C++ For Beginners

Lesson 1 Junior：计算机体系组成

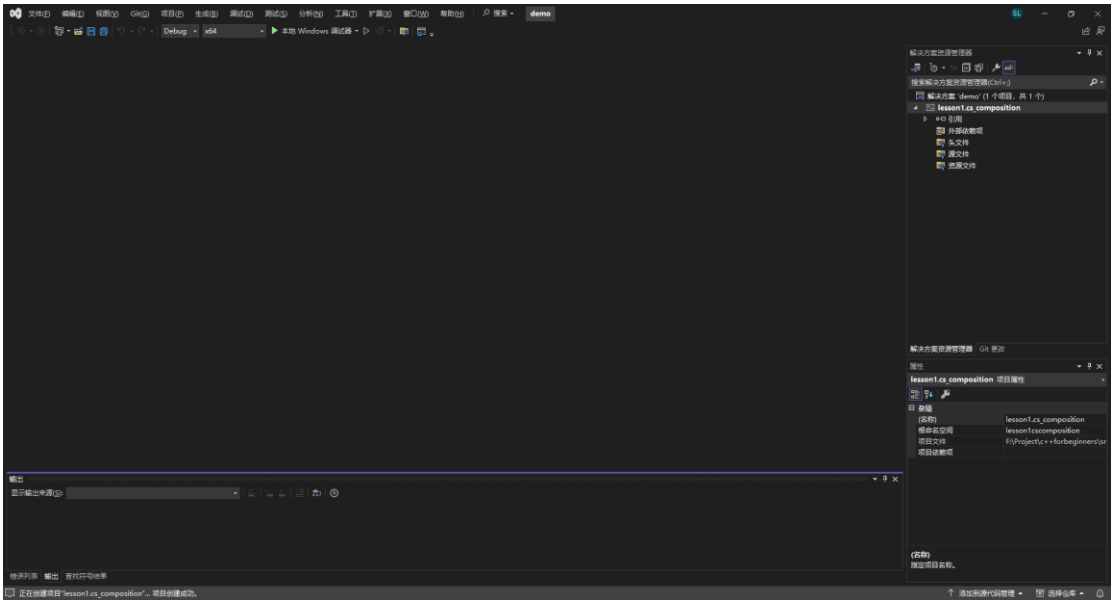
1.1 部署开发环境并运行第一个程序

工欲善其事，必先利其器。对于初学者而言，一个优秀的编程工具可以帮助你找出代码中的大量错误，而智能补全工具则可以帮助你记忆各种复杂的名字。在本课程中，我们使用 Visual Studio 2022 Community Edition（下称 VS2022）作为开发工具。VS2022 是由微软开发的一个集成开发环境（IDE），下载链接是 https://visualstudio.microsoft.com/zh-hans/。

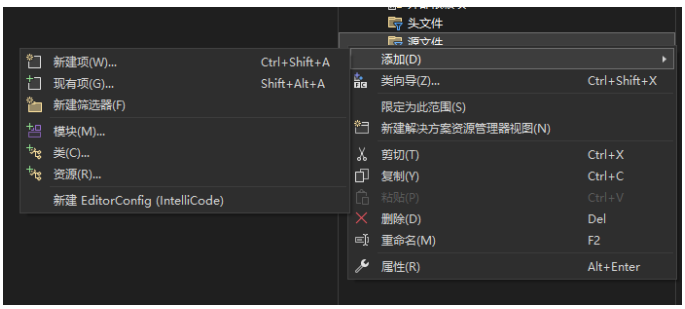
下载完成后双击 VisualStudioSetup.exe 即可进行安装，安装完成后找到 Visual Studio Installer，启动软件后找到“可用”一栏中的 Visual Studio Community 2022，点击“安装”，下拉找到“使用 C++的桌面开发”并勾选，点击“安装”。

安装完成后找到 Visual Studio 2022 并打开，现在界面应该是这样：

选择“创建新项目”，找到“空项目”，点击下一步，给新项目起一个名字，然后点击创建即可。你现在看到的界面应该如下图所示：



右键“源文件”，选择“添加”“新建项”，选择“C++文件（.cpp）”后点击添加。

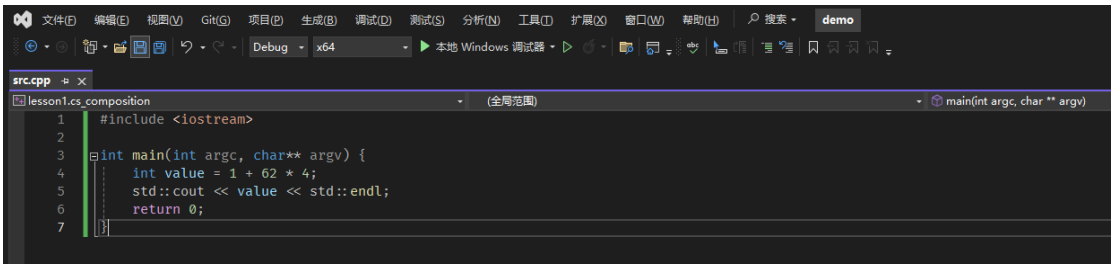


在左边出现的文本编译器中复制并粘贴以下代码：

```
#include <iostream>

int main(int argc, char** argv) {
  int value = 1 + 62 * 4;
  std::cout << value << std::endl;
  return 0;
}
```

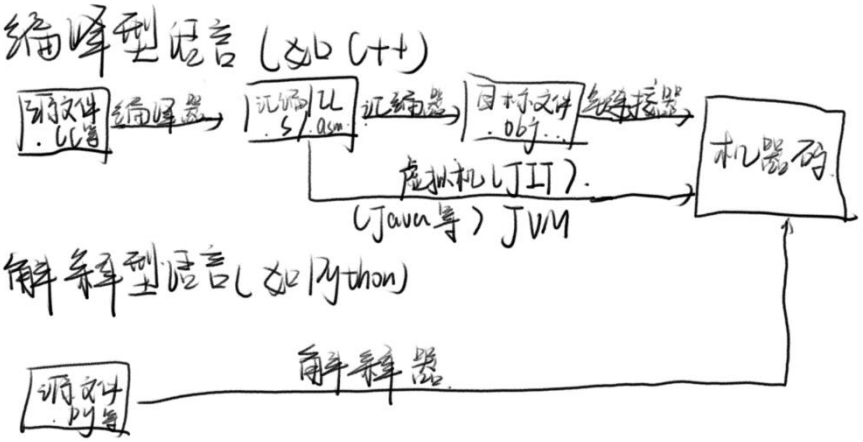
然后点击“本地 Windows 调试器”，等待一会即可出现结果：



这就是你编写的第一个 C++程序，如果你没有相关基础，我会很肯定你理解不了这些英文，不过没关系，在你学完剩下的六节课的基础部分后，你将彻底了解这些代码的含义。

1.2 浅淡编程语言

IDE 为你隐藏了大量代码背后的细节，但是我们并不能忽略这些细节。下面我将讲



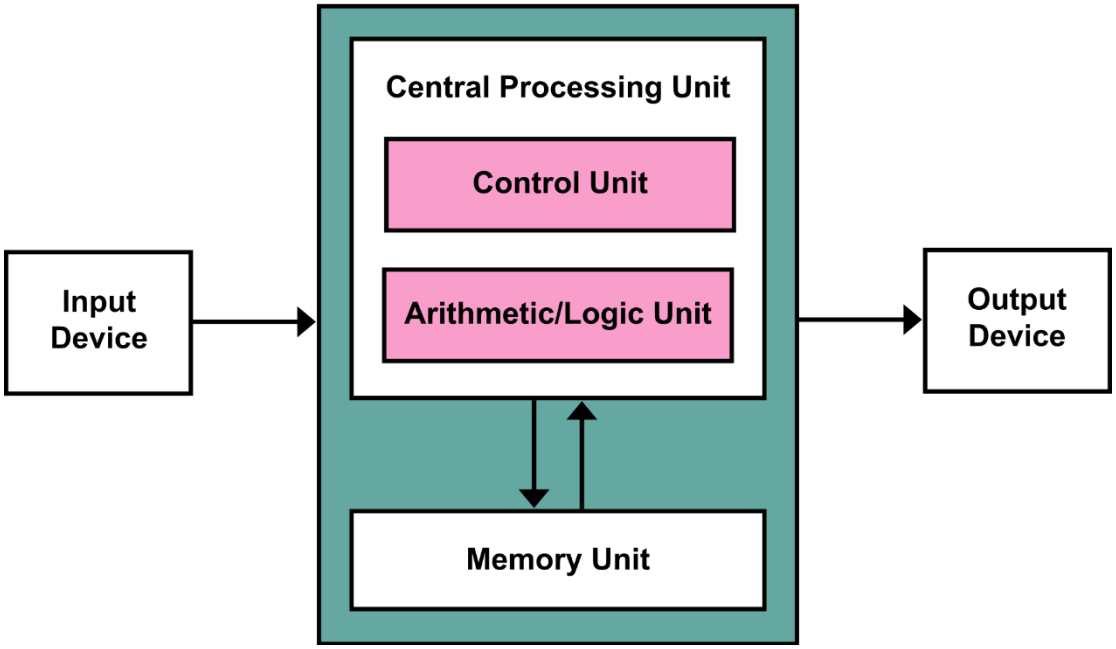
解这些代码是如何变成控制台的黑框框并输出结果的。在上文中，我们编写的叫做“C++语言源代码”，源代码是用编程语言编写而成的一些文本，这些文本都是人类可以理解的内容，而编程语言本质上不过规定一些语法和词法，通过这些法则用自然语言表达出程序执行逻辑，然后由编程语言实现的各种工具对这些逻辑进行处理，翻译成计算机能够理解的所谓机器码（Machine Code），这个过程叫做编译（Compile）。机器码本质上就是一些数字，只不过是 CPU 能够理解的数字。下图详细说明了各种编程语言的编译过程：

对于编译型语言（如 C++/Java），编译器从源文件（.cc/.cpp/.cxx 等）读取源代码，把所有预编译代码进行扩展、内联（例如宏、内联函数、包含文件）后对源代码进行词法、句法、文法分析后构建所谓“语法树”，通过分析语法树将源代码转换成等价的汇编代码或中间代码（IL）例如 VS2022 会调用 MSVC 编译器（cl.exe）将源代码翻译成汇编代码（.asm），而 Java 编译器会将源文件（.java）翻译成可供 JVM 读取的字节码（bytecode, .class）。对于 C++来说，编译只能确定需要执行的指令，但不能确定外部的函数，因为通常源代码文件不止一个，因此需要把每个文件编译成的汇编文件合并到一个二进制文件中，这个过程就叫做链接。需要注意的是，链接过程不只执行这一个动作，出于简化课程的考虑，我们这里只考虑它的合并工作。一旦链接器将整个程序完成链接后即生成可执行文件（.exe），通过 Shell 调用系统加载器即可将可执行文件加载进内存并执行；而对于 Java 来说，编译器不会进行链接并输出成可执行文件，这些工作实际上是在运行时由 JIT（Just In Time Compiler，即时编译器）针对不同系统不同架构完成的，但二者殊途同归，最终都是转换成机器码形式进行执行。

而对于解释型语言（如 Python/JavaScript），这些语言并不存在编译器，相反地，它用一个解释器通过动态分析源代码翻译出程序的行为并进行执行。这样的语言具有优秀的跨平台特性和动态性，但由于文本结构不如二进制结构紧凑，再加上无法通读源代码全文后进行优化（编译型语言可以优化，并且优化效果非常明显），其效率和性能堪忧（一个例外是 Google 实现的 JavaScript 引擎 V8，但 V8 事实上将 JavaScript 编译成机器码进行执行的）。

1.3 计算机结构

现代家用计算机或服务器都是基于冯·伊诺曼体系改良而来。



冯·伊诺曼结构示意图

所谓冯·伊诺曼体系，其实就是一个由 CPU 和内存组成的基本结构，在这种结构下，使用者输入数据，经过 CPU 进行运算，然后将一些中间数据暂存至内存，在运算完成后将结果输出给使用者。所谓 CPU 由一个能够进行数学/逻辑运算的单元和一个控制单元（控制指令的顺序等）组成，你可以暂时把它当成一个计算器；而内存则可以看成一个暂存数据的记事本，通过将数据写到记事本上就可以完成存储，而把数据从记事本上读出来就是读取。举例来说，要计算 1 + (3 + 5)这个式子，计算机应该通过某种输入装置先把 1、3 和 5 读取进内存，然后把数据发送给 CPU 进行计算，计算完成后将数据输出到内存（在这个示例里数据就是 8），然后再把 1 和 8 发送给 CPU 进行计算，计算完成后再将数据输入到内存（现在是 9），最后通过某种输出装置呈现给使用者。

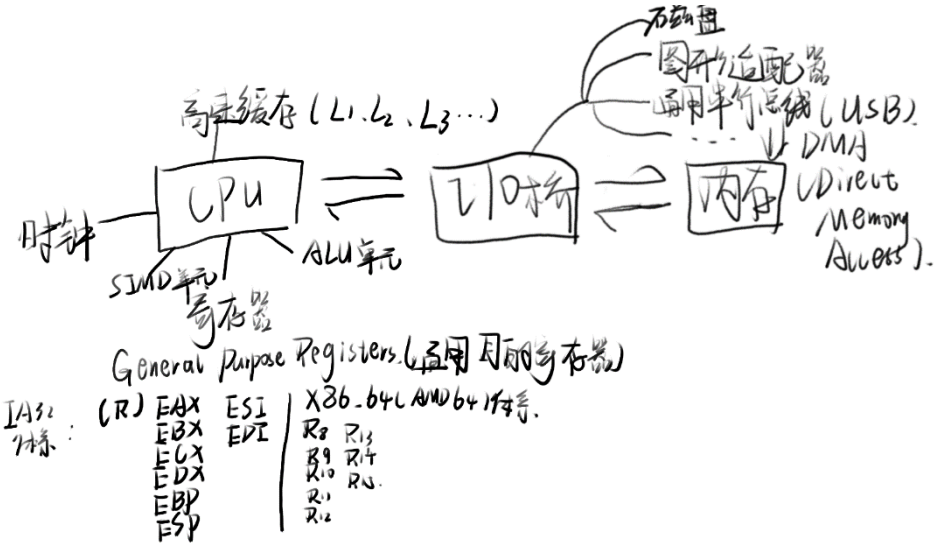
这个过程似乎很完美，但我在上文已经说过，现代计算机由冯·伊诺曼结构改良而来，之所以改良当然是因为它不够优秀。我现在可以向你介绍，在现代计算机的体系中，内存这个组件一般由 DRAM 单元实现，所谓 DRAM 单元可以把他简单理解为一堆电容，通过一个窗口对电容进行充电和放电就可以完成数据的修改，高电位代表 1，低电位代表 0。在这种情况下，如果在计算完成后关机（断电），那么所有的电容此时的电位都是 0，这就造成了数据的丢失。人们可不希望在计算机完成计算后拿一张纸去记录下结果。因此，光有内存进行存储是不够的，还需要引入一个能够持久存储（即掉电后仍然能保持数据）的装置，这个装置迭代了很多代，现在流行的装置有两种，分别是机械硬盘（HDD）和固态硬盘（SSD）。

所谓机械硬盘的原理其实就是通过一根及其尖锐的磁针扫过数片碟片改变碟片中每个存储单元的磁性来完成数据读取和存储的，而有一个电机以极快的速度让碟片做圆周运动以达到让磁针访问碟片的效果（这个速度通常可以达到每分钟几万转）。机械硬盘很慢，笔者的电脑上装有一块机械硬盘，它的连续写入速度通常可以达到 150MB/s（关于数据容量单位定义和转换可以见附录）。想象这个装置读取数据的情景，如果我们需要读取 8 个 Bit 的

数据，那么理论上需要等待磁盘转 8 圈才能找到合适的位置，这个操作叫做寻址；以 7200RPM(Round Per Minute，转/分钟)的机械硬盘为例，每一转约需要 8.3 毫秒，那么 8 圈就是 66 毫秒。这个数字看起来很小，但 8 个 Bit 在计算机中可以算是小的不能再小了，而 66 毫秒对于 CPU 以纳秒为单位的运算速度来说可以算是一个很长的时间了。为了优化连续读取一段数据的性能，我们将磁盘分成了一个个的单位，每一个单位都能存储一定量的数据（通常是 4KB 或 8KB），这样在读取数据的时候就可以一次性读取很多数据，减少寻址次数。固态硬盘是机械硬盘的继任者，所谓“固态”，就是这种装置的内部其实就是一个一个半导体晶体管，用类似于内存的方式进行读写。这样的读写方式速度极快，笔者的电脑里也有一块基于 NVME3.0 协议的固态硬盘，这块硬盘的连续写入速度可以达到 1.8GB/s，并且它的寻址速度也远快于机械硬盘，但代价是价格较机械硬盘稍贵（如果是两年前，那么我会去掉这个“稍”字），且寿命相较机械硬盘较短。

那么在引入一个能够持续存储的装置之后，似乎这个体系也完美了，但是我们还有一个关键的问题，这个持续存储装置应该与哪个部件连接？是内存还是 CPU？答案是，都不是。这个持续存储装置连接到 IO 总线(I/O Bus)，IO 总线的一端连接着 CPU，另一端则连接着内存和持续存储装置。当我们需要访问装置的一些数据时，由 CPU 向 IO 总线发送一些指令，然后 IO 总线把这些指令翻译成装置能理解的指令，紧接着装置把数据读出后发送到 IO 总线，这个过程叫做 DMA（Direct Memory Access，直接内存访问），而在这个过程中 CPU 并不会傻傻的等待数据被读取完成；事实上，对于很多简单的运算来说，等待的时间足够 CPU 把这些数据运算几万甚至几十万遍，所以 CPU 会继续它的工作。总线把数据转发到内存并向 CPU 发送一个叫做“中断”的信号，CPU 在接收到这个信号后从内存中读取数据进行运算。

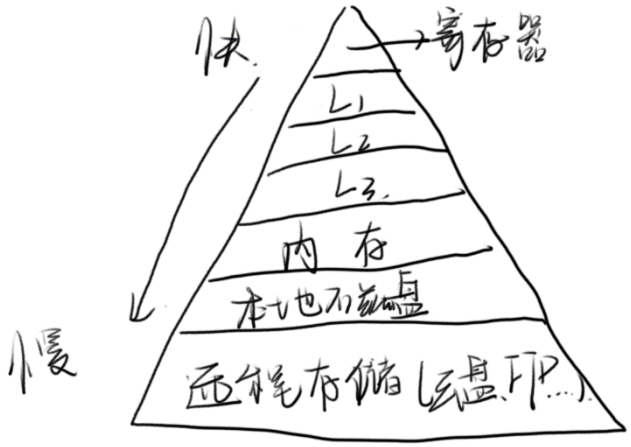
一个计算机可能还需要具备图形运算的能力、连接网络的能力等，这些组件都是由各种装置实现，这些装置也可以被接入 IO 总线，工作原理与磁盘类似，故不再多做赘述。



整个计算机的体系结构

看到这里，一个计算机该有的功能应该都有了，似乎完事大吉了，但别急，还有一个严重的问题，那就是内存到 CPU 的过程太耗时了。CPU 的速率以它内部的一个振荡时钟为基础，这个时钟每振荡一次叫做一个时钟周期，每个周期过去，CPU 就会更新一次晶体管的状态。CPU 完成一次简单的数学运算（加减乘除等）需要的周期在 1-20 不等，而通常从内存中读取一次数据要经过几百甚至几千个周期，如果一个数据需要反复使用，那么将会浪费

大量的周期在读取数据上。现代 CPU 拥有寄存器和高速缓存两个部件，高速缓存是分级的，



通常分为 L1 和 L2，部分 CPU 还有 L3 缓存，其中 CPU 访问寄存器通常需要的时间少于一个周期，而访问 L1 缓存需要 2-4 个周期，访问 L2 缓存需要大于 10 个周期，而访问 L3 则至少需要 20 个周期。如果你玩过游戏《Minecraft》的话，你可以把寄存器当作一个工作台，而缓存则是一个背包，CPU 中的计算单元（ALU Unit）只与寄存器打交道，任何数据需要先逐级从内存加载到寄存器（内存->L3->L2->L1->寄存器），这些存储装置的容量逐级递减，而速度逐级递增。

计算机体系的存储金字塔

这就是现代计算机的体系结构，但很多内容都是经过大量精简过的，不过学生现在不需要对这些概念很明晰，只需要在以后的学习中能够通过这个模型去分析程序的行为就算是掌握本节课所学内容了。

Lesson 1 Senior：剖析程序的行为

1.1 程序的低级表示

对于计算机而言，只有 CPU 具有执行代码的能力。CPU 是一个电子元件，因此它只能读取电信号然后经过一些处理输出电信号。一个程序的运行需要计算机的各个部件配合，将存储在硬盘上的机器码读入到内存，进而读入高速缓存。CPU 内部的一些电路将机器码所表示的电信号逐个输入并转换成 RISC 指令，发射到各个执行单元进行执行；在这个过程中，CPU 不会等待一条指令执行完后再执行下一条指令。相反地，在 CPU 内部存在“流水线”（pipeline），一条流水线上有很多条 RISC 指令，这样的流水线设计保证了 CPU 永远不会处于空闲状态，大幅提高了 CPU 利用率，但对于条件分支这样的指令（例如 jcc）及其不友好。想象一下，如果等待一个条件被判读完毕再执行下一条指令会浪费大量的 CPU 周期；现代 CPU 具有分支预测的能力，通过提前执行分支后的一些指令，如果条件成立则直接往后执行，在这个过程中没有任何的额外开销。而若条件不成立则将流水线状态回退到流水线初始状态，这种行为称为“分支预测

错误惩罚”。现代 CPU 可以将分支预测成功率做到 90%以上，因此现代 CPU 具有非常高的运行效率。机器码作为机器行为（几乎）最底层的表示方法，这些二进制数字通常是人类理解很困难的（为了优化机器码的存储效率，CPU 厂商使用了各种各样的奇技淫巧把机器码大幅缩短，代价是人类几乎不可能正常阅读）；最开始，人们发明了“汇编语言”来把机器码转化成人类能够读懂的一些简短的指令，这些指令的执行非常贴近机器码，相当于是机器码的纯文本表示法，因此能够控制程序的每一个运行细节。再往上就是现代的编程语言，这些编程语言越来越贴近人类语法，而暴露给人们的细节越来越少。考虑到直接编写汇编语言太过麻烦，所以程序员几乎都使用现代的高级编程语言进行编程，然而一个优秀的程序员必须理解汇编，这样程序员能够知晓编程语言隐藏的细节，从而精细把控程序的运行效率、发现并解决隐蔽的 Bug、对程序进行手工的优化。

考虑到本课程是 C++课程而并非汇编课程，因此我只提供程序的反汇编语句并对语句的具体效果进行分析。如果想要知道每一个汇编指令代表的具体含义，请自行学习 x86 汇编语言。

下面是章首的源代码的反汇编语句。这种反汇编语句的生成非常简单，只需要打开 VS2022，进入工程后按下 F10，在编辑器区域的任何地方按下右键，选择“反汇编”即可。

```
#include <iostream>

int main(int argc, char** argv) {
00007FF6253C18D0  mov        qword ptr [rsp+10h],rdx
00007FF6253C18D5  mov        dword ptr [rsp+8],ecx
00007FF6253C18D9  push       rbp
00007FF6253C18DA  push       rdi
00007FF6253C18DB  sub        rsp,108h
00007FF6253C18E2  lea        rbp,[rsp+20h]
00007FF6253C18E7  lea        rcx,[__D08BF3F2_src@cpp
(07FF6253D2066h)]
00007FF6253C18EE  call       __CheckForDebuggerJustMyCode
(07FF6253C137Ah)
  int value = 1 + 62 * 4;
00007FF6253C18F3  mov        dword ptr [value],0F9h
  std::cout << value << std::endl;
00007FF6253C18FA  mov        edx,dword ptr [value]
00007FF6253C18FD  mov        rcx,qword ptr [__imp_std::cout
(07FF6253D0170h)]
00007FF6253C1904  call       qword ptr
[__imp_std::basic_ostream<char,std::char_traits<char> >::operator
<< (07FF6253D0158h)]
00007FF6253C190A  lea
rdx,[std::endl<char,std::char_traits<char> > (07FF6253C1037h)]
00007FF6253C1911  mov        rcx,rax
00007FF6253C1914  call       qword ptr
[__imp_std::basic_ostream<char,std::char_traits<char> >::operator
<< (07FF6253D0150h)]
```

```
  return 0;
00007FF6253C191A  xor       eax,eax
}
00007FF6253C191C  lea       rsp,[rbp+0E8h]
00007FF6253C1923  pop       rdi
00007FF6253C1924  pop       rbp
00007FF6253C1925  ret
```

在第一行和第二行，这些代码代表把当前栈空间的前十二个字节分别加载到两个寄存器rdx和ecx中，rdx中存储的是参数argv这个指针，而ecx中存储的则是argc这个int值。接着，通过把rbp寄存器的值压到当前栈首来完成函数调用的初始化工作，接着把rdi寄存器的值压入栈上进行保存（因为rdi寄存器是非易失型寄存器）。下一步将rsp的值减去0x108这个值，考虑到栈的增长方向是由上至下的，那么这一行的意思实质是为当前执行的函数分配栈内存空间。紧跟着的两行是msvc运行库提供的栈损坏检测，不用管它。接下来的一行汇编是把一个常量0x0F9h（十进制249，也就是1+62*4表达式的结果）压入内存空间。之所以没有体现成一个加法指令和一个乘法指令，是因为C++在编译过程中对于常量参与的表达式进行优化，直接算出结果写进程序中。接下来的六行实际上就是调用C++标准库中的cout函数将该值输出到控制台上。下一行指令代表着将eax寄存器设置为0（两个相同的数进行异或得到的值是什么？），然后恢复寄存器、退栈并返回。这就是文首的代码的运行过程。