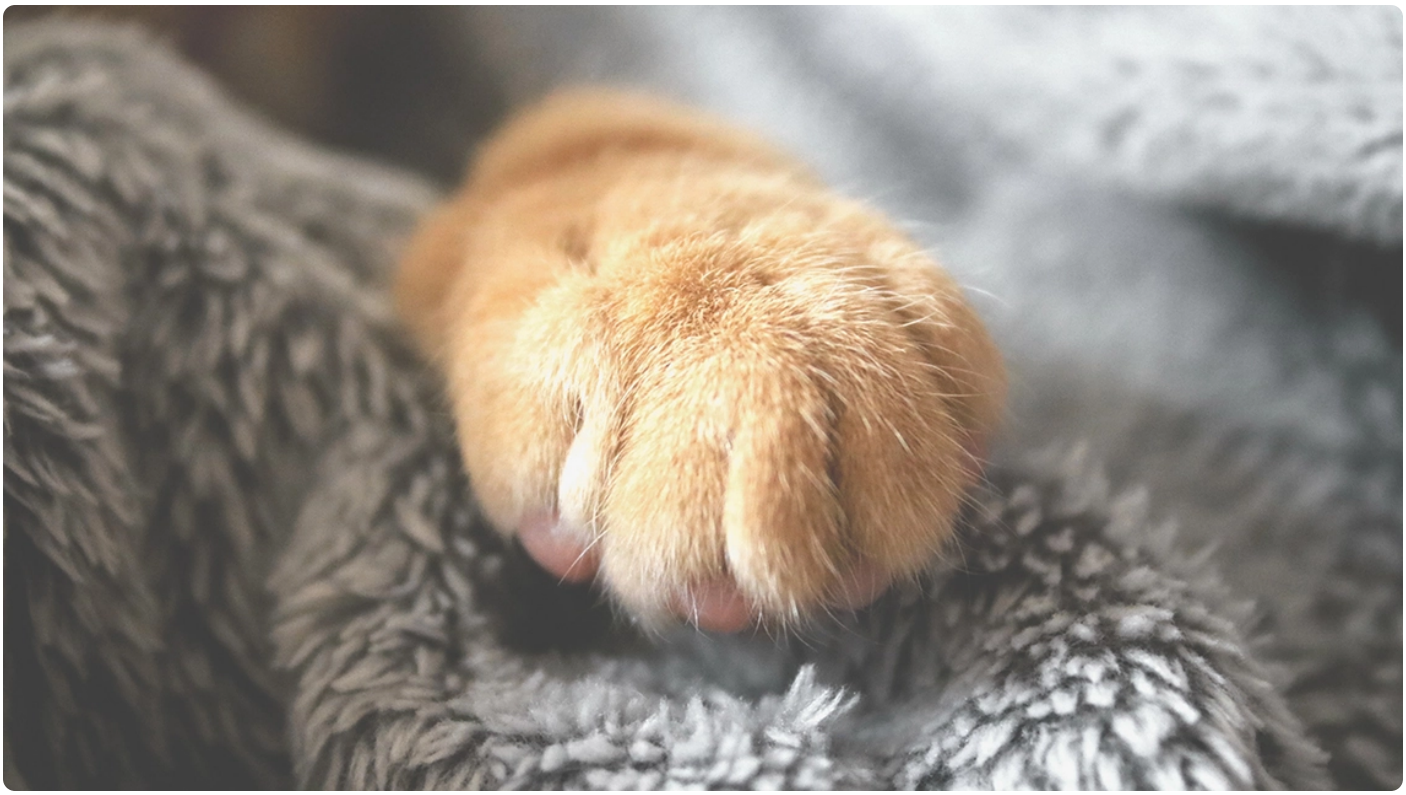
加餐 | 和 C 语言相比，C++ 有哪些不同的语言特性？

2022-03-14 于航

《深入C语言和程序运行原理》 课程介绍 >



讲述：于航

时长 12:22 大小 11.33M

你好，我是于航。

在之前的春节策划三里，我为你介绍了如何使用 C++ 语言来实现一个简单的 JIT 编译器，你能够从中大致感受到 C 和 C++ 这两种语言在使用上的差异。那么这次的加餐，我就用专门的一讲，来为你介绍 C++ 与 C 相比究竟有何不同。

C 语言的语法简单，且贴近底层硬件。使用它，我们能在最大程度上为程序提供较高的运行时性能。但在 C 语言诞生的几年后，它开始逐渐无法满足人们在构建应用程序时对编码范式的需求。而一门基于 C 语言构建的，名为 C++ 的语言便由此诞生。作为一个 C 语言使用者，也许你还在犹豫要不要把 C++ 作为下一门继续学习的编程语言，那么相信在学习完这一讲后，你会有进一步的思考。

在具体比较 C 和 C++ 之前，我们先来看看 C++ 的发展历史和应用场景，从整体视角对它有一个了解。

C++ 发展简史

丹麦计算机科学家 Bjarne Stroustrup（下面简称 BS）从 1979 年开始研发 C++ 这门编程语言。BS 于 1979 年从剑桥大学取得博士学位，在撰写博士论文的过程中，他发现 Simula 语言（该语言被认为是第一个面向对象的编程语言）的某些特性对大型软件的开发十分有帮助，但遗憾的是，该语言本身的执行效率却并不高。

毕业之后，他前往位于美国新泽西州的贝尔实验室，以研究员的身份开始了职业生涯。在工作期间，为了解决遇到的一个棘手问题，BS 决定选择当时被广泛使用的，具备较高性能和兼容性的 C 语言作为基准语言，开始为其添加与 Simula 类似的，面向对象的相关特性。通过直接修改 C 编译器，包括类、继承、默认参数等在内的一系列新特性被“整合”到了 C 语言中，这一新语言在当时也被直观地称为 “C with Classes”。

而直到 1983 年，BS 才将 “C with Classes” 正式更名为 C++，并开始为它添加虚函数、运算符重载、引用类型等更多新特性。同时，BS 也为该语言单独开发了第一款专用的编译器 Cfront。该编译器可用于将 C++ 代码转译为 C 代码，而这些 C 代码需要再通过其他 C 编译器的处理后，才能够生成最终的目标文件。

在接下来的日子里，C++ 开始了不断“进化”的过程。这里，我将其中的一些重要时间节点整理如下：

- 1984 年，BS 为 C++ 添加了第一个流式的 IO 操作库；
- 1985 年，第一本介绍 C++ 的权威书籍 “The C++ Programming Language” 第一版发布；
- 1989 年，Cfront 2.0 发布，该版本为 C++ 语言新增了多重继承、抽象类、静态成员函数等特性；
- 1991 年，ISO C++ 委员会成立，C++ 开始进入语言标准化时代；
- 1998 年，C++98 发布，该版本新增了 RTTI、模板初始化、类型转换操作符等语言特性。同时，标准库也增加了对智能指针，以及 STL 相关容器和算法的支持；
- 2003 年，C++03 发布，该版本修复了 98 中的一些重要 BUG；
- 2011 年，C++11 发布，该版本新增了很多重要的语言特性。可以说，C++11 仍然是目前工业界内使用最多的 C++ 语言版本之一。

在随后的十年里，C++14、17，乃至 20 标准也都相继发布，而下一代的 C++23 标准也在酝酿之中。这些标准版本的迭代，为 C++ 语言注入了大量新鲜“血液”，使得 C++ 变得越来越复杂和庞大（C++20 标准的文档手册已经超过了 1800 页）。好在 C++ 也足够灵活，你可以仅使用它的一小部分重要特性来完成所有基本工作。而如果你想要进一步优化或抽象程序，C++ 也同样具备这些能力，只不过这也会带来相应的使用成本提高。

从整体发展趋势来看，C++ 旨在成为一个大而全的，支持多范式的系统级编程语言。

C++ 应用场景

C++ 与 C 在应用场景上有很大的重合，它们都可以应用于那些需要高运行时性能，或与底层硬件打交道的场景中。但总的来看，C++ 赋予了开发者进一步抽象程序逻辑的能力。但有些时候，这些抽象也会导致编译器无法保证其编译产物在机器代码层面能够获得完全一致的表现。

因此，对于某些较低层次的软件实现（如驱动程序），C 仍然是主流开发语言。同样地，当 C++ 需要与其他编程语言进行“通信”时（如 FFI），它们通常都会使用 `extern "C" {}` 等方式，来将两方的接口调用规范限定为 C 语言的 ABI，以在最大程度上保证兼容性。

不过我们也知道，无论如何，抽象都是会带来成本的。因此，在一些对程序性能有着极端要求的场景下，C 语言通常会表现得更好，更稳定。但另一方面，相较于 C 语言，**C++ 往往可以带来更高的开发效率和可维护性**。比如，C++ 在 STL 中为我们提供了可以直接使用的各种容器类型，如大小可自动伸缩的 `std::vector` 。但在 C 语言中，类似的数据结构则需要我们自行构建。因此，具体应该选择 C 还是 C++，我们就要视情况而定了。

除此之外，使用 C++ 构建的知名开源项目也有很多，比如 JavaScript 引擎 V8、浏览器引擎 Chromium、LLVM Compiler 套件、openJDK，等等。C++ 在 2022 年 3 月份公布的 TIOBE 榜单上位列第 4 名（C 语言为第 2 名），它在工程领域的使用人数之多和应用范围之广毋庸置疑。

接下来，我们具体看看在语言特性方面，C++ 跟 C 究竟有哪些不同。

C++ 和 C 在语言特性上的不同

需要注意的是，C++ 并非完全支持 C 语言的所有语法形式。比如，C++ 会执行比 C 更严格的类型规则检查和初始化要求。举个例子，下面这段代码是合法的 C 代码，但无法被 C++ 编译器正常编译：

复制代码

```
void foo(void) {
  void *ptr;
  int *i = ptr;
}
```

除此之外，在编码体验上，C++ 也为我们提供了更多易用的语言能力。下面我们以 C++11 标准为例，来看看相较于 C 语言，C++ 有哪些完全不同的特性。

类

正如 C++ 的前身 “C with Classes” 的名字所表达的那样，C++ 引入了可用于支持面向对象（OOP）编程的相关特性和语法元素，比如类、继承、虚函数，等等。虽然在 C 中，我们也可以模拟 OOP 这种编程范式，但由于缺少语言层面的相关支持，能够实现的功能有限。且编译器对类定义、继承关系及访问权限等方面的检查往往也不够严格。而 C++ 则直接从语言层面入手解决了这个问题。

这里，我们可以通过下面这段代码，直观感受下 C++ 中与此相关的语法形式。限于篇幅，我不会完整介绍其中使用到的每一个 C++ 特性，但你可以参考注释信息来理解每一段代码的具体作用。

复制代码

```
#include <iostream>
enum class VehicleType {  // 枚举类；
  SUV, MPV, CAR
};
class Vehicle {  // Vehicle 基类，属性默认私有；
  VehicleType type = VehicleType::CAR;
  int seats = 5;
 public:
  Vehicle() = default;  // 默认构造函数；
  Vehicle(VehicleType type, int seats) : type(type), seats(seats) {}  // 普通构造
  Vehicle(const Vehicle& v) { type = v.type; seats = v.seats; }  // 拷贝构造函数；
  void spec(void) {  // 成员函数；
    std::cout << "Vehicle has " << seats << " seats" << std::endl;
  }
};
struct Car : Vehicle {  // 派生类 Car，继承自 Vehicle；
  Car() = default;
  Car(int seats) : Vehicle(VehicleType::CAR, seats) {}
  Car(const Car& c) : Vehicle(c) {}
  void stop(void) {  // 成员函数；
    std::cout << "Car stops immediately!" << std::endl;
```

```
  }
  void stop(int delaySecs) {  // 重载的成员函数；
    std::cout << "Car stops after " << delaySecs << " seconds!" << std::endl;
  }
};
int main(void) {
  Car carA { 7 };  // 初始化一个 Car 类型对象；
  auto carB = carA;  // 拷贝对象 carA；
  carB.spec();
  carA.stop(10);
  return 0;
}
```

可以看到，我们分别在代码的第 5~15 行与 16~26 行定义了不同的两个类。其中，类 Car 继承自类 Vehicle。在每一个类定义中，我们都为它设置了相应的默认构造函数、拷贝构造函数，以及普通构造函数，这些函数控制着每一个类对象的具体构造过程。

除此之外，每一个类在其定义中也都包含有相应的成员属性与成员方法。这些成员反映了类对象的具体状态，以及对外的可交互接口。最后，在 main 函数中，通过列表初始化的方式，我们构造了类 Car 的一个对象，并完成了该对象的拷贝与成员函数的调用。

除此之外，C++ 中还有很多用于支持 OOP 的特性，如多重继承、移动构造函数、虚继承等等。这些特性都极大地增强了 C++ 支持 OOP 编程范式的灵活性。

STL

除了这些用于支持 OOP 的特性外，从标准库方面来看，C 与 C++ 的一个最大不同是，C++ 在其标准库中增加了对常用容器类型（如向量、双向链表、双端队列）的支持。这些数据结构与相应的算法、迭代器和适配器等，一同组成了“标准模板库（Standard Template Library，STL）”的主要内容。对这些容器类型的合理使用可以帮助我们大幅提高生产效率。

我们来看看下面的这个例子：

复制代码

```
#include <iostream>
#include <vector>
#include <algorithm>
int main(void) {
  std::vector<int> v { 1, 2, 3, 4, 5 };  // 构造一个 std::vector 对象 v；
  std::for_each(  // 对 v 的内容进行遍历；
    v.begin(),  // v 的首元素迭代器；
```

```
    v.end(),    // v 的尾后迭代器;
    [](int& n){ std::cout << n << std::endl; });  // 遍历时对每个元素应用的 lambda
  return 0;
}
```

这里，我们在代码的第 5 行创建了一个 `std::vector` 类型的对象 `v`。你可以将这个容器类型简单理解为**一个大小动态可变且内存连续的数组**。在代码的第 6~9 行，我们使用 STL 中的函数模板 `std::for_each`，对该对象内的元素进行了遍历。该模板共接收三个参数，前两个参数用于通过迭代器来定位需要迭代的范围，最后一个参数则指明了对于每个迭代元素的具体处理方式。

在 C++ 中，STL 内所有标准容器类型的数据访问，都可以通过迭代器来进行。迭代器通过提供一组通用接口，屏蔽了不同容器在内部实现上的差异。

模板

模板是 C++ 中用于支持泛型编程的一个重要特性，C++ 编译器可以在编译时对每一处模板使用，根据其调用参数的实际情况进行自动推导和匹配。最简单的一种模板使用方式是函数模板，来看下面这个简单的例子：

复制代码

```
#include <iostream>
template <typename T>  // 函数模板;
T max(T x, T y) {
  return x < y ? y : x;
}
int main(void) {
  double x = 10.1, y = 20.2;
  std::cout << max(x, y) << std::endl;  // 模板实例化并调用;
  return 0;
}
```


这里，我们构建了一个名为 max 的函数模板，该模板将根据输入的实参类型（即符号 T）来实例化相应的具体函数实现。函数在内部会判断两个输入参数的大小，并将其中较大者返回。通过这种方式，我们便能构建支持泛型参数的函数。

当然，模板的功能并不止于此。对于下面这个例子，编译器甚至可以在编译阶段就直接计算出给定参数 N 的阶乘，而不会带来任何运行时负载。

复制代码

```
#include <iostream>
template <unsigned N>
struct factorial {
  static constexpr unsigned value = N * factorial<N - 1>::value;
};
template <>
struct factorial<0> {
  static constexpr unsigned value = 1;
};
int main(void) {
  std::cout << factorial<4>::value << std::endl;
  return 0;
}
```

事实上，C++ 中的模板是图灵完备的，这意味着你可以单纯使用模板来构建一个编译时“运行”的图灵机。当然，前提是你觉得损失一定的代码可读性是能够接受的（笑）。

如今，这种使用模板进行编译期计算的编码方式已经“自成一派”，通常被称为“模板元编程（Template Programming，TMP）”。

智能指针

C++ 中智能指针的出现，主要是为了解决 C 编程中指针的使用不当所可能导致的内存泄露问题。智能指针借助 RAII 机制，使得堆上动态分配的资源可以随着栈对象的创建和销毁，相应地做出自动分配与回收。在这种情况下，我们只要使用方法得当，内存泄露问题便可以得到有效控制。来看下面这个例子：

复制代码

```
#include <iostream>
#include <memory>
#include <string>
struct Person {
  std::string name;
  Person(const std::string& name) : name(name) {}
};
void decorator(std::shared_ptr<Person> p) {
  p->name += ", handsome!";  // 通过智能指针访问对象数据；
}
int main(void) {
  // 创建一个指向 Person 类对象的，基于引用计数的智能指针(shared_ptr)；
  auto personPtr = std::make_shared<Person>("Jason");
  decorator(personPtr);
  std::cout << personPtr->name << std::endl;
```

```
16 }
```

这里，我们首先在代码的第 4~7 行创建了名为 Person 的类。紧接着，在代码的第 13 行，我们通过名为 `std::make_shared` 的函数模板，在堆上创建了一个 Person 类的对象。函数在调用后会返回一个 `std::shared_ptr<Person>` 类型的智能指针，并将其指向该对象。`std::shared_ptr` 是一种基于引用计数的智能指针，它会通过计算所指向对象的实际被引用次数，来在适当时机自动将该对象资源回收。

其他

除了上面介绍的几个重要的 C++ 特性外，C++ 还有一些同样重要的语言功能，比如列表初始化、`decltype` 运算符、cast 类型转换操作符、右值引用，等等。不仅如此，如果上升到 C++20 标准，新加入的 “Big Four” 四大特性（Concepts、Ranges、Coroutines、Modules）也是我们不应错过的 C++ 重要改变。今天的介绍只是为你勾勒了一个 C++ 语言特性的大致图景，至于这些更加丰富的内容，就需要你在接下来的学习旅程中去自行探索了。

总结

这一讲，我向你简单介绍了 C++ 的发展历史和应用场景，然后主要带你看了 C 与 C++ 在语言特性上的一些重要区别。

C 是一种直接抽象于汇编语言之上的高性能编程语言。这种语言语法简单，且没有对机器代码做过多抽象，因此它被广泛应用在操作系统、编译器等需要保持足够性能，且 ABI 稳定可控的底层系统软件上。

和 C 相比，C++ 向上做了更多的抽象和扩展。C++ 提供了 STL、类、模板、智能指针等一系列新特性，这使其可以被应用在多种编程范式中。这些特性也使得基于 C++ 进行的大型软件开发变得更加高效，而且进一步避免了使用原始 C 时容易出现的内存泄露等问题。但有利就有弊，过多的新特性和抽象也使得 C++ 语言变得复杂和臃肿。所以每次面试候选人时，如果对方表示自己精通 C++，我通常也会心存疑虑。

总之，如果看完这一讲后你对 C++ 产生了一些兴趣，那么你可以在 C 语言的基础上继续对它展开学习。但是，是否要在工作项目中使用它，还需要你从多方面衡量，视具体情况而定。

思考题

你之前使用过 C++ 吗？如果用过，你觉得这门语言有哪些优点和缺点呢？欢迎在评论区告诉我你的想法。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

上一篇　大咖助阵丨海纳：C 语言是如何编译执行的？（二）

下一篇　大咖助阵丨海纳：C 语言是如何编译执行的？（三）

精选留言 (2)

写留言

LDxy 置顶
2022-03-14

1，C++在调用C写的函数时感到很困惑，有的人说要在C函数前加上:: 有的人又说不用加可以直接调。2，在C++中包含处理C字符串的头文件，有人说包含string.h就行，有人说要包含cstring。3，智能指针是可以在频繁创建对象且对程序性能有要求的场合用吗？4，源文件的后缀名也很糊涂，命名为*.cpp, *.cc, *.hpp, *.hh, *.h的都有，到底该用哪种？5，std::this_thread::sleep_for(std::chrono::milliseconds(50))和Sleep(50)是等价的吗？

作者回复: 很棒的问题！

1. 这个问题应该取决于你的代码是怎么写的，通常来说，通过 “#include” 包含进来的 C 头文件应该是在全局作用域的，因此，调用的时候自然也是需要使用全局作用域中的那个，但实际上需不需要 “::” 则取决于你的具体调用地；

2. 这个肯定还是建议包含以 “c” 开头的这些头文件；

3. 实际上是可以使用的，但对于性能这类问题，建议还是以具体的 case 入手再进行分析，看瓶颈在哪里，而不是听到智能指针有一些性能损耗就直接放弃。就比如边界检查实际上也有开销，但大多数情况都不是影响程序性能的那 80% 的重要因素，在流水线化的处理器上根本不成问题；

4. 后缀名实际上对编译来说没有区别，选择配套的来用就行，比如 .cc 与 .hh；.cpp 与 .hpp。但混用关系也不大，可能会影响某些老的调试工具的某些功能。

5. 这两个方法的抽象层次不同，一个是语言标准库中定义的，一个是 POSIX 标准中定义的。两者可能在调用上有一定重合，比如编译器在实现 sleep_for 时可能会在底层直接调用操作系统提供的 sleep 方法。具体依编译器而定。

共 2 条评论 >

ahack

2022-03-15

这一节课收获很大，印象最深刻的在于c++是c的内容的选择并增添了类的特性，除此之外还针对频繁的内存管理生产出了智能指针，再就是性能这里有封装有优化就会有性能开销，所以在某些极致的场景下没有添加任何额外开销的c性能更优秀，我这才懂了为啥马斯克说我支持rust

共 1 条评论 >