

C++

---

MRL Liu

(未经许可, 不得传播)

2022 年 02 月 02 日

---

如今，计算机行业已经诞生了无数优秀的现代编程语言，但是现在 C++ 却仍然具有强盛的生命力。显然，C++ 语言具备独特的优势：**和 C 兼容，可以和硬件设备密切交互，完成各种稀奇古怪的底层功能**。系统级软件是数字世界的基础设施，C++ 被誉为系统级编程皇冠上的明珠。现代 C++ 版本一般指的是国际 C++ 标准委员会发布的 4 个版本：C++11/14/17/20。

# 第 1 章 C++程序篇

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**面向过程编程、面向对象编程和泛型编程**。（注意：静态类型是指 C++ 在编译时执行类型检查，而不是在运行时执行类型检查）。

C++ 具备面向对象程序设计的四大特性：**封装、抽象、继承和多态**。（也有的说面向对象的三大特性，则将**抽象**删除）。

标准的 C++ 由三个重要部分组成：

- （1）核心语言。提供了 C++ 的变量、数据类型和常量等。
- （2）C++ 标准库。提供了大量的类和函数，用于操作文件、字符串 `string` 等。
- （3）标准模板库（STL）。提供了常见数据结构的模板和算法等，属于泛型编程。

## 一、C++的文件

C++ 大型程序由多个源代码文件（.cpp）组成，不同的文件可能共享一些数据，每个文件都可以独立修改后由 C++ 编译器（g++）单独编译，最终链接成可执行程序文件（.exe）。

C/C++ 中的变量和函数等都必须先声明后使用。而且 C++ 还有著名的单定义规则（One Definition Rule, ODR），即 C++ 的变量或非内联函数只能有一次定义。C++ 提供了 2 种声明：定义声明（简称为定义）和引用声明（简称为声明）。定义声明一般要显示地进行初始化，它会在程序运行时给该变量分配存储空间，引用声明则不会在程序运行时分配存储空间，因为它引用已有的变量。

由于不同的源代码文件可能使用同一个声明，为了维护这种声明一致性，C++ 鼓励程序员将结构声明都放在头文件（.h）中，然后在每个源文件（.cpp）中包含该头文件，这样修改结构声明只需要在头文件修改一次即可。由此，整个 C++ 程序也可以划分为三个部分：


三类文件	文件内容
头文件（.h）	函数原型、结构声明、类声明、模板声明、内联函数、使用 <code>#define</code> 或 <code>const</code> 定义的符号常量
主程序文件（.cpp）	包含 <code>main()</code> 的文件，程序入口，调用模块文件实现的方法。
模块文件（.cpp）	放置一些函数定义，也称为功能模块。

### 1、头文件的预编译

头文件中包含的各种声明都不会让编译器分配内存，结构声明、类声明、模板声明、函数原型等不会创建变量，只是告诉编译器如何创建该变量；被声明为 `const` 的数据和内联函数有特殊的链接属性。

同一个源文件（.cpp）只能将同一个头文件（.h）包含一次，但是很可能在不识情的

情况下将头文件包含多次。为了避免一个头文件在多个.cpp 文件中被包含而被多次重复编译，现代 C++ 提供了针对头文件的预编译机制，以尽可能减少反复编译同一个头文件。这种预编译机制如下：



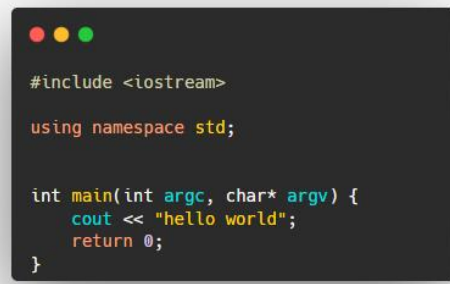
```
/*预编译指令方式一*/
#ifndef _COORDIN_H_
#define _COORDIN_H_
//#include包含各种头文件
#endif

-----
/*预编译指令方式二*/
#pragma once
//#include包含各种头文件
```

这种预编译指令告知编译器只编译一次包含该头文件的指令，这样头文件就不会被多个源文件重复编译，也可以加快大型 C++ 程序的编译速度。C++ 中的头文件会被预处理器进行预编译，包含在不同的源文件中。一个源文件（.cpp）和其对应的头文件（.h）会组成一个编译单元。

## 2、源文件的预处理和主函数

C++ 程序源文件的后缀是 .cpp，一个最简单的 .cpp 文件如下：



```
#include <iostream>

using namespace std;

int main(int argc, char* argv) {
    cout << "hello world";
    return 0;
}
```

这个最简单的 .cpp 文件中也包含了三部分：预编译指令、主函数 main()。

### 1、预处理指令

预处理指令经常用于 C++ 程序文件的开头，用来指定 C++ 预处理器的任务。预处理指令以 # 开头，常见的预处理指令如 #include、#define、#if、#else、#line。这里介绍两种常见的预处理指令。预处理指令不属于 C++ 语句，因为预处理指令都不需要 ‘;’ 结尾。

#### （1）#include 指令

#include 指令有两种使用方法：

#include<>：一般用于包含系统头文件，预处理程序默认在系统默认目录查找文件。

#include""：一般用于包含用户自定义头文件，预处理程序会先在程序源文件中查找，找不到再找系统默认目录。

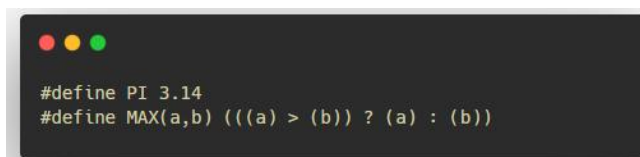
由于历史原因，系统头文件存在几种不同的写法，例如：

`#include<math.h>`是 C 风格旧式写法

`#include<iostream>`是 C++新式风格

## （2）#define 指令

`#define` 经常用于定义一个宏，其为一个标识符定义了宏名，源程序中宏名出现的地方都会用其定义的标识符进行替换，称为宏替换。宏名一般使用大写字母定义。宏经常用于定义一个常量或者简单函数，如下：



```
#define PI 3.14
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

上述示范中用宏定义了一个常量 `PI`，但现代 C++ 中建议使用 `const` 进行常量定义，因为宏替换并不会进行类型匹配之类的安全性检查。

上述示范中也用宏定义了一个 `MAX` 函数，其好处是没有函数调用的额外开销，运行速度较快，但容易出错。

## 2、main 函数

`main` 函数是 C++ 程序的入口函数，所有 C++ 程序的执行从 `main` 函数开始，并由 `main` 的返回结束（Windows 编程中的 DLL 除外）。`main` 函数的返回值为一个整数，返回 0 表示程序正常退出，返回非 0 表示程序出现异常。`main` 函数的原型如下：



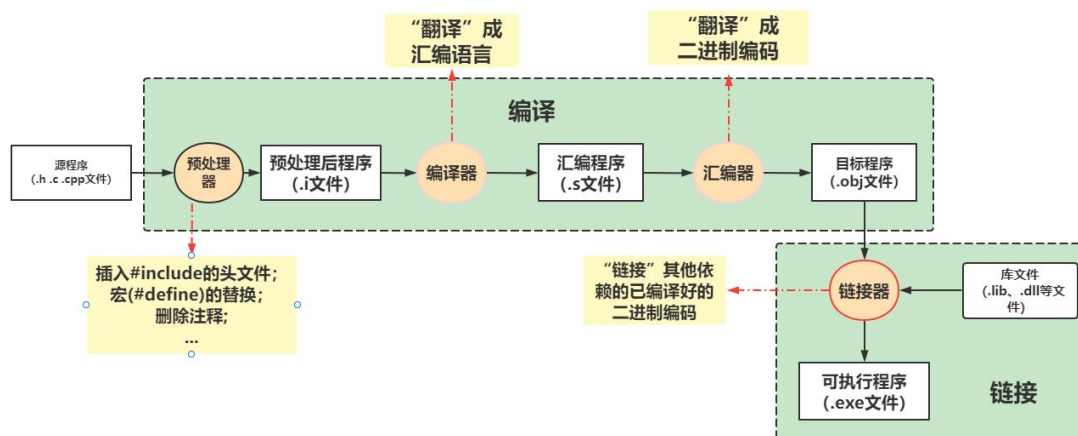
```
int main();
int main(int argc, char* argv[]);
//或
int main(int argc, char** argv);
```

`main()` 函数是 C++ 程序经过特殊处理的函数，其他的返回值类型不是 `void` 的函数，如果没有使用 `return` 语句，编译器将报错，当 `main()` 函数内没有出现 `return` 语句时，同样可以通过编译并正常运行，这是因为编译器在 `main()` 函数的末尾自动添加了 `return 0;` 语句。

`main()` 函数的带参原型可以接收用户向程序输入的参数，其中 `argc` 代表参数个数，`argv` 数组中的每一个元素保存命令行参数内容的字符串。

## 二、C++ 的编译流程

在开发过程中，程序员写的 C++ 源程序文件（.cpp）都要被 IDE 中集成的 C++ 编译器（g++）编译生成 .exe 文件，C++ 编译器就完成了 C++ 的编译流程。C++ 的编译流程总体可以分为两个步骤：**编译**和**链接**，其过程如下：



## 1、编译过程

C++的编译过程首先读取各类源程序文件，头文件（.h）会被预处理器包含进源文件中（.cpp），然后进行宏替换，删除注释等就得到了可以用于编译的预处理后程序文件。预处理后的程序文件会被依次翻译为二进制编码语言，从而得到目标程序文件，此时翻译过程便结束了。

## 2、链接过程

编译过程产生的目标程序文件不能直接执行，因为每个目标程序文件都是由一个源文件（.cpp）单独编译过来的，不同的源文件可能相互引用，所以还需要进行 C++的链接过程将各个目标程序文件相互链接最终生成可执行程序文件（.exe）。

链接过程有静态链接和动态链接 2 种方式。从图中也可以看出链接过程有.lib 和.dll 两种文件。

静态链接就是将各种目标程序文件和静态链接库（.lib 文件）链接起来，实际静态链接库也是一组目标程序文件的集合，所有的目标程序文件会被以复制的形式拷贝到最终的可执行程序文件（.exe）中。

动态链接是指一些目标程序文件不会被复制到可执行程序文件（.exe）中，而是存放在动态链接库（.dll 文件）中，作为一种可执行程序组件，其不可以单独执行，其必须依赖于对应的可执行程序（.exe）执行。

## 三、C++的内存模型

C++的内存模型决定了 C++程序在执行过程中各种数据所在的内存分区。C++的内存分区总体可以划分为栈区、堆区和固定存储区。

内存分区	内存特点	变量类型	声明方式	生命周期	作用域	链接性
栈	系统自动维护	局部变量	在代码块中	自动存储持续性	所在代码块	无链接性

		函数参数 局部常量				
寄存器	系统自动维护	寄存器变量	在代码块中，使用关键字 <code>register</code>	自动存储持续性	所在代码块	无链接性
(静态存储区) 固定存储	系统自动维护	局部静态变量	在代码块中，使用 <code>static</code> 关键字	静态存储持续性	所在代码块	无链接性
		全局静态变量	不在任何函数内，使用 <code>static</code> 关键字		所在文件	内部链接性
		全局变量	不在任何函数内		所在文件	外部链接性
(常量存储区) 固定存储		全局常量 函数指针				
堆	程序员手动维护	动态变量( <code>new</code> 分配的指针变量)	使用 C++ 运算符 <code>new</code> 声明	动态存储持续性		
代码区	系统自动维护	函数体的二进制代码		静态存储持续性		外部链接性

不同内存分区的数据具有不同的生命周期、作用域和链接性，下面依次讲解。

## 1、数据的生命周期

数据的生命周期指的是数据在内存中的保留时间，即分配内存到回收内存的时间。数据生命周期的不同是因为其定义生命的起始时间可能不一样。

C 语言和 C++ 都不允许在一个函数中定义另一个函数，因此函数的生命周期都是静态存储持续期，即从程序开始执行到程序结束。

变量可以在各种代码块中定义，存在不同的生命周期，总体分为如下 4 种：

生命周期	内存分配	相关变量
自动存储持续性	函数被执行时分配内存；函数执行完毕，内存释放	函数定义内部声明的变量和函数参数
静态存储持续性	程序被执行时分配内存；程序执行完毕，内存释放	函数定义外定义的变量和使用关键字 <code>static</code> 声明的变量
动态存储持续性	<code>new</code> 运算符被执行时分配内存； <code>delete</code> 运算符被执行或程序结束内存释放	由 <code>new</code> 运算符分配的变量
线程存储持续性	生命周期和所属线程一样长（属于并行编程内容）	使用关键字 <code>thread_local</code> 声明的变量

## 2、名称的作用域

名称（函数或变量的名字）的作用域描述了名称在文件的多大范围内可见。C++ 函数的作用域可以是整个类或全局的，不能是局部的，因为函数不能在代码块内定义。

变量的作用域从大到小如下：

命名空间内声明的变量作用域为整个命名空间；源文件内代码块外声明的变量为全局作用域（文件作用域），作用于整个文件；类成员的作用域为整个类；局部变量有局部作用域，

作用于所在代码块；函数原型作用域中使用的名称只在包含参数列表的括号中可用。

请注意，因为 C++ 要求函数和变量必须先声明后使用，所以实际作用域都是默认从声明开始到其理论作用域结束。代码块内部的变量的作用域会隐藏代码块外部的相同名称的全局变量的作用域。

### 3、名称的链接性

名称（函数或变量的名字）的链接性描述了名称如何在不同源文件间共享。名称的链接性有三种：

链接性	相关含义	相关代表
外部链接性	可以在其他源文件访问	全局变量（也叫外部变量）、非静态函数
内部链接性	只能在当前文件访问	全局静态变量、静态函数
无链接性	只能在当前函数或代码块中访问	局部静态变量、局部变量

全局变量（外部变量）和函数由于具有外部链接性，可以用来在多文件程序的不同部分之间共享数据。如果要在多个文件中使用全局变量/函数，只需要在一个源文件中包含该变量/函数的定义声明，在其他使用该变量/函数的源文件中使用 `extern` 引用声明它即可。

全局静态变量/全局静态函数由于具有内部链接性，可以用来在同一个文件中的多个函数之间共享数据。

局部静态变量由于具有无链接性，其只能用于所在代码块，但由于其静态的生命周期，所以其可以用于再生，即同一函数调用 2 次，该值第二次调用不会再次初始化而是使用上次计算的值进行计算。

### 4、C++的命名空间

C++11 提供了命名空间（`namespace`），在命名空间中声明的函数和变量都默认具有外部链接性，所以它允许程序的其他部分使用该命名空间中声明的东西，同时，一个命名空间中的名称不会与另一个命名空间的相同名称发生冲突。使用命名空间内部的东西需要添加作用域解析运算符`::`或者 `using` 声明。

命名空间是一个逻辑分组，其用来作为附加信息区分不同库中的相同名称的函数、类和变量等。命名空间的定义需要使用 `namespace` 关键字，C++ 中的关键字都在命名空间 `std` 中。使用其他命名空间的对象时需要在前面加上命名空间的名称，例如使用 `std::cout`。但是为了避免多次添加命名空间前缀的繁琐，C++ 提供了 `using namespace` 指令来告诉编译器，后续代码的关键字将使用指定的命名空间中的名称。

### 5、变量的初始化方式

变量的初始化方式分为静态初始化和动态初始化，其中静态初始化又分为零初始化和常量初始化。静态初始化是值变量在编译时进行初始化，动态初始化是指变量在程序运行时进行初始化。

普通变量必须由程序员显示地初始化，否则会编译报错，即必须进行静态初始化。

静态变量不必须由程序员显示地初始化，所有静态变量在编译时都会自动进行零初始

化，如果静态变量还有显示初始化，则看编译是否可以计算出来，可以则继续进行常量初始化，否则由程序执行时动态初始化。

```
#include <cmath>
int x; // zero-initialization
int y = 5; // constant-expression initialization
long z = 13 * 13; // constant-expression initialization
const double pi = 4.0 * atan(1.0); // dynamic initialization
```

## 四、C++的说明符和限定符

### 1、常见说明符

关键字 **auto** 在 C++11 之前用于将变量声明为自动变量，在 C++11 之后用于自动类型推断。

关键字 **register** 用于在声明中知名函数用寄存器存储，提高访问速度，在 C++11 之后只是显示地指出该变量是自动的。

关键字 **static** 用于代码块外部声明时表示该变量具有内部链接性，用于代码块内部声明时表示该局部变量的存储持续性为静态的。

关键字 **extern** 表明该声明是引用声明，即引用其他源文件中定义的外部变量。

关键字 **thread\_local** 用于表明该变量的生命周期与所属线程相同，可以与 **static** 或 **extern** 结合使用。

### 2、常见限定符

关键字 **const** 用于表明变量的内存空间被初始化后程序不能再次修改它，如果 **const** 修饰全局变量，则将会将全局变量的链接性转换为内部，表明这是该文件独享的一组常量，但是其他文件可以继续使用 **extern** 来覆盖它的内部链接性。

```
const int fingers = 10; // same as static const int fingers = 10;
```

关键字 **volatile** 用于表明硬件可以修改该变量的内存空间，即使程序不修改它。例如程序访问同一个变量 2 次时编译器一般会提前检查其值是否被程序更改过，如果程序没有更改它而自己发生了变化则认为需要优化，而该关键字避免编译器做这样的优化。

关键字 **mutable** 用来表明即使结构或类变量被整体声明为 **const**，其某个使用了 **mutable** 的变量成员可以不受 **const** 限制继续修改。



# 第 2 章 C++语法篇

## 一、数据类型

C++中有常量、数组、字符串、指针、引用、类、结构体、联合体、枚举等各种概念，但是它们**本质上都是变量**，是的，你没听错，编程语言中的常量也是一种特殊的变量（使用 `const` 关键字）。

变量是什么？**变量本质是程序可操作的内存空间的名称，每个变量概念都由 `type` 和 `value` 组成**，其中 `type` 就是其**数据类型**，其决定了数据在内存中的存储方式和运算方式，**常量本质是定义后不可更改 `value` 的变量**。所以准确理解了变量概念，就相当于拿到了 C++ 中的各种变量类型的总把手。

变量的 `value` 没什么好说的，但是变量的 `type` 可是五花八门，不同的 `type` 在内存中的大小也不一样。**C++ 中的变量必须先声明后使用**。

这里先简单说一些基本变量类型、指针和引用。

### 1、基础变量

基础变量就是变量的 `type` 为编程语言内置数据类型的变量，这些类型很常见，但是我们要记住**每种变量所占的内存字节数**。可以使用一个或多个类型修饰符进行修饰：`signed`、`unsigned`、`short` 和 `long`。

#### （1）整型变量

布尔型（`bool`）、字符型（`char`）和整数型（`int`）一起合称为整型，他们可以方便地进行转换。

`bool` 变量与整数变量可以进行隐式转换：`true` 具有值 1，`false` 具有值 0；非零整数转换为 `true`，而 0 转换为 `false`。

`char` 变量占据一个字节数，因此它可以保存  $2^8=256$  种不同的值，其默认和 ASCII 字符集进行对应，因此 `int (c)` 可以将一个 `char` 变量 `c` 显示转换为 ASCII 值。`char` 变量默认是无符号（`unsigned`），`unsigned char` 表示的值是 0~255，`signed char` 表示的值是 -128~127。`wchar_t` 用来保存更大的字符集里的字符。

`int` 默认是有符号的（`signed`），这样默认的 `int` 变量就可以表示正数和负数，`unsigned int` 是无符号整数，其只能用来表示整数，一般不常用。`int` 一共有 3 种大小的用法，`short int`（简写为 `short`）；`int`；`long int`（简写为 `long`）。

#### （2）浮点型变量

浮点数也有 3 种精度大小的：`float`（单精度）、`double`（双精度）和 `long double`（扩展精度）。

#### （3）数据类型的大小

C++对象的大小是用 char 大小的倍数表示的，所以 char 的大小一般为 1，一个 C++对象的字节数可以根据运算符 sizeof 得到。每一种数据类型实际的内存大小与其系统位数有关，下表列出的是目前为主的 64 位系统，一字节为 8 位。

关键字	数据类型	内存大小
void	无类型	—（没有 void 对象，只有 void 指针）
bool	布尔型	1 个字节
char	字符型	1 个字节
wchar_t	宽字符型	2 或 4 个字节
int	整型	4 个字节
float	浮点型	4 个字节
double	双浮点型	8 个字节
	指针类型	8 个字节

## 2、数组

C++中的数组必须在定义时就指定大小，C++中的数组初始化方式如下：

```
int arr1[] = {0,1,2}; //大小为3
int arr2[5] = {0,1,2}; //大小为5，未指定的使用0补全
```

## 3、字符串

这里着重介绍下 C++中的字符串，首先介绍下**字符串常量**。在 C++中一个**单引号**用于单个**字符常量**（'a'），一个**双引号**用于单个**字符串常量**（"A"）。

字符常量本质上就是一个固定 value 大小的字符变量（'a'的类型是 const char[1]），字符串常量本质上是末尾加了一个空字符'\0' 的 const 字符数组（"A"的类型是 const char[2]）。特别地，一个字符串常量可以给一个 char 指针赋值：

```
char* p1 = "hello world"; //只读赋值
p1[2]='a' // 错误用法，不可以修改

char p2[] = "hello world"; //可读写赋值
p2[2]='a' // 正确用法，可以修改
```

C++中有两种表示字符串的形式：C 风格的字符数组和 C++引入的 string 类。string 类本质上也是封装了 C 风格的字符数组，但是其也集成了许多字符串的常见操作，是 C++推荐的表示字符串的方式。string 实际上就是使用 null 字符 '\0' 终止的字符数组

## 3、枚举类型

枚举类型可以保存一组定义的整数常量，其用法一般如下：

```
enum workday{MON,TUE,WED,THU,FRI};

switch(workday){
    case MON:
        break;
    case TUE:
        break;
    case WED:
        break;
    case THU:
        break;
    case FRI:
        break;
}
```

数据结构和类

4、指针和引用变量

接下来我们介绍下**指针**和**引用**，这是面试的高频知识，这里我们只强调一些关键信息。

(1) 指针

**指针本质上是一种特殊的变量，特殊在其 value 是另一个变量的地址，所以无论何种类型的指针，其内存大小都是固定的**（例如在 X86 中为 4 个字节，在 X64 中为 8 个字节）；**指针可以在任何时候初始化或者更改；存在空的指针。**

指针的定义如下：

```
int i = 20;
int * a = &i; // 含义：a是一个指向整型变量i的整型指针变量，注意，此处的&是取地址，不是引用
```

但是实际编程中程序员不会这么无聊地用，基本都是 new 一个类对象，然后访问其属性和方法。new 是一种操作符，和 delete 结合使用：

```
int * a=new int; // 含义：动态分配一段整型内存
delete a; // 含义：释放分配的内存
int *array=new int [m]; // 含义：动态分配一段整型数组内存
delete [] array;// 含义：释放分配的数组内存
```

当然除了 new 和 delete，也可以使用标准库函数 malloc()和 free()：

```
int *b = malloc(sizeof(int));// 动态分配一个整型变量大小的内存
free(b); // 释放分配的内存
```

new 和 malloc()的主要区别：

	new/delete	malloc()/free()
本质	运算符	标准库函数
分配大小	自动计算	作为参数手动输入
主要操作	分配/释放内存时调用对象的构造和析构函数	分配/释放内存时不调用对象的构造和析构函数
安全性	内存分配失败时抛出 bad_alloc 异常	内存分配失败时返回 null

	进行类型识别检测，返回定义时具体类型指针，如果为 int 指针分配 float 变量时报错	不进行类型识别检测，返回 void 指针，需要手动进行类型转换，如果为 int 指针分配 float 大小的字节数不报错。
--	---	---

总结，new 封装了 malloc() 方法，new 一般用来返回一个类对象，malloc() 一般用在基本变量（int、float），前者更加常用。

## (2) 引用

引用可能初学者不经常使用，但是没关系，引用本质就是一个已存在的变量的别名，所以引用的内存分配区域和大小由其指向的变量类型决定。需要注意的是引用必须在创建时初始化，之后不可更改或再次初始化；不存在空的引用；这两点与指针很不同。

引用的定义：

```
int i = 20;
int& b = i; // 含义：b是一个初始化为i的整型引用
```

引用一般用在函数的参数和返回值上：

```
// 最常用用法：用在函数的参数列表和返回值
double& setValues(int &i) {
    double& ref = vals[i];
    return ref;
}
```

## 二、常量

前文我们已经知道，常量本质上就是使用了关键字 const 的变量，使用了 const 的变量在其作用域内都不能更改其 value，const 的最常见用途是作为数组的最大界和分情况标号。

## 二、输入输出

C++标准库提供了丰富的 I/O（输入/输出）功能。首先看下 C++编程中最基本和最常见 I/O 操作。C++的 I/O 发生在流中，流是字节序列。

输入操作被 C++看做字节流从设备（如键盘、磁盘驱动器、网络连接等）流向内存；

输出操作被 C++看做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）。

### 1、标准输入输出流

下面给出标准输入输出流的演示代码：

```
#include <iostream> // 定义了标准输入流cin, 标准输出流cout, 标准错误输出流cerr, 标准日志输出流clog
using namespace std; // 引入C++标识符的命名空间
// 主程序入口
int main()
{
    char name[50];
    cout << "请输入您的名字: " << endl; //标准输出流
    cin >> name; //标准输入流
    cout << "您输入的名字是: " << name << endl; //标准输出流

    system("pause"); //暂停程序（此处防止控制台程序关闭）
    return 0;
}
```

首先要用#include 语句包含头文件<iostream>; using namespace std 语句负责引入 C++标识符的命名空间, 没有该语句的话, 则以下 cout、cin 和 endl 都得换成 std::cout、std::cin 和 std::endl。cout 可以直接输出字符串或字符数组, 只在输出 endl 时换行; cin 以阻塞的方式从控制台读取数据到字符数组中。

# 第 3 章 C++抽象篇

C++语言是面向对象语言，面向对象程序设计的四大特性：抽象、封装、继承和多态。

## 一、类的成员

C++语言通过类的概念来实现抽象和封装。一个类（`class`）中包含了成员变量和成员函数，一般 C++的结构体（`struct`）中只有成员变量，不包含大量成员函数。

### （1）类的访问权限

`class` 通过 `private`、`public`、`protect` 三个关键字来实现类成员的访问控制，`class` 的成员默认都是 `private` 类型，`struct` 的成员默认都是 `public` 类型，这也是两者最大的不同。

`class` 的 `private` 成员只准类本身内部的成员函数访问；

`class` 的 `public` 成员可供类本身、派生类及外部访问；

`class` 的 `protect` 成员可以被类本身以及派生类访问，外部不可访问；

### （2）类的构造函数和析构函数

类的构造函数和析构函数一直被其开发者认为是 C++的核心。

C++中类的构造函数具有与类相同的名字，输入参数可以自定义也可以不定义，其由 C++程序自动调用，不需要程序员显示调用。一个类通常具有多个构造函数，从而为用户提供多种不同的初始化方式。在所有构造函数中，有一类特殊的复制构造函数，其特别之处在于传入的参数是一个自身类型的参数。包含 `const` 成员类不能自动调用默认构造函数。

析构函数

### （3）类的静态成员

一个类可以实例化出多个对象，不同的对象占据不同的内存。在类成员的声明中使用关键字 `static` 就可以定义类的静态成员，不同对象的所有 `static` 成员占据同一份内存。

### （3）类的常量成员函数

在类的成员函数声明的参数表后面使用关键字 `const` 就变成了常量成员函数，常量成员函数不允许其修改类本身的变量。

成员函数

## 二、类的继承

继承是面向对象程序设计中最重要的一個概念，代表了 is-a 关系。从一个类中派生出另一个类中，原始类称为基类（或父类），继承类称为派生类（或子类）。

子类可以访问父类中所有的非私有成员，并继承所有的父类方法，以下情况除外：

- （1）父类的构造函数、析构函数和复制构造函数。
- （2）父类的重载运算符。
- （3）父类的友元函数。

类的继承有三种方式（在大部分情况下几乎都不使用 `protected` 和 `private`）：

（1）公有继承（`public`）。父类的公有成员成为子类的公有成员，父类的保护成员成为子类的保护成员，子类无法直接访问父类的私有成员，只能通过父类的公有和保护成员函数来间接访问。

（2）保护继承（`protected`）。父类的公有成员和保护成员都会成为子类的保护成员。

（3）私有继承（`private`）。父类的公有成员和保护成员都将成为子类的私有成员。

### 三、类的重载

C++ 允许同一个作用域中的某个函数或运算符具有多个定义，这也叫做函数重载和运算符重载。重载声明是指该作用域内声明的函数或方法具有相同名称，但是参数列表和实现不相同。C++ 编译器会自行选用最合适的定义，即重载决策。

### 三、类的多态

类的多态指的是调用类的成员函数时会根据调用函数的对象的类型来执行不同的函数。虚函数是在父类中使用关键字 `virtual` 声明的函数。假设父类有一个方法 `area()`，子类也有一个相同的方法 `area()`。

假设父类的方法 `area()` 没有声明为虚函数，则调用子类的 `area()` 时会默认执行父类的 `area()`，这就是所谓的静态多态（或静态链接），函数调用在程序执行前就准备好了。有时候这也被称为早绑定，因为 `area()` 函数在程序编译期间就已经设置好了。

假设父类的方法 `area()` 声明为虚函数，则调用子类的 `area()` 时会默认执行子类的 `area()`，这就是所谓的动态链接，或后期绑定。虚函数的声明会告诉编译器不要静态链接到该函数。

虚函数必须进行实现，且子类不一定需要覆盖该函数；纯虚函数（也叫抽象函数）则不需要实现，且子类一定需要覆盖该函数。

在上述举例说明中，子类通过将父类方法声明为虚函数就可以在子类覆盖其实现，总结起来，子类方法想要覆盖父类方法需要满足如下要求：

一虚：父类中被覆盖的成员函数必须声明为虚函数。

二容：父类和子类的成员函数的返回类型和异常规格必须兼容

四同：父类和子类的成员函数的名程、形参类型、常量属性和引用限定符必须完全相同。

`override` 是 C++11 中添加的安全声明的关键字，子类要覆盖的方法可以添加 `override` 关键字来声明，编译器将会对该函数和父类的对应函数进行一虚二容四同的比较，错误的地方会报错。注意，不添加 `override` 关键字也是可以的，但是出现问题时编译器可能不会报错。

在子类中调用父类的方法：父类名::`方法名`



# 第 4 章 STL 与泛型编程

C++标准库（Standard Library）可以看做是 STL+其他 API 函数，其中 STL 是系统性的一个开源库。C++标准库中提供的容器和函数都以头文件的形式提供，例如`#include<vector>`同时头文件内的组件封装于命名空间“std”。

STL（Standard Templates Library）是 C++的标准模板库，泛型编程就是使用 `template`（模板）为主要工具来编写程序，在 C++中泛型编程就是使用 STL 进行编程。泛型编程与面向对象编程在设计思想上不一样。面向对象将数据和对其操作封装为一个对象，泛化编程将数据和操作分离。

## 一、STL 的六大部件

### 1、六大部件的关系

STL 中主要有六大部件：

容器（Containers）是存储数据的数据结构，容器封装了内存分配的机制。

分配器（Allocators）为容器的内存分配机制由提供支持。

算法（Algorithms）封装了对于容器内数据的常见操作。

迭代器（Iterators）是一种泛化的指针，可以看做是泛型编程中的指针。算法通过迭代器来操作容器内的数据。

仿函数（Functors）的作用类似于函数。

适配器（Adapters）可以对容器、迭代器、仿函数进行转换。



### 2、六大部件的使用案例

如下代码使用了六大部件，其中常见的是容器、算法和迭代器。

```

01 #include <vector>
02 #include <algorithm>
03 #include <functional>
04 #include <iostream>
05
06 using namespace std;
07
08 int main()
09 {
10     int ia[ 6 ] = { 27, 210, 12, 47, 109, 83 };
11     vector<int, allocator<int>> vi(ia, ia+6);
12
13     cout << count_if(vi.begin(), vi.end(),
14                     not1(bind2nd(less<int>(), 40))) << endl;
15
16     return 0;
17 }

```

Diagram illustrating the components of the code:

- allocator**: Points to `allocator<int>` in the `vector` constructor.
- container**: Points to `vector` in the `vector` constructor.
- iterator**: Points to `vi.begin()` and `vi.end()` in the `count_if` function call.
- algorithm**: Points to `count_if` in the `count_if` function call.
- function adapter (negator)**: Points to `not1` in the `not1(bind2nd(less<int>(), 40))` expression.
- function adapter (binder)**: Points to `bind2nd` in the `not1(bind2nd(less<int>(), 40))` expression.
- function object**: Points to `less<int>()` in the `not1(bind2nd(less<int>(), 40))` expression.
- predicate**: Points to the entire `not1(bind2nd(less<int>(), 40))` expression.

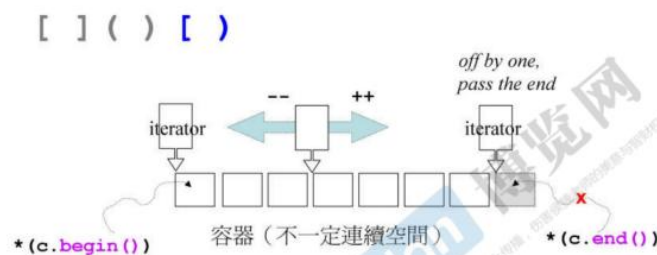
— 侯捷 —

13

## 二、容器的遍历

### 1、前闭后开区间的通用遍历

STL 中的任意一种容器都包含两个迭代器 `c.begin()` 和 `c.end()`。容器内的 `c.begin()` 和 `c.end()` 指定的区间是一个前闭后开的区间，即 `c.begin()` 指向容器内的第一个元素，`c.end()` 指向容器内的最后一个元素的后一个空位：



根据这两个迭代器，遍历 STL 中任何一个容器都可以有以下写法：

```
Container<T> c;
```

```
for(auto iter=c.begin(); iter!=c.end(); ++iter);
```

其中 `auto` 关键字是让编译器自动推断 `iter` 的类型, `iter` 的类型就是其容器对应的迭代器类型 `Container<T>::iterator`。

容器内的 `c.begin()` 和 `c.end()` 指定的区间是一个前闭后开的区间，STL 中的所有容器都包含这两个迭代器，所以遍历任何一个容器都可以有以下写法：

## 2、C++11 新加的遍历写法

在现代 C++11 中添加了遍历容器的一种简便写法：

```
for(auto num:arr) cout<<num;
```

请注意，这种遍历方式只是以只读的形式读取容器中的每个元素，如果想要遍历修改容器内的每个元素，可以采取引用的写法。

```
vector<int> arr={1,2,3,4,5,6};
```

方式一：for(auto& num:arr) {num=num\*3;} // 3,6,9,12,15,18

方式二：for(auto num:arr) {num=num\*3;} // 1,2,3,4,5,6

## 三、容器的结构与分类

STL 中的容器可以分为序列式容器（Sequence Containers）和关联式容器（Associative Containers）。

关联式容器通过 key 来进行快速、大量地查找，set/multiset/map/multimap 的底层基本都使用红黑树，multi 意味着允许 key 重复。

其中 C++11 在关联式容器中添加了一类新的不定序式容器（Associative Containers），其底层使用哈希表的结构（HashTable），查找效率可以接近 O(1)。

这些容器的头文件导入如下：

```
#include<iostream>
#include<algorithm>
using namespace std;
// 序列式容器
#include<array> //一段固定长度的顺序存储的数组容器
#include<vector> //一段可以动态扩充的顺序存储的数组容器

#include<queue> //一段可以单向扩充的顺序存储的单向队列容器
#include<deque> //一段可以双向扩充的顺序存储的双端队列容器

#include<stack> //一段可以单向扩充的顺序存储的栈容器

#include<list> //一段可以双向遍历的链式存储的双向链表容器
#include<forward_list> //一段可以单向遍历的链式存储的单向链表容器
// 关联式容器
#include<set> // 包含 set/multiset
#include<map> // 包含 map/multimap
#include<unordered_set> // 包含 unordered_set/unordered_multiset
#include<unordered_map> // 包含 unordered_map/unordered_multimap
```

# 附录 C++常见面试题

## 一、C++的语言特性

### 1、C 语言和 C++有什么区别？

两者最主要的区别是编程语言的设计思想。C 语言是一种**面向过程的编程语言**，其使用大量的函数来处理数据，数据和函数一般是分离的。C++语言是一种**面向对象的编程语言**，其将数据和操作数据的函数封装为类，通过抽象出一个个的对象模型来解决问题。C 语言目前适合于代码体积小、效率高的场景，例如嵌入式；C++多用于大型的底层系统场景。

其次是语法的不同，例如 C 语言中的 struct 不可以有函数，而 C++中的 struct 可以有函数，**C++包含 C 语言，而且还支持泛型编程。**

### C++和 Java 有什么区别？

	编译流程	语法规则	效率	跨平台	学习难度
C++	<b>编译性语言</b> ，其源代码经过 C++编译器编译和链接后生成可执行代码。	C++支持 <b>面向对象</b> 、 <b>面向过程</b> 和 <b>泛型编程</b> ，可以定义 <b>全局变量</b> 和 <b>全局函数</b> 。 C++ <b>支持指针</b> ，允许程序员动态分配内存	最高	不可以	最高
	<b>静态语言</b> ，变量定义必须有类型声明	<b>支持多继承</b> 、 <b>函数重载</b> 、 <b>运算符重载</b> 等			
Java	<b>解释性语言</b> ，其源代码经过 Java 编译器编译后需由 Java 虚拟机（JVM）解释执行	Java 只支持 <b>面向对象</b> ，所有代码必须在类中实现，不存在全局变量或者全局函数。 Java <b>没有指针概念</b> ，使用垃圾回收器实现内存的自动回收，对程序员来讲，Java 内存管理更加安全	较高	可以	较高
	<b>静态语言</b> ，变量定义必须有类型声明	Java <b>支持函数重载</b> ， <b>不支持多继承</b> 、 <b>运算符重载</b> 等			
Python	<b>解释性的脚本语言</b> ，其源代码直接由 python 解释器生成并输出	Python 支持 <b>面向对象</b> 和 <b>面向过程</b> ，可以定义全局变量和全局函数。 Python <b>没有指针概念</b> ，使用垃圾回收器实现内存的自动回收，对程序员来讲，Python 内存管理更加安全	较低	可以	较低
	<b>动态语言</b> ，变量定义无类型声明	Python <b>支持多继承</b> ，不支持 <b>函数重载</b> 和 <b>运算符重载</b> 等			

### 面向对象的三大特性/四大特性是什么？C++如何实现这些特性？/谈谈对面向对象的理解

面向对象的四大特性分别是抽象、封装、继承和多态。

C++通过类来实现**抽象**和**封装**。一个 C++类将数据和操作数据的方法封装在一起，数据变成了成员变量，方法变成了成员函数。类可以通过控制变量和方法的访问权限（public、protected、private）来达到对外隐藏变量和方法的目的，更加安全。

C++提供类的继承机制，而且允许单继承和多继承，类的继承是让某种类型的对象快速获得另一个类型对象的属性和方法。可以实现代码的重用，节省开发时间。

C++提供类的多态机制，当 C++类存在继承关系时，就可能存在多态。类的多态指的是调用类的成员函数时会根据调用函数的对象的类型来执行不同的函数。

## 二、C++的编译机制

请简述一下 C++为什么要使用头文件？/头文件的作用是什么？

(1) **头文件用来保存程序的声明。**C++是一种静态编译的编程语言，C++中的变量必须先声明后定义，C++需要将一些重复性使用的声明保存在头文件中。

(2) **头文件可以提升程序安全性。**C++中有些核心代码是不能对用户提供的，但是其代码又必须被用户引用。C++可以向用户提供头文件和编译后的二进制的函数库(例如.lib 和.dll)，用户只需要按照头文件中的接口声明就可以调用函数库功能。

(3) **头文件具备说明文档的作用。**C++中代码的使用必须和头文件中的声明一致，这就大大减轻了 C++程序员调试程序的负担。

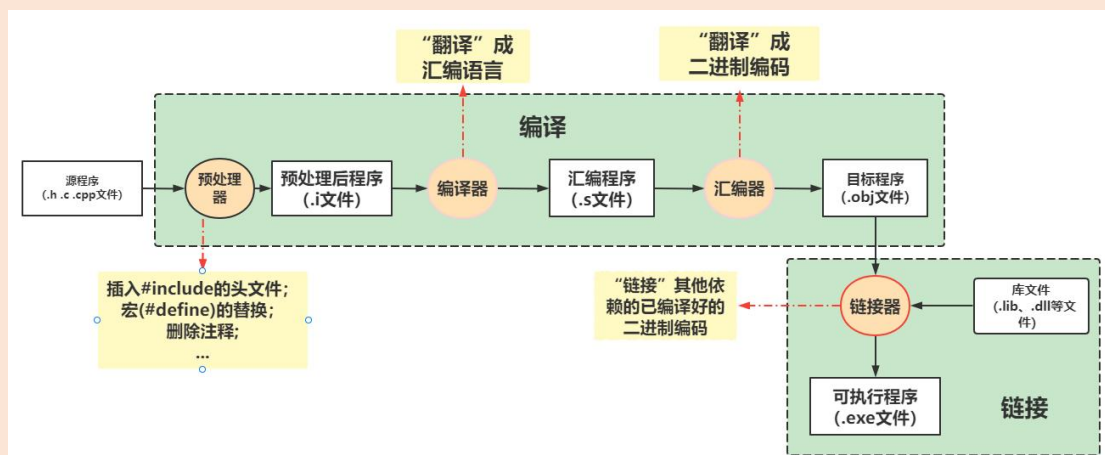
请简述以下 C++的（文件）编译机制？

C++是一门编译性的编程语言，其源文件（.cpp）到编译为（.exe）基本需要以下 4 个阶段：  
第一阶段：**预处理阶段**。主要是一些代码文本的替换工作，处理#开头的预处理指令，比如拷贝#include 包含的文件代码，#define 宏定义的替换，条件编译等。

第二阶段：**编译阶段**。将上一阶段的文件“翻译”成汇编语言。

第三阶段：**汇编阶段**。把汇编语言“翻译”成二进制语言。

第四阶段：**链接阶段**。一个.cpp 文件编译汇编为一个.obj 文件，但是最终的.exe 执行文件只有一个，链接阶段将多个.obj 文件和静态链接库（.lib）、动态链接库（.dll）组装成最终的执行文件。



请简述一下 C++的预编译机制？

C++的预编译机制又称为 C++的预处理，它是 C++在正式编译源程序之前的预处理步骤，其主要是做些代码文本的替换工作。**删除注释**，处理#开头的指令,比如拷贝**#include 包含的文件代码**，**#define 宏定义的替换**，**条件编译**等。请注意，#开头的预处理指令不属于 C++语句，因为其没有以;结尾。

请简述一下#include<>和#include""的区别。

#include<>一般用于包含 C++自身的标准头文件，其从 C++的默认头文件路径导入文件。

#include""一般用于包含程序员自定义或者第三方的头文件，其现在自定义目录寻找，找不到才会寻找 C++的默认头文件路径。

请简述一下 C++的条件编译？简述下#define #endif 和#ifndef 的作用

C++的条件编译是 C++预处理阶段的步骤之一，是根据定义宏来进行代码静态编译的手段，最常见的是在 C++的头文件中引入和#ifndef、#define 和#endif 来防止文件编译中重复包含头文件的内容。

请简述一下静态链接和动态链接的区别。

静态链接在 C++的文件链接阶段就将各种目标程序文件（.obj 文件）和第三方提供的静