

# 声明和定义

严格说，声明（**declaration**）和定义（**definition**）不一样

声明：

- 说明一个事物的存在和它的一些属性，例如变量与其类型。根据声明可以确定被声明事物的使用方式。对象声明并不实际建立对象
- 例如：C 语言的外部（**extern**）变量声明，函数原型声明

定义：

- 是声明，具有说明同样的基本作用
- 对象定义要求程序运行时创建被定义对象
  - 对于子程序，需要提供子程序的体
  - 对于变量，可能提供变量的初始值，造成的实际动作包括为变量分配存储和可能的初始化

一些语言的标准里严格区分了两种情况，有些没严格区分这两个术语

# 作用域

定义（声明）建立名字与事物之间的约束，这一约束在源程序里（在源程序的正文里）的作用范围称为这一约束的作用域（**scope**），也说是这个定义或声明的作用域

- 声明有确定的作用范围（作用域），只在确定范围里有效
- 在声明的作用域里，被声明名字指称相应的对象

我们（人/语言处理器）根据作用域确定一个名字对应的事物

例：C 局部变量在其定义所在的复合语句里可用（从定义位置开始）

局部变量定义的作用域就是这个复合语句，从定义开始

**作用域单位**：通常规定一些程序结构作为名字/对象约束的作用域

不同语言里的规定可能不同

常见作用域单位：子程序/函数，复合语句，类，等等

# 作用域

早期 **Fortran** 的作用域结构很简单

由于人们认识到信息局部化的重要性，引进了许多作用域单位

**Algol 60** 引进块（**block**）结构作为基本作用域单位

允许任意嵌套的局部作用域

子程序看作是有名字且可被调用的块

现在一般语言都以子程序（过程/函数）作为作用域单位，一些语言允许子程序体内部的嵌套作用域（如 **C** 的复合语句）

其他作用域单位：

- 结构/记录的声明，**OO**语言里的类定义（声明）
- 模块结构（如 **C++** 的名字空间，**Ada** 的包）
- **C** 源文件，也是一种作用域单位（**static** 外部函数和变量的作用域）

学习一种语言时，必须弄清楚语言中有哪些作用域单位

2012年3月

25

## 作用域：作用域单位

不同语言对作用域单位和作用域有一些不同的规定。例如：

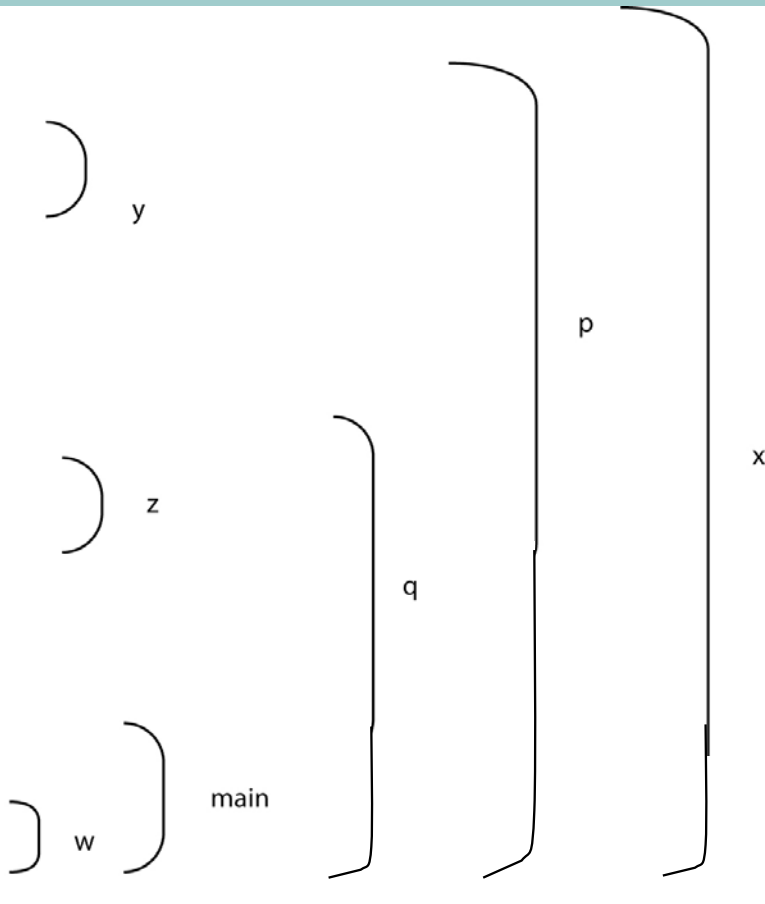
- 一些语言里有全局作用域，供定义全局的标识符与对象（和其他事物）的约束。除全局作用域外的其他作用域都是局部的
- **C** 语言的全局作用域里可以定义/声明变量、函数、类型等。函数只允许定义在全局作用域里（不允许在函数内部嵌套定义函数）
- **C** 语言的局部作用域是复合语句（函数体也是复合语句）。**C++** 对 **C** 的作用域做了许多扩充
- 在 **Java** 语言的全局作用域里只能定义类，其他的实体都只能定义在类内部的各种局部作用域里
- **Fortran** 只有全局作用域和子程序作用域，子程序内部没有嵌套作用域
- **Basic**（老的）只有一个全局作用域，所有变量定义都在一个作用域里。定义在子程序里的变量也具有全局作用域（难以编写大程序）

2012年3月

26

## 作用域

```
int x;  
void p(void)  
{ char y;  
  ...  
} /* p */  
  
void q(void)  
{ double z;  
  ...  
} /* q */  
  
main()  
{ int w[10];  
  ...  
}
```



2012年3月

27

## 作用域：冲突和嵌套

通常规定：

- 在一个作用域里同一个名字不能有多个约束（名字冲突，一个名字具有多个约束的现象称为重载，**overloading**）
- 不同作用域里的名字相互无关

作用域可能嵌套：局部作用域位于全局作用域里，局部作用域里可能还有内部的作用域

同一名字的不同定义可能出现作用域重叠。此时会出现了作用域的“空洞”（图）

规定：内层定义“遮蔽”外层定义

为了易理解，应避免同名对象的作用域嵌套

```
int n;  
  
int p (int n) {  
    double x;  
    ... n ... x ...  
}  
  
int main () {  
    int x; double y;  
    ... n ... x ... y ...  
    if (...) {  
        int n = 3;  
        ... n ... x ... y ...  
        for (... ..) {  
            int n; int y;  
            ... n ... x ... y ...  
        }  
        ... n ... x ... y ...  
    }  
    ... n ... x ... y ...  
}
```

2012年3月

## 作用域：嵌套

在作用域空洞里，（被遮蔽的）外层定义的东西不能用（其定义不可见），直接写出的名字总是表示当时的最内层约束

为了编程方便，一些语言提供了“带修饰”名字形式，通过为名字加前缀修饰的方式指定对外层（定义）约束的访问（有些语言的机制有限制）

例：C++ 的 `::` 是作用域解析运算符，可以用 `::nm` 的形式表示引用全局定义，用 `cn::nm` 的方式引用特定 `namespace` 或者类里的定义

在 Ada 里，每个块（`block`）都可以命名，利用块名作为前缀，可以引用外围块定义的约束（就像引用结构/记录的成员）

一个语言里可能存在多种作用域单位，许多作用域单位允许嵌套。因此，写程序时需要特别注意约束被遮蔽的问题

语言里可命名的不仅是变量，还有函数/过程，类型，常量，结构（记录）的成分，等等。它们都遵循语言的作用域规则，有时有特殊规则

2012年3月

29

## 静态/动态作用域规则

作用域规则（**Scope rule** | **Static, Dynamic**）：对于程序中任一特定位置的一个名字，确定与之约束的对象的方法

局部定义的约束容易确定，关键是非局部定义（非局部）名的约束

静态作用域规则：对非局部名字，其约束根据名字出现位置的静态正文环境确定

动态作用域规则：非局部名的约束根据当时的动态运行环境确定

常规语言都采用静态作用域规则，函数 `f` 中非局部的 `y` 约束于函数的静态环境里的 `y`

有些函数式语言采用动态作用域规则

```
int y = 3;
#define f(x) \
    printf("%d", x+y)
... ..
...{
    int y = 4;
    f(2);
    ... ..
}
```

对子程序中的非局部变量采用静态作用域规则确定约束对象，语义清晰，但实现有些复杂（在有关子程序控制的一章讨论）

2012年3月

30

# 作用域规则和动态约束

在一个语言里，有可能对不同的程序结构采用不同的作用域规则  
完全可能在某些地方采用特殊的规则

例如：**OO** 语言里最重要的方法约束（绑定）问题，也是根据名字确定应该使用的对象的问题（确定应该执行的方法体。动态？静态？）

右边代码方法 **m1** 的约束（方法也是对象）如何确定？是 **B** 类里的方法 **m1**，或也可能是其他类里的方法？（是根据程序正文静态确定？还是根据程序运行中的情况动态确定？）

```
int fun (B & o) {  
    ...  
    o.m1(...);  
    ...  
}
```

**C++** 语言支持两种方式（非虚方法和虚方法）

**Java** 语言统一采用动态规则

**OO** 语言中方法的约束问题在后面讨论

2012年3月

31

## 作用域：实例

不同语言的作用域设计可能不同，每个语言有一套规定

**Pascal** 语言：

- **program** 作用域（类似全局作用域）
- 允许任意嵌套的子程序（函数/过程）作用域
- 只有子程序是局部的作用域单位，子程序里没有嵌套的作用域

**C** 语言：

- 全局的外部作用域
- 子程序定义的局部作用域（不允许嵌套定义的子程序）
- 子程序里任意嵌套的复合语句作用域
- 文件为单位的全局 **static** 作用域（目的是信息局部化）

2012年3月

32

## 作用域：实例

### C++ 语言

- C 语言的各种作用域
- 类定义形成的作用域（类似的还有结构和联合）
- 继承关系形成的作用域嵌套（子类里可以访问父类的成员）
- 方法的作用域嵌套在类作用域里（因此可以直接引用类成员）
- 由 `namespace` 定义的模块（名字空间）作用域

### Java 语言

- 全局作用域里只能定义类（全局作用域里没有对象名）
- 类定义形成作用域
- 方法作用域嵌套在类作用域里
- 由 `package` 形成的模块（包）作用域

2012年3月

33

## 作用域：细节

一个定义/声明的作用域，常见的有两种规定：

- 定义的作用域从定义出现的位置到当前作用域结束
- 定义的作用域包括整个当前作用域

两个例子：

```
int n = 2;
int f (...) {
    int n = n+1, m = n+1;
    ... ..
}
```

C：作用域从声明结束点开始

```
int n = 2;
class C {
    int f() { n++; }
    ... ..
    int n;
}
```

C++：类成员的作用域是整个类定义

语言对于声明的作用域的范围都有明确规定

相关规定有时很繁杂，存在许多细节，需要仔细阅读语言手册

2012年3月

34



# 作用域和可见性

访问控制：C++ 把操作系统的访问控制概念引进了程序设计语言，带来一些新的概念和问题（后面讨论）

作用域进入和退出的特殊描述方式：

- . 运算符。许多语言，由整体进入成分，如对结构变量
- :: 运算符。C++ 等面向对象语言，从类/名字空间到成员
- Pascal 的 with 语句
- 等等

后面讨论模块概念时还会提到这方面的一些细节

... ..

## 其他实体的命名和作用域

程序里的命名实体还很多。现在简单讨论类型、标号的问题

类型通常只在静态处理过程中有效，采用通用的静态作用域规则

类型定义也可能由于嵌套作用域里的重新定义而被覆盖

标号：标明代码中的一个位置，作为 goto 的目标

有些语言里标号的作用域有特殊规定

- C 语言的标号，作用域是整个函数，允许向前或向后跳到标号。允许跳入跳出嵌套的复合语句、内层控制结构（对于跳入位置引起的语义不清晰情况，语言不予定义。如跳入循环后循环变量的值等）
- Fortran 标号的作用域是整个子程序。存在与 C 语言类似的问题
- Pascal 标号要求先声明，用无符号整数表示，作用域是定义所在的整个子程序（包括嵌套定义的子程序）。不允许跳入嵌套的控制结构，但允许从内层子程序里跳回外层子程序（非局部转跳，导致复杂的控制转移）

# 名字分类

程序里有多种命名事物（命名对象、类型、结构的成分名，标号等等），有些语言对名字做了分类。同一作用域里属于不同类别的名字互不冲突

一般规定（在一个作用域里）同类事物的名字不能冲突。但也有语言允许名字重载（一个名字有多重意义，下面讨论）

例：C 语言规定了三个名字类（C 语言手册里称为“名字空间”）

- 标号名
- `struct/union/enum` 标志
- 变量名、函数名、类型名、`typedef` 名、枚举常量名

此外，每个结构或联合声明也是一个“名字空间”（包含其中的各成分名）

规定程序里的名字分属不同类别，要求语言的实现能区分它们

采用怎样的规定，与语言里各种特征的设计有关

# 名字重载

在同一作用域里，一个名字约束于多个对象的情况称为**重载**

最常见的重载实例是算术运算符。`+` 运算符通常表示多种加法运算

一些语言允许程序员定义的名字重载，主要目的是为了提高程序的可读性，以及支持某些很有重要的程序设计技术

例：C++ 的重载对于继承、定义构造函数完成初始化，支持面向对象编程的基本特征等都是必不可少的

允许重载，就必须能有效进行**重载解析**：对重载名的每个使用，都能确定应该实际使用该名字的哪个定义（哪个与之关联的对象）

方法：根据名字出现的静态或动态环境，确定应使用的对象

- 静态解析：在编译时利用静态上下文信息确定名字的正确约束，主要是利用类型信息，确定应该使用的定义（例：加法运算符的解析）
- 动态解析：利用动态的上下文信息。如 OO 语言的方法动态指派



# 对象和值：值

程序对象通常都有约束值。根据可做的操作不同，值有如下分类：

一级值（**first-class value**）：可以赋给变量、作为参数传入子程序、作为函数返回值的数据或对象。（可能有更多要求，如可以运行中构造）

二级值：可以传入子程序，但不能返回也不能赋值

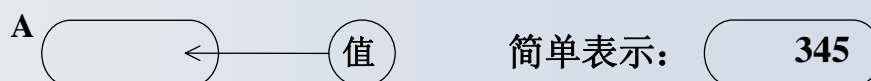
三级值：不能传入子程序（除二级值的限制外）

- 在大部分语言里，简单类型（整数、字符等）的值都是一级值
- C 语言里的指针和结构是一级的值，数组是三级值
  - 注意：在 C 里，数组名放在赋值符右边不是做数组赋值，作为参数也不是传递数组。还应注意：字符串“也是”数组
- 在一些语言里，子程序也是一级值
  - Scheme 里子程序可以赋值、传入传出子程序。可以在运行中动态地构造子程序值（由此发展出许多高级程序设计技术）
  - C# 和 Java 支持 **lambda** 表达式，就是想支持一级的子程序值构造

# 对象和值

现在讨论对象和值的关系。对象的值约束有两种基本方式：

1，值保存在对象的存储区里，这种方式称为**值语义**（值模型）



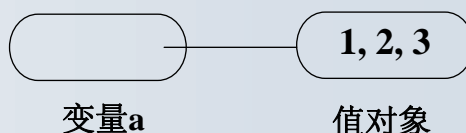
2，对象（例如 A）的值是另一对象（例如 B），对象 A 的存储区里存着对 B 的引用，对象 B 里保存实际数据。称为**引用语义**（引用模型）



被引用的对象常常是匿名对象，有的语言支持对有名字的对象引用

- C 语言里的变量都采用值语义
- Java 里基本类型的变量采用值语义，其他类型的变量都采用引用语义

例：`int[] a = {1,2,3};`



# 变量

- 变量是一种命名程序对象，它的作用是保存值，所保存的值可以在程序执行中改变（与值的约束可动态改变，一般是通过赋值操作）

常变量：一种变量，其值在创建时给定，生存期间不变

许多语言里的变量/常变量都是通过“变量/常量定义”创建

- 在一些语言里，变量定义具有特殊的语法地位（如 **Pascal**, **Ada**, **C**），只能出现在特定的语法位置
- 一些语言把变量定义看作一种语句，可出现在任何可以写语句的地方

例：C++ 把变量定义看作语句。还允许出现在一些特殊位置：

```
while (int n = ...) { ... }
```

```
for (int n = ...; ...; ...) { ... }
```

还可写在 if 和 switch 头部（基本想法：使定义点尽量接近使用点）

# 变量和属性

有些语言里变量不需要定义，遇到新标识符自动定义新变量。这时要解决：

- 所出现的变量的使用范围（相关规定可能有很多细节）
- 变量的属性如何确定（例如类型，可以如何使用）

**Fortran** 语言有 **I-N** 规则，名字以 **I** 到 **N** 开头的是整型变量，其余是实型。采用这种方式的新语言主要是脚本语言、解释性语言，其变量无类型

变量的基本属性：      名字      类型      常性（**const**）等

复合变量对象：

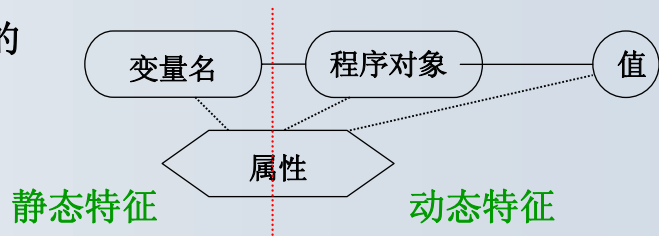
- 数组的维数和各维的上下界
- 结构（记录）的成员名字和类型
- 其他
  - 如 C 语言的存储类 **static**（局部变量），可见性 **static**（外部变量）

# 变量和属性

由于语言不同或者情况不同，变量的属性可能：

- 关联于变量的标识（变量名），只是静态可用（静态属性）
  - 关联于程序对象或者值（对象），是动态属性，在执行中可用
- 在源程序里，变量名（变量标识）代表相应的变量
- 与变量名关联的程序对象是变量的运行时的实体体现
  - 其他属性可能只是静态可用，或者动态也可以用
    - 例如 C 语言的数组大小，只为分配空间用
    - 一些语言的数组大小在运行环境里可用，支持动态越界检查

变量名与相应对象的约束是静态的  
对象与值的约束是动态的，可变



2012年3月

43

# 变量和属性

- “纯编译的”语言（如 C 和 Fortran）设计目标之一就是保证变量的所有属性都可作为静态属性，只在编译中使用和检查
  - 运行时的变量对象没有值以外的属性
  - 运行中不做任何动态检查（没有信息，不可能检查）
  - 以保证程序的紧凑性和执行效率
- C 的目标模块里没有任何类型信息
  - 没有函数的参数和返回值信息，没有全局变量的类型信息
  - 连接时只考虑名字对应，类型安全性只靠源程序文件里的局部信息

许多语言要求为某些程序对象保留一些属性信息：

- 要在运行时做数组访问越界检查，就必须保留维数与上下界信息
- OO语言支持方法动态约束和基于类的指派，需要在对象里保存信息

2012年3月

44

# 初始化和赋值

变量取得值的基本方式

- 初始化：在定义时给被定义的变量提供一个初值
- 赋值：为已有值的变量赋予新值

提供初始化机制的意义：

- 静态分配的变量的初始化如果能静态做，可以避免动态运行开销
- 可帮助避免由于忘记初始化就使用，或者不正确的初始化而造成的程序错误（很常见的一类程序错误）

一些语言规定了默认的初始值

- 如果变量定义时没有给初始值，就自动赋予默认的初始值
- 常用0、空指针等作为变量的默认初值

但提供默认初始值，也就排除了检查，有可能影响程序的安全性

# 初始化和赋值

例：C 语言

- 对静态分配的所有变量用全 0 二进制序列初始化。因此，如果没有另外初始化，数值变量正好取值 0，指针变量取空指针值
- 动态创建的变量（自动变量）没有默认初始化，是为了程序效率
- 动态分配的匿名变量可以选择分配函数，做或不做默认初始化

例：Java 要求在每个变量使用前都必须做“定义性”赋值（初始化）

- 规定：在到达每个变量的每个使用点的每条控制路径上，都必须明确地出现对该变量的赋值
- 这样可以保证不出现使用未初始化的变量的情况，基于设计 Java 语言时对程序安全性的总体考虑
- 这个规定是静态要求，可以在编译时静态检查
- 这是一个充分条件，可能并不必要