简介

- C++ 语言基础知识

C++ 简介

- C++ 是一种静态类型的、编译式的、通用的、大小写敏感的、不规则的编程语言，支持过程化编程、面向对象编程和泛型编程。
- C++ 被认为是一种中级语言，它综合了高级语言和低级语言的特点。
- C++ 是由 Bjarne Stroustrup 于 1979 年在新泽西州美利山贝尔实验室开始设计开发的。C++ 进一步扩充和完善了 C 语言，最初命名为带类的C，后来在 1983 年更名为 C++。
- C++ 是 C 的一个超集，事实上，任何合法的 C 程序都是合法的 C++ 程序。
- 注意：使用静态类型的编程语言是在编译时执行类型检查，而不是在运行时执行类型检查。

面向对象程序设计

- C++ 完全支持面向对象的程序设计，包括面向对象开发的四大特性：
    - 封装（Encapsulation）：封装是将数据和方法组合在一起，对外部隐藏实现细节，只公开对外提供的接口。这样可以提高安全性、可靠性和灵活性。
    - 继承（Inheritance）：继承是从已有类中派生出新类，新类具有已有类的属性和方法，并且可以扩展或修改这些属性和方法。这样可以提高代码的复用性和可扩展性。
    - 多态（Polymorphism）：多态是指同一种操作作用于不同的对象，可以有不同的解释和实现。它可以通过接口或继承实现，可以提高代码的灵活性和可读性。
    - 抽象（Abstraction）：抽象是从具体的实例中提取共同的特征，形成抽象类或接口，以便于代码的复用和扩展。抽象类和接口可以让程序员专注于高层次的设计和业务逻辑，而不必关注底层的实现细节。

标准库

- 标准的 C++ 由三个重要部分组成：
    - 核心语言，提供了所有构件块，包括变量、数据类型和常量，等等。
    - C++ 标准库，提供了大量的函数，用于操作文件、字符串等。
    - 标准模板库（STL），提供了大量的方法，用于操作数据结构等。

ANSI标准

- ANSI 标准是为了确保 C++ 的便携性 —— 您所编写的代码在 Mac、UNIX、Windows、Alpha 计算机上都能通过编译。
- 由于 ANSI 标准已稳定使用了很长的时间，所有主要的 C++ 编译器的制造商都支持 ANSI 标准。

常见C++编译器制造商

C++ 编译器是将 C++ 程序源代码转换为可执行程序的工具。它们负责将源代码翻译成机器代码，以便计算机可以理解和执行。C++ 编译器制造商是开发和维护这些编译器的公司或组织。以下是一些常见的 C++ 编译器制造商以及他们的一些特点：

1. **GNU Compiler Collection (GCC)**:

- 制造商：自由软件基金会（Free Software Foundation）
- 特点：GCC 是一个开源编译器集合，支持多种编程语言，包括 C++。它是许多 Unix 和 Linux 发行版的默认编译器，并且在跨平台开发中广泛使用。GCC 遵循开放标准并且具有强大的优化能力。

2. **Clang**:

   - 制造商：LLVM 项目
   - 特点：Clang 是 LLVM 编译器基础设施的一部分，它支持 C++ 和其他编程语言。Clang 以其快速的编译速度和准确的诊断能力而闻名。它被广泛用于 C++ 开发和跨平台移动应用程序开发。

3. **Microsoft Visual C++**:

   - 制造商：微软公司
   - 特点：Visual C++ 是 Windows 平台上的主要 C++ 编译器。它集成在 Visual Studio 开发环境中，为 Windows 开发提供了丰富的工具和库。Visual C++ 遵循微软的特定标准，并且与 Windows API 紧密集成。

4. **Intel C++ Compiler**:

   - 制造商：英特尔公司
   - 特点：Intel C++ 编译器专注于优化，特别是针对英特尔处理器的优化。它通常用于高性能计算和科学计算领域，以提高代码在英特尔架构上的性能。

5. **IBM XL C/C++**:

   - 制造商：IBM
   - 特点：IBM 的 XL C/C++ 编译器主要用于 IBM 的大型服务器和超级计算机上。它支持多种操作系统和架构，并提供了许多高级编译器优化。

6. **ARM Compiler**:

   - 制造商：ARM Holdings
   - 特点：ARM Compiler 是专门为 ARM 架构处理器设计的编译器。它用于嵌入式系统和移动设备的开发，以及一些嵌入式 Linux 系统。

7. **其他厂商**：除了上述制造商之外，还有许多其他公司和组织开发了各种 C++ 编译器，包括商业和开源编译器，以满足不同需求和平台的开发者。

选择合适的 C++ 编译器通常取决于项目的要求、目标平台和性能需求。不同的编译器可能会对标准的支持程度、代码优化和诊断能力等方面有所不同，因此开发者需要根据项目的具体需求来选择合适的编译器。此外，一些项目可能会选择多个编译器进行测试和优化，以确保代码在不同环境下都能正常运行和性能出色。

学习C++

- 学习 C++，关键是要理解概念，而不应过于深究语言的技术细节。
- 学习程序设计语言的目的是为了成为一个更好的程序员，也就是说，是为了能更有效率地设计和实现新系统，以及维护旧系统。
- C++ 支持多种编程风格。您可以使用 Fortran、C、Smalltalk 等任意一种语言的编程风格来编写代码。每种风格都能有效地保证运行时间效率和空间效率。

C++的使用

- C++ 语言在许多行业和领域都有广泛应用，包括：
  - 游戏开发：C++ 是游戏开发领域中最常用的编程语言之一，因为它具有高效的性能和直接控制硬件的能力。许多主要的游戏引擎，如 Unreal Engine 和 Unity，都使用 C++ 编写。
  - 嵌入式系统开发：C++ 可以在嵌入式系统中发挥重要作用，如智能手机、汽车、机器人和家电等领域。由于嵌入式系统通常具有严格的资源限制和实时要求，因此 C++ 的高效性能和内存控制功能非常有用。
  - 金融领域：C++ 在金融领域中被广泛应用，如高频交易、算法交易和风险管理等领域。由于这些应用程序需要高效的性能和对硬件的直接控制，C++ 语言是一个合适的选择。
  - 图形图像处理：C++ 可以用于开发图形和图像处理应用程序，如计算机视觉、计算机图形学和人工智能领域。由于这些应用程序需要高效的计算能力和对硬件的控制，因此 C++ 是一个很好的选择。
  - 科学计算和数值分析：C++ 可以用于开发科学计算和数值分析应用程序，如数值模拟和高性能计算等领域。由于这些应用程序需要高效的计算能力和对硬件的直接控制，C++ 语言是一个很好的选择。

C++基本语法

- C++ 程序可以定义为对象的集合，这些对象通过调用彼此的方法进行交互。现在让我们简要地看一下什么是类、对象，方法、即时变量。
  - 对象 - 对象具有状态和行为。例如：一只狗的状态 - 颜色、名称、品种，行为 - 摇动、叫唤、吃。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。

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。
- 例如，下面是三个不同的语句：

```
x = y;
y = y+1;
add(x, y);
```

- 语句块是一组使用大括号括起来的按逻辑连接的语句。例如：

```
{
   cout << "Hello World"; // 输出 Hello World
   return 0;
}
```

- C++ 不以行末作为结束符的标识，因此，您可以在一行上放置多个语句。

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。因此，在 C++ 中，Manpower 和 manpower 是两个不同的标识符。

三字符组

- 三字符组就是用于表示另一个字符的三个字符序列，又称为三字符序列。三字符序列总是以两个问号开头。
- 三字符序列不太常见，但 C++ 标准允许把某些字符指定为三字符序列。以前为了表示键盘上没有的字符，这是必不可少的一种方法。
- 三字符序列可以出现在任何地方，包括字符串、字符序列、注释和预处理指令。

C++中的空格

- 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，您可以根据需要适当增加一些空格。

C++注释

- 程序的注释是解释性语句，您可以在 C++ 代码中包含注释，这将提高源代码的可读性。所有的编程语言都允许某种形式的注释。
- C++ 支持单行注释和多行注释。注释中的所有字符会被 C++ 编译器忽略。
- C++ 注释一般有两种：
    - // - 一般用于单行注释。
    - /* ... */ - 一般用于多行注释。

C++数据类型

- 使用编程语言进行编程时，需要用到各种变量来存储各种信息。变量保留的是它所存储的值的内存位置。这意味着，当您创建一个变量时，就会在内存中保留一些空间。
- 您可能需要存储各种数据类型（比如字符型、宽字符型、整型、浮点型、双浮点型、布尔型等）的信息，操作系统会根据变量的数据类型，来分配内存和决定在保留内存中存储什么。

基本的内置类型

- 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：
    - 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
- typedef 声明
    - 可以使用 typedef 为一个已有的类型取一个新的名字。下面是使用 typedef 定义一个新类型的语法：
    - `typedef type newname`
- 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。
    - 创建枚举，需要使用关键字 enum。枚举类型的一般形式为：

```
enum 枚举名{
      标识符[=整型常数],
      标识符[=整型常数],
...
    标识符[=整型常数]
} 枚举变量;
```

- 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的值。
    - 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。
- 动态转换（Dynamic Cast）
    - 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，如果不能进行转换则返回空指针或引发异常

```
class Base {};
class Derived : public Base {};
```

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

- 常量转换（Const Cast）
  - 常量转换用于将 const 类型的对象转换为非 const 类型的对象。
  - 常量转换只能用于转换掉 const 属性，不能改变对象的类型

```
const int i = 10;
int& r = const_cast<int&>(i); // 常量转换，将const int转换为int
```

- 重新解释转换（Reinterpret Cast）
  - 重新解释转换将一个数据类型的值重新解释为另一个数据类型的值，通常用于在不同的数据类型之间进行转换
  - 重新解释转换不进行任何类型检查，因此可能会导致未定义的行为

```
int i = 10;
float f = reinterpret_cast<float&>(i); // 重新解释将int类型转换为float类型
```

C++ 变量类型

- 变量其实只不过是程序可操作的存储区的名称
- 在 C++ 中，有多种变量类型可用于存储不同种类的数据。
- C++ 中每个变量都有指定的类型，类型决定了变量存储的大小和布局，该范围内的值都可以存储在内存中，运算符可应用于变量上。变量的名称可以由字母、数字和下划线字符组成。它必须以字母或下划线开头
- 大写字母和小写字母是不同的，因为 C++ 是大小写敏感的
- 基本的变量类型
  - bool：布尔类型，存储值 true 或 false，占用 1 个字节。
  - char：字符类型，用于存储 ASCII 字符，通常占用 1 个字节。
  - int：整数类型，通常用于存储普通整数，通常占用 4 个字节。
  - float：单精度浮点值，用于存储单精度浮点数。单精度是这样的格式，1 位符号，8 位指数，23 位小数，通常占用4个字节。
  - double：双精度浮点值，用于存储双精度浮点数。双精度是 1 位符号，11 位指数，52 位小数，通常占用 8 个字节。
  - 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
- 整数类型（Integer Types）：
  - int：用于表示整数，通常占用4个字节。

    - short：用于表示短整数，通常占用2个字节。
    - long：用于表示长整数，通常占用4个字节。
    - long long：用于表示更长的整数，通常占用8个字节。

- 浮点类型（Floating-Point Types）：

    - float：用于表示单精度浮点数，通常占用4个字节。
    - double：用于表示双精度浮点数，通常占用8个字节。
    - long double：用于表示更高精度的浮点数，占用字节数可以根据实现而变化。

- 字符类型（Character Types）：

    - char：用于表示字符，通常占用1个字节。
    - wchar_t：用于表示宽字符，通常占用2或4个字节。
    - char16_t：用于表示16位Unicode字符，占用2个字节。
    - char32_t：用于表示32位Unicode字符，占用4个字节。

- 布尔类型（Boolean Type）：

    - bool：用于表示布尔值，只能取true或false。

- 枚举类型（Enumeration Types）：

    - enum：用于定义一组命名的整数常量。

- 指针类型（Pointer Types）：

    - type*：用于表示指向类型为type的对象的指针。

- 数组类型（Array Types）：

    - type[]或type[size]：用于表示具有相同类型的元素组成的数组。

- 结构体类型（Structure Types）：

    - struct：用于定义包含多个不同类型成员的结构。

- 类类型（Class Types）：

    - class：用于定义具有属性和方法的自定义类型。

- 共用体类型（Union Types）：

    - union：用于定义一种特殊的数据类型，它可以在相同的内存位置存储不同的数据类型。

- C++ 中的变量定义

    - 变量定义就是告诉编译器在何处创建变量的存储，以及如何创建变量的存储。

- 变量定义指定一个数据类型，并包含了该类型的一个或多个变量的列表，如下所示：

    - type variable_list;

- 在这里，type 必须是一个有效的 C++ 数据类型，可以是 char、wchar_t、int、float、double、bool 或任何用户自定义的对象，variable_list 可以由一个或多个标识符名称组成，多个标识符之间用逗号分隔。

- 变量可以在声明的时候被初始化（指定一个初始值）。初始化器由一个等号，后跟一个常量表达式组成，如下所示：
- type variable_name = value;
- 不带初始化的定义：带有静态存储持续时间的变量会被隐式初始化为 NULL（所有字节的值都是 0），其他所有变量的初始值是未定义的。
- C++ 中的变量声明
    - 变量声明向编译器保证变量以给定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明
    - 当您使用多个文件且只在其中一个文件中定义变量时（定义变量的文件在程序连接时是可用的），变量声明就显得非常有用。您可以使用 extern 关键字在任何地方声明一个变量。虽然您可以在 C++ 程序中多次声明一个变量，但变量只能在某个文件、函数或代码块中被定义一次
- C++ 中的左值(Lvalues)和右值(Rvalues)
- C++ 中有两种类型的表达式：
    - 左值（lvalue）：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边
    - 右值（rvalue）：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。
- 变量是左值，因此可以出现在赋值号的左边。数值型的字面值是右值，因此不能被赋值，不能出现在赋值号的左边。下面是一个有效的语句：
    - int g = 20;
- 但是下面这个就不是一个有效的语句，会生成编译时错误：
    - 10 = 20;

C++ 变量作用域

- 一般来说有三个地方可以定义变量：
    - 在函数或一个代码块内部声明的变量，称为局部变量。
    - 在函数参数的定义中声明的变量，称为形式参数。
    - 在所有函数外部声明的变量，称为全局变量
- 作用域是程序的一个区域，变量的作用域可以分为以下几种：
    - 局部作用域：在函数内部声明的变量具有局部作用域，它们只能在函数内部访问。局部变量在函数每次被调用时被创建，在函数执行完后被销毁。
    - 全局作用域：在所有函数和代码块之外声明的变量具有全局作用域，它们可以被程序中的任何函数访问。全局变量在程序开始时被创建，在程序结束时被销毁。
    - 块作用域：在代码块内部声明的变量具有块作用域，它们只能在代码块内部访问。块作用域变量在代码块每次被执行时被创建，在代码块执行完后被销毁。

    - 类作用域：在类内部声明的变量具有类作用域，它们可以被类的所有成员函数访问。类作用域变量的生命周期与类的生命周期相同。

- 注意：如果在内部作用域中声明的变量与外部作用域中的变量同名，则内部作用域中的变量将覆盖外部作用域中的变量。

- 局部变量

    - 在函数或一个代码块内部声明的变量，称为局部变量。它们只能被函数内部或者代码块内部的语句使用。

- 全局变量

    - 在所有函数外部定义的变量（通常是在程序的头部），称为全局变量。全局变量的值在程序的整个生命周期内都是有效的。
    - 全局变量可以被任何函数访问。也就是说，全局变量一旦声明，在整个程序中都是可用的。
    - 在程序中，局部变量和全局变量的名称可以相同，但是在函数内，局部变量的值会覆盖全局变量的值

- 初始化局部变量和全局变量

    - 当局部变量被定义时，系统不会对其初始化，您必须自行对其初始化。定义全局变量时，系统会自动初始化为下列值：
        - int : 0
        - char : '\0'
        - float : 0
        - double : 0
        - pointer : NULL
    - 正确地初始化变量是一个良好的编程习惯，否则有时候程序可能会产生意想不到的结果

- 类作用域

    - 类作用域指的是在类内部声明的变量

C++ 常量

- 常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做字面量

- 常量可以是任何的基本数据类型，可分为整型数字、浮点数字、字符、字符串和布尔值

- 常量就像是常规的变量，只不过常量的值在定义后不能进行修改

- 整数常量

    - 整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制
    - 整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。
    - 以下是各种类型的整数常量的实例：

```
85          // 十进制
0213        // 八进制
0x4b        // 十六进制
30          // 整数
30u         // 无符号整数
30l         // 长整数
30ul        // 无符号长整数
```

- 浮点常量
  - 浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。
  - 当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时， 必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。
  - 下面列举几个浮点常量的实例：

```
3.14159       // 合法的
314159E-5L    // 合法的
510E          // 非法的：不完整的指数
210f          // 非法的：没有小数或指数
.e55          // 非法的：缺少整数或分数
```

- 布尔常量
  - 布尔常量共有两个，它们都是标准的 C++ 关键字：
    - true 值代表真。
    - false 值代表假。
  - 我们不应把 true 的值看成 1，把 false 的值看成 0。
- 字符常量
  - 字符常量是括在单引号中。如果常量以 L（仅当大写时）开头，则表示它是一个宽字符常量（例如 L'x'），此时它必须存储在 wchar_t 类型的变量中。否则，它就是一个窄字符常量（例如 'x'），此时它可以存储在 char 类型的简单变量中。
  - 字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）
  - 在 C++ 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：
- 字符串常量
  - 字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。
  - 您可以使用 \ 做分隔符，把一个很长的字符串常量进行分行。
- 定义常量：
  - 在 C++ 中，有两种简单的定义常量的方式：
    - 使用 #define 预处理器。

        - 使用 const 关键字。

- #define 预处理器
    - 下面是使用 #define 预处理器定义常量的形式：
    - #define identifier value

- const 关键字
    - 您可以使用 const 前缀声明指定类型的常量，如下所示：
    - const type variable = value;
    - 请注意，把常量定义为大写字母形式，是一个很好的编程实践。

C++ 修饰符类型

- C++ 允许在 char、int 和 double 数据类型前放置修饰符。
- 修饰符是用于改变变量类型的行为的关键字，它更能满足各种情境的需求。
- 下面列出了数据类型修饰符：
    - signed：表示变量可以存储负数。对于整型变量来说，signed 可以省略，因为整型变量默认为有符号类型。
    - unsigned：表示变量不能存储负数。对于整型变量来说，unsigned 可以将变量范围扩大一倍。
    - short：表示变量的范围比 int 更小。short int 可以缩写为 short。
    - long：表示变量的范围比 int 更大。long int 可以缩写为 long。
    - long long：表示变量的范围比 long 更大。C++11 中新增的数据类型修饰符。
    - float：表示单精度浮点数。
    - double：表示双精度浮点数。
    - bool：表示布尔类型，只有 true 和 false 两个值。
    - char：表示字符类型。
    - wchar_t：表示宽字符类型，可以存储 Unicode 字符。
- 修饰符 signed、unsigned、long 和 short 可应用于整型，signed 和 unsigned 可应用于字符型，long 可应用于双精度型。
- 这些修饰符也可以组合使用，修饰符 signed 和 unsigned 也可以作为 long 或 short 修饰符的前缀。例如：unsigned long int。
- C++ 允许使用速记符号来声明无符号短整数或无符号长整数。您可以不写 int，只写单词 unsigned、short 或 long，int 是隐含的。例如，下面的两个语句都声明了无符号整型变量。

```
signed int num1 = -10; // 定义有符号整型变量 num1，初始值为 -10
unsigned int num2 = 20; // 定义无符号整型变量 num2，初始值为 20

short int num1 = 10; // 定义短整型变量 num1，初始值为 10
long int num2 = 100000; // 定义长整型变量 num2，初始值为 100000

long long int num1 = 10000000000; // 定义长长整型变量 num1，初始值为
10000000000
```

```
float num1 = 3.14f; // 定义单精度浮点数变量 num1，初始值为 3.14
double num2 = 2.71828; // 定义双精度浮点数变量 num2，初始值为 2.71828

bool flag = true; // 定义布尔类型变量 flag，初始值为 true

char ch1 = 'a'; // 定义字符类型变量 ch1，初始值为 'a'
wchar_t ch2 = L'你'; // 定义宽字符类型变量 ch2，初始值为 '你'
```

- C++ 中的类型限定符
    - 类型限定符提供了变量的额外信息，用于在定义变量或函数时改变它们的默认行为的关键字。
    - const：const 定义常量，表示该变量的值不能被修改。
    - volatile：修饰符 volatile 告诉该变量的值可能会被程序以外的因素改变，如硬件或其他线程。
    - restrict：由 restrict 修饰的指针是唯一一种访问它所指向的对象的方式。只有 C99 增加了新的类型限定符 restrict。
    - mutable：表示类中的成员变量可以在 const 成员函数中被修改。
    - static：用于定义静态变量，表示该变量的作用域仅限于当前文件或当前函数内，不会被其他文件或函数访问。
    - register：用于定义寄存器变量，表示该变量被频繁使用，可以存储在CPU的寄存器中，以提高程序的运行效率。

- const 实例:

```
const int NUM = 10; // 定义常量 NUM，其值不可修改
const int* ptr = &NUM; // 定义指向常量的指针，指针所指的值不可修改
int const* ptr2 = &NUM; // 和上面一行等价
```

- volatile 实例

```
volatile int num = 20; // 定义变量 num，其值可能会在未知的时间被改变
```

- mutable 实例

```
class Example {
public:
    int get_value() const {
        return value_; // const 关键字表示该成员函数不会修改对象中的数据成员
    }
    void set_value(int value) const {
        value_ = value; // mutable 关键字允许在 const 成员函数中修改成员变量
    }
private:
    mutable int value_;
};
```

- static 实例

```
void example_function() {
    static int count = 0; // static 关键字使变量 count 存储在程序生命周期内都存在
    count++;
}
```

- register 实例

```
void example_function(register int num) {
    // register 关键字建议编译器将变量 num 存储在寄存器中
    // 以提高程序执行速度
    // 但是实际上是否会存储在寄存器中由编译器决定
}
```

C++ 存储类

- 存储类定义 C++ 程序中变量/函数的范围（可见性）和生命周期。这些说明符放置在它们所修饰的类型之前。下面列出 C++ 程序中可用的存储类：
    - auto
    - register
    - static
    - extern
    - mutable
    - thread_local(C++11)
- 从C++17开始，auto关键字不再是C++存储类说明符，且register关键字被弃用
- auto 存储类
    - 自 C++ 11 以来，auto 关键字用于两种情况：声明变量时根据初始化表达式自动推断该变量的类型、声明函数时函数返回值的占位符。
    - C++98标准中auto关键字用于自动变量的声明，但由于使用极少且多余，在 C++17 中已删除这一用法。
    - 根据初始化表达式自动推断被声明的变量的类型，如：

```
auto f=3.14;      //double
auto s("hello");  //const char*
auto z = new auto(9); // int*
auto x1 = 5, x2 = 5.0, x3='r';//错误，必须是初始化为同一类型
```

- register 存储类
    - register 存储类用于定义存储在寄存器中而不是 RAM 中的局部变量。这意味着变量的最大尺寸等于寄存器的大小（通常是一个词），且不能对它应用一元的 '&' 运算符（因为它没有内存位置）。

```
{
   register int  miles;
}
```

- 寄存器只用于需要快速访问的变量，比如计数器。还应注意的是，定义 'register' 并不意味着变量将被存储在寄存器中，它意味着变量可能存储在寄存器中，这取决于硬件和实现的限制。

- static 存储类

    - static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。
    - static 修饰符也可以应用于全局变量。当 static 修饰全局变量时，会使变量的作用域限制在声明它的文件内。
    - 在 C++ 中，当 static 用在类数据成员上时，会导致仅有一个该成员的副本被类的所有对象共享。

- extern 存储类

    - extern 存储类用于提供一个全局变量的引用，全局变量对所有的程序文件都是可见的。当您使用 'extern' 时，对于无法初始化的变量，会把变量名指向一个之前定义过的存储位置。
    - 当您有多个文件且定义了一个可以在其他文件中使用的全局变量或函数时，可以在其他文件中使用 extern 来得到已定义的变量或函数的引用。可以这么理解，extern 是用来在另一个文件中声明一个全局变量或函数。
    - extern 修饰符通常用于当有两个或多个文件共享相同的全局变量或函数的时候

- mutable 存储类

    - mutable 说明符仅适用于类的对象，这将在本教程的最后进行讲解。它允许对象的成员替代常量。也就是说，mutable 成员可以通过 const 成员函数修改。

- thread_local存储类

    - 使用 thread_local 说明符声明的变量仅可在它在其上创建的线程上访问。 变量在创建线程时创建，并在销毁线程时销毁。 每个线程都有其自己的变量副本。
    - thread_local 说明符可以与 static 或 extern 合并。
    - 可以将 thread_local 仅应用于数据声明和定义，thread_local 不能用于函数声明或定义。
    - 以下演示了可以被声明为 thread_local 的变量：

```
thread_local int x;  // 命名空间下的全局变量
class X
{
    static thread_local std::string s; // 类的static成员变量
};
static thread_local std::string X::s;  // X::s 是需要定义的

void foo()
{
    thread_local std::vector<int> v;  // 本地变量
}
```

C++ 运算符

- 运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C++ 内置了丰富的运算符，并提供了以下类型的运算符：
    - 算术运算符
    - 关系运算符
    - 逻辑运算符
    - 位运算符
    - 赋值运算符
    - 杂项运算符
- 算术运算符
    - `+`：把两个操作数相加
    - `-`：从第一个操作数中减去第二个操作数
    - `*`：把两个操作数相乘
    - `/`：分子除以分母
    - `%`：取模运算符，整除后的余数
    - `++`：自增运算符，整数值增加 1
    - `--`：自减运算符，整数值减少 1
- 关系运算符
    - `==`：检查两个操作数的值是否相等，如果相等则条件为真。
    - `!=`：检查两个操作数的值是否相等，如果不相等则条件为真。
    - `>`：检查左操作数的值是否大于右操作数的值，如果是则条件为真。
    - `<`：检查左操作数的值是否小于右操作数的值，如果是则条件为真。
    - `>=`：检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
    - `<=`：检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。
- 逻辑运算符
    - `&&`：称为逻辑与运算符。如果两个操作数都 true，则条件为 true。
    - `||`：称为逻辑或运算符。如果两个操作数中有任意一个 true，则条件为 true。
    - `!`：称为逻辑非运算符。用来逆转操作数的逻辑状态，如果条件为 true 则逻辑非运算符将使其为 false。
- 位运算符
    - `&`：按位与操作，按二进制位进行"与"运算
    - `|`：按位或运算符，按二进制位进行"或"运算。
    - `^`：异或运算符，按二进制位进行"异或"运算。
    - `~`：取反运算符，按二进制位进行"取反"运算。
    - `<<`：二进制左移运算符。将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。
    - `>>`：二进制右移运算符。将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。

- 赋值运算符
  - =：简单的赋值运算符，把右边操作数的值赋给左边操作数
  - +=：加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数
  - -=：减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数
  - *=：乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数
  - /=：除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数
  - %=：求模且赋值运算符，求两个操作数的模赋值给左边操作数
  - <<=：左移且赋值运算符
  - >>=：右移且赋值运算符
  - &=：按位与且赋值运算符
  - ^=：按位异或且赋值运算符
  - |=：按位或且赋值运算符
- 杂项运算符
  - sizeof：sizeof 运算符返回变量的大小。例如，sizeof(a) 将返回 4，其中 a 是整数。
  - Condition ? X : Y：条件运算符。如果 Condition 为真 ? 则值为 X：否则值为 Y。
  - ,：逗号运算符会顺序执行一系列运算。整个逗号表达式的值是以逗号分隔的列表中的最后一个表达式的值。
  - .（点）和 ->（箭头）：成员运算符用于引用类、结构和共用体的成员。
  - Cast：强制转换运算符把一种数据类型转换为另一种数据类型。例如，int(2.2000) 将返回 2。
  - &：指针运算符 & 返回变量的地址。例如 &a; 将给出变量的实际地址。
  - *：指针运算符 * 指向一个变量。例如，*var; 将指向变量 var。

C++ 循环

- 有的时候，可能需要多次执行同一块代码。一般情况下，语句是顺序执行的：函数中的第一个语句先执行，接着是第二个语句，依此类推。
- 编程语言提供了允许更为复杂的执行路径的多种控制结构。
- 循环语句允许我们多次执行一个语句或语句组，下面是大多数编程语言中循环语句的一般形式：
- 循环类型
  - while 循环：当给定条件为真时，重复语句或语句组。它会在执行循环主体之前测试条件。
  - for 循环：多次执行一个语句序列，简化管理循环变量的代码。
  - do...while 循环：除了它是在循环主体结尾测试条件外，其他与 while 语句类似。
  - 嵌套循环：您可以在 while、for 或 do..while 循环内使用一个或多个循环。
- 循环控制语句
  - 循环控制语句更改执行的正常序列。当执行离开一个范围时，所有在该范围中创建的自动对象都会被销毁。
  - C++ 提供了下列的控制语句。点击链接查看每个语句的细节。
    - break 语句：终止 loop 或 switch 语句，程序流将继续执行紧接着 loop 或 switch 的下一条语句。
    - continue 语句：引起循环跳过主体的剩余部分，立即重新开始测试条件。
    - goto 语句：将控制转移到被标记的语句。但是不建议在程序中使用 goto 语句。

- 无限循环
    - 如果条件永远不为假，则循环将变成无限循环。for 循环在传统意义上可用于实现无限循环。由于构成循环的三个表达式中任何一个都不是必需的，您可以将某些条件表达式留空来构成一个无限循环。
    - 当条件表达式不存在时，它被假设为真。您也可以设置一个初始值和增量表达式，但是一般情况下，C++ 程序员偏向于使用 for(;😉 结构来表示一个无限循环。

C++ 判断

- 判断结构要求程序员指定一个或多个要评估或测试的条件，以及条件为真时要执行的语句（必需的）和条件为假时要执行的语句（可选的）。
- 判断语句
    - C++ 编程语言提供了以下类型的判断语句。点击链接查看每个语句的细节。
    - if 语句 : 一个 if 语句 由一个布尔表达式后跟一个或多个语句组成。
    - if...else 语句 : 一个 if 语句 后可跟一个可选的 else 语句，else 语句在布尔表达式为假时执行。
    - 嵌套 if 语句 : 您可以在一个 if 或 else if 语句内使用另一个 if 或 else if 语句。
    - switch 语句 : 一个 switch 语句允许测试一个变量等于多个值时的情况。
    - 嵌套 switch 语句 : 您可以在一个 switch 语句内使用另一个 switch 语句。
- ? : 运算符
    - 我们已经在前面的章节中讲解了 条件运算符 ? :，可以用来替代 if...else 语句。它的一般形式如下：
    - Exp1 ? Exp2 : Exp3;
    - 其中，Exp1、Exp2 和 Exp3 是表达式。请注意，冒号的使用和位置。
    - ? 表达式的值是由 Exp1 决定的。如果 Exp1 为真，则计算 Exp2 的值，结果即为整个 ? 表达式的值。如果 Exp1 为假，则计算 Exp3 的值，结果即为整个 ? 表达式的值。

C++ 函数

- 函数是一组一起执行一个任务的语句。每个 C++ 程序都至少有一个函数，即主函数 main() ，所有简单的程序都可以定义其他额外的函数。
- 您可以把代码划分到不同的函数中。如何划分代码到不同的函数中是由您来决定的，但在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。
- 函数声明告诉编译器函数的名称、返回类型和参数。函数定义提供了函数的实际主体
- C++ 标准库提供了大量的程序可以调用的内置函数。例如，函数 strcat() 用来连接两个字符串，函数 memcpy() 用来复制内存到另一个位置。
- 函数还有很多叫法，比如方法、子例程或程序，等等。
- 定义函数
    - C++ 中的函数定义的一般形式如下：

```
return_type function_name( parameter list )
{
   body of the function
}
```

- 在 C++ 中，函数由一个函数头和一个函数主体组成。下面列出一个函数的所有组成部分：
    - 返回类型：一个函数可以返回一个值。return_type 是函数返回的值的数据类型。有些函数执行所需的操作而不返回值，在这种情况下，return_type 是关键字 void。
    - 函数名称：这是函数的实际名称。函数名和参数列表一起构成了函数签名。
    - 参数：参数就像是占位符。当函数被调用时，您向参数传递一个值，这个值被称为实际参数。参数列表包括函数参数的类型、顺序、数量。参数是可选的，也就是说，函数可能不包含参数。
    - 函数主体：函数主体包含一组定义函数执行任务的语句。
- 函数声明
    - 函数声明会告诉编译器函数名称及如何调用函数。函数的实际主体可以单独定义。
    - 函数声明包括以下几个部分：
        - return_type function_name(parameter list);
    - 针对上面定义的函数 max()，以下是函数声明：
        - int max(int num1, int num2);
    - 在函数声明中，参数的名称并不重要，只有参数的类型是必需的，因此下面也是有效的声明：
        - int max(int, int);
    - 当您在一个源文件中定义函数且在另一个文件中调用函数时，函数声明是必需的。在这种情况下，您应该在调用函数的文件顶部声明函数。
- 调用函数
    - 创建 C++ 函数时，会定义函数做什么，然后通过调用函数来完成已定义的任务。
    - 当程序调用函数时，程序控制权会转移给被调用的函数。被调用的函数执行已定义的任务，当函数的返回语句被执行时，或到达函数的结束括号时，会把程序控制权交还给主程序。
    - 调用函数时，传递所需参数，如果函数返回一个值，则可以存储返回值
- 函数参数
    - 如果函数要使用参数，则必须声明接受参数值的变量。这些变量称为函数的形式参数。
    - 形式参数就像函数内的其他局部变量，在进入函数时被创建，退出函数时被销毁
    - 当调用函数时，有三种向函数传递参数的方式：
        - 传值调用 : 该方法把参数的实际值赋值给函数的形式参数。在这种情况下，修改函数内的形式参数对实际参数没有影响。
        - 指针调用 : 该方法把参数的地址赋值给形式参数。在函数内，该地址用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。
        - 引用调用 : 该方法把参数的引用赋值给形式参数。在函数内，该引用用于访问调用中要用到的实际参数。这意味着，修改形式参数会影响实际参数。
    - 默认情况下，C++ 使用传值调用来传递参数。一般来说，这意味着函数内的代码不能改变用于调用函数的参数。
- 参数的默认值

    - 当您定义一个函数，您可以为参数列表中后边的每一个参数指定默认值。当调用函数时，如果实际参数的值留空，则使用这个默认值。
    - 这是通过在函数定义中使用赋值运算符来为参数赋值的。调用函数时，如果未传递参数的值，则会使用默认值，如果指定了值，则会忽略默认值，使用传递的值。

- Lambda 函数与表达式

    - C++11 提供了对匿名函数的支持,称为 Lambda 函数(也叫 Lambda 表达式)。
    - Lambda 表达式把函数看作对象。Lambda 表达式可以像对象一样使用，比如可以将它们赋给变量和作为参数传递，还可以像函数一样对其求值。
    - Lambda 表达式本质上与函数声明非常类似。Lambda 表达式具体形式如下:
        - capture->return-type{body}
    - 例如：
        - [](int x, int y){ return x < y ; }
    - 如果没有返回值可以表示为：
        - capture{body}
    - 例如：
        - []{ ++global_x; }
    - 在一个更为复杂的例子中，返回类型可以被明确的指定如下：
        - [](int x, int y) -> int { int z = x + y; return z + x; }
    - 如果 lambda 函数没有传回值（例如 void），其返回类型可被完全忽略。
    - 在Lambda表达式内可以访问当前作用域的变量，这是Lambda表达式的闭包（Closure）行为。与JavaScript闭包不同，C++变量传递有传值和传引用的区别。可以通过前面的[]来指定：

```
[]       // 沒有定义任何变量。使用未定义变量会引发错误。
[x, &y] // x以传值方式传入（默认），y以引用方式传入。
[&]      // 任何被使用到的外部变量都隐式地以引用方式加以引用。
[=]      // 任何被使用到的外部变量都隐式地以传值方式加以引用。
[&, x]   // x显式地以传值方式加以引用。其余变量以引用方式加以引用。
[=, &z] // z显式地以引用方式加以引用。其余变量以传值方式加以引用。
```

- 另外有一点需要注意。对于[=]或[&]的形式，lambda 表达式可以直接使用 this 指针。但是，对于[]的形式，如果要使用 this 指针，必须显式传入：
    - this { this->someFunc(); }();

C++ 数字

- 通常，当我们需要用到数字时，我们会使用原始的数据类型，如 int、short、long、float 和 double 等等。这些用于数字的数据类型，其可能的值和数值范围，我们已经在 C++ 数据类型一章中讨论过。

- C++ 数学运算

    - 在 C++ 中，除了可以创建各种函数，还包含了各种有用的函数供您使用。这些函数写在标准 C 和 C++ 库中，叫做内置函数。您可以在程序中引用这些函数。
    - C++ 内置了丰富的数学函数，可对各种数字进行运算。下表列出了 C++ 中一些有用的内置的数学函数。
    - 为了利用这些函数，您需要引用数学头文件 。
        - double cos(double);

        - 该函数返回弧度角（double 型）的余弦。
    - double sin(double);
        - 该函数返回弧度角（double 型）的正弦。
    - double tan(double);
        - 该函数返回弧度角（double 型）的正切。
    - double log(double);
        - 该函数返回参数的自然对数。
    - double pow(double, double);
        - 假设第一个参数为 x，第二个参数为 y，则该函数返回 x 的 y 次方。
    - double hypot(double, double);
        - 该函数返回两个参数的平方总和的平方根，也就是说，参数为一个直角三角形的两个直角边，函数会返回斜边的长度。
    - double sqrt(double);
        - 该函数返回参数的平方根。
    - int abs(int);
        - 该函数返回整数的绝对值。
    - double fabs(double);
        - 该函数返回任意一个浮点数的绝对值。
    - double floor(double);
        - 该函数返回一个小于或等于传入参数的最大整数。

- C++ 随机数
    - 在许多情况下，需要生成随机数。关于随机数生成器，有两个相关的函数。一个是 rand()，该函数只返回一个伪随机数。生成随机数之前必须先调用 srand() 函数。
    - 下面是一个关于生成随机数的简单实例。实例中使用了 time() 函数来获取系统时间的秒数，通过调用 rand() 函数来生成随机数：

```
#include <iostream>
#include <ctime>
#include <cstdlib>

using namespace std;

int main ()
{
   int i,j;

   // 设置种子
   srand( (unsigned)time( NULL ) );

   /* 生成 10 个随机数 */
   for( i = 0; i < 10; i++ )
   {
      // 生成实际的随机数
      j= rand();
      cout <<"随机数： " << j << endl;
   }
```

```
    return 0;
}
```

C++ 数组

- C++ 支持数组数据结构，它可以存储一个固定大小的相同类型元素的顺序集合。数组是用来存储一系列数据，但它往往被认为是一系列相同类型的变量。
- 数组的声明并不是声明一个个单独的变量，比如 number0、number1、...、number99，而是声明一个数组变量，比如 numbers，然后使用 numbers[0]、numbers[1]、...、numbers[99] 来代表一个个单独的变量。数组中的特定元素可以通过索引访问。
- 所有的数组都是由连续的内存位置组成。最低的地址对应第一个元素，最高的地址对应最后一个元素。
- 声明数组
    - 在 C++ 中要声明一个数组，需要指定元素的类型和元素的数量，如下所示：
        - type arrayName [arraySize];
    - 这叫做一维数组。arraySize 必须是一个大于零的整数常量，type 可以是任意有效的 C++ 数据类型。例如，要声明一个类型为 double 的包含 10 个元素的数组 balance，声明语句如下：
        - double balance[10];
    - 现在 balance 是一个可用的数组，可以容纳 10 个类型为 double 的数字。
- 初始化数组
    - 在 C++ 中，您可以逐个初始化数组，也可以使用一个初始化语句，如下所示：
        - double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
    - ▢ 中指定的元素数目。
    - 如果您省略掉了数组的大小，数组的大小则为初始化时元素的个数。因此，如果：
        - double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
    - 您将创建一个数组，它与前一个实例中所创建的数组是完全相同的。下面是一个为数组中某个元素赋值的实例：
        - balance[4] = 50.0;
    - 上述的语句把数组中第五个元素的值赋为 50.0。所有的数组都是以 0 作为它们第一个元素的索引，也被称为基索引，数组的最后一个索引是数组的总大小减去 1。以下是上面所讨论的数组的的图形表示：
- 访问数组元素
    - 数组元素可以通过数组名称加索引进行访问。元素的索引是放在方括号内，跟在数组名称的后边。
    - double salary = balance[9];
    - 上面的语句将把数组中第 10 个元素的值赋给 salary 变量。下面的实例使用了上述的三个概念，即，声明数组、数组赋值、访问数组：

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

#include <iomanip>
```

```
using std::setw;

int main ()
{
   int n[ 10 ]; // n 是一个包含 10 个整数的数组

   // 初始化数组元素
   for ( int i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; // 设置元素 i 为 i + 100
   }
   cout << "Element" << setw( 13 ) << "Value" << endl;

   // 输出数组中每个元素的值
   for ( int j = 0; j < 10; j++ )
   {
      cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
   }

   return 0;
}
```

- 上面的程序使用了 setw() 函数 来格式化输出。当上面的代码被编译和执行时，它会产生下列结果：

```
Element        Value
      0          100
      1          101
      2          102
      3          103
      4          104
      5          105
      6          106
      7          107
      8          108
      9          109
```

- C++ 中数组详解
    - 在 C++ 中，数组是非常重要的，我们需要了解更多有关数组的细节。下面列出了 C++ 程序员必须清楚的一些与数组相关的重要概念：
        - 多维数组 : C++ 支持多维数组。多维数组最简单的形式是二维数组。
        - 指向数组的指针 : 您可以通过指定不带索引的数组名称来生成一个指向数组中第一个元素的指针。
        - 传递数组给函数 : 您可以通过指定不带索引的数组名称来给函数传递一个指向数组的指针。
        - 从函数返回数组 : C++ 允许从函数返回数组。

C++ 字符串

- C++ 提供了以下两种类型的字符串表示形式：

    - C 风格字符串
    - C++ 引入的 string 类类型

- C 风格字符串

    - C 风格的字符串起源于 C 语言，并在 C++ 中继续得到支持。字符串实际上是使用 null 字符 \0 终止的一维字符数组。因此，一个以 null 结尾的字符串，包含了组成字符串的字符
    - 下面的声明和初始化创建了一个 RUNOOB 字符串。由于在数组的末尾存储了空字符，所以字符数组的大小比单词 RUNOOB 的字符数多一个。
    - C++ 中有大量的函数用来操作以 null 结尾的字符串:
        - strcpy(s1, s2);
            - 复制字符串 s2 到字符串 s1。
        - strcat(s1, s2);
            - 连接字符串 s2 到字符串 s1 的末尾。连接字符串也可以用 + 号，例如:
        - strlen(s1);
            - 返回字符串 s1 的长度。
        - strcmp(s1, s2);
            - 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回值小于 0；如果 s1>s2 则返回值大于 0。
        - strchr(s1, ch);
            - 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。
        - strstr(s1, s2);
            - 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

- C++ 中的 String 类

    - C++ 标准库提供了 string 类类型，支持上述所有的操作，另外还增加了其他更多的功能

C++ 指针

- 学习 C++ 的指针既简单又有趣。通过指针，可以简化一些 C++ 编程任务的执行，还有一些任务，如动态内存分配，没有指针是无法执行的

- 每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。

- 什么是指针？

    - 指针是一个变量，其值为另一个变量的地址，即，内存位置的直接地址。就像其他变量或常量一样，您必须在使用指针存储其他变量地址之前，对其进行声明。指针变量声明的一般形式为：
        - type *var-name;
    - 在这里，type 是指针的基类型，它必须是一个有效的 C++ 数据类型，var-name 是指针变量的名称。用来声明指针的星号 * 与乘法中使用的星号是相同的。但是，在这个语句中，星号是用来指定一个变量是指针。以下是有效的指针声明：

```
int    *ip;    /* 一个整型的指针 */
double *dp;    /* 一个 double 型的指针 */
float  *fp;    /* 一个浮点型的指针 */
char   *ch;    /* 一个字符型的指针 */
```

- 所有指针的值的实际数据类型，不管是整型、浮点型、字符型，还是其他的数据类型，都是一样的，都是一个代表内存地址的长的十六进制数。不同数据类型的指针之间唯一的不同是，指针所指向的变量或常量的数据类型不同。

- C++ 中使用指针

    - 使用指针时会频繁进行以下几个操作：定义一个指针变量、把变量地址赋值给指针、访问指针变量中可用地址的值。这些是通过使用一元运算符 * 来返回位于操作数所指定地址的变量的值。下面的实例涉及到了这些操作：

```
#include <iostream>

using namespace std;

int main ()
{
   int  var = 20;   // 实际变量的声明
   int  *ip;        // 指针变量的声明

   ip = &var;       // 在指针变量中存储 var 的地址

   cout << "Value of var variable: ";
   cout << var << endl;

   // 输出在指针变量中存储的地址
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // 访问指针中地址的值
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

- C++ 指针详解
    - 在 C++ 中，有很多指针相关的概念，这些概念都很简单，但是都很重要。下面列出了 C++ 程序员必须清楚的一些与指针相关的重要概念：
    - C++ Null 指针 : C++ 支持空指针。NULL 指针是一个定义在标准库中的值为零的常量。
    - C++ 指针的算术运算 : 可以对指针进行四种算术运算：++、--、+、-
    - C++ 指针 vs 数组 : 指针和数组之间有着密切的关系。
    - C++ 指针数组 : 可以定义用来存储指针的数组。
    - C++ 指向指针的指针 : C++ 允许指向指针的指针。
    - C++ 传递指针给函数 : 通过引用或地址传递参数，使传递的参数在调用函数中被改变。
    - C++ 从函数返回指针 : C++ 允许函数返回指针到局部变量、静态变量和动态内存分配。

C++ 引用

- 引用变量是一个别名，也就是说，它是某个已存在变量的另一个名字。一旦把引用初始化为某个变量，就可以使用该引用名称或变量名称来指向变量。
- C++ 引用 vs 指针
    - 引用很容易与指针混淆，它们之间有三个主要的不同：
        - 不存在空引用。引用必须连接到一块合法的内存。
        - 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
        - 引用必须在创建时被初始化。指针可以在任何时间被初始化。
- C++ 中创建引用
    - 试想变量名称是变量附属在内存位置中的标签，您可以把引用当成是变量附属在内存位置中的第二个标签。因此，您可以通过原始变量名称或引用来访问变量的内容。例如：
        - int i = 17;
    - 我们可以为 i 声明引用变量，如下所示：

```
int&  r = i;
double& s = d;
```

- 在这些声明中，& 读作引用。因此，第一个声明可以读作 "r 是一个初始化为 i 的整型引用"，第二个声明可以读作 "s 是一个初始化为 d 的 double 型引用"。下面的实例使用了 int 和 double 引用：
- 引用通常用于函数参数列表和函数返回值。下面列出了 C++ 程序员必须清楚的两个与 C++ 引用相关的重要概念：
    - 把引用作为参数 : C++ 支持把引用作为参数传给函数，这比传一般的参数更安全。
    - 把引用作为返回值 : 可以从 C++ 函数中返回引用，就像返回其他数据类型一样。

C++ 日期 & 时间

- C++ 标准库没有提供所谓的日期类型。C++ 继承了 C 语言用于日期和时间操作的结构和函数。为了使用日期和时间相关的函数和结构，需要在 C++ 程序中引用 头文件。
- 有四个与时间相关的类型：clock_t、time_t、size_t 和 tm。类型 clock_t、size_t 和 time_t 能够把系统时间和日期表示为某种整数。
- 结构类型 tm 把日期和时间以 C 结构的形式保存，tm 结构的定义如下：

```
struct tm {
  int tm_sec;   // 秒，正常范围从 0 到 59，但允许至 61
  int tm_min;   // 分，范围从 0 到 59
  int tm_hour;  // 小时，范围从 0 到 23
  int tm_mday;  // 一月中的第几天，范围从 1 到 31
  int tm_mon;   // 月，范围从 0 到 11
  int tm_year;  // 自 1900 年起的年数
  int tm_wday;  // 一周中的第几天，范围从 0 到 6，从星期日算起
  int tm_yday;  // 一年中的第几天，范围从 0 到 365，从 1 月 1 日算起
  int tm_isdst; // 夏令时
};
```

- 下面是 C/C++ 中关于日期和时间的重要函数。所有这些函数都是 C/C++ 标准库的组成部分，您可以在 C++ 标准库中查看一下各个函数的细节。
    - time_t time(time_t *time);
        - 该函数返回系统的当前日历时间，自 1970 年 1 月 1 日以来经过的秒数。如果系统没有时间，则返回 -1。
    - char *ctime(const time_t *time);
        - 该返回一个表示当地时间的字符串指针，字符串形式 day month year hours:minutes:seconds year\n\0。
    - struct tm *localtime(const time_t *time);
        - 该函数返回一个指向表示本地时间的 tm 结构的指针。
    - clock_t clock(void);
        - 该函数返回程序执行起（一般为程序的开头），处理器时钟所使用的时间。如果时间不可用，则返回 -1。
    - char * asctime (const struct tm * time);
        - 该函数返回一个指向字符串的指针，字符串包含了 time 所指向结构中存储的信息，返回形式为：day month date hours:minutes:seconds year\n\0。
    - struct tm *gmtime(const time_t *time);
        - 该函数返回一个指向 time 的指针，time 为 tm 结构，用协调世界时（UTC）也被称为格林尼治标准时间（GMT）表示。
    - time_t mktime(struct tm *time);
        - 该函数返回日历时间，相当于 time 所指向结构中存储的时间。
    - double difftime (time_t time2, time_t time1);
        - 该函数返回 time1 和 time2 之间相差的秒数。
    - size_t strftime();
        - 该函数可用于格式化日期和时间为指定的格式。
- 当前日期和时间
    - 下面的实例获取当前系统的日期和时间，包括本地时间和协调世界时（UTC）。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
   // 基于当前系统的当前日期/时间
   time_t now = time(0);

   // 把 now 转换为字符串形式
   char* dt = ctime(&now);

   cout << "本地日期和时间：" << dt << endl;

   // 把 now 转换为 tm 结构
   tm *gmtm = gmtime(&now);
   dt = asctime(gmtm);
```

```
    cout << "UTC 日期和时间："<< dt << endl;
}
```

- 使用结构 tm 格式化时间
  - tm 结构在 C/C++ 中处理日期和时间相关的操作时，显得尤为重要。tm 结构以 C 结构的形式保存日期和时间。大多数与时间相关的函数都使用了 tm 结构。下面的实例使用了 tm 结构和各种与日期和时间相关的函数。
  - 在练习使用结构之前，需要对 C 结构有基本的了解，并懂得如何使用箭头 -> 运算符来访问结构成员。

```
#include <iostream>
#include <ctime>

using namespace std;

int main( )
{
   // 基于当前系统的当前日期/时间
   time_t now = time(0);

   cout << "1970 到目前经过秒数:" << now << endl;

   tm *ltm = localtime(&now);

   // 输出 tm 结构的各个组成部分
   cout << "年: "<< 1900 + ltm->tm_year << endl;
   cout << "月: "<< 1 + ltm->tm_mon<< endl;
   cout << "日: "<<  ltm->tm_mday << endl;
   cout << "时间: "<< ltm->tm_hour << ":";
   cout << ltm->tm_min << ":";
   cout << ltm->tm_sec << endl;
}
```

C++ 基本的输入输出

- C++ 标准库提供了一组丰富的输入/输出功能，我们将在后续的章节进行介绍。本章将讨论 C++ 编程中最基本和最常见的 I/O 操作。

- C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做输入操作。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做输出操作。

- I/O 库头文件

  - 下列的头文件在 C++ 编程中很重要。
    - : 该文件定义了 cin、cout、cerr 和 clog 对象，分别对应于标准输入流、标准输出流、非缓冲标准错误流和缓冲标准错误流。
    - : 该文件通过所谓的参数化的流操纵器（比如 setw 和 setprecision），来声明对执行标准化 I/O 有用的服务。

    - : 该文件为用户控制的文件处理声明服务。我们将在文件和流的相关章节讨论它的细节。

- 标准输出流（cout）
    - 预定义的对象 cout 是 iostream 类的一个实例。cout 对象"连接"到标准输出设备，通常是显示屏。cout 是与流插入运算符 << 结合使用的，
    - C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。
    - 流插入运算符 << 在一个语句中可以多次使用，如上面实例中所示，endl 用于在行末添加一个换行符。

- 标准输入流（cin）
    - 预定义的对象 cin 是 iostream 类的一个实例。cin 对象附属到标准输入设备，通常是键盘。cin 是与流提取运算符 >> 结合使用的，如下所示：
    - C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储在给定的变量中。
    - 流提取运算符 >> 在一个语句中可以多次使用，如果要求输入多个数据，可以使用如下语句：
        - cin >> name >> age;
    - 这相当于下面两个语句：
        - cin >> name;
        - cin >> age;

- 标准错误流（cerr）
    - 预定义的对象 cerr 是 iostream 类的一个实例。cerr 对象附属到标准输出设备，通常也是显示屏，但是 cerr 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

- 标准日志流（clog）
    - 预定义的对象 clog 是 iostream 类的一个实例。clog 对象附属到标准输出设备，通常也是显示屏，但是 clog 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲区，直到缓冲填满或者缓冲区刷新时才会输出。
    - clog 也是与流插入运算符 << 结合使用的，如下所示：

```
#include <iostream>

using namespace std;

int main( )
{
   char str[] = "Unable to read....";

   clog << "Error message : " << str << endl;
}
```

- 通过这些小实例，我们无法区分 cout、cerr 和 clog 的差异，但在编写和执行大型程序时，它们之间的差异就变得非常明显。所以良好的编程实践告诉我们，使用 cerr 流来显示错误消息，而其他的日志消息则使用 clog 流来输出。

C++ 数据结构

- C/C++ 数组允许定义可存储相同类型数据项的变量，但是结构是 C++ 中另一种用户自定义的可用的数据类型，它允许您存储不同类型的数据项。
- 结构用于表示一条记录，
- 定义结构
    - 为了定义结构，您必须使用 struct 语句。struct 语句定义了一个包含多个成员的新的数据类型，struct 语句的格式如下：

```
struct type_name {
member_type1 member_name1;
member_type2 member_name2;
member_type3 member_name3;
.
.
} object_names;
```

- type_name 是结构体类型的名称，member_type1 member_name1 是标准的变量定义，比如 int i; 或者 float f; 或者其他有效的变量定义。在结构定义的末尾，最后一个分号之前，您可以指定一个或多个结构变量，这是可选的
- 访问结构成员
    - 为了访问结构的成员，我们使用成员访问运算符（.）。成员访问运算符是结构变量名称和我们要访问的结构成员之间的一个句号。
- 结构作为函数参数
    - 您可以把结构作为函数参数，传参方式与其他类型的变量或指针类似。您可以使用上面实例中的方式来访问结构变量：

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books book );

// 声明一个结构体类型 Books
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
```

```
{
   Books Book1;        // 定义结构体类型 Books 的变量 Book1
   Books Book2;        // 定义结构体类型 Books 的变量 Book2

    // Book1 详述
   strcpy( Book1.title, "C++ 教程");
   strcpy( Book1.author, "Runoob");
   strcpy( Book1.subject, "编程语言");
   Book1.book_id = 12345;

   // Book2 详述
   strcpy( Book2.title, "CSS 教程");
   strcpy( Book2.author, "Runoob");
   strcpy( Book2.subject, "前端技术");
   Book2.book_id = 12346;

   // 输出 Book1 信息
   printBook( Book1 );

   // 输出 Book2 信息
   printBook( Book2 );

   return 0;
}
void printBook( struct Books book )
{
   cout << "书标题 : " << book.title <<endl;
   cout << "书作者 : " << book.author <<endl;
   cout << "书类目 : " << book.subject <<endl;
   cout << "书 ID : " << book.book_id <<endl;
}
```

- 指向结构的指针
  - 您可以定义指向结构的指针，方式与定义指向其他类型变量的指针相似，如下所示：
  - struct Books *struct_pointer;
  - 现在，您可以在上述定义的指针变量中存储结构变量的地址。为了查找结构变量的地址，请把 & 运算符放在结构名称的前面，如下所示：
  - struct_pointer = &Book1;
  - 为了使用指向该结构的指针访问结构的成员，您必须使用 -> 运算符，如下所示：
  - struct_pointer->title;
  - 让我们使用结构指针来重写上面的实例，这将有助于您理解结构指针的概念：

```
#include <iostream>
#include <cstring>

using namespace std;
void printBook( struct Books *book );

struct Books
{
```

```
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   Books Book1;        // 定义结构体类型 Books 的变量 Book1
   Books Book2;        // 定义结构体类型 Books 的变量 Book2

    // Book1 详述
   strcpy( Book1.title, "C++ 教程");
   strcpy( Book1.author, "Runoob");
   strcpy( Book1.subject, "编程语言");
   Book1.book_id = 12345;

   // Book2 详述
   strcpy( Book2.title, "CSS 教程");
   strcpy( Book2.author, "Runoob");
   strcpy( Book2.subject, "前端技术");
   Book2.book_id = 12346;

   // 通过传 Book1 的地址来输出 Book1 信息
   printBook( &Book1 );

   // 通过传 Book2 的地址来输出 Book2 信息
   printBook( &Book2 );

   return 0;
}
// 该函数以结构指针作为参数
void printBook( struct Books *book )
{
   cout << "书标题  : " << book->title <<endl;
   cout << "书作者 : " << book->author <<endl;
   cout << "书类目 : " << book->subject <<endl;
   cout << "书 ID : " << book->book_id <<endl;
}
```

- typedef 关键字
  - 下面是一种更简单的定义结构的方式，您可以为创建的类型取一个"别名"。例如：

```
typedef struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
}Books;
```

- 现在，您可以直接使用 Books 来定义 Books 类型的变量，而不需要使用 struct 关键字。下面是实例：
- Books Book1, Book2;
- 您可以使用 typedef 关键字来定义非结构类型，如下所示：

```
typedef long int *pint32;

pint32 x, y, z;
```

- x, y 和 z 都是指向长整型 long int 的指针。