程序设计语言原理

Principle of Programming Languages

裘宗燕

北京大学数学学院

2012.2～2012.6

1. 引　言

- 为什么学习“程序设计语言原理”课程？
- 程序设计语言的发展
- 计算模型和语言分类
- 语言：规范和实现
- 程序设计语言的实现
- 程序设计语言的设计目标及其演化

为什么学习程序设计语言原理？

程序设计语言是大多数计算机工作者使用的“计算机”（描述和完成计算的工具）。要很好完成与计算机有关的工作，必须对语言有深入理解

深入理解“程序设计语言”，有助于：

- 正确使用语言，更好完成程序设计和软件开发工作。特别是：
  - 理解语言中比较复杂，容易误解的特征，正确使用
  - 理解如何在语言的不同表达方式中做出正确选择
  - 正确使用各种与编程相关的工具，如 **Debugger** 等
  - 在缺乏某些特征的语言里正确有效地模拟这些特征
- 正确讨论和解释与程序设计语言有关的问题（工作中经常需要）
- 学习和理解遇到的新语言和新语言概念
- 根据实际的需要正确地选择所用的语言
- 在需要时设计和实现语言（人们需要在不同程度上做这种事情）

为什么学习？ （问题实例）

1，我学的是 **VC** 程序设计

2，我学的是 **6.0**（**VC 2008, VC 2010**）版的 **C++**

3，谁知道下面 **C** 语句给 **n** 赋什么值？

```
m = 1; n = m+++m++;
```

4，为什么我的程序调试时完全正确，**Release** 生成的程序却出错？

可以看到很多类似的错误说法和问题，实践中也常遇到许多不理解的问题

出现的很多问题的根源是因为对程序设计语言缺乏正确完整的理解

读过有关“程序设计语言”的书籍，或上过这方面课程，提高了相关认识，就可能把计算机领域的工作做得更好。

程序设计语言原理 计算机原理

程序设计语言的发展

程序是计算机科学技术领域最基本最核心的概念，有计算机就必须有程序，有描述程序的手段和方式，即，必须有描述程序的程序设计语言

硬件计算机，就是机器语言程序的执行器（解释器）

计算机硬件只能处理二进制形式的程序（机器语言程序）。如：

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000002 00832023
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020
03e00008 00001025
```

MIPS机器语言：用欧几里得算法求GCD的程序

机器语言，人很难阅读、理解、使用

使用机器开发程序的成本高昂，耗时巨大，容易出错，难以检查，依赖于具体计算机，难以移植到其他计算机，……

发展

为解决程序的易写和可读性，人们发展了符号形式的汇编语言。例：

```
    addiu   sp,sp,-32
    sw      ra,20(sp)
    jal     getint
    nop
    jal     getint
    sw      v0,28(sp)
    lw      a0,28(sp)
    move    v1,v0
    beq     a0,v0,D
    slt     at,v1,a0
A:  beq     at,zero,B
    nop

        b       C
        subu    a0,a0,v1
B:      subu    v1,v1,a0
C:      bne     a0,v1,A
        slt     at,v1,a0
D:      jal     putint
        nop
        lw      ra,20(sp)
        addiu   sp,sp,32
        jr      ra
        move    v0,zero
```

MIPS汇编语言：GCD程序

- 每条指令都很容易理解，容易书写和阅读
- 仍用与机器语言直接对应的简单线性形式。没有高级结构，不支持程序的高级组织，大型程序难以理解和开发，依赖具体计算机，…

发展

1954 年 IBM 公司 John Backus 领导的小组开发并实现了第一个后来被广泛使用的高级程序设计语言 Fortran，标志着程序设计的新时代开始了

高级语言的主要特点：

- 抽象的描述形式（形式上类似于自然语言和数学公式）
- 易写性和可读性（利于开发者的使用）
- 可实现性（写出的程序可由计算机执行，通常是间接的通过转换或软件）
- 机器无关性（有利于移植）

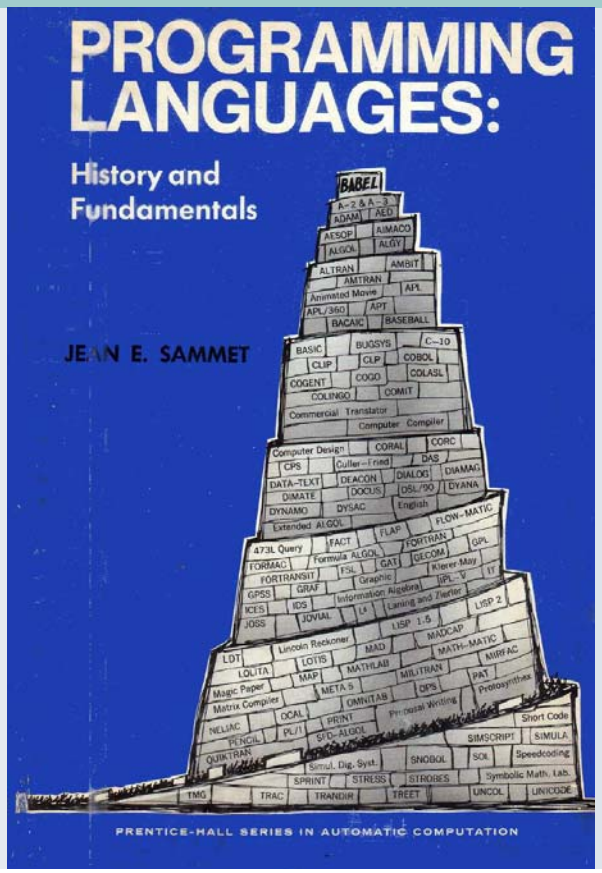
术语程序设计语言

广义：指所有能用于为计算机写程序的语言

狭义：专指各种高级语言

此后是程序设计语言的大发展。（下页的图）

Jean E. Sammet的书和程序语言巴比塔(1969)

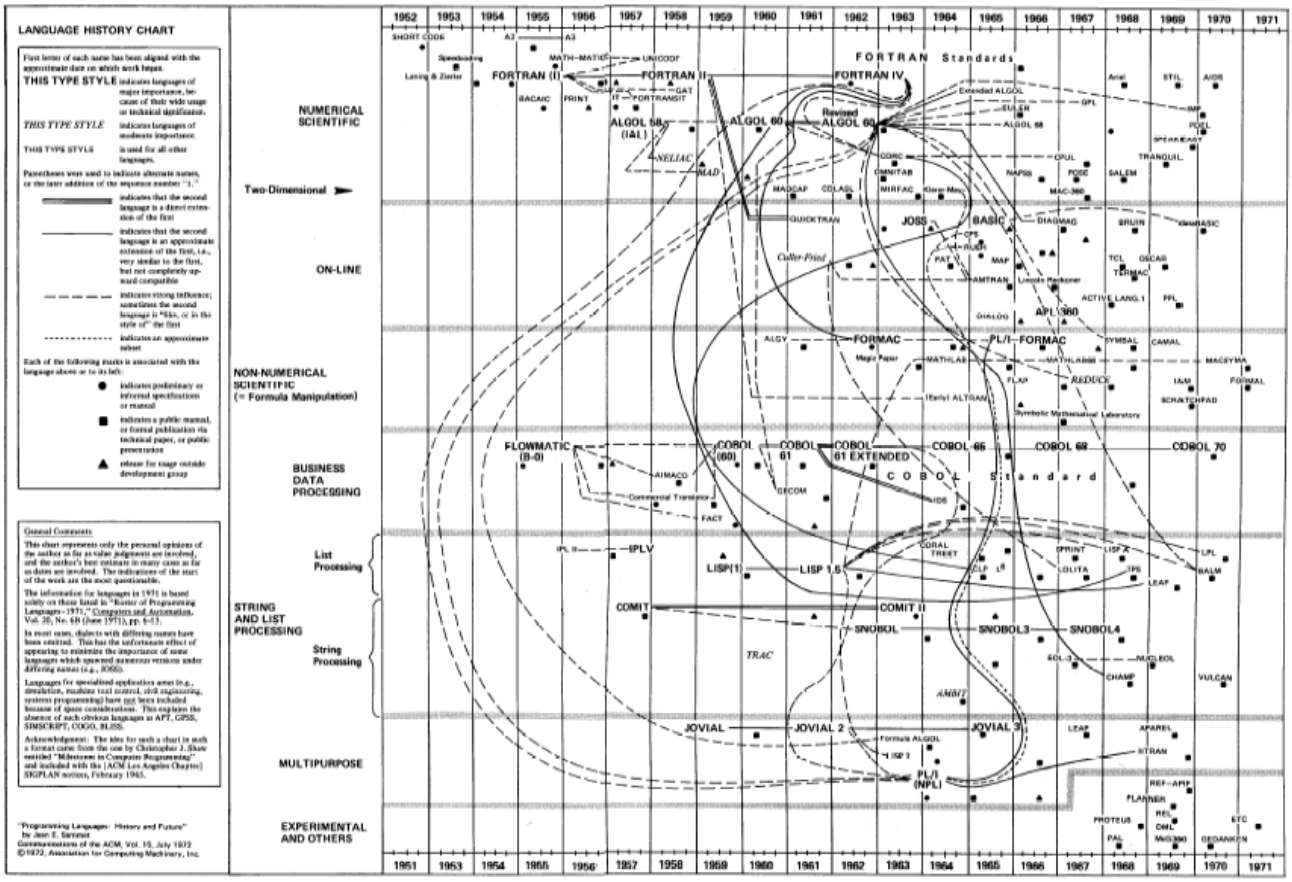


Tower of Babel, by Pieter Bruegel

老彼得 布鲁格尔

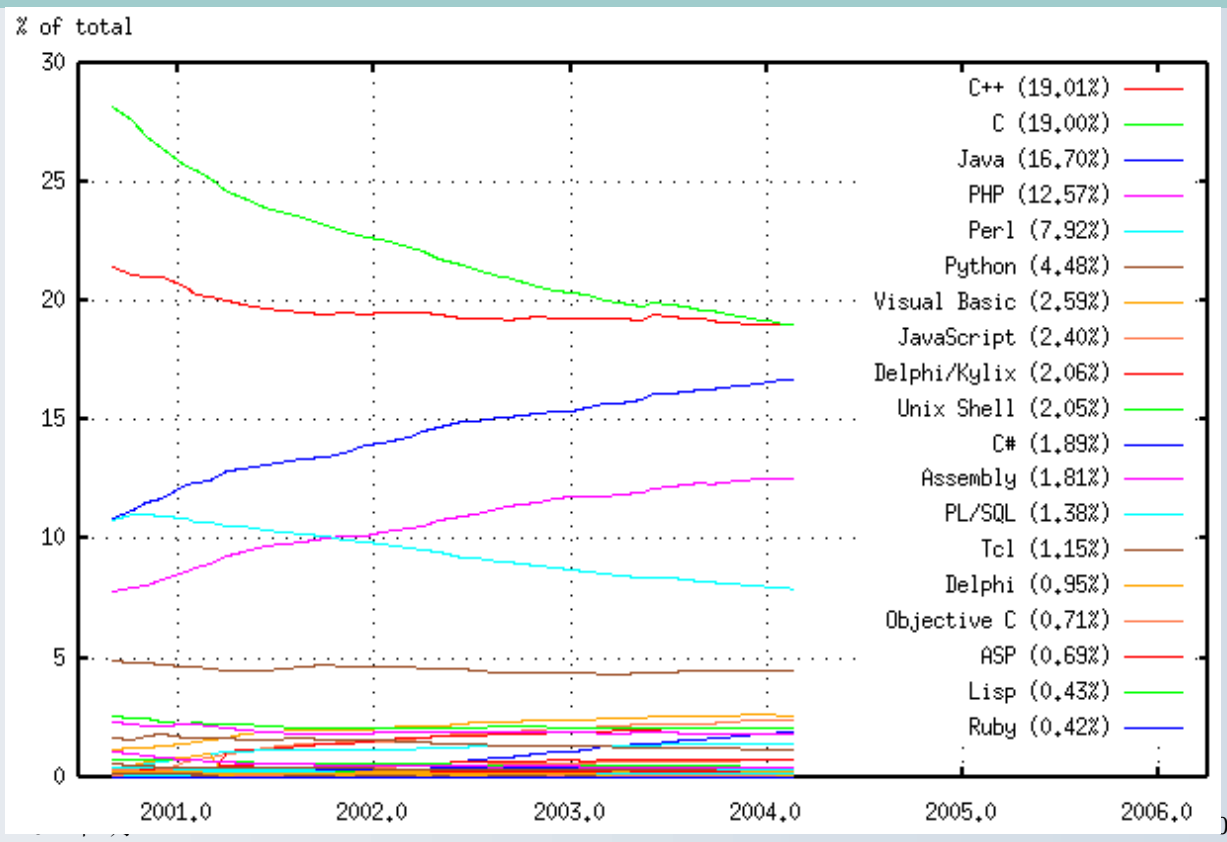
然后上帝说：看，他们是同一群人，都使用同一种语言，而这还只是事情的开始。

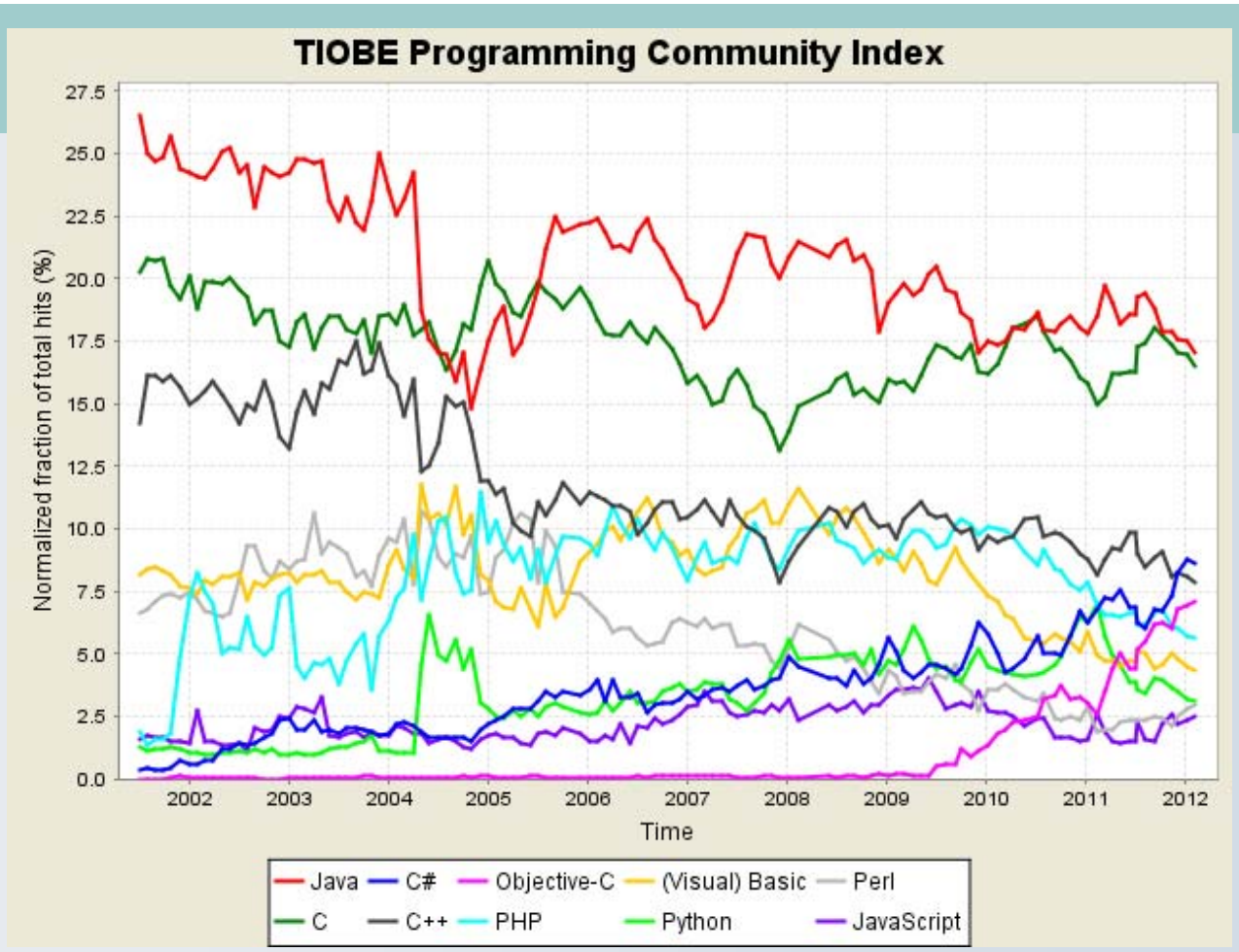
——《创世记》**11:6**



常用语言

Compiled by François Labelle from statistics on open-source projects at SourceForge





为什么有这么多语言

为什么没能出现一种适应所有计算机应用的唯一语言？

- 计算机的应用领域丰富多彩
  - 不同领域、不同具体应用对于合适的语言有不同要求
  - 一种语言无法最好地满足所有应用的需要
- 语言设计的许多方面有很多选择
  - 不同的设计选择通常是各有各的优点和缺点
  - 常常是“鱼和熊掌不可兼得”
- 新应用领域和应用需求不断出现
  - 不同领域（任务）有特殊的功能或描述方式的要求
  - 为满足新需要，人们常常设计出新的语言
  - 如网络、实时应用、图形用户界面，业务处理等

为什么有这么多语言

- 硬件发展，例如速度、容量、并行机、RISC等
  - 有些过去认为“奢侈”的特征，由于有助于软件系统的开发者，慢慢变成必备的特征（如“废料收集”）
  - 例：近来多核多处理器系统正在快速发展，要求新的编程方式和技术，将推动语言里支持并行程序设计的特征的大发展
- 程序技术发展，如
  - 结构化程序设计（结构化程序设计语言）
  - 面向对象的程序设计（面向对象的语言）
  - 脚本技术，等
- 程序理论的研究和实际应用（提出对语言设计的指导性意见）
- 标准化的需要
- 向后兼容性，历史遗产和人的习惯。等等

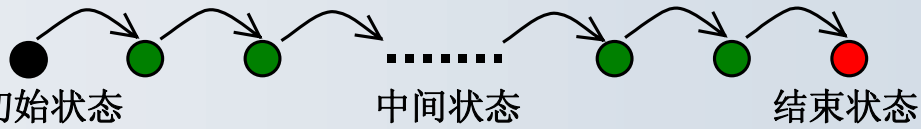
计算模型和语言范型

形成许多程序设计语言的另一重要原因：对计算过程可以有多种不同看法，由此产生了不同的计算模型（范型，**Paradigms**）

基于不同计算范型产生了不同的语言类（语言范型）。主要范型有：

过程式范型（**Procedural**，或称命令式，**Imperative**）：

把计算看成一系列操作的执行，基本计算元素是一组基本操作。计算在一个环境里进行，操作的效果就是改变环境的状态：



语言提供一组基本操作和一组描述组合操作的手段，提供的抽象手段是定义新操作（定义过程）。过程实现大步的状态变换

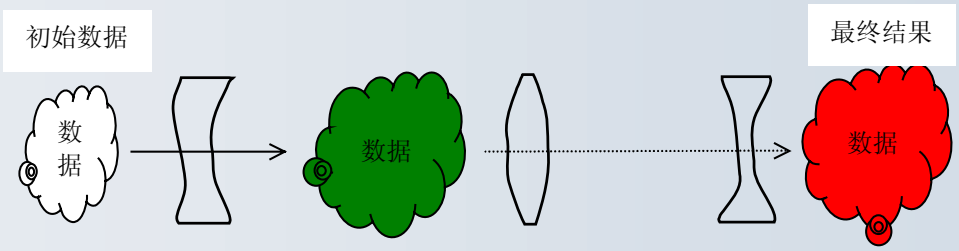
写程序就是描述操作执行的顺序过程，描述状态和状态的变化

常规语言，如 **C**，**Pascal**，**Fortran** 等，都是命令式语言

语言范型

函数式范型（Functional）：

- 把计算看成对数据的函数变换，一个计算就是一系列函数变换
- 基本计算元素是一组基本函数，语言提供各种函数组合机制（复合、函数选择），抽象手段是定义新函数（允许递归定义）



最早的函数式语言是 John McCarthy 开发的 Lisp。目前最重要的函数式语言包括 Common Lisp，Scheme，ML 和 Haskell 等

网上有很多有关函数式语言和函数式程序设计的文献。课程网页上也有一些这方面的材料

语言范型

面向对象的范型（Object-Oriented）

- 把计算看成是一批独立对象相互作用的效果。OO语言提供
  - 描述（定义）对象及其行为的机制
  - 以及描述对象之间相互作用的机制
- 比较纯的 OO 语言
  - Smalltalk，Java 等基于类的语言，程序中定义类，对象是类的实例
  - JavaScript，是基于对象的语言，通过原型概念定义新的类似对象
- 其他支持 OO 概念和编程方法的语言包括 C++，Ada95 等
- 这些都是基于命令式行为描述的 OO 语言

  也有采用函数式的或者其他范型做行为描述的 OO 语言
- 还有一些支持数据抽象，但不支持继承和动态约束的语言

语言范型

- 说明式语言（**Declarative**），包括
  - 逻辑式语言（**Logical**）
  - 关系式语言（**Relational**）
  - 基于限制的语言（**Constraint**）
- 基本想法是只描述需要做什么，不描述怎样做
  - 例如：可能是只描述被计算对象之间的逻辑关系；或者描述问题的解应该满足的条件约束
  - 语言的实现通过很复杂机制提供一种通用的计算过程，它能根据具体“程序”的说明性描述产生出所需的结果
- 例：逻辑式语言 **Prolog**；数据库查询语言 **Datalog**

  近年很受重视的 **Constraint Language** 和 **Constraint Programming**

语言范型

脚本语言（**Scripting Language**）

- 从操作系统命令语言和文本处理语言发展起来的一类语言。通常：
- 提供一批高级数据结构，提供灵活的变量、函数、对象等机制
- 采用解释方式实现，使用灵活方便
- 许多脚本语言提供了高级的文本处理功能

其他语言：

文本描述语言 **Postscript**，也是一种完整的程序设计语言

仪器控制语言 **Forth**，是一种完整的基于栈的程序设计语言

硬件描述语言 **Verilog** 和 **VHDL** … …

本课程主要以命令式语言作为讨论的对象，适当涉及其他语言

语言：标准和实现

要弄清用某个语言写的一个程序的意义，我们应该怎么做？

问题：什么定义了一个程序设计语言？

- 将程序输入计算机，编译后试运行，看结果
- 找懂得这个语言的人询问
- 参考语言的标准文本（或称语言规范），它定义了
  - 语言的语法形式（合法程序的形式）
  - 形式合法的程序结构的语义
  - 明确说明“规范”对某些东西未予定义
- 并适当参考广泛使用的实现，作为帮助理解的辅助工具

同一语言可能有若干标准文本。如：**Fortran 66，Fortran 77，Fortran 90**

标准和实现

一个语言的规范文本常常没有把该语言的一切问题都定义清楚，而是留下一些不加定义的事项。为什么出现这种情况？

例：C语言未规定函数参数的求值顺序，二元运算符运算对象的求值顺序。

除可能是语言设计者的失误外，许多不完全是有意的，目的是：

- 使语言处理程序（例如编译器）的开发者有更大的工作空间，使之能选择最适合实际运行环境的实现方式
- 为使定义的程序设计语言具有更好的普遍适应性，能适应不同计算机体系结构，容易在不同计算机上实现
- 为目标程序的优化提供更多可能性
- 规范的不完全性，有可能导致同一个程序在不同实现上（如gcc/VC/…）的行为不同。影响用该语言写出的程序的可移植性（提示：应尽可能写不依赖于具体实现特征的程序，或将可能有依赖性的代码局部化）
- 增加了学习和掌握语言的负担

标准和实现

例外：Java 语言规范几乎把所有的东西都规定清楚了

例如各种数据的二进制表示方式

表达式求值顺序的各个方面，等等

这样做的优点和缺点：

- 提高了程序的可移植性（一次书写处处使用）
- 提高了程序的安全性
- 限制了新实现方式和新实现技术的应用
- 限制了目标代码优化的可能性
- 一些机器可能不适合Java语言的具体需求，因此难以高效实现Java
- 影响用 Java 写出的程序的效率
- 等等

语言和实现

语言的实现

根据某语言的标准文本开发的一个语言处理系统称为该语言的一个实现

早期的一个语言实现只是一个语言处理程序（例如编译器）

今天，语言的实现已经发展为一类规模巨大的软件系统，通常包括：

- 语言的处理器。使用该语言写出的程序可以在计算机上执行
- 程序开发工具。如编辑器、调试器、代码浏览工具、代码检查工具、代码优化工具、模拟运行工具等等
- 标准库和丰富的增强库

C++ 的一个实现，就是根据 C++ 标准做的一个处理 C++ 程序的软件系统

例如：Borland C++ 系统，gcc 系统，Visual C 系统等

同一个实现还可能有许多不同版本（例如 Visual C）

标准和实现

语言的实现可能与标准不同：

- 语言规范中所有未明确定义的东西，实现时都必须确定（可任意选择）
- 可能对语言规范做一些扩充（扩充语法、语义、库等）
- 可能没有实现语言规范的一些特征
- 可能对某些特征的实现是错误的

一个语言可能有一些不同的实现，每个实现可能有多个不同版本

- 不同实现可以都符合语言标准，但相互之间又存在许多差异（不兼容）
- 不同实现可能采用不同的技术，尤其是可能采用不同程度的优化。因此在程序加工速度和程序执行效率方面都可能有很大差异
- 具体语言实现需要考虑：
  - 尽可能地利用硬件和平台软件的功能和特点
  - 为所在平台上的应用系统开发提供方便

标准和实现

注意：无论哪个实现

- 开发厂商会尽可能努力去保证自己的实现符合语言标准
- 但是，同一厂商对同一语言的下一实现版本也可能与目前实现不同

  从自身利益出发，厂商会努力保证所提供的不同实现之间的兼容性

在编程时

- 完全符合语言规范，只采用语言的标准功能，不受语言规范未定义特征影响的程序具有最大可能的兼容性
- 如果必须用与实现相关的功能，也应将这种部分隔离，使程序更容易修改，有利于移植，适应语言的其他实现（包括同一种实现的新版本）

语言的标准化非常重要

- 没有经过标准化的语言，变化的可能性更大（例如一些由公司所拥有的语言，如 **Visual Basic**，**C#**，**Java** 等）