：$(SolutionDir)\bin\$(Platform)\$(Configuration)\

中间目录：$(SolutionDir)\bin\intermediates\$(Platform)\$(Configuration)\

## 6.构建和调试

构建项目后，Visual Studio会将生成的可执行文件和中间文件放在指定的目录中。调试和发布版本的输出通常放在不同的文件夹中，以区分不同构建配置。

## 7.清理和优化

在项目设置完成后，可以清理不再需要的文件和文件夹，如旧的调试目录，以保持项目目录的整洁。

# C++指针

## 1.指针的基本概念

指针：指针是一种变量，其值为内存地址。它用于存储和操作内存地址，允许程序员直接访问和修改内存中的数据。

内存地址：计算机内存可以被想象成一条线性排列的“房子”，每个“房子”都有一个唯一的地址，即内存地址。

## 2.指针的类型

void指针：没有特定类型的指针，可以指向任何类型的数据。它不关心存储的数据类型，只保存内存地址。

具体类型的指针：如int指针、char指针等，它们指向特定类型的数据。编译器使用这些信息来正确地读写内存。

## 3.指针的操作

初始化指针：创建指针变量时，可以将其初始化为特定的内存地址，或者使用nullptr（C++11引入）表示空指针，即不指向任何有效地址。

获取变量的地址：使用&运算符可以获取任何变量的内存地址，并将其赋值给指针。

解引用指针：使用\*运算符可以访问指针指向的内存地址中的数据。这称为解引用指针。

## 4.指针的使用场景

直接内存操作：通过指针可以实现对内存的直接读写，这对于需要精细控制内存的应用程序（如游戏开发）非常重要。

动态内存分配：使用new关键字可以动态地在堆上分配内存，并通过指针访问这些内存。使用完毕后，需要使用delete释放内存，避免内存泄漏。

## 5.指针的高级概念

多级指针：指针可以指向另一个指针，形成多级指针（如int\*\*），这在处理复杂数据结构时非常有用。

指针算术：指针可以进行算术运算，如递增（++）或递减（--），这通常用于遍历数组或内存块。

# C++引用

## 1.引用与指针的关系

引用本质上是别名：引用提供了一种方式来为已存在的变量创建别名，而不是创建一个新的变量。

引用不是变量：引用本身不占用内存空间，它们是对其他变量的引用。

引用与指针的区别：指针可以指向NULL或被重新指向另一个地址，而引用一旦初始化后，必须始终引用同一个对象。

## 2.引用的声明和使用

声明引用：声明引用时，需要在变量类型后加上&符号，并立即初始化，如int& ref = a;。

引用的使用：一旦声明，引用就可以像使用原变量一样使用，例如ref = 2;实际上修改的是a的值。

## 3.引用与函数参数

传递引用到函数：通过引用传递参数可以修改函数外部的变量，因为传递的是变量的引用而非其副本。

引用与指针的比较：虽然可以通过指针达到类似的效果，但使用引用可以使代码更简洁易读。

## 4.引用的限制

初始化必须：声明引用时必须立即初始化，因为引用必须始终指向一个有效的对象。

不能更改引用的对象：一旦引用被初始化，它就不能被重新指向另一个对象。

## 5.引用与指针的代码示例对比

通过值传递：函数接收参数的副本，不会影响原始变量。

通过引用传递：函数接收参数的引用，可以修改原始变量。

通过指针传递：函数接收参数的内存地址，通过解引用可以修改原始变量。

# C++类

## 1.类和面向对象编程（OOP）

面向对象编程（OOP）：是一种编程范式，它使用“对象”来表示数据和功能。OOP强调将数据和操作数据的函数封装在一起，以提高代码的模块化和可重用性。

类：在C++中，类是创建对象的蓝图或模板。类定义了一组属性（数据）和方法（函数），这些属性和方法共同构成了对象的状态和行为。

## 2.类的定义和使用

定义类：使用关键字class后跟类名来定义一个类。类的定义包括属性（变量）和方法（函数）。

创建对象：通过类定义创建的对象称为实例。创建对象时，需要指定类名和对象名。

访问控制：类中的成员（属性和方法）可以设置为public（公有）或private（私有）。公有成员可以从类的外部访问，而私有成员只能在类的内部访问。

## 3.类的实例化和方法

实例化：创建类的实例（对象）的过程称为实例化。

方法：类中的函数称为方法。方法可以操作类的属性，并提供与对象交互的接口。

## 4.类的优势

封装：类将数据和操作数据的函数封装在一起，有助于隐藏实现细节，只暴露必要的接口。

代码组织：类有助于组织代码，使代码更加清晰和易于维护。

代码重用：通过继承和多态等面向对象的特性，可以创建可重用的代码模块。

# C++类与结构体对比

## 1.类和结构体的区别

默认可见性：在C++中，类的成员默认是私有的（private），而结构体的成员默认是公有的（public）。

使用场景：类通常用于定义具有复杂行为和数据封装的对象，而结构体则用于表示简单的数据集合。

继承：类和结构体都支持继承。结构体通常用于简单的数据表示，而类则更常用于构建复杂的系统和层次结构。

## 2.结构体的使用

数据结构表示：结构体适合表示简单的数据结构，提供清晰的语义，如数学向量或点。

避免复杂性：由于结构体不强制实现复杂的行为，因此在不需要继承或封装的情况下，它们有助于保持代码简洁。

## 3.类的使用

复杂对象表示：类用于创建具有复杂行为和属性的对象，适合构建复杂的应用程序，如游戏中的玩家、游戏世界等。

封装和继承：类支持封装和继承，适合构建复杂的系统和层次结构，有助于构建灵活且可扩展的系统。

## 4.个人编程风格

偏好使用结构体：作者倾向于使用结构体来表示简单的数据集合，而使用类来表示具有复杂行为的对象。

避免混合使用：建议避免将类和结构体混合使用，以保持代码的清晰和一致性。

# 如何写一个C++类

## 1.目标

目标是创建一个基本的日志类，用于打印消息到控制台，用于调试目的。

## 2.日志类的目的

用于管理日志消息，帮助开发者了解程序运行状态。

可以很简单，也可以很复杂，但对调试和开发非常重要。

## 3.日志类的复杂性

可以简单到只有几行代码，也可以复杂到几千行代码。

功能可以包括打印到控制台、不同颜色、输出到文件或通过网络。

## 4.日志级别的概念

有三个日志级别：错误（Error）、警告（Warning）、信息（Info）。

可以通过设置日志级别来控制打印哪些消息。

## 5.类的设计

通过实例化和使用日志类来设计API。

包括设置日志级别的函数和记录不同类型消息的函数。

## 6.代码实现

创建日志类，定义日志级别。

实现设置日志级别的函数。

实现记录不同类型消息的函数。

## 7.信息打印

使用std::cout来打印消息。

使用if语句来根据日志级别决定是否打印消息。

# C++中的静态（static）

## 1.类或结构体外部的static

当static用在类或结构体外部时，它表示该变量或函数的链接性是内部的，仅在定义它的翻译单元（Translation Unit）中可见。这意味着它们不会被其他编译单元看到。

如果你在另一个翻译单元中定义了同名的全局变量，编译时不会有问题，但在链接阶段会出现重复定义的错误。

要解决这个问题，可以使用extern关键字来声明变量，这样链接器会在其他翻译单元中寻找该变量的定义。

## 2.类或结构体内部的static

当static用于类的成员变量或方法时，意味着这些成员属于类本身，而不是类的任何特定实例。所有类的实例共享同一个static成员变量，即内存中只有一份拷贝（副本）。

static方法不能访问非静态成员变量或方法，因为它们不依赖于类的任何实例。

对于静态方法，调用时不会传递类的实例。

在不需要全局访问的情况下，应尽量使用static关键字，以避免潜在的全局变量问题。全局变量虽然方便，但可能导致难以追踪的错误和冲突。

# C++类和结构体中的静态（static）

## 1.静态变量（Static Variables）

当在类或结构体中声明一个静态变量时，意味着这个变量在所有类的实例之间共享。也就是说，无论创建多少个类的实例，都只会有一个该变量的副本。

如果一个实例改变了这个静态变量的值，那么这个改变会影响到所有其他的实例，因为它们都指向同一个内存地址。

## 2.静态方法（Static Methods）

静态方法可以不通过类的实例来调用，意味着你不需要创建一个类的对象就可以使用这个方法。

静态方法不能访问非静态成员变量，因为静态方法没有类的实例上下文。

## 3.使用场景

当需要在多个实例之间共享数据时，静态变量非常有用。例如，可以将与所有实体实例共享的数据存储在实体类中。

静态方法适用于那些不需要访问类实例状态的操作。

# C++中的局部静态（local Static）

## 1.静态局部变量（Static Local Variables）

在函数中声明静态局部变量意味着该变量的生命周期贯穿整个程序运行期间，但其作用域仍然限制在声明它的函数内。

与普通局部变量不同，静态局部变量不会在函数调用结束时被销毁，而是保持其值直到下一次函数调用。

## 2.静态局部变量与全局变量的区别：

虽然静态局部变量的生命周期与全局变量类似，但其作用域仅限于声明它的函数内，这提供了更好的封装性和避免全局命名空间的污染。

## 3.使用场景：

静态局部变量适用于需要在函数调用间保持状态或计数的场景。

使用静态局部变量来实现单例模式（Singleton Pattern），即确保一个类只有一个实例，并提供一个全局访问点。

## 4.单例模式的实现：

在函数中创建并返回类的唯一实例。

这种方法避免了全局变量的使用，同时确保了类的实例化逻辑只执行一次。

# C++枚举（Enumerations，简称Enum）

## 1.枚举（Enum）的定义

枚举是一种数据类型，它包含一组命名的值，通常用于表示一组相关的常量。

枚举允许为一组值赋予有意义的名称，从而提高代码的可读性和可维护性。

## 2.枚举与整数的比较

与直接使用整数相比，枚举可以限制变量只能取预定义的几个值，避免了错误的赋值。

枚举的每个值都有一个默认的整数表示，从0开始，也可以自定义每个枚举值的整数。

## 3.枚举的限制和优势

枚举类型限制了变量可以取的值，使得代码更加安全。

枚举值可以被赋予有意义的名称，使得代码更易于理解和维护。

## 4.枚举在实际编程中的应用

例如在日志记录系统中使用枚举来表示不同的日志级别。

枚举可以与函数结合使用，通过函数参数传递枚举值来执行特定的操作。

## 5.枚举的命名冲突问题

当函数名与枚举值相同时，会导致编译错误。

解决方法是使用命名空间或为枚举值添加前缀，以区分枚举值和函数名。

# C++构造函数（Constructors）

## 1.构造函数的定义

构造函数是一种特殊类型的方法，每次创建对象时自动调用。

它没有返回类型，并且名称必须与类名相同。

## 2.构造函数的使用

通过定义构造函数，可以在创建对象时自动初始化成员变量，例如将值设置为零。

构造函数可以重载，即可以定义多个构造函数，只要它们的参数列表不同。

如果不手动初始化对象的成员变量，它们将包含内存中残留的随机值。

## 3.构造函数与默认行为

如果没有定义构造函数，某些语言会提供默认的构造函数，但C++不会自动初始化非静态成员变量。

对于基本类型，如整数或浮点数，某些语言会自动初始化为零，但C++不会。

## 4.构造函数的限制

如果类中只包含静态方法，可能不希望允许创建类的实例。

为此，可以将构造函数声明为私有（private），或者使用delete关键字显式删除默认构造函数。

另外还有一些特殊构造函数，如复制构造函数和移动构造函数。

# C++析构函数（Destructors）

## 1.构造函数（Constructors）

构造函数是一种特殊的方法，当创建对象时自动调用。

它用于初始化对象的状态，例如设置成员变量的初始值。

构造函数可以重载，即可以定义多个具有不同参数列表的构造函数。

## 2.析构函数（Destructors）

析构函数与构造函数相对应，当对象生命周期结束时被调用。

它用于执行清理工作，如释放分配的资源或执行其他必要的清理操作。

析构函数的名称是在类名前加上波浪号（~）。

## 3.析构函数的特殊用法

析构函数通常用于释放对象在构造函数中分配的资源，如动态分配的内存。

如果对象是自动存储期（栈对象），析构函数会在对象生命周期结束时自动调用。

对于动态分配的对象，析构函数会在使用delete操作符时被调用。

## 4.构造函数和析构函数的注意事项

应正确使用构造函数和析构函数，以避免资源泄露和其他潜在问题。

构造函数和析构函数的正确使用有助于维护代码的健壮性和可读性。

# C++继承（Inheritance）

## 1.继承的概念

继承是面向对象编程的核心概念之一，允许创建一个类（子类）继承另一个类（父类）的属性和方法。

通过继承，子类可以复用父类的代码，减少代码重复，提高开发效率。

## 2.继承的用途

主要用于将共通的功能放在一个基类（父类）中，然后通过子类继承并扩展或修改这些功能。这种机制有助于避免在多个类中重复编写相同的代码，使得代码更加简洁和易于维护。

## 3.继承的实现

通过一个游戏开发的例子来说明继承的实现，其中定义了一个基类Entity，包含所有实体共有的属性和方法，如位置信息和移动方法。

然后创建了一个Player类，继承自Entity类，并添加了额外的属性和方法，如玩家的名字和特定的行为。

通过继承，Player类自动获得了Entity类的所有属性和方法，同时可以添加自己特有的功能。

## 4.多态性（Polymorphism）

多态性是继承的一个重要特性，允许使用父类类型的引用来指向子类对象。

这意味着可以在需要父类对象的地方使用子类对象，提供了灵活性和扩展性。

## 5.内存大小和继承

继承对类内存大小的影响：子类对象的大小会包含父类的所有成员变量。

举例来说，如果 Entity 类有两个浮点数成员，Player 类有一个浮点数成员，那么继承自 Entity 的 Player 类的大小将等于 Entity 类的大小（4+4）加上 Player 类自己的成员变量的大小（4），即12字节。

# C++虚函数（virtual functions）

## 1.虚函数的重要性

虚函数对于实现多态性至关重要。多态性允许我们通过基类的指针或引用来操作派生类的对象，从而执行相应派生类的方法。

如果不使用虚函数，通过基类指针调用方法时，将总是调用基类中的方法，而不是派生类中重写的方法。

## 2.虚函数的使用

在基类中声明函数时，使用 virtual 关键字，这样派生类中同名函数就会覆盖基类中的函数。

使用 override 关键字在派生类中声明覆盖的函数，这有助于编译器检查是否正确覆盖了基类中的虚拟函数。

## 3.实现细节

虚函数通过虚函数表（vtable）实现，该表存储了指向类中虚函数的指针。

当调用虚函数时，程序会通过vtable查找并调用正确的函数实现，这个过程称为动态绑定。

## 4.注意事项

使用虚函数会引入额外的运行时开销，因为需要通过vtable来解析函数调用。

正确使用 override 关键字可以避免一些常见的错误，如拼写错误导致的未覆盖函数。

## 5.总结

虚函数是实现多态性的基础，允许在运行时根据对象的实际类型来调用相应的方法。正确使用虚函数和 override 关键字可以提高代码的可读性和健壮性。

# C++纯虚函数（pure virtual functions）、

## 1.纯虚函数和接口

纯虚函数是一种特殊的虚拟函数，它在基类中没有实现（即函数体为空），并且必须在派生类中被实现。

接口在C++中通常指的是一种只包含纯虚函数的类，它定义了一组方法，但不提供这些方法的具体实现。

## 2.接口的使用场景

有时候，我们希望创建一个类，它只包含方法的声明而不包含实现，强制派生类提供这些方法的具体实现。

这种设计模式常见于需要确保派生类遵循特定接口的情况，例如，要求所有派生类都必须实现一个特定的方法。

## 3.纯虚函数的声明

在基类中声明纯虚函数时，需要使用 virtual 关键字，并在函数声明后加上 = 0。

这样做意味着基类不能被实例化，只有实现了所有纯虚函数的派生类才能被实例化。

## 4.接口的实际应用

通过接口，可以创建通用函数，这些函数可以接受任何实现了接口的类的对象，并调用接口中定义的方法。

这种方式增加了代码的灵活性和可重用性。

## 5.总结

纯虚函数和接口是面向对象编程中实现多态性和抽象的关键概念。

它们定义一组方法规范，确保派生类遵循这些规范，同时提供具体的实现。

接口在设计模式中非常有用，特别是在需要确保类具有特定功能时。

# C++可见性

## 1.访问控制的概念

访问控制（visibility）是面向对象编程中的一个概念，它决定了类成员（变量和方法）的可见性和可访问性。

访问控制不会影响程序的运行性能，它主要是为了帮助开发者编写更清晰、易于维护的代码。

## 2.私有（Private）

私有成员只能在定义它们的类内部访问。

通过使用 friend 关键字，可以允许特定的类或函数访问私有成员。

## 3.受保护（Protected）

受保护成员在基类和派生类中都是可见的，但对类外不可见。

这种访问级别允许派生类访问基类的成员，阻止了外部代码的访问。

## 4.公有（Public）

公有成员对所有代码都是可见的，包括类的外部。

公有接口是类与外界交互的主要方式。

## 5.访问控制的重要性

使用访问控制可以保护类的内部状态，防止外部代码直接修改类的私有成员，从而降低错误和潜在的破坏。

通过指定明确哪些成员是公有的，哪些是私有的，可以清晰地指示如何正确使用类。

## 6.编程风格和实践

建议不要将所有成员都设置为公有。

根据实际需要合理使用私有和公有成员，以提高代码的可维护性和可读性。

# C++数组

## 1.数组基础

数组是元素的集合，通常包含相同类型的变量。

在C++中，数组的索引从0开始，例如第一个元素的索引是0，第二个1，依此类推。

数组的声明需要指定类型和大小，例如 int ptr[5];声明了个包含5个整数的数组。

## 2.数组与指针

数组名可以被视为指向数组第一个元素的指针。

通过指针可以访问数组中的元素，例如 \*ptr + 2 实际上访问的是数组的第三个元素。

## 3.数组的内存管理

数组在内存中是连续存储的，每个元素占用固定大小的空间（例如，整数占用4字节）。

使用 new 关键字可以在堆上动态创建数组，需要使用 delete[] 来释放内存。

## 4.数组的边界检查

C++标准数组不提供边界检查，因此访问数组边界之外的元素会导致未定义行为。

C++11引入的 std::array 提供了边界检查和大小跟踪，但使用时会有一定的性能开销。

## 5.数组的使用场景

当需要处理一组数据时，使用数组可以避免创建大量单独的变量。

在函数中返回数组时，需要使用 new 关键字动态分配内存。

## 6.数组的大小

在栈上分配的数组可以通过计算内存大小除以单个元素的大小来确定数组的元素数量。

在堆上分配的数组则需要手动管理大小信息，因为编译器不会自动跟踪。

## 7.安全性和性能权衡

使用标准数组（如 std::array）比原始数组更安全，但可能牺牲一些性能。

原始数组在性能上可能更优，但需要程序员自己管理边界和内存。

## 8.资源管理

动态分配的数组需要程序员负责释放内存，否则会导致内存泄漏。

使用智能指针如 std::unique\_ptr 或 std::shared\_ptr 可以帮助自动管理内存。

## 9.总结

建议在可能的情况下使用标准数组来提高代码的安全性和可维护性。

数组使用中可能遇到的内存管理和边界检查的问题。

# C++字符串

## 1.字符串基础

字符串是由字符组成的文本，可以是单个字符、单词、句子或段落。

在C++中，字符串通常通过字符数组来表示，也可以使用标准库中的std::string类。

## 2.字符和字符串的关系

字符是构成字符串的基本单位，可以是字母、数字、符号等。

C++中的char类型用于表示单个字符，而字符串则是char类型的数组。

## 3.C++中的字符串处理

C++标准库提供了std::string类，用于方便地处理字符串。

std::string类提供了许多成员函数，如size()、append()、find()等，用于获取字符串长度、拼接字符串、查找子字符串等操作。

## 4.字符串的内存表示

字符串在内存中以字符数组的形式存储，以空字符（null terminator）'\0'标识字符串的结束。

当使用std::string时，内存管理（如分配和释放内存）由类自动处理，无需手动操作。

## 5.字符串与字符指针

字符串字面量（如"Hello"）在C++中是常量字符数组，可以被隐式转换为const char\*类型。使用std::string可以避免直接处理字符指针和手动内存管理的复杂性。

## 6.字符串的传递和复制

将字符串作为参数传递给函数时，推荐使用常量引用（const std::string&），以避免不必要的复制。

直接传递std::string对象会创建副本，而使用引用则不会。

## 7.字符串的使用场景

std::string适用于大多数需要字符串处理的场景，特别是当涉及到动态文本处理时。

对于性能敏感的应用，了解字符串的内部实现和内存管理是很有帮助的。

# C++字符串字面量

## 1.字符串字面量

字符串字面量是由双引号括起来的一系列字符，例如 "Cherno"。

字符串字面量在内存中以字符数组的形式存储，并以空字符（null terminator）'\0'结尾，表示字符串的结束。

## 2.字符类型

char 类型用于表示单个字符，如字母、数字、符号等。

C++11引入了宽字符类型，如 wchar\_t、char16\_t 和 char32\_t，用于支持更广泛的字符集，如UTF-16 和 UTF-32。

## 3.字符串的内存表示

字符串字面量通常存储在程序的只读数据段中。

字符串的长度计算不包括结尾的空字符。

## 4.字符串处理

使用标准库中的 std::string 类可以更方便地处理字符串，包括字符串的拼接、查找、替换等操作。

字符串的传递和复制应当使用常量引用以避免不必要的性能开销。

## 5.字符串字面量的高级特性

C++14 引入了 std::string\_view，它提供了一种不拥有数据但可以查看字符串的方式。

字符串字面量前可以使用 u、U、L 或 u8 前缀来指定字符集，如 u"Hello" 表示使用 UTF-16 编码的字符串。

## 6.字符串的内存操作

直接操作字符串字面量的内存是未定义行为，因为它们位于只读内存段。

字符串字面量的修改尝试会导致运行时错误或异常。

## 7.字符串的优化和性能

字符串的处理应当考虑性能，避免不必要的复制和内存分配。

使用 std::string 可以自动管理内存，减少内存泄漏的风险。

# C++中的const

## 1.const关键字基础

const用于声明变量为常量，即其值在程序运行期间不可改变。

const可以用于指针，有两种形式：const int\* ptr（指针指向的值不可变）和int\* const ptr（指针本身不可变）。

## 2.const与类成员函数

在类的成员函数声明中，const放在函数参数列表之后，表示该函数不会修改类的任何成员变量。

const成员函数可以被const对象调用，而非常量成员函数则不能。

使用mutable关键字可以允许const成员函数修改某些特定的成员变量。

## 3.const的其他用途

const可以用于类的实例，表示该实例不可变。

const可以用于引用，表示引用的对象不可变。

const可以用于指针，表示指针指向的内存区域不可变。

## 4.const与代码优化

使用const可以提高代码的可读性和安全性，因为它明确告诉编译器和阅读代码的人哪些值是不可变的。

const还可以帮助编译器进行优化，如编译器可以将const变量存储在只读内存段中。

## 5.const与指针

当const放在指针声明的星号前时，表示指针指向的值不可变。

当const放在指针声明的星号后时，表示指针本身不可变，即不能指向其他地址。

## 6.const与类成员变量

在类中，const成员变量必须在构造函数初始化列表中初始化，一旦初始化后不可更改。

## 7.const与函数重载

const可以用于区分函数重载，例如，可以有void func()和void func() const两个版本，分别用于修改和不修改对象状态的场景。

## 8.const与引用

const引用可以绑定到临时对象或右值上，而非常量引用则不能。

## 9.const与数组

const可以用于数组，表示数组中的元素不可变。

## 10.const与指针的组合使用

可以组合使用const和指针，例如const int\* const ptr表示指针本身和指向的值都不可变。

# C++中的mutable关键字

## 1.可变关键字与常量

在C++中，可变关键字（mutable）允许在常量成员函数中修改类的某些成员变量。

这意味着即使对象被声明为常量（const），使用可变关键字标记的成员变量仍然可以被修改。

例如，如果有一个类成员变量需要在调试过程中被修改（如记录函数调用次数），即使整个对象是常量，也可以通过可变关键字来实现。

## 2.可变关键字与Lambda表达式

Lambda表达式是C++中用于创建匿名函数对象的一种方式。

当Lambda表达式通过值捕获外部变量时，默认情况下这些变量是不可修改的。

如果需要在Lambda表达式内部修改这些通过值捕获的变量，可以使用可变（mutable）关键字来声明Lambda表达式，允许修改这些变量的副本。

# C++中的成员初始化列表

## 1.成员初始化列表的概念

成员初始化列表是一种初始化类成员的方式，它在构造函数中使用。

它允许在构造函数体执行之前初始化类成员。

## 2.初始化方法

传统方法是在构造函数体内使用赋值操作符（=）来初始化成员。

成员初始化列表使用冒号（:）后跟成员名称和初始化值。

## 3.初始化顺序

成员初始化列表中的初始化顺序应与类成员声明的顺序一致。

不遵循此顺序可能导致依赖性问题。

## 4.性能优势

使用成员初始化列表可以避免不必要的对象构造和析构，从而提高性能。

例如，如果在构造函数体内先默认构造一个对象，然后赋值，实际上进行了两次构造和一次析构。

## 5.代码风格

使用成员初始化列表可以使构造函数体更简洁，代码更易读。

建议在所有情况下使用成员初始化列表，包括原始类型和类类型。

# C++中的三元运算符

## 1.三元运算符简介

三元运算符是C++中的一种条件运算符，使用问号（?）和冒号（:）表示。

它可以看作是if-else语句的简写形式。

## 2.基本用法

三元运算符的基本形式为 条件 ? 值1 : 值2。

如果条件为真，则表达式的结果为值1；如果条件为假，则结果为值2。

## 3.性能优势

三元运算符可以减少代码量，使代码更易读。

在某些情况下，使用三元运算符可以提高代码的执行效率。

## 4.嵌套使用

三元运算符可以嵌套使用，但应谨慎使用，以避免代码过于复杂。

# 创建并初始化C++对象

## 1.对象创建的两种方式

栈（Stack）：对象在函数内部创建，具有自动寿命，当变量超出作用域时自动销毁。

堆（Heap）：对象在堆上创建，需要手动管理内存，使用new关键字分配，使用delete关键字释放。

## 2.栈对象

具有自动寿命，由声明范围控制。内存超出作用域时自动释放。

适用于对象生命周期短且不需要在函数外访问的情况。

## 3.堆对象

需要手动管理内存，使用new和delete。

适用于对象生命周期长或需要在函数外访问的情况。

适用于对象大小较大或需要创建大量对象的情况。

## 4.栈和堆的比较

栈通常较小，适合存储小型对象。堆较大，适合存储大型对象或大量对象。

## 5.智能指针

用于自动管理堆上的对象，避免内存泄漏。

例如，shared\_ptr和unique\_ptr可以自动释放对象。

# C++new关键字

## 1.new关键字的作用

new关键字用于在堆上分配内存。

它不仅分配内存，还会调用构造函数来初始化对象。

与malloc函数相比，new不仅分配内存，还负责调用构造函数。

## 2.栈和堆的区别

栈（Stack）上的对象具有自动寿命，当变量超出作用域时自动销毁。

堆（Heap）上的对象需要手动管理内存，使用new和delete进行分配和释放。

## 3.内存分配的细节

new关键字在分配内存时，会先检查是否有足够的连续内存块。

如果没有足够的连续内存，它会使用“空闲列表”来维护可用的内存地址。

new关键字返回一个指向分配内存的指针。

## 4.内存释放

使用new关键字分配的内存必须使用delete关键字来释放。

如果忘记释放内存，会导致内存泄漏。

# C++隐式转换与explicit关键字

## 1.隐式构造

隐式构造指的是编译器自动将一种数据类型转换为另一种数据类型的过程。

C++允许编译器执行隐式转换，例如将整数转换为字符串，或者将字符串转换为自定义类对象。

这种转换通常发生在构造函数可以接受转换的类型作为参数时。

## 2.显式关键字

explicit关键字用于防止构造函数进行隐式转换。

当构造函数前加上explicit关键字时，该构造函数只能通过显式调用进行对象构造，不能自动转换。

这有助于避免意外的类型转换，提高代码的清晰度和安全性。

## 3.使用场景

explicit关键字在数学库或低级编程中特别有用，可以防止意外的类型转换，避免性能问题或错误。

在某些情况下，显式构造可以提高代码的安全性和可读性。

# C++运算符及其重载

## 1.运算符基础

运算符定义：运算符是用于执行操作的符号，如加号、减号等。

常见运算符：除了数学运算符，还有其他常用的运算符，如解引用运算符、箭头运算符、取地址运算符、左移运算符等。

运算符类型：包括算术运算符、关系运算符、逻辑运算符、位运算符等。

## 2.运算符重载

运算符重载含义：允许给运算符赋予新的含义或改变其行为。

支持情况：Java不支持运算符重载，C#部分支持，而C++提供了完全控制。

优缺点：虽然提供了灵活性，但也可能导致代码难以理解，因此应谨慎使用。

## 3.运算符重载注意事项

代码可读性：过度使用运算符重载可能导致代码难以理解，应避免。

使用场景：建议仅在确实提高代码可读性和简洁性时使用运算符重载。

函数与运算符：运算符本质上是函数，但使用运算符可以使代码更直观。

## 4.结论

选择性使用：在创建库时，可以提供运算符重载和普通函数两种方式，让用户选择。

代码风格：应避免过度使用运算符重载，以免降低代码的可读性。

# C++中的this关键字

## 1.this关键字概述

定义：this是一个指向当前对象的指针，只能在类的成员函数中使用。

用途：用于区分成员变量和局部变量，或者在成员函数中访问其他成员函数。

## 2.this关键字的使用场景

区分成员变量和局部变量：当成员变量和局部变量同名时，使用this指针来明确指定成员变量。

在构造函数中赋值：在构造函数中，可以使用this指针来为成员变量赋值。

在非静态成员函数中使用：this指针在非静态成员函数中是有效的，可以用于访问对象的其他成员。

## 3.this指针的类型

类型：this是一个指向当前对象的指针，其类型是ClassName\*。

常量成员函数：在常量成员函数中，this指针是const ClassName\*类型，表示不能通过this修改对象。

## 4.this指针的使用示例

赋值操作：使用this->x = x;或this->x = this->x;来为成员变量赋值。

调用其他成员函数：可以使用this->functionName();来调用当前对象的其他成员函数。

## 5.特殊用法

删除对象自身：可以使用delete this;来删除当前对象，但这种做法非常危险，通常不推荐使用。

传递给其他函数：可以将this指针传递给其他函数，如printEntity(this);。

# C++中的对象生存期（栈作用域生存期）

## 1.对象生命周期和栈

栈上对象：在C++中，栈上对象的生命周期与它们所在的代码块（作用域）直接相关。

作用域：函数、if语句、循环等都可以是作用域。

生命周期：当进入一个作用域时，会将一个栈帧（stack frame）推入栈中，当退出作用域时，栈帧被移除，栈上的对象随之被销毁。

## 2.栈上对象的自动销毁

自动销毁：栈上的对象不需要手动删除，它们会在作用域结束时自动被销毁。

好处：这种自动销毁机制简化了内存管理，减少了内存泄漏的风险。

## 3.错误示例：返回栈上对象的指针

错误示例：尝试返回一个栈上对象的指针，当函数结束时，栈上的对象被销毁，返回的指针将指向无效内存。

正确做法：如果需要返回对象的指针，应使用堆分配（new）或复制对象到堆上。

## 4.使用栈上对象的正确示例

示例：使用ScopedPointer类，该类在构造时自动在堆上分配对象，在析构时自动删除对象。

智能指针：ScopedPointer类似于智能指针，可以自动管理堆上的对象。

## 5.其他使用栈上对象的例子

计时器：创建一个计时器类，用于自动计时代码段的执行时间。

互斥锁：创建一个自动锁定和解锁互斥锁的类，用于线程同步。

# C++的智能指针

## 1.智能指针概述

智能指针：是C++中用于自动管理内存的工具，可以自动释放分配的内存，减少内存泄漏的风险。

智能指针类型：包括unique\_ptr和shared\_ptr。

## 2.unique\_ptr

定义：unique\_ptr是一种作用域指针，当指针超出作用域时，它会自动释放所管理的内存。

特点：不能复制，只能移动，保证了内存的唯一所有权。

使用场景：适用于不需要共享内存所有权的情况。

异常安全：由于其作用域特性，unique\_ptr在异常发生时可以保证不会造成内存泄漏。

## 3.shared\_ptr

定义：shared\_ptr通过引用计数机制管理内存，当最后一个shared\_ptr被销毁时，内存被释放。

特点：可以复制和共享，适用于需要共享内存所有权的情况。

开销：由于引用计数机制，shared\_ptr有一定的性能开销。

使用场景：适用于需要多个对象共享同一内存的情况。

## 4.shared\_ptr的实现

引用计数：shared\_ptr通过引用计数来跟踪有多少个shared\_ptr实例指向同一内存。

控制块：shared\_ptr需要分配一个额外的内存块（控制块）来存储引用计数。

## 5.weak\_ptr

定义：weak\_ptr是shared\_ptr的补充，不增加引用计数，用于访问shared\_ptr管理的对象，但不阻止对象被销毁。

使用场景：适用于需要访问共享对象但不增加引用计数的场景，如缓存。

## 6.智能指针的选择

首选unique\_ptr：由于其作用域特性，unique\_ptr通常更推荐使用，因为它更简单且没有额外的性能开销。

shared\_ptr的使用：当需要共享内存所有权时使用shared\_ptr，但要注意其性能开销。

# C++的复制与拷贝构造函数

## 1.复制的概念

复制：复制是指复制数据，包括内存的复制，或者从一个地方到另一个地方的数据复制。

复制的目的：复制可以用于修改对象，或者在不修改原对象的情况下读取值。

## 2.复制的影响

性能影响：复制需要时间，不必要的复制会浪费性能。

避免不必要的复制：了解复制在C++中的工作原理，以及如何避免不必要的复制。

## 3.复制构造函数

定义：复制构造函数是当一个对象被复制时调用的构造函数。

默认行为：C++默认提供一个复制构造函数，它执行浅复制，即复制指针而不是指针指向的内容。

深度复制：深度复制需要完整复制对象，包括指针指向的内容。

## 4.深度复制的实现

自定义复制构造函数：可以通过编写自定义的复制构造函数来实现深度复制。

复制构造函数的签名：复制构造函数的签名是接受对同一类的常量引用的构造函数。

5.复制操作的注意事项

避免不必要的复制：通过引用传递对象可以减少不必要的复制。

通过常量引用传递：总是通过常量引用传递对象，除非函数本身需要修改对象。

# C++的箭头操作符

## 1.箭头运算符的基本用途

用于访问指向结构体或类的指针的成员。

如果有一个指向实体的指针，不能直接使用点操作符（.）来访问成员，因为这会导致编译错误。

为了访问成员，需要先对指针进行解引用（使用\*），然后使用点操作符（例如：(\*ptr).member）。

箭头运算符提供了一种更简洁的替代方式（例如：ptr->member）。

## 2.箭头运算符的重载

在C++中，箭头运算符可以被重载，允许在自定义类中使用。

通过重载，可以使得自定义类型的指针或引用能够使用箭头操作符访问成员。

例如，可以重载箭头运算符来实现一个智能指针类，该类在对象超出作用域时自动删除对象。

## 3.使用箭头运算符获取成员的偏移量

通过将指针转换为整数类型，可以使用箭头运算符来获取成员变量在内存中的偏移量。

这种方法可以用于计算数据序列化时的偏移量，或者在图形编程和游戏引擎开发中处理字节流。

# C++的动态数组

## 1. vector 的基本概念

vector 是C++标准模板库（STL）中的一个容器，用于存储元素的动态数组。

它允许动态地添加和删除元素，而不需要预先指定数组的大小。

vector 的大小可以随时改变，它会根据需要自动调整。

## 2. vector 的使用

创建 vector 时，需要指定存储的元素类型。

可以使用 push\_back 方法向 vector 添加元素。

可以通过索引操作符 [] 或 at 方法访问 vector 中的元素。

使用 size() 方法可以获取 vector 的当前大小。

## 3. vector 的性能和优化

vector 在扩容时会创建一个新的更大的数组，并将旧数组的内容复制到新数组中，这可能会影响性能。

为了避免不必要的复制，建议使用引用传递来传递 vector，特别是当 vector 中存储的是大型对象时。

# C++的std vector使用优化

## 1.vector 的复制问题

在使用 vector 时，对象复制可能导致的性能问题。

通过添加复制构造函数和调试，发现 vector 在添加元素时进行了多次不必要的复制。

这种复制发生在将对象从 main 函数复制到 vector 内部存储时。

## 2.优化策略

优化策略一： 在 vector 分配的内存中直接构造对象，避免从 main 函数复制到 vector 内部。

优化策略二： 在创建 vector 时，预先指定足够的容量，避免多次调整大小和复制。

通过这些优化，可以显著减少不必要的对象复制，提高程序性能。

## 3.实现优化

通过使用 vector::reserve 方法预先分配足够的内存，避免了多次调整大小和复制。

通过直接在 vector 内部构造对象，避免了从 main 函数复制对象。

# C++中使用库（静态链接）

## 1.外部库的使用

库文件的结构：通常包含两部分，一个是包含文件（头文件），另一个是库文件（二进制文件）。

头文件：提供函数声明，告诉编译器哪些函数可用。

库文件：提供函数定义，允许链接器将函数调用链接到实际的函数实现。

## 2.添加和链接库文件

添加头文件：在Visual Studio中，通过修改项目的附加包含目录来指定头文件的位置。

链接库文件：在链接器设置中指定库文件的位置和名称。

## 3.静态库与动态库

静态库：在编译时链接到可执行文件中，生成的可执行文件较大。

动态库：在运行时链接，允许共享库文件，生成的可执行文件较小。

## 4.注意事项

路径问题：确保指定的路径正确，避免因路径错误导致编译失败。

函数声明与定义：确保头文件中声明的函数与库文件中的定义匹配。

C与C++混合使用：如果库是用C语言编写的，需要在C++代码中使用 extern "C" 来避免名称修饰问题。

# C++中使用动态库

## 1.动态链接基础

定义：动态链接是在运行时发生的链接，与编译时发生的静态链接相对。

区别：静态链接将库内容直接编译进可执行文件或动态库中，而动态链接在运行时加载外部库文件。

优化：静态链接允许编译器和链接器进行更多优化，因为它们完全了解应用程序代码。

## 2.动态链接的使用

步骤：

1.包含头文件（与静态链接相同）。

2.在链接器设置中指定动态库文件（.dll 或 .so 文件）。

3.确保动态库文件在可执行文件的搜索路径中。

## 3.动态链接与静态链接的比较

静态链接：编译时链接，生成的可执行文件较大，但运行时不需要额外的库文件。

动态链接：运行时链接，可执行文件较小，但需要确保动态库文件可用。

## 4.动态链接的应用场景

静态链接：适用于库文件大小较小，且库文件与应用程序紧密集成的情况。

动态链接：适用于库文件较大，或需要在运行时动态加载和卸载库文件的情况。

# C++中创建与使用库

## 1.项目设置

在解决方案中添加两个项目：一个作为可执行文件项目，另一个作为库项目。

## 2.项目配置

可执行文件项目配置为应用程序类型。

库项目配置为静态库类型。

## 3.代码编写

为可执行文件项目创建一个应用程序CPP文件，作为应用程序的基本来源。

为库项目创建一个头文件和相对应的CPP文件。

## 4.项目链接

在可执行文件项目中包含库项目的头文件，使用相对路径和绝对路径两种方法。

使用Visual Studio的引用功能将可执行文件项目链接到库项目，确保可执行文件项目依赖于库项目。

## 5.总结

在Visual Studio中可以设置多个项目和创建可重用库。

通过静态链接，可以将所有依赖项包含在最终的可执行文件中，简化了部署和分发过程。

# C++中如何处理多返回值

## 解决方案

结构体方法：创建一个自定义结构体，包含所有需要返回的变量。这种方法的优点是代码清晰易读，变量命名明确。

元组方法：使用C++标准库中的std::tuple来返回多个值。这种方法的优点是灵活，可以返回不同类型的值，但缺点是代码风格不佳，难以理解。

配对方法：使用std::pair来返回两个值。这种方法比元组方法稍微好一些，但仍然存在命名不明确的问题。

推荐使用结构体方法，因为它提供了最好的代码清晰度和可读性。

通过结构体，可以为每个返回值命名，避免混淆。

# C++的模板

## 1.模板概念

模板类似于其他语言中的泛型，但功能更强大。

模板允许编译器根据一组规则编写代码。

## 2.模板语法

使用template关键字和尖括号来定义模板参数。

通过模板参数，可以创建一个函数或类的蓝图，编译器根据实际使用情况生成具体的代码。

## 3.模板类

除了函数，模板也可以用于创建类。

例如创建一个模板类Array，它允许在栈上创建一个固定大小的数组。

通过模板参数，可以指定数组的类型和大小。

## 4.模板的使用建议

模板非常强大，但过度使用可能导致代码难以理解和维护。

建议在需要处理多种类型时使用模板，例如日志系统或渲染中的材质系统。

# C++的堆与栈内存的比较

## 1.堆栈和堆的概念

堆栈是具有预定义大小的内存区域，通常在两兆字节左右。

堆是一个可以增长和变化的内存区域。

这两个区域在物理位置上是相同的，但它们在程序中的使用方式不同。

## 2.内存分配

使用new关键字在堆上分配内存，而堆栈分配则通过移动堆栈指针来实现。

## 3.内存分配的性能差异

在堆栈上分配内存非常快速，因为它只涉及移动堆栈指针。

在堆上分配内存则需要调用malloc函数，涉及更多的簿记工作，因此速度较慢。

## 4.堆栈和堆的使用建议

建议尽可能在堆栈上分配内存，因为堆栈分配速度快且不需要手动释放。

只有在堆栈无法满足需求时（例如需要比函数作用域更长的生命周期或处理大量数据时），才考虑在堆上分配。

## 5.性能优化

在某些情况下，堆栈分配的性能差异可能不明显，例如在处理大量缓存丢失时。

但在实时应用（如游戏引擎）中，堆栈分配的性能优势非常重要。

# C++的宏

## 1.宏的概念

宏是一种预处理器指令，以#开头，如“#define”。

宏在编译前的预处理阶段被处理，可以用来替换文本或代码块。

宏可以非常简单，也可以非常复杂，甚至可以包含参数和变量。

## 2.宏的使用

宏可以用来替换代码中的文本，例如将std::cin.get()替换为wait。

宏也可以用来控制代码的编译，例如在调试模式下包含日志记录代码，在发布模式下不包含。

## 3.预处理器的使用建议

建议不要过度使用宏，因为它们可能会使代码难以理解和维护。

宏应该用于简化代码和提高效率，而不是使代码变得混乱。

## 4.宏的高级用法

宏可以用于调试目的，例如跟踪内存分配和释放。

宏可以用来自动记录错误和日志信息，提高代码的可维护性。

# C++的auto关键字

## 1.自动类型推断

auto关键字允许编译器自动推断变量的类型，这在初始化变量或从一个变量分配到另一个变量时非常有用。

## 2.简化代码

使用auto可以避免重复书写长类型名，使代码更简洁易读。

## 3.类型安全

尽管auto可以简化代码，但它也可能隐藏类型信息，使得代码难以理解，特别是对于不熟悉代码的开发者。

## 4.使用场景

auto在处理复杂类型（如容器或模板）时特别有用，因为它可以减少代码的冗余和复杂性。

## 5.尾随返回类型

auto也可以用于函数的尾随返回类型，这在C++11及以后的版本中特别有用。

## 6.模板编程

在模板编程中，auto有时是必要的，因为类型可能在编译时才能确定。

# C++的静态数组

## 1.标准数组（std::array）介绍

定义：std::array是C++标准模板库（STL）中的一个类，用于处理静态数组。

特点：

静态大小：数组大小在创建时定义，不可更改。

类型安全：需要指定数组元素的类型和数组的大小。

内存位置：存储在栈上，与std::vector不同，后者在堆上创建数据存储。

边界检查：在调试模式下，可以进行边界检查，防止越界访问。

## 2.与传统C风格数组的对比

优势：

类型安全：std::array是类型安全的，避免了类型错误。

边界检查：std::array在调试模式下提供边界检查，防止越界访问。

易用性：std::array提供了size()函数，可以直接获取数组大小，无需额外传递大小参数。

STL兼容性：支持STL算法，如std::sort，可以方便地进行数组操作。

## 3.性能和内存占用

性能：std::array在优化级别下与传统数组性能相当。

内存占用：std::array不存储数组大小信息，大小作为模板参数传递，不会增加额外内存开销。

## 4.结论

std::array提供了类型安全、边界检查和STL兼容性等优势，是处理静态数组的推荐方式。它在调试模式下提供边界检查，而在发布模式下性能与传统数组相当，且不增加额外内存开销。

使用std::array：建议在大多数情况下使用std::array代替传统数组，因为它提供了更好的安全性和易用性。

# C++的函数指针

## 函数指针介绍

定义：函数指针是将函数赋给变量的一种方式，允许将函数作为参数传递给其他函数或存储函数地址。

用途：函数指针可以用于实现回调函数、事件处理、动态函数调用等场景。

## 使用示例

1.定义函数：

void helloWorld() {

std::cout << "Hello World" << std::endl;

}

2.创建函数指针变量：

void (\*function)(void) = helloWorld;

3.调用函数指针：

function();

## 与lambda表达式的对比

lambda表达式：提供了一种更简洁的方式来定义匿名函数，可以直接在代码中定义和使用，无需单独的函数声明。

示例：

auto lambda = []() { std::cout << "Hello World" << std::endl; };

lambda();

## 总结

函数指针：是C++中一种基础且强大的特性，允许将函数作为参数传递或存储函数地址。

lambda表达式：是C++11引入的新特性，提供了一种更简洁、更灵活的方式来定义匿名函数，适用于需要临时函数的场景。

# C++的lambad

## Lambda表达式介绍

定义：Lambda表达式是一种定义匿名函数的方式，允许快速创建小型、一次性使用的函数。

用途：Lambda表达式常用于需要函数作为参数的场景，如排序、查找等操作，或在需要延迟执行的代码中。

## 使用示例

1.定义Lambda表达式：

auto lambda = []() { std::cout << "Hello World" << std::endl; };

2.调用Lambda表达式：

lambda();

## 与函数指针的对比

函数指针：需要显式定义函数原型，然后将函数地址赋给指针变量。

Lambda表达式：直接在需要的地方定义和使用，无需单独的函数声明，代码更简洁。

## 总结

Lambda表达式：是C++11引入的新特性，提供了一种更简洁、更灵活的方式来定义匿名函数，适用于需要临时函数或回调函数的场景。

函数指针：虽然在C++中仍然重要，但Lambda表达式因其简洁性和易用性在现代C++编程中越来越受欢迎。

# 使用using namespace std;的利弊

## 优点

代码简洁：不需要在每个标准库函数前加std::前缀，使代码看起来更简洁。

减少打字：对于长命名空间，可以减少重复输入的字符数量。

## 缺点

命名空间冲突：可能导致命名空间中的函数或变量名冲突，特别是当使用多个库时。

代码可读性降低：使用using namespace std;后，难以区分标准库函数和其他自定义函数或变量。

潜在的命名冲突：如果在代码中使用了与标准库同名的函数或变量，可能会导致意外的行为。

## 结论

不推荐使用：using namespace std;虽然在某些情况下看起来方便，但会降低代码的清晰度和可维护性。

明确命名空间：建议明确写出std::前缀，以清楚地表明函数或变量来自标准库，避免潜在的命名冲突。

使用范围限制：如果确实需要使用using namespace，建议限制其使用范围，例如在函数内部或特定的代码块中。

在大多数情况下应避免使用using namespace std;，特别是在大型项目或库中，以保持代码的清晰度和可维护性。

# C++的命名空间

## 1.命名空间的定义和用途

命名空间用于避免命名冲突，允许在不同的上下文中使用相同的名称。

通过将代码组织在命名空间中，可以更好地管理代码库，避免全局命名空间的混乱。

## 2.命名空间的使用示例

通过定义命名空间（如namespace Apple和namespace Orange），可以将函数或变量封装在特定的命名空间内。

使用命名空间时，可以通过命名空间::函数名的方式调用特定命名空间中的函数。

## 3.命名空间的高级特性

可以为命名空间创建别名，简化代码的书写。

可以嵌套命名空间，进一步组织代码结构。

C++17引入了内联命名空间，使得父命名空间的所有内容都可用，但这个特性不是所有编译器都支持。

## 4.命名空间的建议使用方式

建议将命名空间的使用限制在尽可能小的范围内，避免在全局范围内滥用命名空间。

在头文件中使用命名空间时要特别小心，避免引起全局命名冲突。

C语言没有命名空间的概念，因此在使用C代码时，命名空间可以避免命名冲突。

即使在C++项目中，如果使用C代码，仍然需要考虑命名冲突的问题。

# C++的线程

## 1.线程的基本概念和用途

线程允许程序在多个逻辑处理线程上并行执行代码，提高程序性能。

在现代计算机和设备中，多线程是常见的，可以同时执行多个任务。

## 2.线程的使用示例

通过创建线程对象（如std::thread），可以将函数指针传递给线程构造函数，从而启动线程。

使用thread.join()方法可以等待线程完成其工作，这在主线程中尤其重要。

## 3.线程的高级特性

线程可以用于执行需要长时间计算的任务，从而避免阻塞主线程。

线程可以用于同时执行多个任务，例如在等待用户输入时执行其他操作。

## 4.线程的建议使用方式

在使用线程时，应避免过度使用，以防止CPU使用率过高。

可以使用std::this\_thread::sleep\_for方法让当前线程暂停一段时间，以减少CPU使用率。

## 5.线程与C代码的兼容性

C++中的线程API最终会转换为平台特定的代码，但基本概念和使用方法是通用的。

# C++的计时

## 1.计时的重要性

计时对于理解代码执行时间、优化性能、评估基准等场景非常重要。

可以用于确定特定操作或代码段的执行时间。

## 2.使用<chrono>库进行计时

std::chrono是C++标准库的一部分，提供了高分辨率计时器。

使用std::chrono::high\_resolution\_clock获取当前时间点。

使用std::chrono::duration来计算时间差。

## 3.计时器类的实现

使用对象的生命周期来自动开始和结束计时。

在构造函数中开始计时，在析构函数中结束计时。

计算并打印出以毫秒为单位的持续时间。

## 4.性能优化

通过修改代码（例如在std::cout中添加换行符）来优化性能。

在发布模式下进行性能测试。

# C++多维数组

## 1.多维数组的定义

多维数组可以看作是数组的数组，例如二维数组是数组的数组，三维数组是数组的数组的数组。

在C++中，多维数组的实现是通过指针的指针来完成的。

## 2.二维数组的实现

二维数组可以看作是数组的数组，即一个数组，其中每个元素本身也是一个数组。

在C++中，实现二维数组需要使用指针的指针（int\*\*），即一个指向整数指针的指针。

## 3.三维数组的实现

三维数组可以看作是数组的数组的数组，即一个数组，其中每个元素本身也是一个二维数组。

在C++中，实现三维数组需要使用指针的指针的指针（int\*\*\*），即一个指向整数指针的指针的指针。

## 4.内存分配和访问

在C++中，多维数组的内存分配和访问比普通数组复杂，因为需要逐层分配和访问指针。

通过嵌套循环可以访问多维数组中的元素。

## 5.内存碎片和性能问题

多维数组可能导致内存碎片，因为每个数组可能被分配在内存的不同位置。

这可能导致缓存未命中，从而影响程序性能。

## 6.优化建议

通常建议使用一维数组来存储二维或三维数据，以优化内存访问和提高性能。

例如，可以使用一维数组来存储图像数据，而不是使用二维数组。

# C++的排序

## 1.排序的重要性

排序是编程中常见的需求，用于组织和管理数据。

C++标准库提供了std::sort函数，可以对任何类型的容器进行排序。

## 2.std::sort函数的使用

std::sort函数接受两个迭代器参数，分别指向容器的开始和结束。

可以通过传递一个比较函数或lambda表达式来自定义排序规则。

## 3.排序规则的自定义

比较函数或lambda表达式需要返回一个布尔值，指示第一个参数是否应该排在第二个参数之前。

# C++的类型双关

## 1.简介

类型双关是指在C++中绕过类型系统，直接操作内存的一种技术。

C++是一种强类型语言，通常需要声明变量类型，如整数、浮点数、结构体或类等。

与JavaScript等语言不同，C++的类型系统较为严格，但可以通过直接访问内存来绕过。

## 2.应用

类型双关可以用于将同一块内存以不同的类型进行解释，例如将整数内存解释为浮点数。

这种技术在某些情况下非常有用，如将类或结构体作为字节流输出，或在不进行复制或转换的情况下访问数据。

## 3.正确使用

如果需要将一个整数变量a的内存地址直接作为浮点数使用，可以使用指针和引用的组合，而不是进行类型转换。

这种方法可以避免不必要的内存复制，但同样需要小心处理，以避免内存越界。

## 4.危险性

类型双关虽然强大，但使用不当可能导致程序崩溃或数据损坏。

在实际应用中，应尽量避免使用类型双关，除非有充分的理由和对内存操作有深入的理解。

# C++的联合体

## 1.联合体简介

联合体是一种特殊的数据类型，类似于类或结构体，但只能同时存储一个成员。

联合体的大小等于其最大成员的大小，因为它只能存储一个成员。

## 2.联合体的应用

联合体常用于类型转换，例如将同一块内存以不同的类型解释。

它可以用于节省空间，尤其是在需要存储多种类型数据但不同时使用时。

## 3.联合体的限制

联合体不能包含虚函数，因为它们需要虚函数表，这会增加额外的内存开销。

联合体通常不用于包含虚函数的类。

## 4.联合体的使用场景

联合体常用于实现类型双关，例如将一个数学向量类同时作为RGB颜色处理。

这种技术允许开发者以不同的方式解释同一块内存，从而节省空间或实现特定功能。

## 5.联合体的可读性

联合体的使用可以提高代码的可读性，尤其是在处理复杂数据结构时。

但过度使用或不当使用可能导致代码难以理解和维护。

# C++的虚析构函数

## 1.虚析构函数简介

虚析构函数是C++中用于处理多态和继承时非常重要的概念。

当基类的析构函数被声明为虚函数时，它确保在删除指向派生类对象的基类指针时，派生类的析构函数也会被调用。

## 2.虚析构函数的应用

在多态场景中，如果基类的析构函数不是虚函数，删除基类指针时只会调用基类的析构函数，可能导致派生类资源未被正确释放，从而造成内存泄漏。

虚析构函数确保在删除基类指针时，会先调用派生类的析构函数，再调用基类的析构函数，从而正确释放所有资源。

## 3.虚析构函数的重要性

在设计允许继承的基类时，应将析构函数声明为虚函数，以确保派生类的析构函数能够被正确调用。

这对于避免内存泄漏和资源管理至关重要。

## 4.建议

确保将基类的析构函数声明为虚函数。

这样可以确保在通过基类指针删除派生类对象时，派生类的析构函数也会被调用，从而安全地释放所有资源。

# C++的类型转换

## 1.类型转换简介

类型转换是C++中处理类型转换和类型转换时使用的一种技术。

C++是一种强类型语言，意味着类型系统是强制的，需要明确转换才能改变类型。

## 2.类型转换的应用

类型转换分为隐式转换和显式转换。

隐式转换是编译器自动进行的，例如将整数转换为浮点数。

显式转换需要程序员明确指定，例如使用static\_cast或reinterpret\_cast。

## 3.C风格转换

C风格转换使用(type)variable的形式，例如(int)variable。

这种方式虽然灵活，但缺乏类型安全检查，可能导致错误。

## 4.C++风格转换

C++提供了多种类型安全的转换方式，包括static\_cast、reinterpret\_cast、dynamic\_cast和const\_cast。

static\_cast用于非多态类型之间的转换，如基本数据类型之间的转换。

reinterpret\_cast用于重新解释内存中的数据，例如将整数指针转换为字符指针。

dynamic\_cast用于多态类型之间的转换，可以检查转换是否成功，并在失败时返回nullptr。

const\_cast用于移除或添加const或volatile属性。

## 5.类型转换的好处

类型转换提供了代码的可读性和安全性。

使用C++风格的转换可以避免一些常见的错误，并在编译时进行类型检查。

## 6.动态转换（Dynamic Cast）

dynamic\_cast主要用于多态类型之间的转换，如基类指针转换为派生类指针。

它在运行时检查转换是否成功，并在失败时返回nullptr。

这种转换主要用于安全地处理多态类型，避免类型错误。

## 建议

对于新项目或小型项目，建议使用C++风格的转换，如static\_cast、dynamic\_cast等，以提高代码的安全性和可读性。

# 条件与操作断点

## 1.条件断点

条件断点允许开发者指定断点触发的条件，例如只有当内存中的某个值满足特定条件时才触发断点。

## 2.动作断点

动作断点允许在断点触发时执行特定动作，例如打印信息到控制台，但不中断程序执行。

## 3.条件断点的高级应用

条件断点在循环中的应用，例如在游戏的更新循环中，只在特定条件下触发断点。

例如使用条件断点快速定位游戏中的特定实体。

## 4.性能影响

条件断点和动作断点可能会降低程序运行速度，建议在性能敏感的应用中谨慎使用。

建议在需要时停止程序，重新编译代码，以避免调试工具对性能的影响。