

# 第三章 过程式程序设计语言

基本观点:

计算实现的模型如果按冯·诺依曼原理强制改变内存中的值叫命令(或译指令、强制Imperative式)的。由于强制改变值,程序状态的变化没有一定规则,程序大了就很难查错,很难调试,不易证明其正确。

组织程序的范型即:算法过程+数据结构 (计算控制+计算对象)

3.1 计算对象表示—值与类型

3.2 计算对象实现—存储

3.3 计算对象连接—束定

3.4 计算组织---程序控制

3.5 计算组织---函数与过程

3.6 计算组织---抽象与封装

# 3.1 计算对象-值与类型

类型是计算机可能实现的结构和约定对客观世界差异的刻划。

同一类型的外延，即同一结构表示所有可能的值构成一个域。

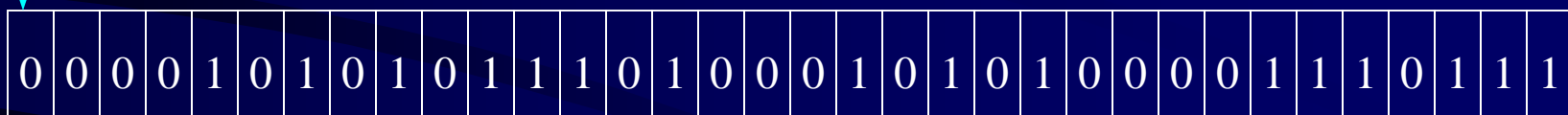
分类原则：同样表示结构, 同样语义解释, 同样的操作。

同类型值运算结果：同类型。

无符号整数：二进制解释的值

整数：符号

值



浮点数：

符号

阶码

尾数

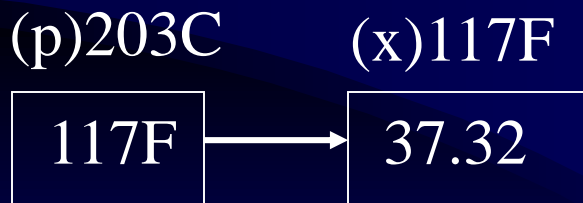
程序语言中：基元(primitive)类型：整型/实（定，浮）/字符/真值/枚举

结构（structured）类型：元组/数组/记录（结构）/表/串

## 3.1.1 字面量、变量、常量

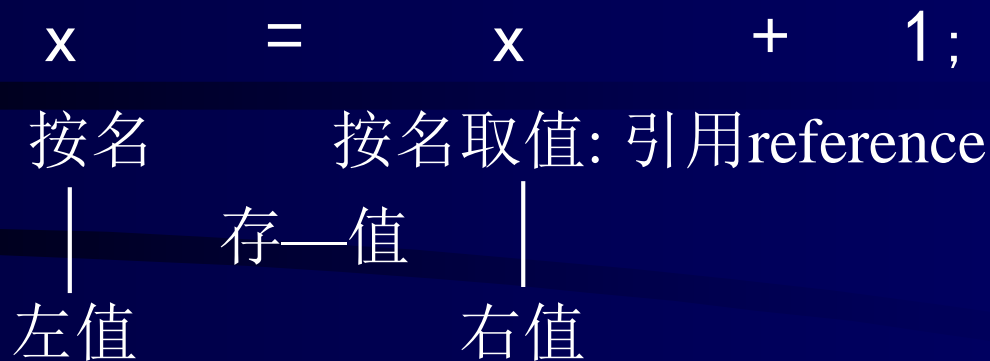
- 名字操纵值, 名: 字面量 从名字即知类型 字面值不变  
变量 符号名要声明类型 值可变  
常量 符号名要声明类型 值不变
- 基元值的名字是地址的别名, 地址值在计算机中不恒定
- 操纵地址的名字是指针(地址变量)

```
float *p, x = 37.32;    p=&x;
```



# 续

- 名值可分导致



y = x ++ ; x 既是右值也是左值

- 别名：多个变量可以具有同一个地址。当用多个变量名来访问单个存储地址时，这些变量名就称为别名。

(违反无二义原则吗?)

int & Rint = Svar //&在此不是求地址运算，而是起标识作用。

Rint 是引用类型的变量，即左值变量必须给一右值，因而成为Svar别名

## 3.1.2 值是头等程序对象

### 程序语言中的值

- 字面量(整、实、布尔、字符、枚举、串)
- 复合量(记录、数组、元组、结构、表、联合、集合、文件)
- 指针值
- 变量引用(左值、右值)
- 函数和过程抽象，数学对象参与运算的权利是一样的，值是计算对象也要按一致性原则：
  - 可出现在表达式中并求值
  - 可作函数返回值
  - 可单独存储
  - 可以构成复杂的数据结构
  - 可作函数参数

- 程序中的求值方式
- 表达式中求值靠运算符
- 函数、过程引用改变值

函数得到函数值，过程改变环境中值（也是 改变状态）。

函数和过程是操作集合的概括，函数是从原始值到函数值的映射，是一种事物的度量到另一种事物度量的联系。

运算符是特殊的函数

单目运算：  $\oplus E$       等价表示为  $\oplus(E)$

双目运算：  $E1 \oplus E2$  等价表示为  $\oplus(E1, E2)$

**有没有多目运算**

# 几种值类型讨论：字符串类型

- 设计问题：
  - 字符串应该是一种特殊的字符数组，还是一种基本类型
  - 字符串应该具有静态长度，还是动态长度
- 操作
  - 赋值、拼接、子串引用、比较、模式匹配
- 长度的设计选项
- 实现

# 用户定义的序数类型

- 枚举类型：
  - 是否允许枚举常量出现在多个类型定义中？  
如果允许，程序中出现该常量时，如何对它进行类型检查
  - 枚举值会自动强制转换成整数吗？
  - 其他类型会自动强制转换成枚举类型吗？



# 数组类型

- 设计问题：
  - 哪些类型对于下标是合法的
  - 要对元素引用中的下标表达式检查其范围吗？
  - 何时绑定下标范围
  - 何时进行数据存储空间的分配
  - 支持不规则数据或多维数据吗？
  - 给数组分配了存储空间后，能对数组初始化吗？
  - 如果支持数组片，是哪种类型的数组片呢？
- 数组类型的实现
  - 一维和多维

## 3.1.3 类型系统

- 类型定义 值的集合和值上操作集合  $(V, Op)$
- 类型系统 一组可直接使用的类型
- 类型规则
- 类型检查机制

# 3.1.3 类型系统

- 静态与动态

	静	动	
变量	有类型	无类型	动态简洁、灵活
参数	有类型	无类型	静态清晰、死板
值	有类型	有类型	

- 弱/强类型

- 无类型      LISP, Smalltalk
- 弱类型      变量有类型。类型兼容性大, 系统不作检查
- 强制类型    隐式类型强制 (转换), 自动截尾, 补零。显式类型强制 PL/1
- 伪强类型    静态均有类型且作检查, 由于不严, 导出等价准则 Pascal
- 强类型      类型有严格定义, 均作检查 Ada

# 续

- 类型等价

按结构等价

type A is array (range 1.. 100) of INTEGER;

type B is array (range 1..100) of INTEGER;

OA1, OA2: A;

OB1, OB2: B;

OC1: array (range 1.. 100) of INTEGER;

OD1, OD2: array (range 1..100) of INTEGER;

OE1: A;

OA1, OA2, OB1, OB2, OC1, OD1, OD2, OE1均  
等价

# 续

## 按名等价

OA1, OA2 是同一类型(都用A声明)

OA1, OB1, OC1是不同类型(类型名为A, B, 无)

OD1, OD2 是同一类型(同时声明, 虽无名)

OA1, OC1 是不同类型(两次声明)

OA1, OE1 是同一类型(虽两次声明, 但同名)

- 类型完整性准则

涉及值的类型中不能随意限定操作,

力求没有第二类的值

## 3.1.4 类型兼容

- 不同类型值混合运算，人为定出计算级别，由低层升格为高层，结果值是高层的
- 隐式转换 弱类型       $I := R;$   
  显式转换 强类型       $I := \text{Integer}(R);$
- 强类型按名判定，不同类型名则不兼容只有子类型不同名可以兼容
- 显式和隐式混合

# 续

```
type BASE is INTEGER;
```

```
subtype SON_TYPE is BASE range 1..1000;    --子类型
```

```
type DIVERSE is new BASE range 1..1000;    --派生类型
```

```
A, B:  BASE;
```

```
C, D:  SON_TYPE;
```

```
E:    DIVERSE;
```

```
...
```

```
A:= B+C,    --合法，结果为B类型赋给A
```

```
A:= C+E;    --不合法
```

```
A:= C + SON_TYPE(E);    --合法，有显式强制
```

```
A:= E ;    --不合法，两个类型
```

```
E:= B+BASE(E);    --不合法
```

## 3.2 计算对象的实现—存储

存储对象是程序对象在计算机中的实现

程序对象不一一对应为存储对象

$x:=0$ ;  $x, 0$ 是两程序对象

只有一个存储对象 $x$ 加指令清零

初值常量也不作为单独存储对象



# 需要存储的元素

- 已解释的用户程序的代码段
- 系统运行时程序
- 用户定义的数据结构和常量
- 子程序返回点
- 引用环境

# 需要存储的元素

- 表达式赋值中的临时变量
- 参数传递时的临时变量
- 输入输出缓冲
- 其他系统数据
- 子程序调用和返回操作
- 数据结构创建和析构操作
- 元素的插入和删除操作
- 其他.....

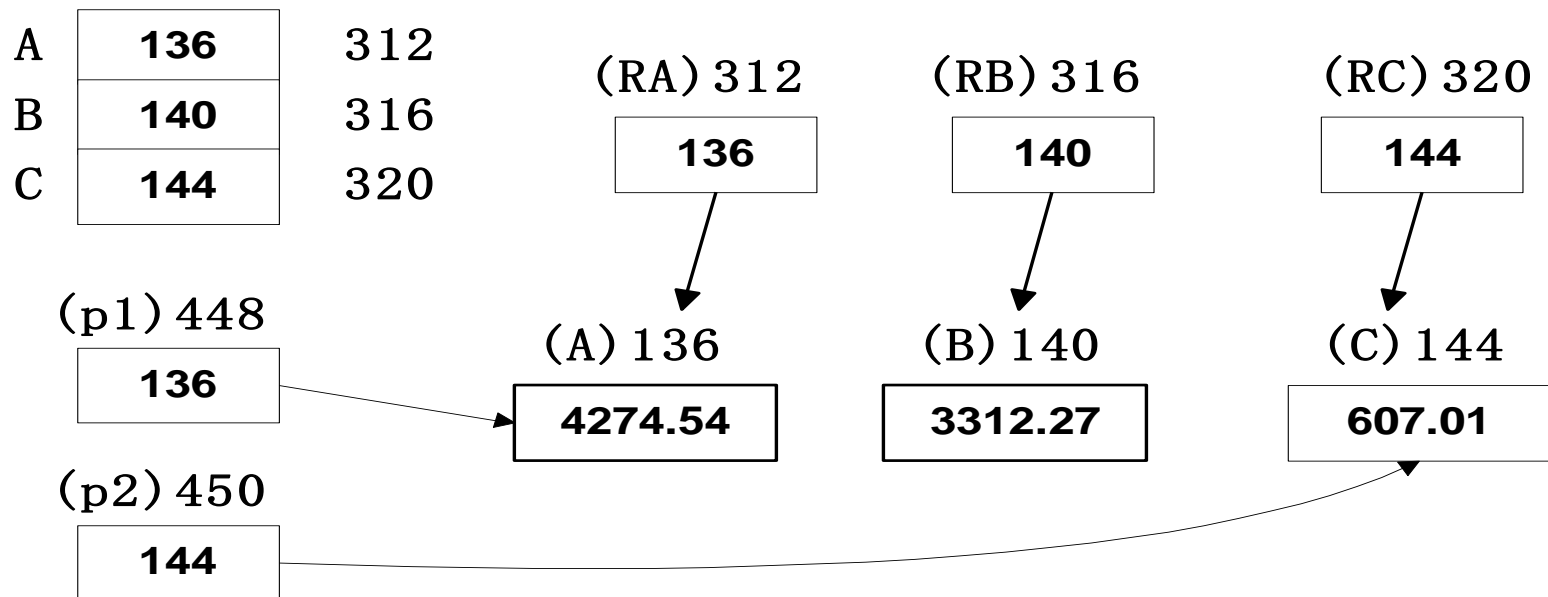
# 指针变量

```
int i;  
int *p, *q;  
:  
*p=i;      //p指向整变量i  
p=i;       //语义错误, p中只放地址值  
p=&i;      //正确, 效果同*p=i, &取地址符  
q=p;       //正确, 同类型赋值, q, p都指向i  
*q=*p+1;   //q指向i, 此时i已成为i+1  
q=p+1;     //‘1’仅是名, 值取决于所指对象每个单元  
           占多少字节。
```

# 3.2.1 程序变量的时空特性

## 引用和指针

P指针是地址变量 \*P是P所指的内容, 也有左值和右值  
\*P左值是P所指地址值, 即P的值  
\*P右值是所指地址内的内容值



引用是常指针是变量的别名, 但实现是不一样的

# 递引用 dereference

通过指针变量引用变量的值为递引用

\*P1右值即递引用

有些语言显式递引用算符如FORTH的@

1 13 VARIABLE xx (声明变量xx并赋初值13)

2 0 VARIABLE Y (声明变量Y并赋初值0)

3 xx @ 2 \* Y! (相当于 $Y = xx * 2$ )

如果只写xx 2 \* 则为将xx的地址乘以2放在Y之中

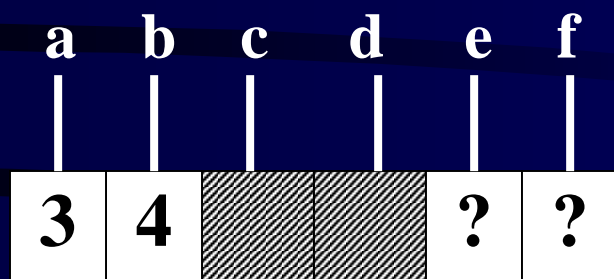
## 3.2.1 变量的时态

- 分配/未分配/除分配
  - 分配: 为程序对象创建—存储对象  
编译时分配叫静态分配 `allocate`  
运行时分配叫动态分配如声明指针 `p`, 执 `new` 才分配
  - 未分配: 声明了未分配运行时分配
  - 除分配: 取消存储对象(程序对象) `delete` 操作显式
  - 自动除配: 无用单元收集 `Garbage collection`  
动态语言有, 静态可有 `Ada` 可没有 `C`

# 续

## • 定义/未定义/失去定义

只要分配存储对象必然有残值，无意义即未定义  
赋值或初值变量得以定义



a,b:分配且有定义

c,d:分配未定义或失去定义

e,f:未分配或除配

声明和定义：定义必然声明；反之不然

声明的两个作用：给出对象, 该对象的时间有效性

出了声明的作用域该对象失去定义。在声明的作用域内显式删除也失去定义

# 存储管理阶段

- 初始分配：在程序开始执行时，存储器中每一块或者被分配用于存储相应数据，或者是作为可利用的空间而闲置。
- 存储单元回收：如果分配和使用的存储器一旦不再使用，则它们可由存储管理器回收以便再次使用。
- 压缩和再用：回收的存储空间可能立即被再用（与初始分配机制相同），还需要对回收的空间进行整理。



# 存储管理方式

- 栈

- 堆栈都是一种数据项按序排列的数据结构，只能在一端(称为栈顶(top))对数据项进行插入和删除。

- 静态的存储管理

- 一种最简单的分配技术，在翻译过程中完成，在程序执行过程中保持不变

- 堆的存储管理

- 一块存储空间，其中各个片段的分配和释放相对而言不以结构化的方式进行，需要存储分配、回收、压缩和重用。

## 3.2.2 存储模型

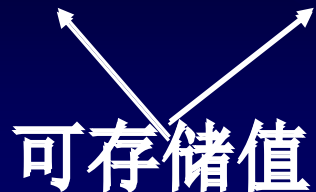
**可存储值Storable:** 指最小的可直接访问的可存储单元中的值  
**Pascal可存储值:**集合不选择更新某一元素是可存储值, Pascal, C , Ada数组可选择更新, 不是可存储值。

**引用非可存储(C++可存储), 过程和函数名也非可存储**  
**Java的集合与引用?**

**ML几乎都是可存储值, 它们是:**

- 基元类型值 仅除数组
- 记录、构造、表 不可更新其中一元素
- 函数抽象, ——ML重过程
- 变量引用

**( if exp then sin else cos ) (x) 得sin(x) | cos(x)**

**可存储值**

**也带来毛病:**  
**每次更新结构数据都要重来。**

# 存储对象的生存期

每个存储对象都有创建(生), 可用(活着), 撤销(死)的生命期。按生命期长短分:

- 全局变量 和引用程序寿命一样长
- 局部变量 和程序中的一个模块寿命一样长
- 持久变量 比程序寿命长除非显式撤销 文件变量
- 瞬间变量 (transient)持久变量的逆
- 静态变量
- 栈动态变量
- 显式堆动态变量
- 隐式堆动态变量

# 静态存储对象

- 编译时分配存储对象, 近代语言类属对象直到装入后确立 (elaboration) 之时才定下存储对象叫静态分配
- 一旦执行不再改动其存储, 直至所在存储单元无效叫静态 (Static) 存储对象
- 全局变量均为隐式的静态对象, COBOL, BASIC 全静态, ALGOL, C 是显式声明静态, Ada 不能。
- C 语言的静态变量是既私有又不随所在声明块中消逝, 全局于所在文件。
- Auto 是静态分配动态装入不叫静态对象。
- Extern 是静态对象。

**extern**

**static**

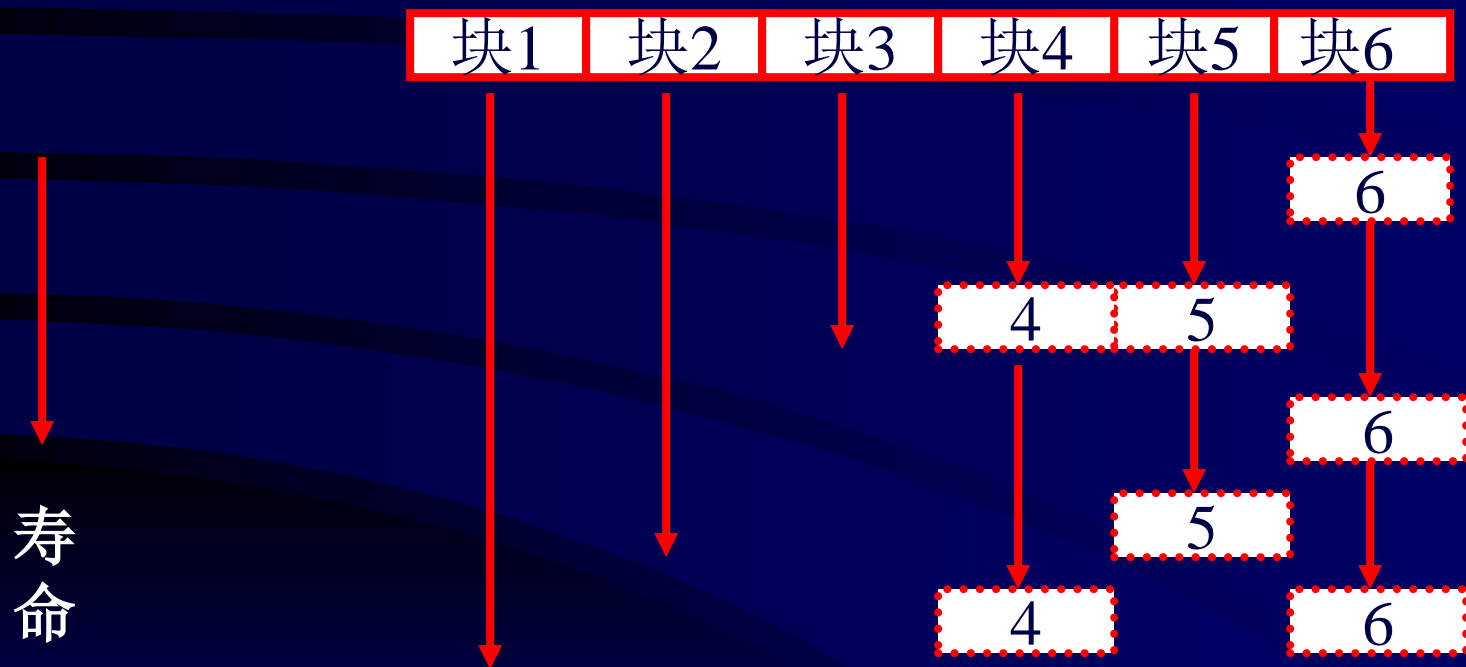
**auto**

# 动态存储方式与静态存储方式

- 存储空间可以分为三部分，即：
  - (1) 程序区
  - (2) 静态存储区
  - (3) 动态存储区
    - A、数形式参数。在调用函数时给形参分配存储空间。
    - B、函数中的自动变量（未加static声明的局部变量）。
    - C、函数调用时的现场保护和返回地址等。

# 动态存储对象

把寿命特长的(如文件, 全局量)排出来归到栈底的某一组, 把寿命特短的(如循环控制变量)另立嵌套组, 这个问题也就解决。

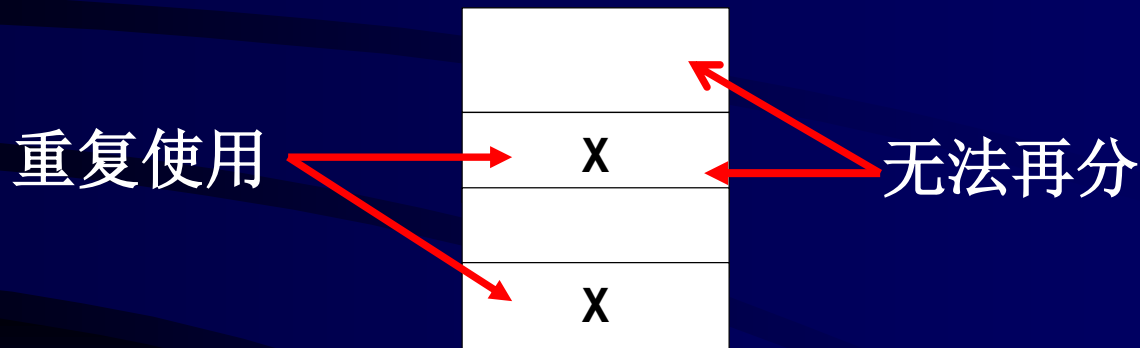


# 堆和栈的区别

- 管理方式不同;
- 空间大小不同;
- 能否产生碎片不同;
- 生长方向不同;
- 分配方式不同;
- 分配和存取效率不同;

# 动态存储对象

二叉树其大小由输入值定在运行中确立。内存开辟堆（heap）随生成随堆放动态存储对象。指针（即堆变量）所指程序对象用堆存放



问题：多次重分，内存成了小洞

解决办法：按寿命分组寿命最长的放在较低（按其所在块生命期）。



# 堆栈帧管理

由动态堆栈联想到一般嵌套式语言可按动态堆栈式管理,多数变量和所在块寿命一样长(语言称之为自动变量)

- 动态堆栈式存储

按程序动态执行,以动态堆栈管理局部数据和动态生成数据

运行时堆栈Run-time stack其底压入程序代码和全局,静态量。每执行到调用时生成一个堆栈帧(frame)记录该块数据信息,每当返回则局部量自动撤销。对于递归块的局部量可多次生成多次消除。

动态链描述调用父辈地址,返回地址是继续执行的下一地址。

静态链描述词法父辈lexical parent块地址,按该块复制局部变量。

续

本帧词法父辈



调用块首地址

程序代码  
全局静态存储

首先调用块  
堆栈帧

第二调用块  
堆栈帧

.....

最新调用块  
堆栈帧

栈顶

临时变量空间

堆栈帧组织

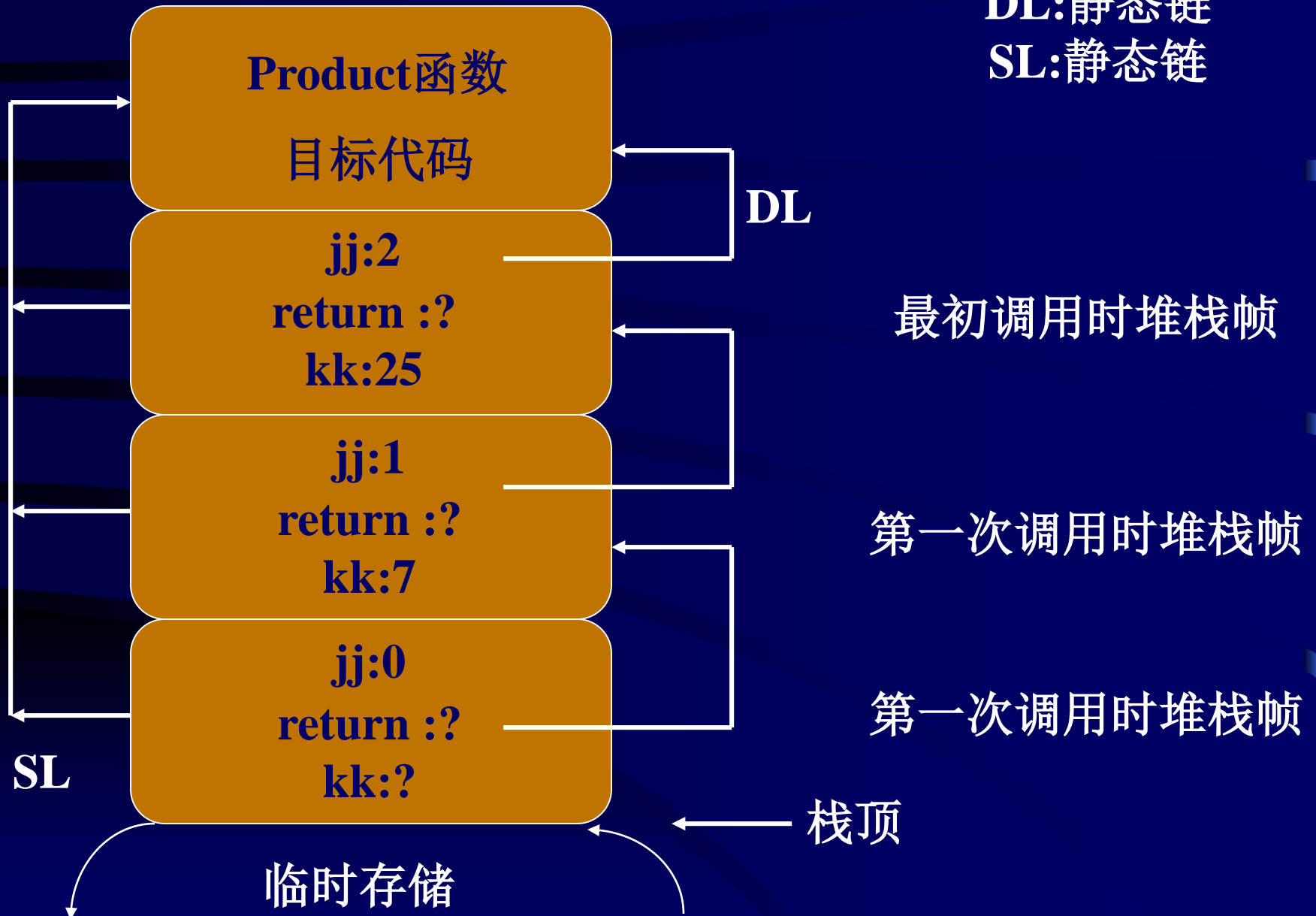
运行时堆栈

- 举例

求整数连乘积之递归程序:

```
function product (jj: Integer): Integer;  
var kk: Integer;  
begin  
    if jj <= 0 then product:=1  
    else begin readln (kk);  
              product:=kk * product(jj-1)  
            end  
end;  
end;
```

DL:静态链  
SL:静态链



# 动态堆存储

动态堆栈缺点：开始帧不知有多大, 要求存储对象比创建它的块寿命长。

指针和显式动态分配依然少不了堆。按帧也设 heap

- 各种语言堆分配

	FORTH	LISP	C	Pascal	Ada
分配	Here	cons	malloc	new	new
除配	无回收	有回收	显式除	显式除	有回收

# 死堆对象处理

- 死堆对象也叫无用单元garbage。(垃圾)
- 死堆的处理方法
  - 忽略死对象 不超过一页浪费, 若寿命差不多浪费不大
  - 保持一个自由表(链表)8个字节头说明数据
  - 按类型长度保持多个表减少识别域开销(Ada)

```
int * dangle (int ** ppp) { //参数是指针的指针
```

```
    int p=5;
```

```
    int m=21;
```

```
    *ppp=&p; //传回指向p的指针
```

```
    return &m; } //返回m的地址
```

```
main() {
```

```
    int k =17;
```

```
    int * pm, *pk=&k; //pk指向k
```

```
    pm = dangle (&pk); //返回时pm, pk均指向已失去定义的  
                        //指针函数的局部量, 即p和m
```

```
}
```

## 3.2.3 悬挂引用Dangling Reference

- 当堆式管理同时提供显式除配命令KILL时；堆栈式管理外块指针指向内块对象时；函数抽象作为第一类值时，都会产生悬挂引用

- 解决办法

Pascal把指针限制为堆对象且不用dispose，不提供地址运算。操作数组不能按指针寻址，快速索引。

C语言比较自由，悬挂引用留给程序员

局部函数作为返回值产生的悬挂指针。ML, Miranda完全作为堆变量且无KILL。Algol 68是引用（常指针），不赋比局部量寿命更长的值



## 3.3 计算对象的连接--- 束定 Binding

- 名字操纵程序对象。名字和存储对象联系起来才成为程序对象。
- 把声明名字（地址）和存储对象或语义实体连接起来叫束定。
- 束定 绑定 定连 连编（编译中连接模块）

# 名字与束定


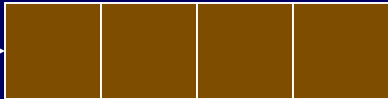
- 一名字可束定到多个对象。
- 一对象可以束定到多个名字。
- 在一个程序的声明期内一旦束定不再改变叫静态（早）束定
- 运行中一个名字束定到（多个）对象叫动态（晚）束定

# 3.3.1 静态束定

编译按数据类型为名字分配合适大小的存储对象，该对象首地址填入符号表即为静态束定，编译后变量表销毁，故静态束定不能变。

符号表

运行时内存

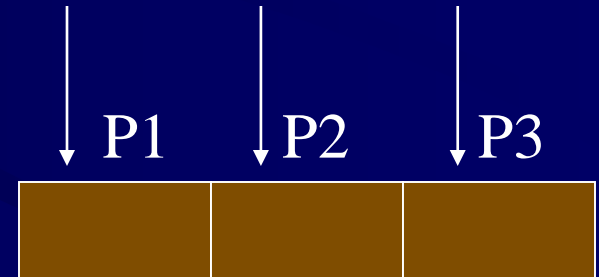
类型	名字	束定	存储对象
real	length	(地址)	
array[1..4] of integer	age	(地址)	

多重束定 FORTRAN等价语句，以内部引用实现。

Real P2, P3

Dimension P1(3)

Equivalence (P1,P2),(P1(2),P3)



其它语言多重束定是:

COBOL REDEFINES

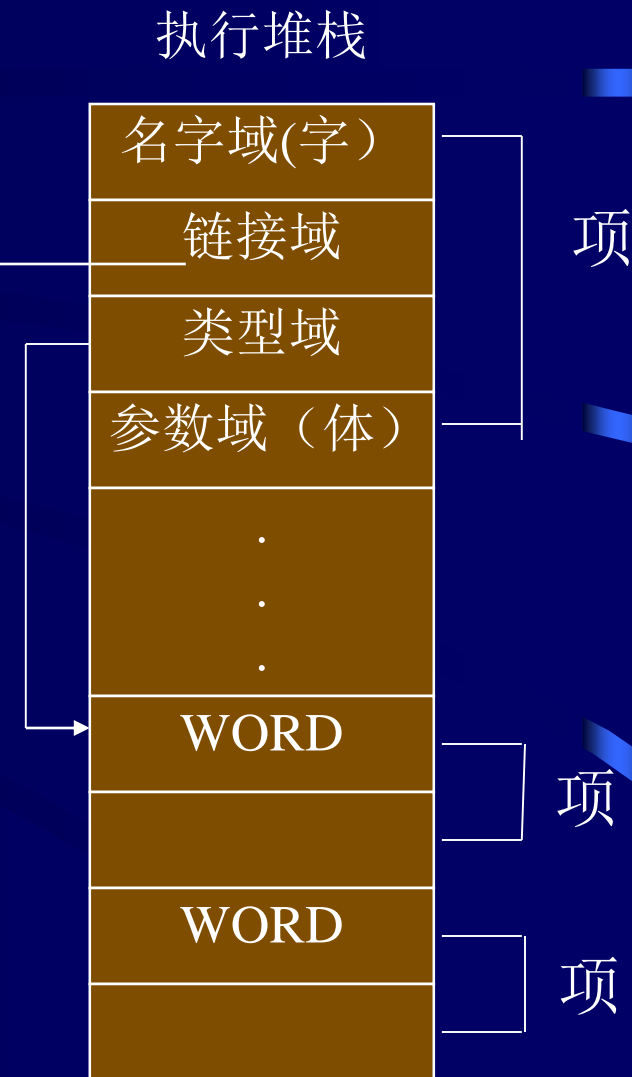
Pascal 变体记录

C 联合

## 3.3.2 动态束定

- 运行时将名（字）束定于其体（得其语义操作）
- 解释型语言一般保留名字表，运行中束定。
- FORTH语言动态束定它是编译 — 解释型**
- 每当生成word出一字项，有两片代码一为初始化和该名应执行的代码（编译完成）。
- 新字体中用到旧字，则运行中束定。
- 解释执行另一片代码时，动态束定编译块。
- FORGET命令可以撤销当前字定义。**

指前一字指针



## 例 FORTH 的动态束定

0 : 2by3array (' : '表示编译开始, 后为类型声明符)

1 create (编译动作: 将类型声明符装入字典项)

2 2, 3 (在字典项中存入  $2 \times 3$  维数)

3 12 allot (为六个短整数分配12个字节)

4 does> (运行时动作指令, 取下标)

5 rangecheck (函数调用, 检查下标)

6 if (如果不越界)

7 linearsub (函数调用, 计算线性下标值)

8 then (给出数组基地址和位移)

9 ; (' ; '切换成解释执行, 数据类型定义毕)

10

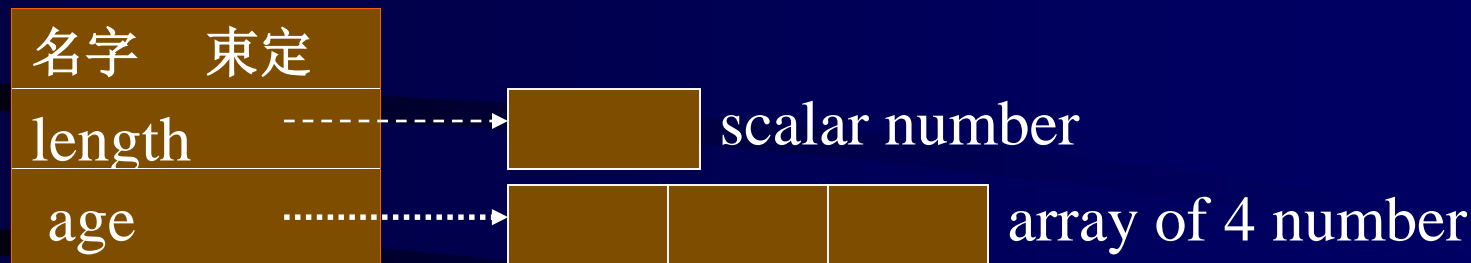
11 2by3array box (声明并分配名为box的数组变量)

12 10 1 2 box (给box(1, 2)赋值10)

# 3.3.3 无类型语言束定

符号表

运行时内存存储对象：类型标签



不同时刻可以将一名字束定到不同存储对象上。

# APL语言可显式操纵束定：束定符号‘←’

APL是无类型解释执行语言，以下是计算税金的程序：

▽ TAXCALC

[1] 'ENTER GROSS PAY'

[2] GROSS ← □ //输入什么值GROSS就是什么类型,□表示终端输入。

[3] →LESS × GROSS < 18000 //条件表达式为真转LESS标号句

[4] TAX ← .25 × GROSS //表达式结果值束定到TAX

[5] →DISPLAY //转到标号DISPLAY句

[6] LESS : TAX ← .22 × GROSS

[7] DISPLAY: 'THE TAX IS \$', Φ TAX

[8] ▽

## 3.3.4 声明declaration

- 声明指明本程序用到的所有程序对象给出所有束定集合  
声明给翻译必要信息
- 声明的确立产生事实上的束定
- 声明有显式的和隐式的

例：Ada的显式声明：

基本声明::=对象 |数 |类型 |子类型 |包  
|子程序 |任务 |类属 |异常 |类属设例  
|换名 |延迟

Ada的隐式声明：

块名字，循环名字，语句标号，循环控制变量，预定义运算符



# 声明种类

- 定义

- 为对象名提供完整的束定信息。声明可多次，定义只一次。
- 定义是从已有信息定义新信息，Ada称之为rename。
- 函数式语言，无类型语言只有值定义，没有变量声明。

val even = fn(n: int) => ( n mod 2 = 0 )

函数型构

函数体

《束定》

函数定义

值定义

fun even (n: int) = ( n mod 2 = 0 )

函数抽象

体

## • 顺序声明的Ada例子

```
package MANAGER is                                --声明程序包规格说明
    type PASSWORD is private;                    --声明私有类型未定义
    NULL_PASSWORD:constant PASSWORD;             --立即用私有类型声明变量
    function GET return PASSWORD;                 --返回私有类型函数
    function IS_VALID(P: in PASSWORD) return BOOLEAN;
Private
    type PASSWORD is range 0..700;                --定义私有类型
    NULL_PASSWORD:constant PASSWORD:=0 ;          --此时才定义
end MANAGER;
```

# • 顺序与并行声明

— 声明是顺序的, 声明后立即有用, 次序是重要的

D1; D2; D3 // D2中可利用D1的声明, D3可利用D1,  
//D2的声明 (见Ada例)

— 并行声明D1 | D2它们一般相互独立, 确立次序不影响语义  
ML的例子:

```
Val Pi = 3.14159
```

```
and sin = fn (x : real ) => (...)
```

```
and cos = fn (x : real ) => (...)
```

```
and tan = fn(x : real ) => sin(x)/con(x) 非法
```

并行声明仅当声明语句结束, 各并行子声明同时生效

## • 递归声明 用声明定义自己的声明

$D = \dots D \dots$  直接递归。名字一样就是递归

$D1 = \dots D2 \dots$   
 $D2 = \dots D1 \dots$  } 间接递归，或相互递归

指明递归 (ML)

```
val rec power = fn ( x : real, n : int ) =>  
    if n = 0 then 1.0  
    else if n < 0 then 1.0/power (x, -n)  
    else x * power (x, n-1)
```

如果没有rec两次power应用出现则调其它地方定义的power而不是本身

# 声明作用域

作用起始	顺序	每一子声明结束即起作用
	并行	所有子声明结束才起作用
	递归	只要遇见相同被声明符, 就起作用

- 嵌套块与可见性

可见性Visiability指简单标识符可引用该对象。嵌套有复盖名字冲突, 同样名字出现在嵌套块中, 按最近声明解释。被覆盖的只能加块名前缀(称全名), 如outer.x才可使用。C++用‘::’作用域分辨符。

- 标识符和全名

正因为有冲突所以内部束定时只能束定全名, A.x, B.y

# 块声明

- 把一个语句扩大到块，封闭的块有局部声明

Begin D; C end;

D是声明集，C是语句集。说明作用域只限于此块

- 块声明：含有局部声明的声明，子声明的束定用于声明。

ML: local

fun multiple (n:int,d:int) = (n mod d = 0) //函数说明

in

fun leap (y:int) = (multiple (y,4) //用于另一函数定义  
andalso not multiple (y,1000))

orelse multiple (y,400)

end

## 3.3.5 束定作用域与释义

### 束定与环境

在束定集的作用范围内称环境(environment)。

### 词法作用域

程序正文给出的嵌套声明的作用域称词法作用域，是词法子集和覆盖父辈

```
PROGRAM A;  
  VAR x, y:Integer;  
  FUNCTION B(d:Integer): Integer;  
    BEGIN b:=x+d END;  
  FUNCTION C(d:Integer):Real;  
    VAR x:Real;  
    BEGIN x:= 5.1;  C:=x+B(d+1) END;  
BEGIN  x:= 3;  y := 0  
      Writeln (C(y))  
END.
```

当 (x=3,y=0)时 C(0)=9.1

# 引用环境

- 语句的引用环境是指这条语句中所有可见变量的集合。
- 静态作用域
  - 语句局部作用域中声明的变量
  - 和祖先作用域中声明的所有可见变量的集合
  - 为编译提供信息，生成代码和数据结构，运行时引用其他作用域的变量



# • 动态作用域和递归

- 递归每展开一层束定一次，且保留至递归终止。运行中动态束定。因声明作用域大小不同。到底多大静态不得而知。
- 词法作用域和动态作用域求值差异

我们沿用上例以LISP求C (0)：

```
(defun A
  (x, y)
  (printc( C y)))      //以最近束定释义
(defun B (d)
  (+d x))
(defun C (d)
  (let (x'5.1))
  (+x(B (+d 1))))
(A 3 0) =(printc (C 0))      //x=3, y=0
        =(printc(+5.1(B(+ 0 1)))) //x=5.1, d=0
        = (printc(+5.1 (B(1))))
        =(printc(+5.1(+1 5.1))) //d=1, x就近取值=5.1
        =printc(+5.1 6.1))
        =(printc 11.2)      //上例是9.1
```

# 静态和动态作用域评价

- 静态作用域
  - 允许的访问权限比需要的更多
  - 不能适应代码的改进/重构需求
  - 替代方式：封装
- 动态作用域
  - 子程序不同执行过程引用不同的非局部变量
  - 子程序中的局部变量对任何其他正在执行的子程序都可见
  - 不能进行非局部变量引用的静态类型检查
  - 必须知道子程序调用顺序才能确定非局部变量的引用含义，阅读困难
  - 更消耗时间

# 作用域与生命期匹配

- 形参与实参结合(束定)实参（程序对象）寿命比该函数/过程长
- 静态(static)对象声明其寿命比所在块长。
- 指针实现的记录或结构的树或链其第一头指针在堆中分配，寿命比堆栈帧内束定的变量长。
- 文件变量的寿命一般比程序中对象长。

# 命名常量

- 命名常量只与值绑定一次
- 命名常量的一个重要用途是将程序参数化
- 某些语言允许动态地将值绑定于命名常量

```
const int result = 2 * width + 1;
```

# 束定机制与语言翻译器

类型语言——编译高效，符号表不存在。局部束定要保留

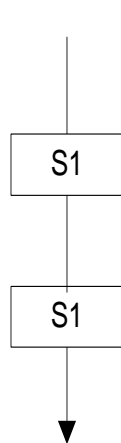
- 无类型语言——解释方便，符号表存在。
- 词法作用域高效, 排错易，但要事先知道如何束定
- 动态作用域灵活，方便，低效。有递归不可少。

## 3.4 计算组织-程序控制

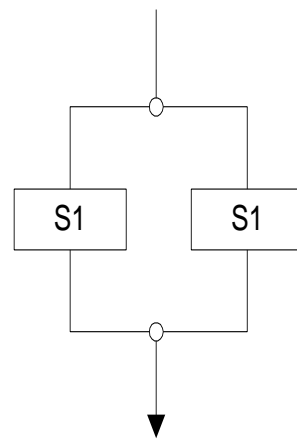
- 冯·诺依曼机器模型变量的时空特性对程序中求值的次序是十分敏感的
- 表达式的求值次序是最低层的程序控制，在它的上层是四类控制：顺序控制、选择控制、重复(迭代)、函数或过程调用
- 再上一层是对程序模块的控制。包括一个程序的各模块组织以及它们与环境的相互关系
- 并发控制也是一类控制，它可以在语句级，特征块和模块级实施并发控制

## 3.4.1 一般概述

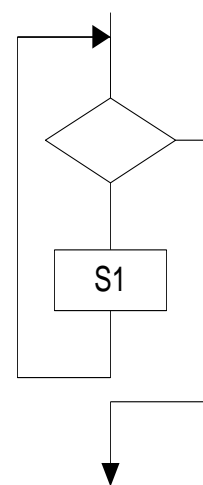
- 命令式语言只要有赋值语句 $V := \text{EXP}$ ，简单的逻辑条件 $\text{IF}(e)$ 和 $\text{GOTO}$ 语句就可以编出一切计算程序(输入/出除外)。
- 语句级控制由于 $\text{GOTO}$ 危害导致结构化程序。
- 1966年Boehm和Jacopini回答了这个问题：任何流程图的计算逻辑都可以用顺序组、条件选择组、迭代组三种程序结构实现。
- 保留 $\text{GOTO}$ 的积极作用限制 $\text{GOTO}$ 的副效应，把它们改头换面变为比较安全的顺序控制器(sequencer)。



顺序



条件选择



迭代

三种最基本的程序结构

# 顺序控制

- 表达式是程序语句的基本组成，体现了程序控制和数据改变的方法，像优先级和括号这样的属性，决定了表达式的计算顺序。
- 用在语句间或一组语句中的结构，如条件和循环语句，决定了程序流程的跳转。
- 有时程序定义不依赖于语句的执行模式，但仍然能使程序运行。（如Prolog）
- 像子程序调用和协同程序这样的子程序结构，将一程序段的控制转移到另一程序段。（调用、协同、子程序间通信）



# 顺序控制

- S1; S2

进一步扩展可为: S1; S2; ...; Sn

- Ada语句较全:

简单\_语句 ::= 空\_语句

| 赋值\_语句 | 过程调用\_语句

| goto\_语句 | 入口调用\_语句

| 出口\_语句 | 返回\_语句

| 引发\_语句 | 夭折\_语句

| 延迟\_语句 | 代码\_语句

其中空\_语句, 赋值\_语句, 延迟语句, 代码\_语句不影响控制和转移, exit(出口)语句, raise(引发)语句, abort(夭折)语句, return(返回)语句都是顺序控制器。

# 条件选择控制

- if(e)无结构。Algol60改为if...then...else结构，退化是if...then

- 悬挂else

if E1 then if E2 then S1 else S2 E1为‘真’‘假’均可执行S2

解决if E1 then begin if E2 then S1 else S2 end

Pascal,Algol,C

if E1 then begin if E2 then S1 end else S2

if E1 then if E2 then S1 endif else S2 endif

Fortran-77,Ada

if E1 then if E2 then S1 else S2 endif endif

└——就近匹配——┘

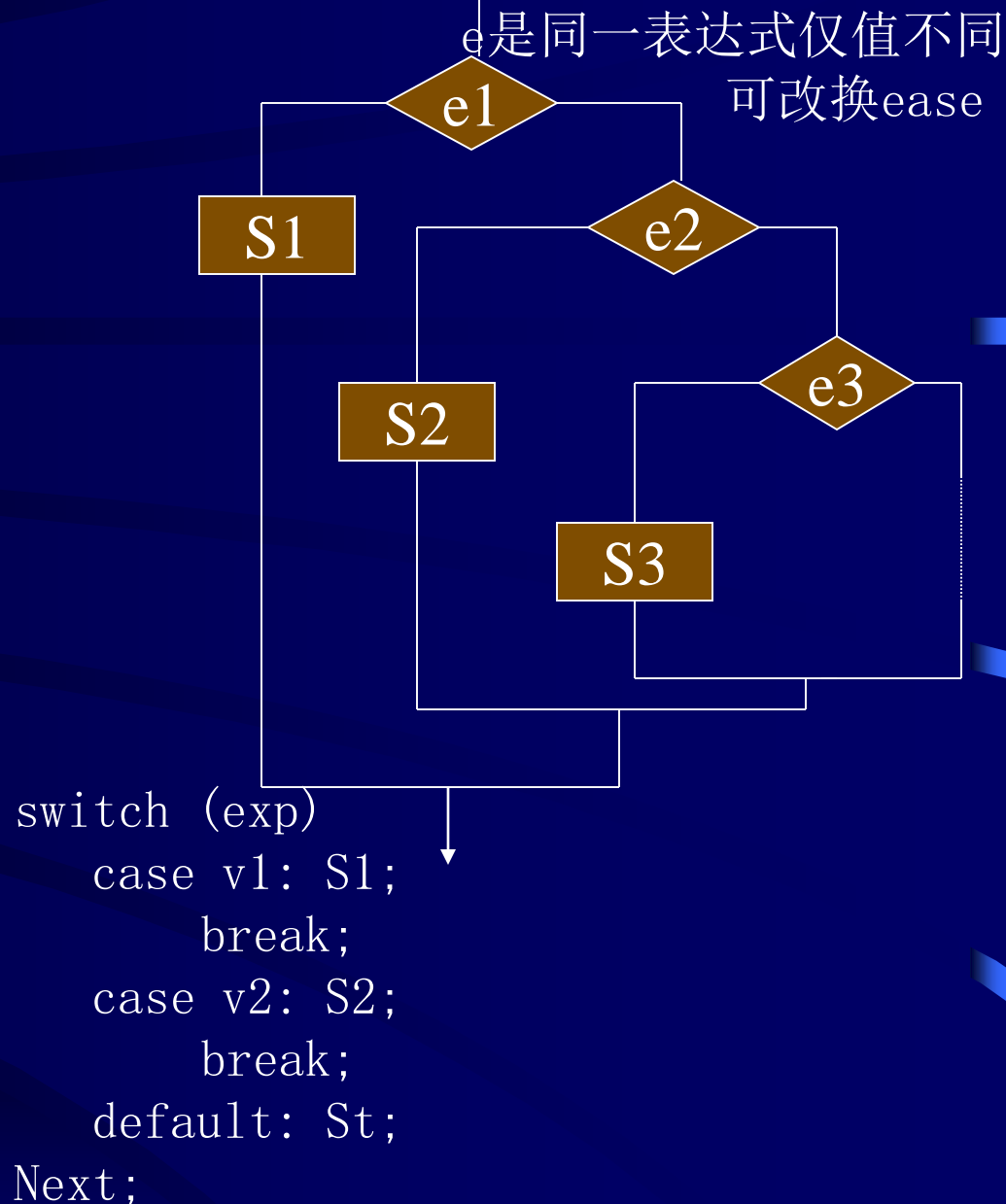
## • 嵌套if和case

Ada 的case语句

```
IF exp1 THEN
    ST1
ELSEIF exp2 THEN
    ST2
ELSEIF exp...
    ...
ELSE
    SF3
ENDIF
```

```
case Exp is
    when v1=> S1;
    when v2=> S2;
    ...
    when vm|vn=>Sn;
    when others => St;
end case;
```

执行一个Si跳到end case, 没有break要顺序执行, 直到next;



## C语言以条件表达式实现选择控制

$a=b<c ? b:c$                       `if (b<c) a=b; else a=c;`

条件表达式

条件语句

命令式语言大量依赖顺序命令，纯函数式语言完全依赖写嵌套函数调用。

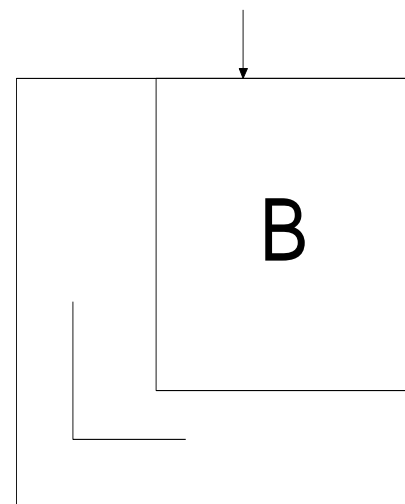
## 迭代控制

迭代一般用于处理同构的、聚集数据结构的重复操作，诸如数组、表、文件。

• 显式迭代（循环），显示迭代有以下几种：

无限循环

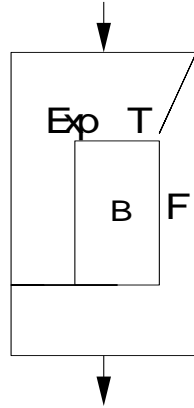
```
loop [name];  
    B  
end loop;
```



- 有限循环的三种形式:

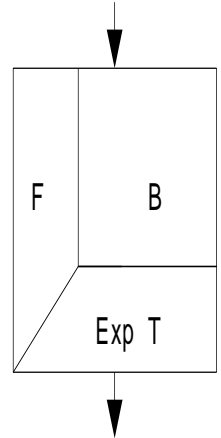
while\_do

```
while Exp
  loop
    B:
  end loop
```



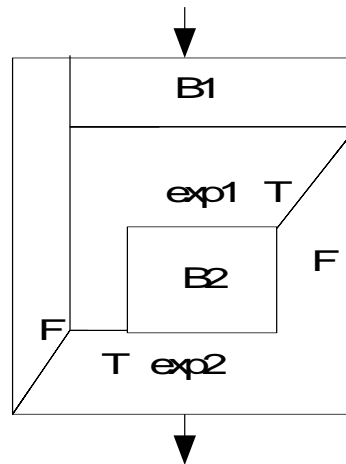
do\_until

```
REPEAT
  B
UNTIL Exp;
```



do\_while\_do

```
BEGIN
  B1;
  WHILE Exp1
    B2;
  REPEAT Exp2
```



## - 计数循环

FORTRAN

初始 上界 增量

DO L I=EXP1, EXP2, EXP3

Ada

for BACKDAY in reverse DAY ' RANGE loop

在循环控制变量和表达式约定上几十年来一直在演变  
C采用迭代元素(Iteration element)

iterate (<控制元素>)<迭代域>

iterate (<a>; <b>; <c>)<迭代域>

对迭代元素不加任何限制。语句，表达式，连续赋值，空均可。  
先算<a>,再算<b>,若不为零，则算<迭代域>然后执行<c>。若为  
‘零’出口。若不为‘零’再算<b>,若不为‘零’重复以上步  
骤，若为‘零’则出循环。

例：用for循环计算表中元素之和

设表list是一简单结构，一个成分的值value，另一个成分是指向下一表元素的指针next。可以有六种写法：

[1] 先赋sum初值进入正规for 循环：

```
sum=0;
sum += current ->value;
for (current=list; current != NULL;
      current =current -> next)
sum += current->value
//循环控制变量current就用表list
```

[2] sum在循环内赋初值：

```
for (current=list, sum=0; current != NULL;
      current =current -> next)
sum += current->value
```

[3] 循环控制变量可以在for以外赋初值， 增量在体内赋值：

```
current = list, sum=0;
for ( ; current != NULL; )
sum += current->value, current = current->next;
//用', '号连接的两语句成为一个命令表达式
```

[4] 循环条件为空，用条件表达式完成：

```
current = list, sum = 0;
for(;;)
    if (current == NULL) break;
    else sum += current->value, current->next;
    //不提倡此种风格
```

[5] 完全不用循环体，全部在循环条件中完成

```
for (current = list, sum = 0;
     (current = current->next) != NULL;
     sum += current->value);
//效率最高. 但list不能为NULL表
```

[6] 逻辑混乱，也能正确计算：

```
current = list, sum = 0;
for(;current=current->next;sum += current->value)
    if (current == NULL) break;
//最坏的风格，list也不能为NULL表
```



# 隐式迭代控制

- 所谓隐式迭代是语言系统自动实施迭代。
- 这对第四代语言，声明式语言(函数式逻辑式)都非常重要，而且是语言发展方向上的重要机制。
  - 对聚集对象的迭代
  - 回溯

# 隐式迭代控制

## ● 对聚集对象的迭代

APL的隐式迭代

▽ ANSWER ← A FUNC B //声明定义两参数函数名为FUNC

[1] ANSWER ← (3×A) - B //函数定义

▽

X← 4 9 16 25 //X, Y束定为整数向量

Y← 1 2 3 4

Z← (3 4) ρ 1 2 3 4 1 4 9 16 1 8 27 64

//Z束定为3×4矩阵

# 一种类似dBASE的dBMAN查询的隐循环

设每个雇员一个文件， 当前记录类型域是：

lastname , sex, hours, grosspay, withheld

有以下dBMAN命令：

- [1]. display all for last name = 'Jones' and sex='M'
- [2]. copy to lowpay.dat all for grosspay < 10000
- [3]. sum grosspay, withheld to grosstot, wheltdtot all for hours >0

# 回溯 Backtracking

## Prolog查询的回溯实现

设数据库中已列出以下事实：

f1. likes (Sara, John)	f9. does (Mike, skating)
f2. likes (Jana, Mike)	f10. does (Jane, swimming)
f3. likes (Mary, Dave)	f11. does (Sara, skating)
f4. likes (Beth, Sean)	f12. does (Dave, skating)
f5. likes (Mike, Jana)	f13. does (John, skating)
f6. likes (Dave, Mary)	f14. does (Sean, swimming)
f7. likes (John, Jana)	f15. does (Mary, skating)
f8. likes (Sean, Mary)	f16. does (Beth, swimming)

查询相爱且有共同运动爱好的两个人

```
?-likes (X,Y), likes(Y,X), does (X,Z), does(Y,Z)  
      (Prolog)
```

匹配	结果
[1]. 沿数据库匹配第一子目标( $f1 \leftarrow g1$ )	$X \leftarrow \text{Sara}, Y \leftarrow \text{John}$
[2]. 匹配g2, 沿 $f1 \rightarrow f7$ . Y对X不对	失败
[3]. 回溯重配g1, ( $f2 \leftarrow g1$ )	$X \leftarrow \text{Jana}, Y \leftarrow \text{Mike}$
[4]. 匹配g2, 沿 $f1 \rightarrow f5$ ( $f5 \leftarrow g2$ )	g2成功
[5]. 匹配g3, 沿 $f9 \rightarrow f10$ ( $f10 \leftarrow g3$ )	$X \leftarrow \text{Jana}, Z \leftarrow \text{swimming}$
[6]. 匹配g4, f9, Y对, X不对,	失败
[7]. 回溯第[5]步 $f11 \rightarrow f16$ 无新匹配	失败
回溯第[4]步 $f6 \rightarrow f8$ 无新匹配	失败
回溯第[3]步重配g1( $f3 \leftarrow g1$ )	$X \leftarrow \text{Mary}, Y \leftarrow \text{Dave}$
[8]. 重匹配g2, ( $f6 \leftarrow g2$ )	g2成功
[9]. 匹配g3 ( $f15 \leftarrow g3$ )	$X \leftarrow \text{Mary}, Z \leftarrow \text{skating}$
[10]. 匹配g4, 按 Z找Y( $f12 \leftarrow g4$ )	$Y \leftarrow \text{Mary}, Z \leftarrow \text{skating}$
[11]. 查询全部谓词满足( $f3 \leftarrow g1$ ) ( $f6 \leftarrow g2$ ) ( $f15 \leftarrow g3$ ) ( $f12 \leftarrow g4$ ) $X = \text{Mary}, Y = \text{Dave}, Z = \text{skating}$	
[12]. 从( $f4 \leftarrow g1$ )再查找有无另一组解重复第[7]-[11]步, 结果 无新解	

## 3.4.2 异常处理

程序无法执行下去，也就是出现了异常(exception)情况。

在早期语言的程序中，出现了这种情况就中断程序的执行，交由操作系统的运行程序处理。

现在向用户开放，Ada, C++, Java。

# 异常定义与处理段

定义用户定义： <异常名>: exception;

系统定义的异常（Ada）：

- CONSTRAINT\_ERROR 凡取值超出指定范围叫约束异常
- NUMBER\_ERROR 数值运算结果值实现已不能表达  
叫数值异常
- STORAGE\_ERROR 动态存储分配不够时，叫存储异常
- TASKING\_ERROR 任务通信期间发生的异常叫任务异常。
- PROGRAM\_ERROR 除上四种而外程序中发生的一切异常，  
都叫程序异常.如子程序未确立就调用等。

- 预定义的可以显式也可以隐式引发，而用户定义的异常必须显式引发
- 引发语句：  
    raise [<异常名>];
- 引发语句可出现在任何可执行语句所在的地方。一旦引发，则本程序块挂起转而执行本块异常处理段中同名的异常处理程序。



每个程序块都可以在该块末尾设立异常处理段:

declare:

    <正常程序声明>;    [<异常声明>];

begin:

    <正常程序代码>;    [<引发语句>];

exception:

    when<异常名> => <处理代码>                    --异常处理段

    when。。。                    --一般再次引发原异常（缺省名字）

    [<引发语句>]

end;

- Ada的嵌套异常及异常传播的

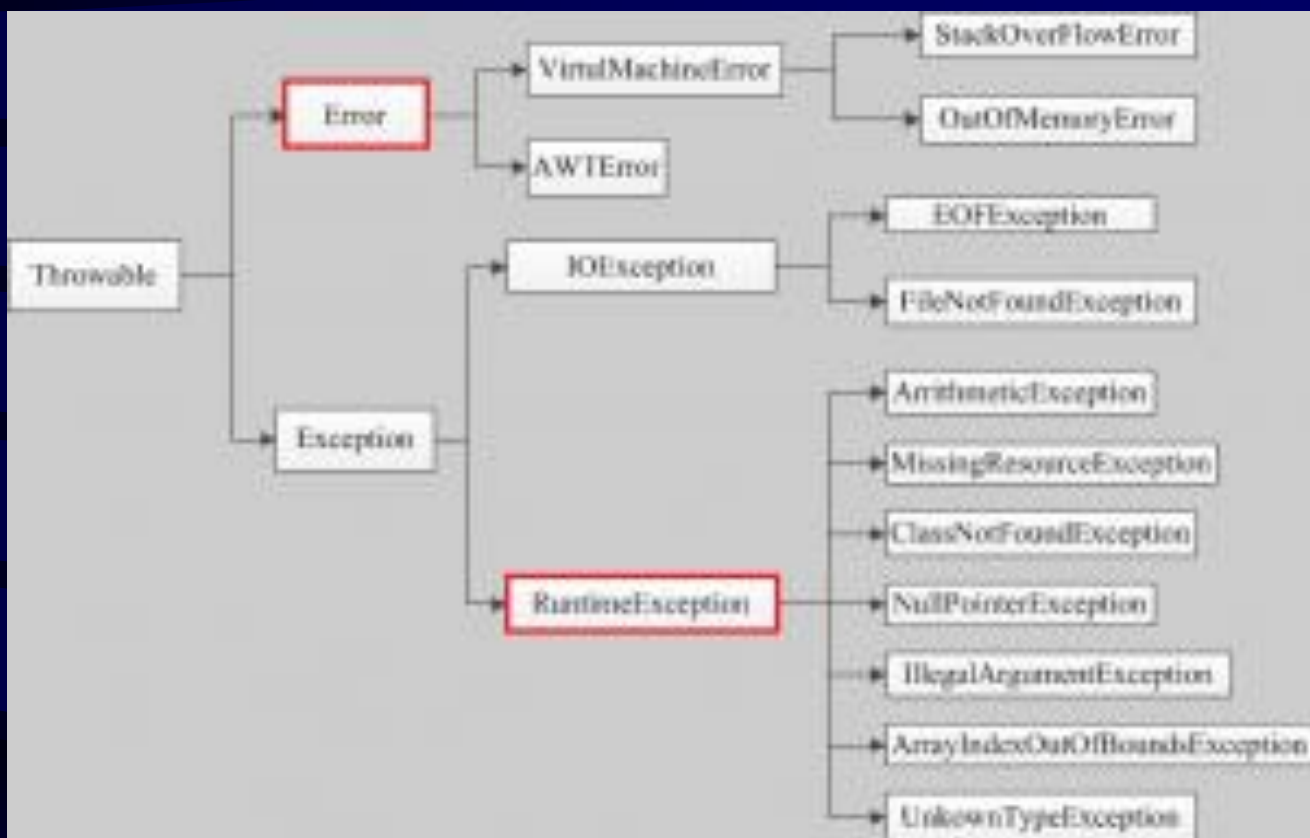
```
with TEXT_IO, MATH_FUNCTIONS;  
  use TEXT_IO, MATH_FUNCTIONS;  
procedure ONE_CIRCLE (DIAM:Float) is  
  RADIUS, CIRCUMFERENCE, AREA:Float;  
begin  
  CIRCUMFERENCE:=DIAM*22.0/7.0;  
  RADIUS:=DIAM/2.0;  
  AREA:=RADIUS*RADIUS*22.0/7.0;  
  PRINT_CIRCLE (DIAM, RADIUS, CIRCUMFERENCE, AREA);  
[7] exception  
  when NUMBER_ERROR =>  
    raise;  
end ONE_CIRCLE;
```

```

procedure CIRCLES is
    DIAMETER:Float;
    MAXIMUM: constant Float:=1.0e6;
[0] TOO_BIG_ERROR: exception;
    PUT ("please enter an float diameter.");
    begin
[1] loop
        begin
            GET (DIAMETER);
            if DIAMETER > MAXIMUM then
[2]   raise TOO_BIG_ERROR;
            else
[3]   exit when DIAMETER <=0;
            end if;
[8] ONE_CIRCLES(DIAMETER);
            PUT("Please enter another diameter(0 toquit):");
[4] exception
            when TOO_BIG_ERROR=>
                PUT ("Diameter too large ! please reenter.");
[5] end ;
[6] end loop;
        PUT ("processing finished.");
        PUT  New_line;
    end CIRCLE;

```

# Java异常类型



**checked exception (by compiler):**  
可控异常，要求必须在方法声明中列出来，否则无法通过编译。  
继承自Exception

**unchecked exception (by compiler):** 不可控异常，可以不在方法声明中列出。继承自RuntimeException

# 不可控异常类型

```
public class ExceptionTest {  
    public static void main(String[] args) {  
        int i = 10/0;  
    }  
}  
Exception in thread "main"  
java.lang.ArithmeticException: / by zero  
at ExceptionTest.main(ExceptionTest.java:5)  
  
public class ExceptionTest {  
    public static void main(String[] args) {  
        int arr[] = {'0','1','2'};  
        System.out.println(arr[4]);  
    }  
}  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 4  
at ExceptionTest.main(ExceptionTest.java:6)  
  
import java.util.ArrayList;  
public class ExceptionTest {  
    public static void main(String[] args) {  
        String string = null;  
        System.out.println(string.length());  
    }  
}  
Exception in thread "main"  
java.lang.NullPointerException  
at ExceptionTest.main(ExceptionTest.java:5)
```

# 不可控异常类型

## RuntimeException



# 可控异常类型

```
try {  
    String input = reader.readLine();  
    System.out.println("You typed : "+input);  
    // Exception prone area  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Exception



FileNotFoundException  
ParseException  
ClassNotFoundException  
CloneNotSupportedException  
InstantiationException  
InterruptedException  
NoSuchMethodException  
NoSuchFieldException

# 异常类型定义

- 选择扩展Exception或RuntimeException
- 只需定义构造函数

```
public class NewKindOfException extends Exception {  
  
    public NewKindOfException( ) { super( ); }  
    public NewKindOfException(String s) { super(s); }  
}
```

```
Exception e1=new NewKindOfException( “this is the reason” );  
String s = e1.toString();
```

→ “NewKindOfException: this is the reason”



# 异常的抛出与捕捉处理

- 如果一个方法m没有使用try...catch来捕捉和处理可能出现的异常，则会产生如下两种情况
  - 如果抛出的是不可控异常，则Java会自动把该异常扩散至m的调用者
  - 如果抛出的是可控异常，且在m的标题中列出了该异常或者该异常的某个父类异常，则Java自动把该异常扩散至m的调用者
- 由于不可控异常的产生在运行时才能确定，因此需要格外小心其捕捉与处理

```
try { x=y[n];}  
catch (IndexOutOfBoundsException e) {  
    //handle IndexOutOfBoundsException from the array access y[n]  
}  
i=Arrays.search(z, x);
```

# 关于异常的处理方式

- 反射

- 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后抛出**另一个异常**e2给其调用者
- “我”处理了一种意外情况，根据软件需求，这种情况也需要报告给”上层”

- 屏蔽

- 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后不再抛出异常给其调用者
- “我”处理了一种意外情况，根据软件需求，没必要让”上层”知道是否发生了这种意外

# 关于异常的处理方式

- 对于在给定数组中搜索某个元素而言，考虑数组对象为null，或者对数组访问越界两种意外情况
  - NullPointerException需要通知调用者
    - Hey, 你给了一个不存在的数组！
  - IndexOutOfBoundsException呢？
    - Hey, 你给的数组我没访问好？！

```
public static int min (int[ ] a) throws
NullPointerException, EmptyException {
    /* @EFFECTS: if a is null throws
    NullPointerException else if a is empty
    throws EmptyException else returns the
    minimum value of a
    */
    int m;
    try { m = a[0]; }
    catch (IndexOutOfBoundsException e)
    {
        throw new EmptyException
        ("Arrays.min"); }
    for (int i=1; i < a.length; i++)
        if (a[i] < m) m = a[i];
    return m;
}
```

# 何时使用异常

- 当需要使用一个特殊返回值来告知调用者输入有异常情况时
  - 特殊返回值和异常相比，所表达的语义模糊，且容易被忽略
- 当一个方法期望可以在多处被重用时
  - 全局适用过程

# 使用可控异常还是不可控异常

- 如果期望调用者提供的输入数据不会引发异常，就应该使用不可控异常 //隐式处理
  - 不可控异常默认逐层“上报”，确保会有方法进行处理，否则会中断程序的运行
  - 不可控异常对于调用者要求较高，如果忘记捕捉，会带来潜在的程序崩溃风险
- 如果不对调用者做特殊要求，应该使用可控异常 //显式处理
  - 能够提高程序的健壮性

# 防御编程(Defensive Programming)

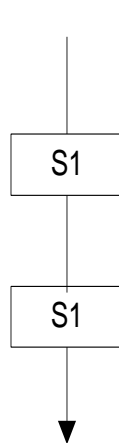
- 异常处理机制提供了一种在主流程处理之外的程序防护能力
  - 确保主流程逻辑的清晰性
  - 通过异常类型有效管理程序需要关注的各种意外情况
  - 反射和屏蔽机制为异常处理带来了灵活性
- 在设计类的方法时，需要问如下问题
  - 有哪些输入？
  - 输入会出现哪些“例外”情况？
  - 这些“例外”情况如何通知调用者？

## 3.4 计算组织-程序控制

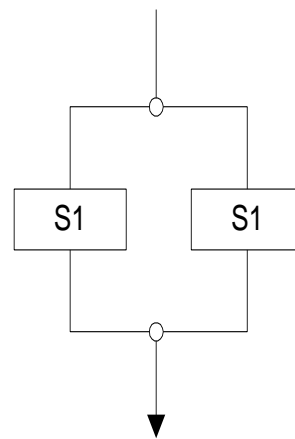
- 冯·诺依曼机器模型变量的时空特性对程序中求值的次序是十分敏感的
- 表达式的求值次序是最低层的程序控制，在它的上层是四类控制：顺序控制、选择控制、重复(迭代)、函数或过程调用
- 再上一层是对程序模块的控制。包括一个程序的各模块组织以及它们与环境的相互关系
- 并发控制也是一类控制，它可以在语句级，特征块和模块级实施并发控制

## 3.4.1 一般概述

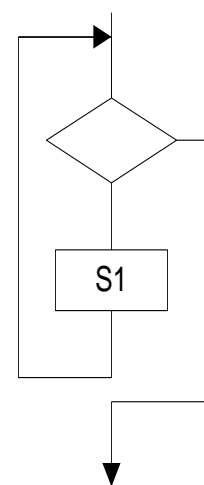
- 命令式语言只要有赋值语句 $V := EXP$ ，简单的逻辑条件 $IF(e)$ 和 $GOTO$ 语句就可以编出一切计算程序(输入/出除外)。
- 语句级控制由于 $GOTO$ 危害导致结构化程序。
- 1966年Boehm和Jacopini回答了这个问题：任何流程图的计算逻辑都可以用顺序组、条件选择组、迭代组三种程序结构实现。
- 保留 $GOTO$ 的积极作用限制 $GOTO$ 的副效应，把它们改头换面变为比较安全的顺序控制器(sequencer)。



顺序



条件选择



迭代

三种最基本的程序结构



# 顺序控制

- 表达式是程序语句的基本组成，体现了程序控制和数据改变的方法，像优先级和括号这样的属性，决定了表达式的计算顺序。
- 用在语句间或一组语句中的结构，如条件和循环语句，决定了程序流程的跳转。
- 有时程序定义不依赖于语句的执行模式，但仍然能使程序运行。（如Prolog）
- 像子程序调用和协同程序这样的子程序结构，将一程序段的控制转移到另一程序段。（调用、协同、子程序间通信）

# 顺序控制

- S1; S2

进一步扩展可为: S1; S2; ...; Sn

- Ada语句较全:

简单\_语句 ::= 空\_语句

| 赋值\_语句 | 过程调用\_语句

| goto\_语句 | 入口调用\_语句

| 出口\_语句 | 返回\_语句

| 引发\_语句 | 夭折\_语句

| 延迟\_语句 | 代码\_语句

其中空\_语句, 赋值\_语句, 延迟语句, 代码\_语句不影响控制和转移, exit(出口)语句, raise(引发)语句, abort(夭折)语句, return(返回)语句都是顺序控制器。

# 条件选择控制

- if(e)无结构。Algol60改为if...then...else结构，退化是if...then

- 悬挂else

if E1 then if E2 then S1 else S2 E1 如何执行

解决if E1 then begin if E2 then S1 else S2 end

Pascal,Algol,C

if E1 then begin if E2 then S1 end else S2

if E1 then if E2 then S1 endif else S2 endif

Fortran-77,Ada

if E1 then if E2 then S1 else S2 endif endif

└——就近匹配——┘

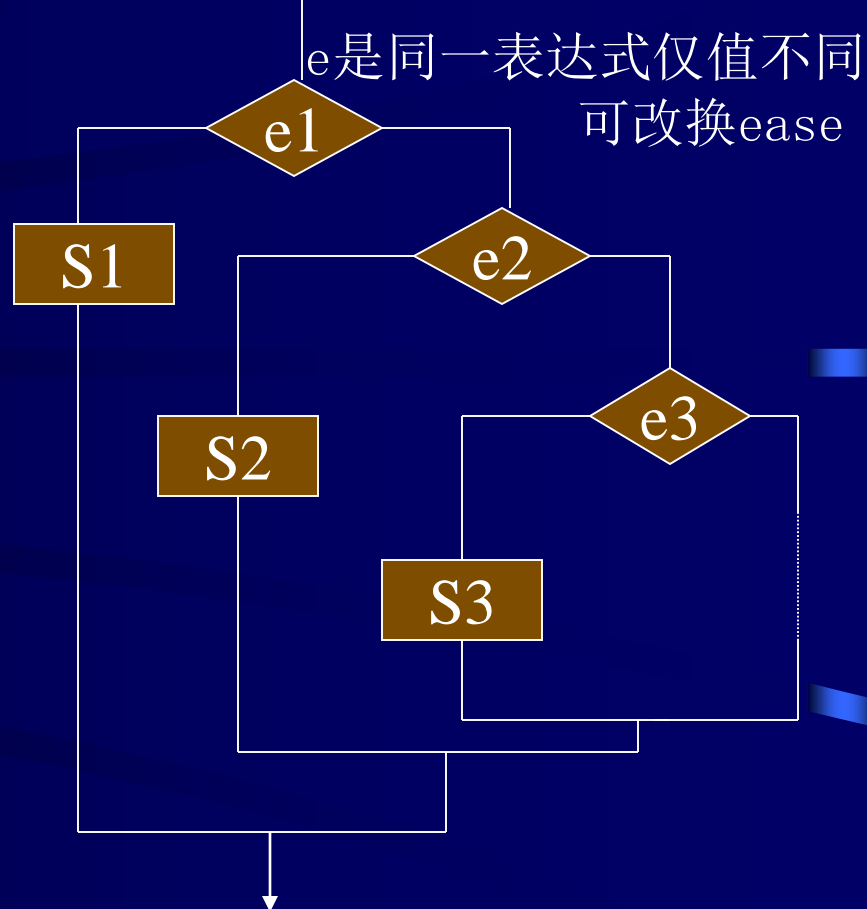
- 嵌套if和case

Ada 的case语句

```
IF exp1 THEN
    ST1
ELSEIF exp2 THEN
    ST2
ELSEIF exp...
    ...
ELSE
    SF3
ENDIF
```

```
case Exp is
    when v1=> S1;
    when v2=> S2;
    ...
    when vm|vn=>Sn;
    when others => St;
end case;
```

执行一个Si跳到end case



```
switch (exp)
    case v1: S1;
                break;
    case v2: S2;
                break;
    default: St;
```

next;

没有break要顺序执行，直到next

## C语言以条件表达式实现选择控制

$a=b<c ? b:c$

条件表达式

`if (b<c) a=b; else a=c;`

条件语句

命令式语言大量依赖顺序命令，纯函数式语言完全依赖写嵌套函数调用。

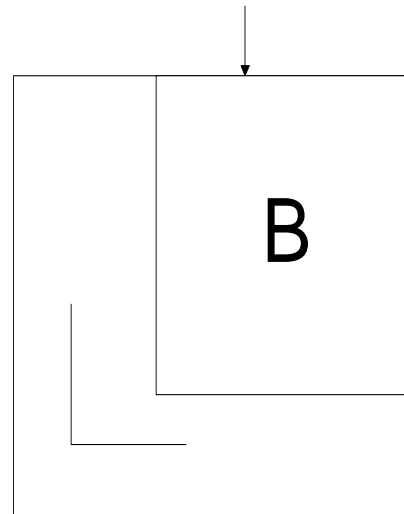
# 迭代控制

迭代一般用于处理同构的、聚集数据结构的重复操作，诸如数组、表、文件。

- 显式迭代（循环），显示迭代有以下几种：

无限循环

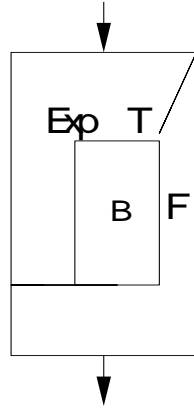
```
loop [name];  
    B  
end loop;
```



- 有限循环的三种形式:

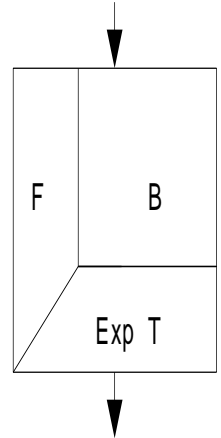
while\_do

```
while Exp
  loop
    B:
  end loop
```



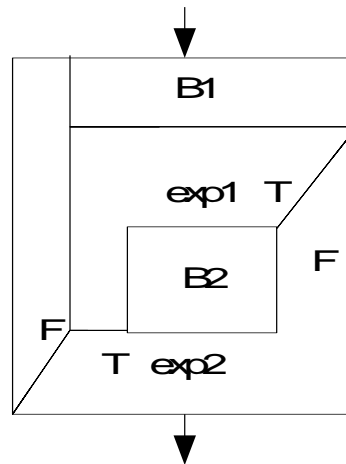
do\_until

```
REPEAT
  B
UNTIL Exp;
```



do\_while\_do

```
BEGIN
  B1;
  WHILE Exp1
    B2;
  REPEAT Exp2
```



## - 计数循环

- FORTRAN

```
DO L I=EXP1, EXP2, EXP3
```

- Ada

```
for BACKDAY in reverse DAY ' RANGE loop
```

在循环控制变量和表达式约定上几十年来一直在演变  
C采用迭代元素(Iteration element)



例：用for循环计算表中元素之和

设表list是一简单结构，一个成分的值value，另一个成分是指向下一表元素的指针next。可以有以下六种写法：

[1] 先赋sum初值进入正规for 循环：

```
sum=0;
sum += current ->value;
for (current=list; current != NULL;
      current =current -> next)
    sum += current->value
//循环控制变量current就用表list
```

[2] sum在循环内赋初值：

```
for (current=list, sum=0; current != NULL;
      current =current -> next)
    sum += current->value
```

[3] 循环控制变量可以在for以外赋初值， 增量在体内赋值：

```
current = list, sum=0;
for ( ; current != NULL; )
    sum += current->value, current = current->next;
//用', '号连接的两语句成为一个命令表达式
```

[4] 循环条件为空，用条件表达式完成：

```
current = list, sum = 0;
for(;;)
    if (current == NULL) break;
    else sum += current->value, current->next;
    //不提倡此种风格
```

[5] 完全不用循环体，全部在循环条件中完成

```
for (current = list, sum = 0;
     (current = current->next) != NULL;
     sum += current->value);
    //效率最高. 但list不能为NULL表
```

[6] 逻辑混乱，也能正确计算：

```
current = list, sum = 0;
for(;current=current->next;sum += current->value)
    if (current == NULL) break;
    //最坏的风格，list也不能为NULL表
```

- $\text{iterate}(\langle \text{控制元素} \rangle) \langle \text{迭代域} \rangle =$   
 $\text{iterate}(\langle a \rangle; \langle b \rangle; \langle c \rangle) \langle \text{迭代域} \rangle$
- a: 初始/进入, b: 上界/跳出, c: 增量/控制
- 对迭代元素不加任何限制。语句, 表达式, 连续赋值, 空均可。先算 $\langle a \rangle$ , 再算 $\langle b \rangle$ , 若不为零, 则算 $\langle \text{迭代域} \rangle$ 然后执行 $\langle c \rangle$ 。若为‘零’出口。若不为‘零’再算 $\langle b \rangle$ , 若不为‘零’重复以上步骤, 若为‘零’则出循环。

# 隐式迭代控制

- 所谓隐式迭代是语言系统自动实施迭代。
- 这对第四代语言，声明式语言(函数式逻辑式)都非常重要，而且是语言发展方向上的重要机制。
  - 对聚集对象的迭代
  - 回溯

# 隐式迭代控制

## ● 对聚集对象的迭代

APL的隐式迭代

▽ ANSWER ← A FUNC B //声明定义两参数函数名为FUNC

[1] ANSWER ← (3×A) - B //函数定义

▽

X← 4 9 16 25 //X, Y束定为整数向量

Y← 1 2 3 4

Z← (3 4) ρ 1 2 3 4 1 4 9 16 1 8 27 64

//Z束定为3×4矩阵

# 一种类似dBASE的dBMAN查询的隐循环

设每个雇员一个文件， 当前记录类型域是：

lastname , sex, hours, grosspay, withheld

有以下dBMAN命令：

- [1]. display all for last name = 'Jones' and sex='M'
- [2]. copy to lowpay.dat all for grosspay < 10000
- [3]. sum grosspay, withheld to grosstot, whelddtot all for hours >0

# 回溯 Backtracking

## Prolog查询的回溯实现

设数据库中已列出以下事实：

f1. likes (Sara, John)	f9. does (Mike, skating)
f2. likes (Jana, Mike)	f10. does (Jane, swimming)
f3. likes (Mary, Dave)	f11. does (Sara, skating)
f4. likes (Beth, Sean)	f12. does (Dave, skating)
f5. likes (Mike, Jana)	f13. does (John, skating)
f6. likes (Dave, Mary)	f14. does (Sean, swimming)
f7. likes (John, Jana)	f15. does (Mary, skating)
f8. likes (Sean, Mary)	f16. does (Beth, swimming)

查询相爱且有共同运动爱好的两个人

```
?-likes (X,Y), likes(Y,X), does (X,Z), does(Y,Z)  
      (Prolog)
```

匹配	结果
[1]. 沿数据库匹配第一子目标( $f1 \leftarrow g1$ )	$X \leftarrow \text{Sara}, Y \leftarrow \text{John}$
[2]. 匹配g2, 沿 $f1 \rightarrow f7$ . Y对X不对	失败
[3]. 回溯重配g1, ( $f2 \leftarrow g1$ )	$X \leftarrow \text{Jana}, Y \leftarrow \text{Mike}$
[4]. 匹配g2, 沿 $f1 \rightarrow f5$ ( $f5 \leftarrow g2$ )	g2成功
[5]. 匹配g3, 沿 $f9 \rightarrow f10$ ( $f10 \leftarrow g3$ )	$X \leftarrow \text{Jana}, Z \leftarrow \text{swimming}$
[6]. 匹配g4, f9, Y对, X不对,	失败
[7]. 回溯第[5]步 $f11 \rightarrow f16$ 无新匹配	失败
回溯第[4]步 $f6 \rightarrow f8$ 无新匹配	失败
回溯第[3]步重配g1( $f3 \leftarrow g1$ )	$X \leftarrow \text{Mary}, Y \leftarrow \text{Dave}$
[8]. 重匹配g2, ( $f6 \leftarrow g2$ )	g2成功
[9]. 匹配g3 ( $f15 \leftarrow g3$ )	$X \leftarrow \text{Mary}, Z \leftarrow \text{skating}$
[10]. 匹配g4, 按 Z找Y( $f12 \leftarrow g4$ )	$Y \leftarrow \text{Mary}, Z \leftarrow \text{skating}$
[11]. 查询全部谓词满足( $f3 \leftarrow g1$ ) ( $f6 \leftarrow g2$ ) ( $f15 \leftarrow g3$ ) ( $f12 \leftarrow g4$ ) $X = \text{Mary}, Y = \text{Dave}, Z = \text{skating}$	
[12]. 从( $f4 \leftarrow g1$ )再查找有无另一组解重复第[7]-[11]步, 结果 无新解	



## 3.4.2 异常处理

程序无法执行下去，也就是出现了异常(exception)情况。

在早期语言的程序中，出现了这种情况就中断程序的执行，交由操作系统的运行程序处理。

现在向用户开放，Ada, C++, Java。

# 异常定义与处理段

定义用户定义： <异常名>: exception;

系统定义的异常（Ada）：

- CONSTRAINT\_ERROR 凡取值超出指定范围叫约束异常
- NUMBER\_ERROR 数值运算结果值实现已不能表达  
叫数值异常
- STORAGE\_ERROR 动态存储分配不够时，叫存储异常
- TASKING\_ERROR 任务通信期间发生的异常叫任务异常。
- PROGRAM\_ERROR 除上四种而外程序中发生的一切异常，  
都叫程序异常.如子程序未确立就调用等。

- 预定义的异常可以显式也可以隐式引发，而用户定义的异常必须显式引发
- 引发语句：  
    raise [<异常名>];
- 引发语句可出现在任何可执行语句所在的地方。一旦引发，则本程序块挂起转而执行本块异常处理段中同名的异常处理程序。

每个程序块都可以在该块末尾设立异常处理段:

declare:

<正常程序声明>;    [<异常声明>;

begin:

<正常程序代码>;    [<引发语句>;

exception:

when<异常名> => <处理代码>                    --异常处理段

when。。。                    --一般再次引发原异常（缺省名字）

[<引发语句>

end;

- Ada的嵌套异常及异常传播的

```
with TEXT_IO, MATH_FUNCTIONS;  
  use TEXT_IO, MATH_FUNCTIONS;  
procedure ONE_CIRCLE (DIAM:Float) is  
  RADIUS, CIRCUMFERENCE, AREA:Float;  
begin  
  CIRCUMFERENCE:=DIAM*22.0/7.0;  
  RADIUS:=DIAM/2.0;  
  AREA:=RADIUS*RADIUS*22.0/7.0;  
  PRINT_CIRCLE (DIAM, RADIUS, CIRCUMFERENCE, AREA);  
[7] exception  
  when NUMBER_ERROR =>  
    raise;  
end ONE_CIRCLE;
```

procedure CIRCLES is

    DIAMETER:Float;

    MAXIMUM: constant Float:=1.0e6;

[0]TOO\_BIG\_ERROR: exception;

    PUT ("please enter an float diameter.");

    begin

[1]loop

    begin

        GET (DIAMETER);

        if DIAMETER > MAXIMUM then

[2]        raise TOO\_BIG\_ERROR;

        else

[3]        exit when DIAMETER <=0;

        end if;

[8]    ONE\_CIRCLES(DIAMETER);

    PUT("Please enter another diameter(0 toquit):");

[4]  exception

    when TOO\_BIG\_ERROR=>

        PUT ("Diameter too large ! please reenter.");

[5]end ;

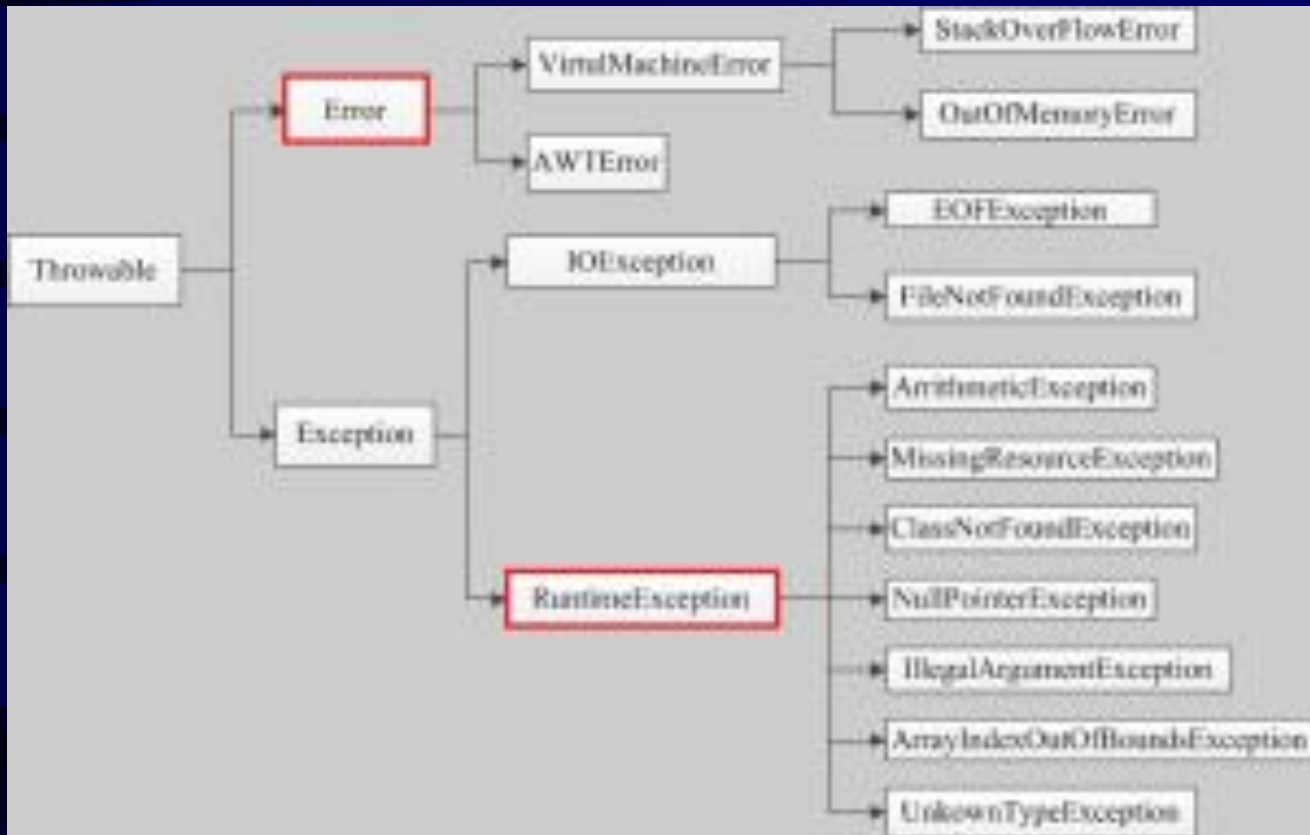
[6]end loop;

    PUT ("processing finished.");

    PUT New\_line;

end CIRCLE;

# Java异常类型



**checked exception (by compiler):**  
可控异常，要求必须在方法声明中列出来，否则无法通过编译。  
继承自Exception

**unchecked exception (by compiler):** 不可控异常，可以不在方法声明中列出。继承自RuntimeException

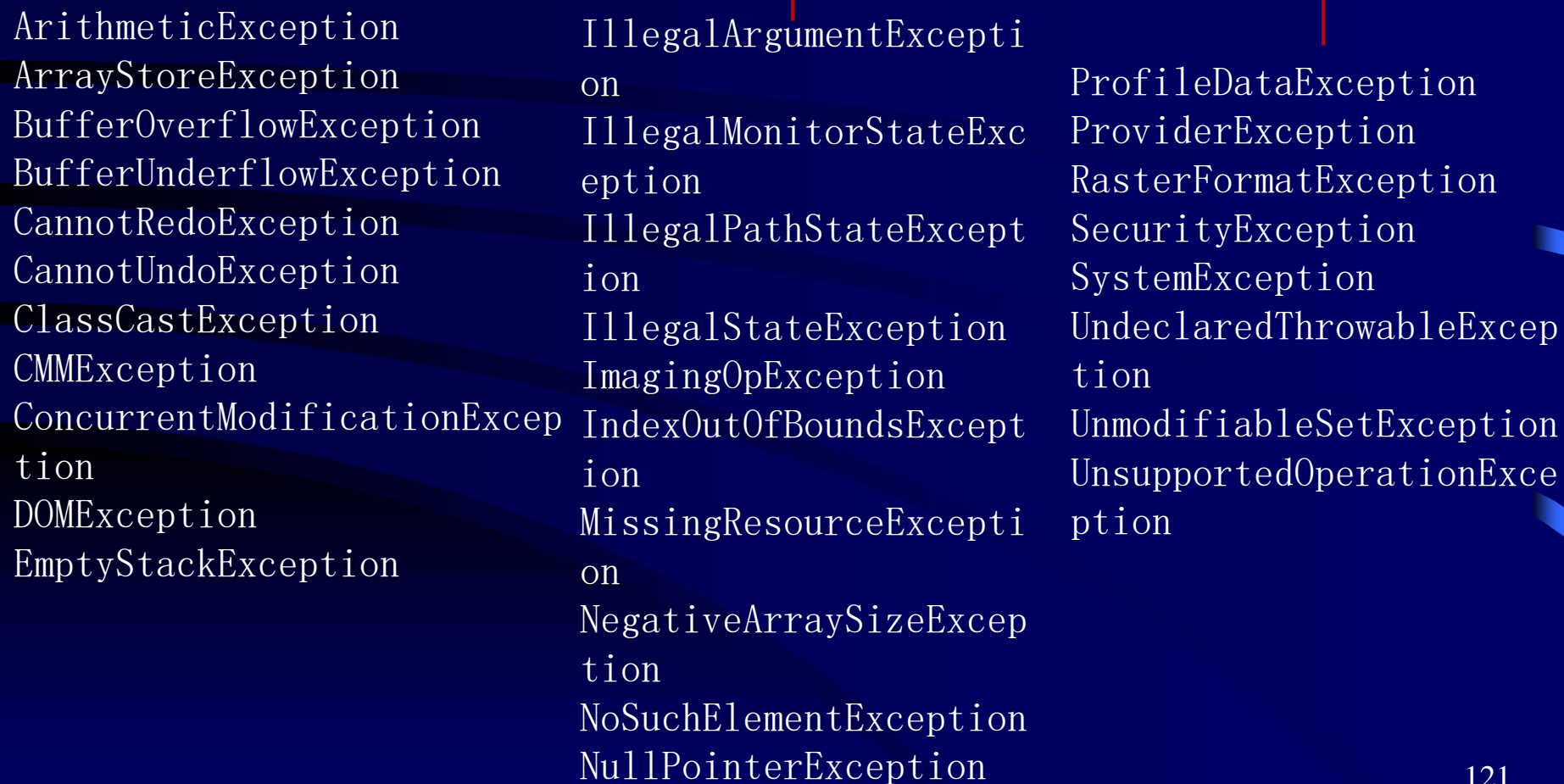
# 不可控异常类型

```
public class ExceptionTest {  
    public static void main(String[] args) {  
        int i = 10/0;  
    }  
}  
Exception in thread "main"  
java.lang.ArithmeticException: / by zero  
at ExceptionTest.main(ExceptionTest.java:5)  
  
public class ExceptionTest {  
    public static void main(String[] args) {  
        int arr[] = {'0','1','2'};  
        System.out.println(arr[4]);  
    }  
}  
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: 4  
at ExceptionTest.main(ExceptionTest.java:6)  
  
import java.util.ArrayList;  
public class ExceptionTest {  
    public static void main(String[] args) {  
        String string = null;  
        System.out.println(string.length());  
    }  
}  
Exception in thread "main"  
java.lang.NullPointerException  
at ExceptionTest.main(ExceptionTest.java:5)
```



# 不可控异常类型

## RuntimeException



# 可控异常类型

```
try {  
    String input = reader.readLine();  
    System.out.println("You typed : "+input);  
    // Exception prone area  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Exception



FileNotFoundException  
ParseException  
ClassNotFoundException  
CloneNotSupportedException  
InstantiationException  
InterruptedException  
NoSuchMethodException  
NoSuchFieldException

# 异常类型定义

- 选择扩展Exception或RuntimeException
- 只需定义构造函数

```
public class NewKindOfException extends Exception {  
  
    public NewKindOfException( ) { super( ); }  
    public NewKindOfException(String s) { super(s); }  
}
```

```
Exception e1=new NewKindOfException( “this is the reason” );  
String s = e1.toString();
```

→ “NewKindOfException: this is the reason”

# 异常的抛出与捕捉处理

- 如果一个方法m没有使用try...catch来捕捉和处理可能出现的异常，则会产生如下两种情况
  - 如果抛出的是不可控异常，则Java会自动把该异常扩散至m的调用者
  - 如果抛出的是可控异常，且在m的标题中列出了该异常或者该异常的某个父类异常，则Java自动把该异常扩散至m的调用者
- 由于不可控异常的产生在运行时才能确定，因此需要格外小心其捕捉与处理

```
try { x=y[n];}  
catch (IndexOutOfBoundsException e) {  
    //handle IndexOutOfBoundsException from the array access y[n]  
}  
  
i=Arrays.search(z, x);
```

# 关于异常的处理方式

- 反射

- 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后抛出**另一个异常**e2给其调用者
- “我”处理了一种意外情况，根据软件需求，这种情况也需要报告给”上层”

- 屏蔽

- 方法m被方法p调用，方法m在运行过程中抛出异常e1，方法p捕捉到e1，经过处理后不再抛出异常给其调用者
- “我”处理了一种意外情况，根据软件需求，没必要让”上层”知道是否发生了这种意外

# 关于异常的处理方式

- 对于在给定数组中搜索某个元素而言，考虑数组对象为 `null`，或者对数组访问越界两种意外情况
  - `NullPointerException` 需要通知调用者
    - Hey, 你给了一个不存在的数组！
  - `IndexOutOfBoundsException` 呢？
    - Hey, 你给的数组我没访问好？！

```
public static int min (int[ ] a) throws  
    NullPointerException, EmptyException {  
    /* @EFFECTS: if a is null throws  
    NullPointerException else if a is empty  
    throws EmptyException else returns the  
    minimum value of a  
    */  
    int m;  
    try { m = a[0]; }  
    catch (IndexOutOfBoundsException e)  
    {  
        throw new EmptyException  
            ("Arrays.min"); }  
    for (int i=1; i < a.length; i++)  
        if (a[i] < m) m = a[i];  
    return m;  
}
```

# 何时使用异常

- 当需要使用一个特殊返回值来告知调用者输入有异常情况时
  - 特殊返回值和异常相比，所表达的语义模糊，且容易被忽略
- 当一个方法期望可以在多处被重用时
  - 全局适用过程

# 使用可控异常还是不可控异常

- 如果期望调用者提供的输入数据不会引发异常，就应该使用不可控异常 //隐式处理
  - 不可控异常默认逐层“上报”，确保会有方法进行处理，否则会中断程序的运行
  - 不可控异常对于调用者要求较高，如果忘记捕捉，会带来潜在的程序崩溃风险
- 如果不对调用者做特殊要求，应该使用可控异常 //显式处理
  - 能够提高程序的健壮性



# 防御编程(Defensive Programming)

- 异常处理机制提供了一种在主流程处理之外的程序防护能力
  - 确保主流程逻辑的清晰性
  - 通过异常类型有效管理程序需要关注的各种意外情况
  - 反射和屏蔽机制为异常处理带来了灵活性
- 在设计类的方法时，需要问如下问题
  - 有哪些输入？
  - 输入会出现哪些“例外”情况？
  - 这些“例外”情况如何通知调用者？

## 3.5 函数和过程

- 命令式语言中子程序有两种形式：函数(必须返回值)也叫函数，过程(实施一组动作)也叫子例程subroutine。它们是程序的第一次分割，这种分割的好处：
  - 实施的功能单一，便于调试；
  - 相对独立，便于多人分工完成，且时间不受约束；
  - 相对封闭，人们易于控制，是分解复杂性的有力措施。
- 子程序和主程序联系的接口特别重要。在这个界面上要指出该例程的数据特征，即输入什么输出什么。而整个子程序体是完成从输入到输出的实现手段。
  - 界面指出“做什么？”，而子程序体回答“怎么做”。
  - 80年代程序完成第二次分割：将子程序定义(即界面)和子程序体显式的分开，成为相对独立的规格说明(Specification)和体(body)。

# 接口与界面

- 子程序和主程序联系的接口特别重要。在这个界面上要指出该例程的数据特征，即输入什么输出什么。而整个子程序体是完成从输入到输出的实现手段。
  - 界面指出“做什么？”，而子程序体回答“怎么做”。
  - 80年代程序完成第二次分割：将子程序定义(即界面)和子程序体显式的分开，成为相对独立的规格说明(Specification)和体(body)。

## 3.5.1 函数和过程抽象

- 函数抽象是用一个简单的名字抽象代表一个函数。
- 函数由函数型构 (Signature) 和函数体 (body) 组成。
- 函数计算的目的是求值。
- 函数体等同于一个复合的表达式。

函数抽象是对表达式的抽象

- 过程抽象是用一个简单的名字抽象代表一个计算过程。
- 过程由过程型构和过程体组成。
- 过程调用的目的是执行一组命令

过程抽象是对命令 (即语句) 集的抽象

# 函数的形式

- 函数由函数型构和函数体组成。形式是：

```
function FUNC (fp1, fp2, ...): returntype; //函数型构
      B; //函数体，可包括任何声明和语句
其中fp1, fp2, ...为形式参数，也叫形式变元(argument)
      returntype 为函数返回值的类型
```

函数引用是应用函数的唯一手段，它在同名的函数名之下给出实在参数(实在变元)：

```
FUNC (ap1, ap2, ...);
```

- 各种语言函数定义

- (a) FORTRAN

```
INTEGER FUNCTION FACT(N)
```

//前缀指明返回类型

```
INTEGER N, I, F
```

//参数类型在此声明

```
F = 1
```

```
DO 10 I = 2, N
```

```
    F = F*I
```

```
10 CONTINUE
```

```
FACT = F
```

//必须至少定义函数名一次

```
RETURN
```

//至少有一返回语句

```
END
```

- (b) Pascal

```
FUNCTION fact (n:Integer) :Integer;
```

//参数类型在变元表中定义,

```
BEGIN
```

//后缀指明返回类型

```
    fact := 1;
```

```
    IF n = 1 OR n = 0 THEN
```

```
        Return
```

```
    ELSE
```

```
        Fact := n*fact(n-1);
```

//也要定义函数名

```
END
```

(c) C

```
int fact (n) {  
    int n;  
    int i, f;  
    f = 1;  
    if(n>1)  
        for (i = 2; i<= n; i++)  
            f *= i;  
    return (f); }
```

//前缀指明返回类型  
//参数在体中声名类型  
//ANSI C改参数原型

//返回表达式f的结果值

(d) Ada

```
function FACT (N: POSITIVE) return POSITIVE is  
begin  
    if N = 1 then  
        return(1);  
    else  
        return (N*FACT(N-1));  
    endif;  
end FACT;
```

//参数类型在表中定义,  
//后缀指明返回类型

//返回表达式值

## • 引用或调用的形式

形—实参数表中元素个数，次序，类型应一致。早期语言都严格遵此准则。近代语言提供了较多的灵活表示法。

Ada引入缺省参数，实参个数可少于形参个数；指明参数结合不考虑次序；Ada引入参数模式in,out,inout指明只读，只写，读写参数。

C语言允许任意多个参数的调用。如内定义函数printf()调用时可以写任意个输出，只是第一参数中的格式个数与参数个数对应。

## • 过程定义与调用

过程子程序定义形式

```
procedure PROC (fp1,fp2,...) //过程型构  
    B; //子程序体包含局声明
```

对应的过程调用是：

```
PROC (ap1, ap2, ...);
```

C语言一切过程，包括主程序都是函数过程。它以void(无值)关键字代替函数类型指明符，实施子程序过程语义



- 无参过程

- 函数和过程的参数表均可为空。有的语言保留(), 有的只有一个名字。一般无参过程也要更新过程内部的值。函数过程还会返回不同的值。全局量在函数中有效。改变了全局量两次调用结果值当然不一样。这就是函数的副作用(side effect)。

- 有副作用的函数

- C 在很大程度上利用函数副作用, 例如, 当需要跳过空白时写:

- ```
while ((c = getch()) == "');
```

- 无参过程

-- Ada中常用的随机数:

```
function RANDOM return FLOAT range 0.0...1.0;
```

引用时, 若FIELD已声明为常量:

```
RESULT := RANDOM * FIELD;
```

RANDOM若无副作用RESULT值不可能改变。

不确定参数的函数或过程

## 3.5.2 参数机制

语言中第一类对象均可作函数/过程参数。  
由于变量的时空特性，传递的形-实参数可以有  
许多不同的实现结合的办法，即参数机制。

### 传值调用 (call-by-value)

- [1]实参表达式先求值
- [2]将值复制给对应的形参。形参和实参有同样大小的存储
- [3]过程运行后一般不再将改变了的形参值复制回实参

# Pascal中的传值调用

```
PROCEDURE test1(J2,A2:Integer; P2:list)
BEGIN
    Writeln(J2, A2, P2↑.value);
    J2 := J2 + 1;
    P2 := P2↑.next;
    Writeln(J2, A2, P2↑.value)
END.
```

调用程序有：

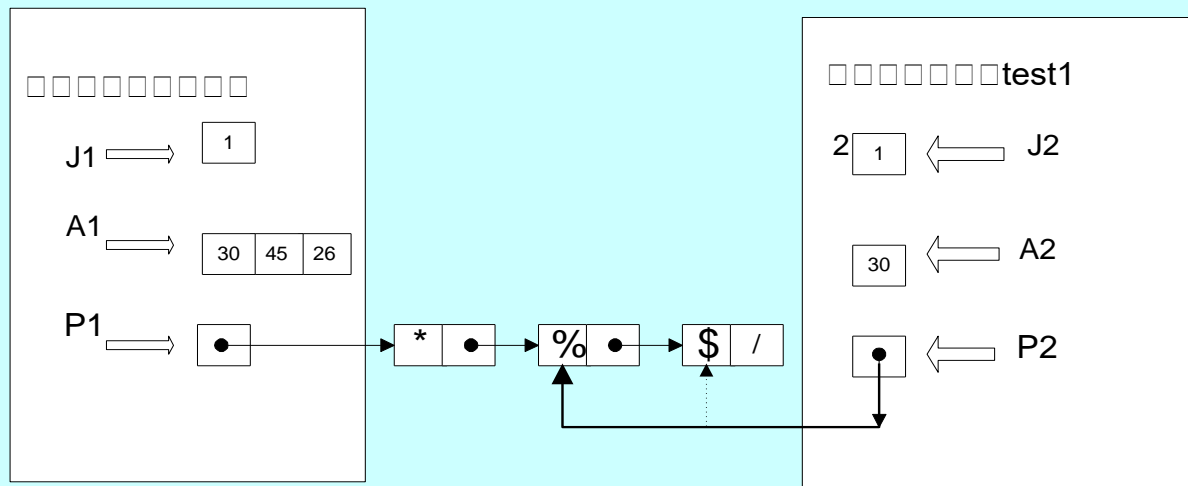
```
test1(J1, A1[J1], P1↑.next);
```

第一次打印为：

1 30 %

第二次打印为：

2 30 \$



→ 指针  
⇒ 束定

传值调用图示

# 传名调用(call-by-name)

传名在过程/函数中加工的就是实参已分配的值，因此不需付出双倍存储代价。但传名过程的虚实结合是将程序体中所有形参出现的地方均以实参变元名置换。这样出现几次算几次效率是低的。

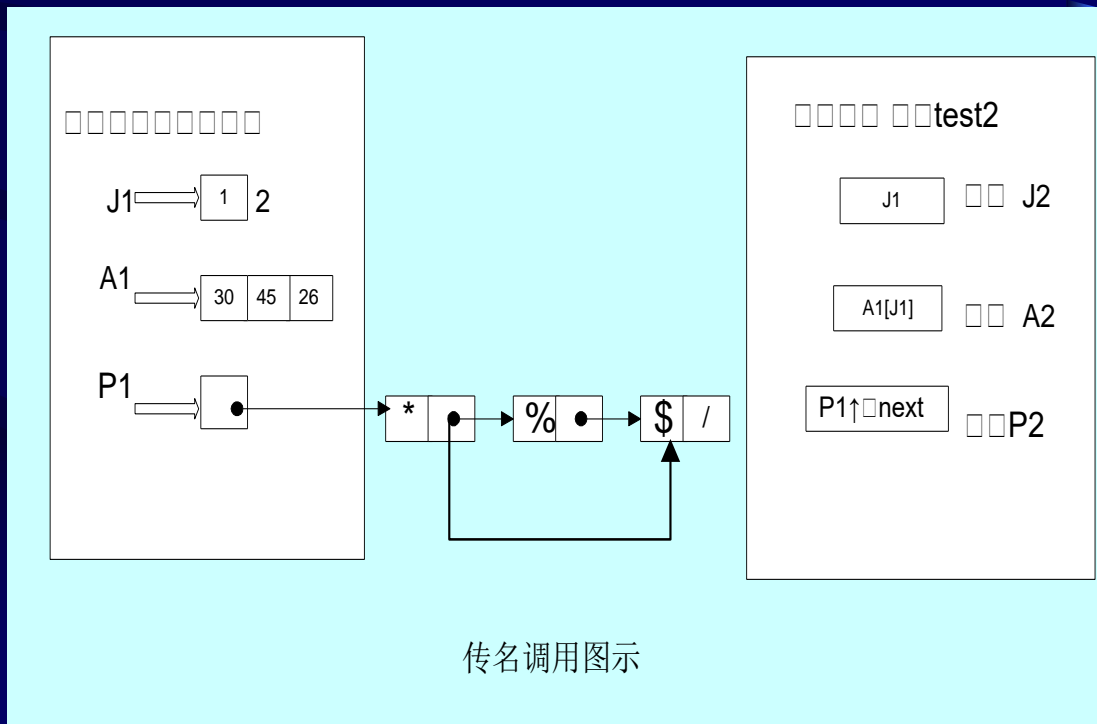
传名调用程序示例

由于Pascal无传名机制，此处作一点扩充：

```
PROCEDURE test2(NAME J2,A2:Integer;NAME P2:List);  
BEGIN  
  Writeln(J2, A2, P2↑.value);  
  J2 := J2 + 1;  
  P2 := P2↑.next;  
  Writeln(J2, A2, P2↑.value)  
END
```

执行同样调用：

test2(J1, A1[J1], P1↑.next);



```

PROCEDURE test1(J2, A2:Integer;P2:list)
BEGIN
    Writeln(J2, A2, P2 ↑ .value);
    J2 := J2 + 1;
    P2 := P2 ↑ .next;
    Writeln(J2, A2, P2 ↑ .value)
END.

```

```

PROCEDURE test2(NAME J2, A2:Integer;NAME P2:List);

```

函数体同test1

执行同样调用:

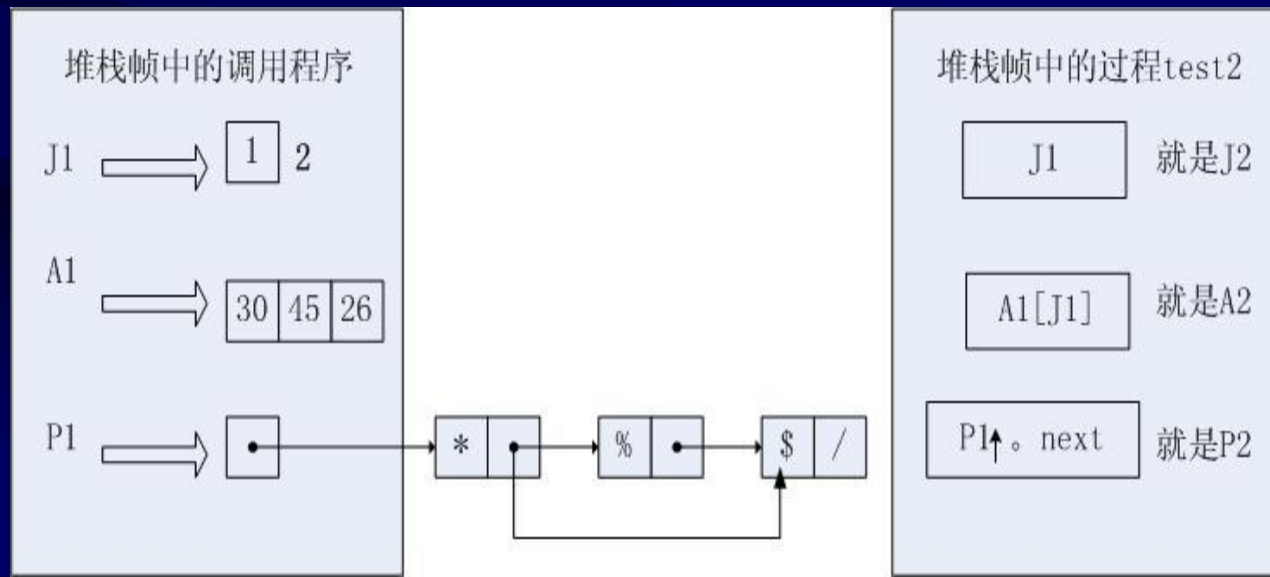
```
test2(J1, A1[J1], P1 ↑ .next);
```

名结合后打印:

1      30      %

执行后结果是:

2      45      \$



# 引用调用(call\_by\_reference)

引用参数实现时，编译器在子程序堆栈中设许多中间指针，将形参名束定于此指针，而该指针的值是实参变量的地址(在主程序堆栈框架内)，在子程序中每次对形参变量的存取都是自动地递引用到实参存储对象上。

引用调用的Pascal示例：

```
PROCEDURE test3(VAR J2,A2:Integer;VAR P2:List);
```

```
BEGIN
```

```
  Writeln(J2, A2, P2↑.value);
```

```
  J2 := J2 + 1;
```

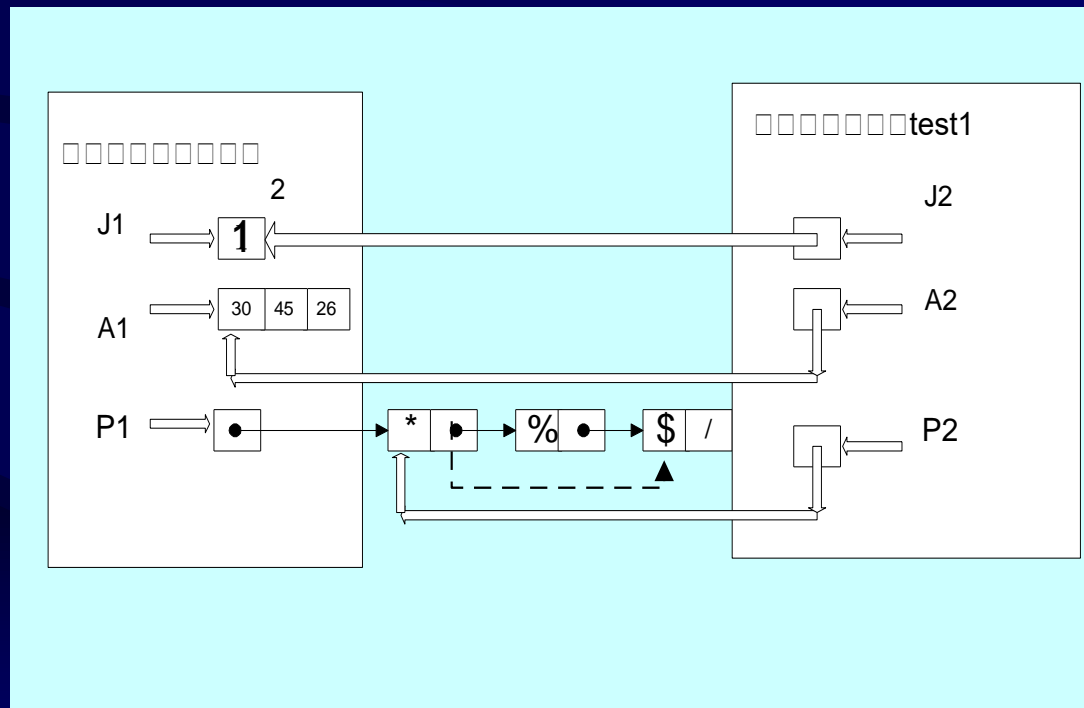
```
  P2 := P2↑.next;
```

```
  Writeln(J2, A2, P2↑.value)
```

```
END
```

相应的调用程序是：

```
test3(J1, A1[J1], P1↑.next);
```



引用调用图示

引用调用的Pascal示例:

```
PROCEDURE test3(VAR J2, A2:Integer;VAR P2:List);
```

函数体同test1

相应的调用程序是:

```
test3(J1, A1[J1], P1 ↑ .next);
```

```
PROCEDURE test1(J2, A2:Integer;P2:list)
```

```
BEGIN
```

```
WriteIn(J2, A2, P2 ↑ .value);
```

```
J2 := J2 + 1;
```

```
P2 := P2 ↑ .next;
```

```
WriteIn(J2, A2, P2 ↑ .value)
```

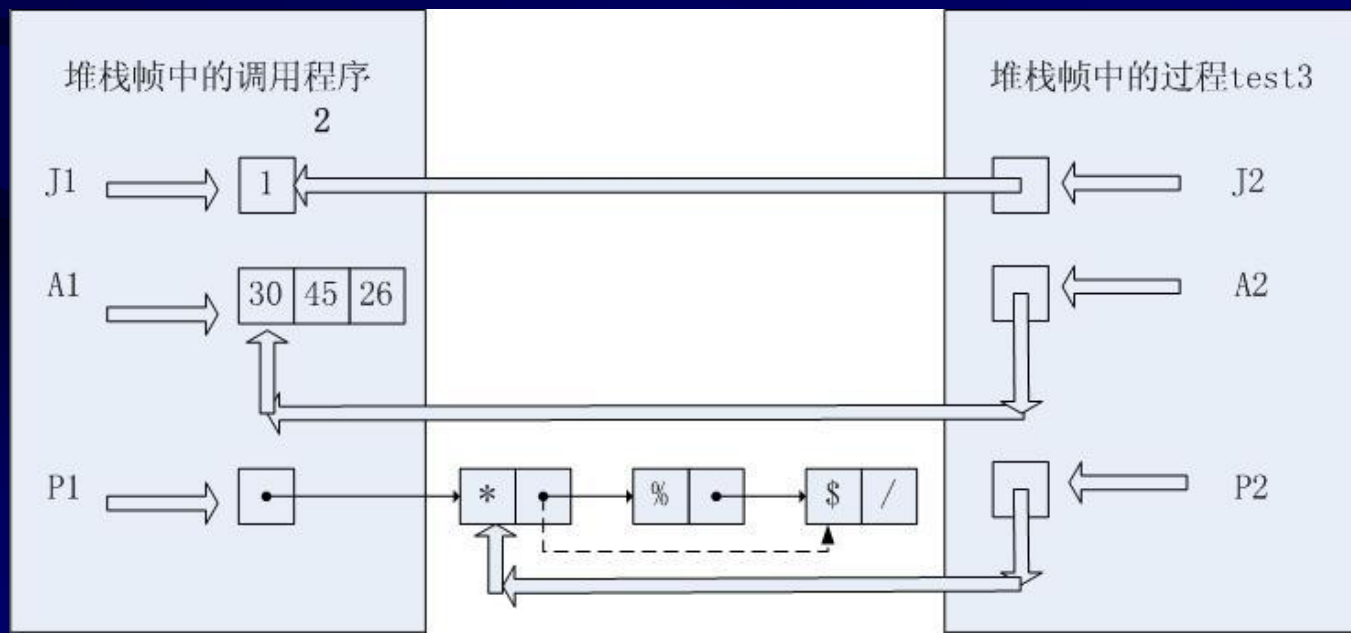
```
END.
```

第一次打印是:

1 30 %

第二次打印是:

2 30 \$





# 参数模式与返回调用(call-by-return)

显式指明参数传递模式,可以为编译实现提供信息

```
fun_name(x,y:Real; VAR s,q:Integer)...
```

x, y传值实现, 它只读。s, q引用实现, 可读/写

Ada只规定参数模式in, out, inout, 传递方向的模式(mode)。

由编译选择实现方式:

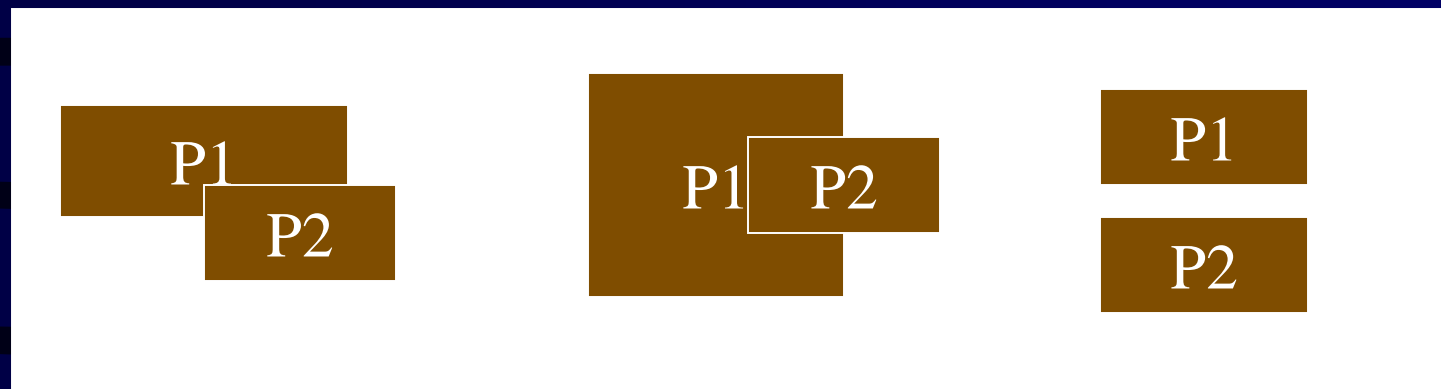
```
proc_name (X, Y: in Real; S: inout Integer; Q: outInteger)...
```

in模式可不写出(缺省)。函数只能有in的模式, 过程都有, 且出现次序不受限制。x, y因在子程序中只读,传值实现可保证不受破坏。s读/写用引用实现, 而q是只写参数, 传值和引用都不能保证”只”写

实现返回调用机制有两种办法:其一是复制。另一种办法是引用实现增加”只写”保护

# 值-返回调用（call-by-value-and-return）

是对by-reference的改进，因多进程竞争数据资源时多重引用(束定)易于引起混乱



返回值由P2定

返回值由P1定

正常顺序执行

对于并发多任务宜只读——只写

值与返回调用机制是把值调用和返回调用组合起来，实现调用程序双向通道，这对于有多个存储器的多处理器系统和网络分布式系统值调用极度安全。

在子程序执行期间因不是束定，形参变量的值不会中途改变，复制回去和拷贝进来处可设断点检查

# 指针参数 (call\_by\_point)

- 指针作为参数其实现方式一般是复制机制，它复制的是地址(指针内容)
- 注意和引用调用之同异

例：指针Pascal引用版： 交换两变量的内容

```
PROCEDURE swap1( VAR a,b:Integer);
```

```
  VAR t:Integer;
```

```
  BEGIN
```

```
    T := a; a := b; b := t
```

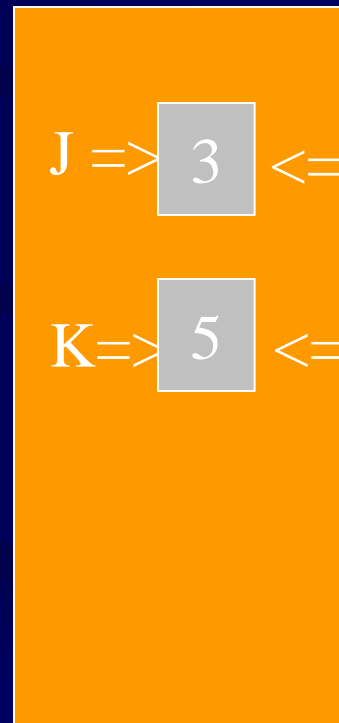
```
  END;
```

调用程序片断：

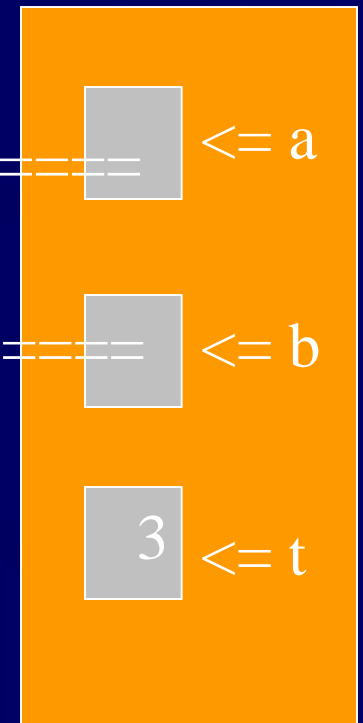
```
  j = 3; k = 5;
```

```
  swap1(j,k); //结果j = 5, k = 3
```

Caller-frame



Swap1-frame



指针版：变换两变量的内容

```
TYPE int_ptr = ↑Integer;
```

```
VAR jp, kp:int_ptr;
```

```
PROCEDURE swap2 (a,b:int_ptr);
```

```
VAR t:Integer;
```

```
BEGIN
```

```
  t := a↑; a↑ := b↑;
```

```
  b↑ := t
```

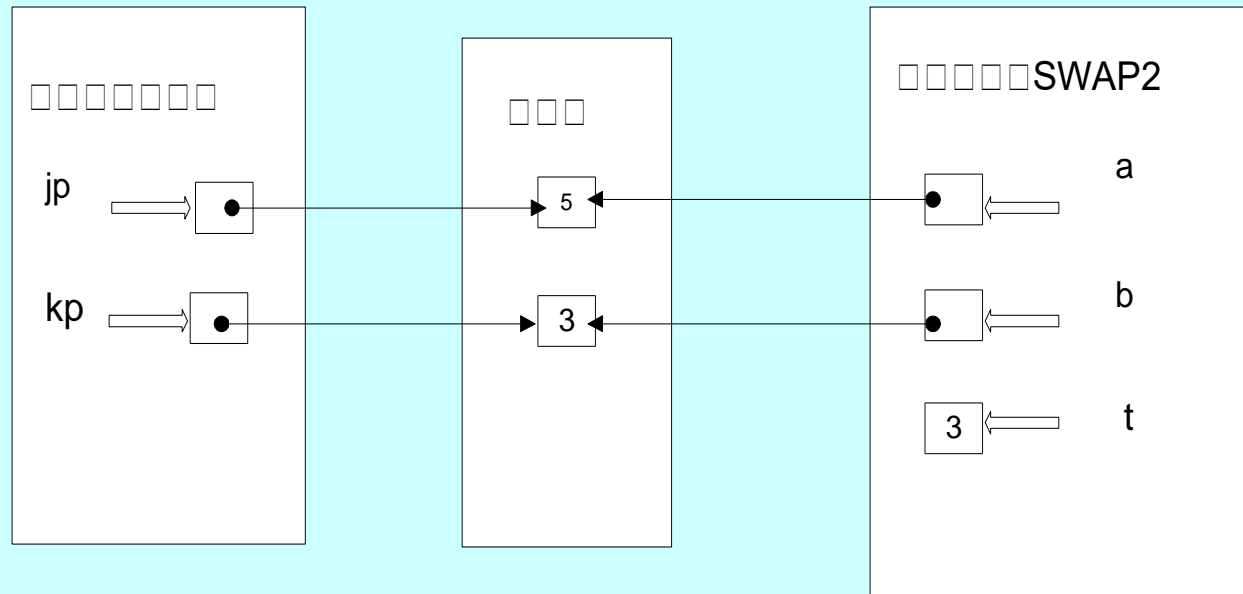
```
END;
```

相应调用程序片断：

```
NEW (jp); jp↑= 3;
```

```
NEW (kp); kp↑ := 5;
```

```
Swap2(jp,kp);
```



指针调用图示

## C语言的指针参数传递

```
void swap3(int *a, int*b)
```

```
{ int t;
```

```
  t = *a; *a = *b; *b = t;
```

```
}
```

形参是两指针， 实参不用指针的版本：

```
main()
```

```
{
```

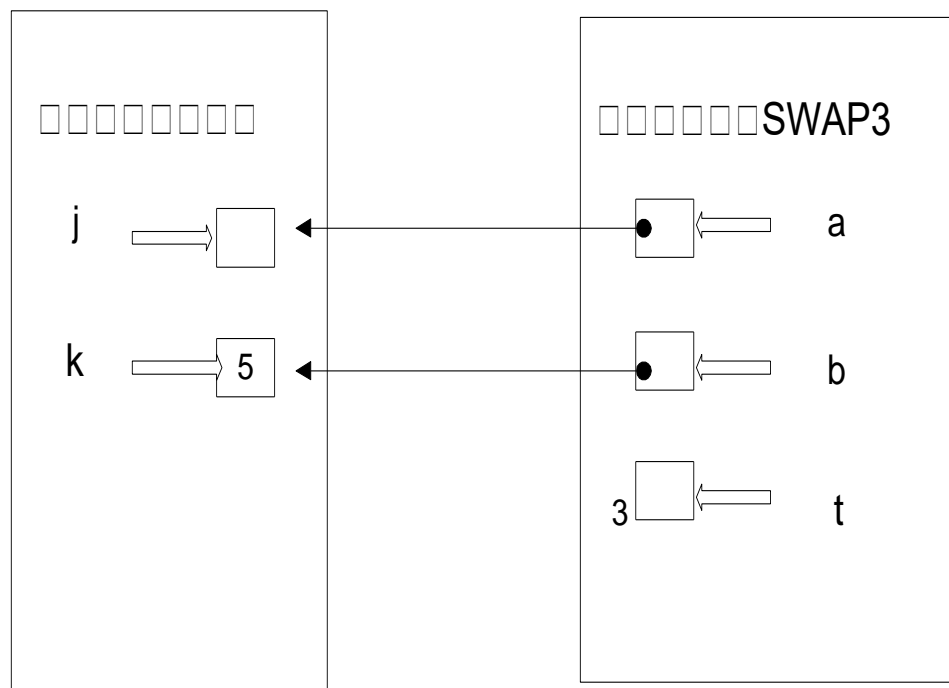
```
  int j = 3; k = 5;
```

```
  //声明并初始化两整数
```

```
  swap3(&j,&k);
```

```
  //类型匹配吗?
```

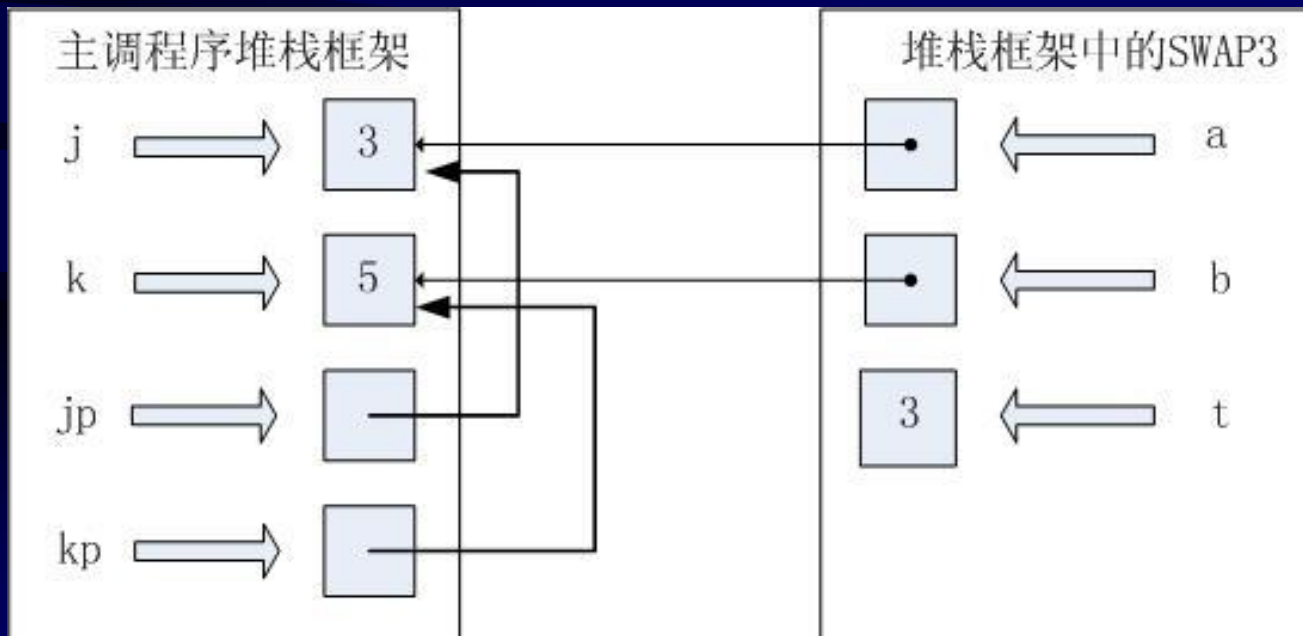
```
}
```



C语言的指针--引用调用

# 实参是指针的版本:

```
main()  
{  
    int j = 3, k = 5;  
    int *jp = &j, *kp = &k;  
    swap3(jp, kp);  
}
```



# 高阶函数

- 以函数或过程 作为实参变元或返回值的 函数或过程，我们统称高阶函数
- 函数作为变元LISP有映射函数(mapping function)它把单目、双目运算扩充到多个数据对象的数组或表上。映射函数本身以简单运算函数和表(或数组)作实参变元

# 函数作为变元

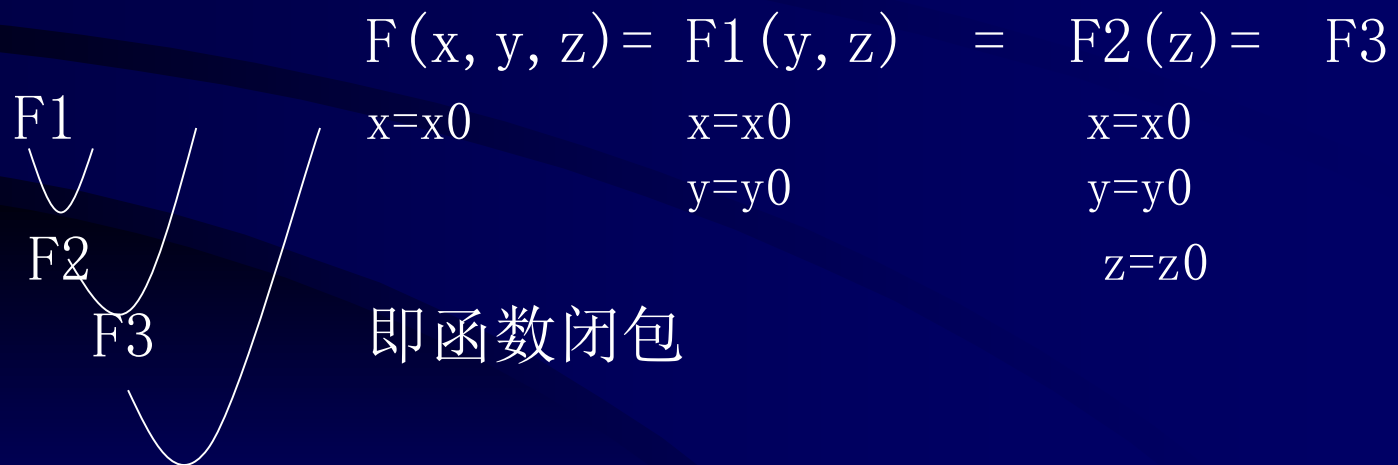
- 函数作为变元的需求主要是为通用化程序单元而提出的。
  - FORTRAN最早就直接允许在参数表中写虚、实函数名。只要在主调程序写上该实函数是外部的(`EXTERNAL FA, FB`), 连编程序就可以找出FA, FB, 并在每个虚函数名F处以FA或FB置换。
  - 是程序库, 程序包程序主要的形式。
- 第二种用途是作映射函数(mapping function)。它把单目、双目运算扩充到多个数据对象的数组或表上。映射函数本身以简单运算函数和表(或数组)作实参变元。



# • 函数作为返回值

- C语言、C++其函数返回值可以是指向函数的指针。但C和C++均不能在函数中创建一个函数并把它作为返回值返回。

- 函数式语言作用于任何一变元返回值是新函数  
`fun F(x) (y) (z) = <函数值>`



- 闭包(closure)是可用到表达式上的操作。闭包最有用和最容易理解的应用是部分参数化。例如，有n个变元的函数，我们将其中一个变元束定于局部定义的值上就得到一个n-1个变元的新函数。
- ML闭包的应用

```
fun powerC(n) (b)= if n=0 then 1.0  
                  else b * powerC(n-1) (b)
```

隐式返回函数

可以显式给出

```
val  sqr=powerC 2  and  cube=powerC 3  
sqr (b) ≡          cube (b) ≡ b 3
```

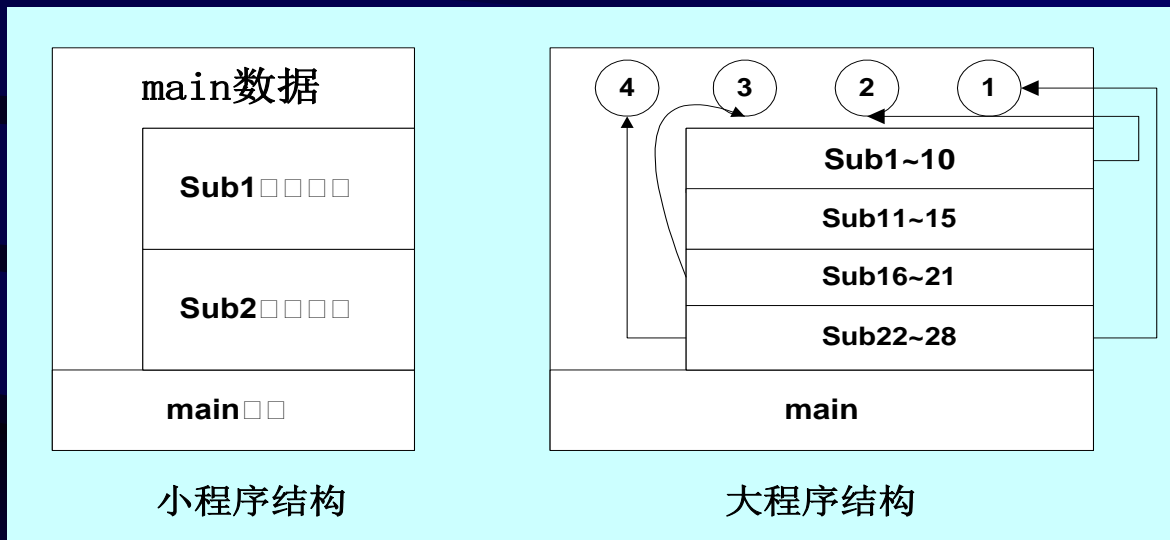
显示的新函数

闭包closure产生一系列函数如：

f(a) (b) (c), fa(b) (c), fb(a) (c), fc(b) (a), fab(c)  
, fbc(a), fca(b), fabc

## 3.6 抽象与封装

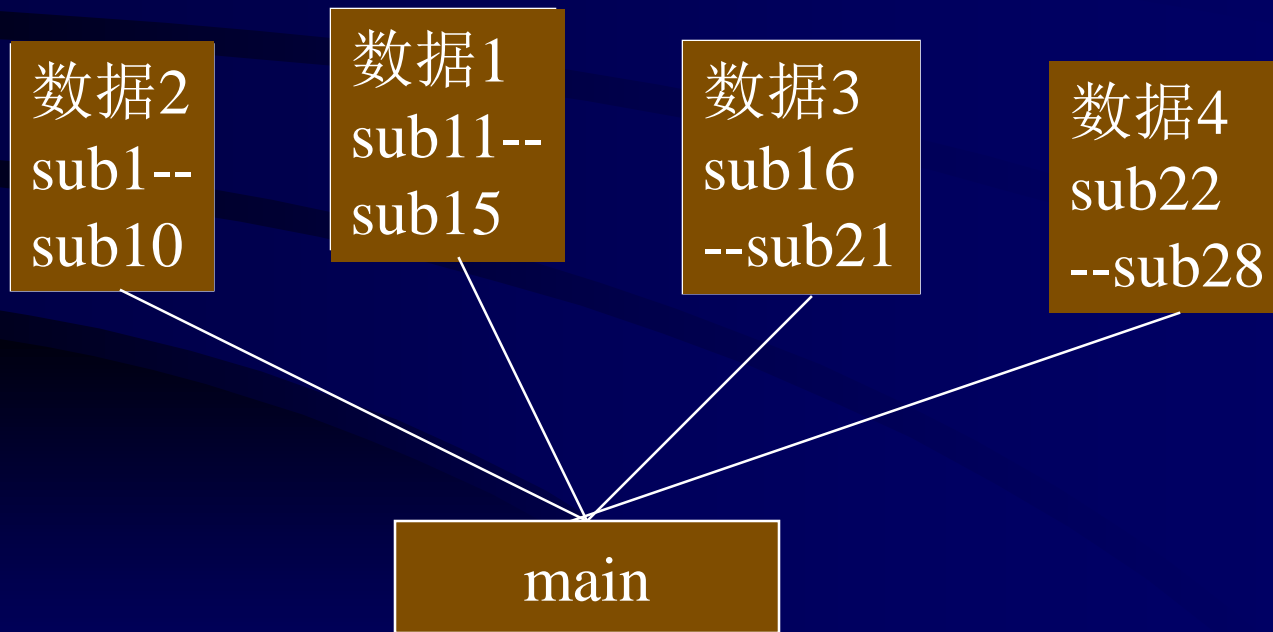
- 函数和过程是封装的程序实体，它有数据和操作，规格说明（型构）和过程体，便于人们控制复杂性
- Pascal统一的嵌套结构不造于大型程序



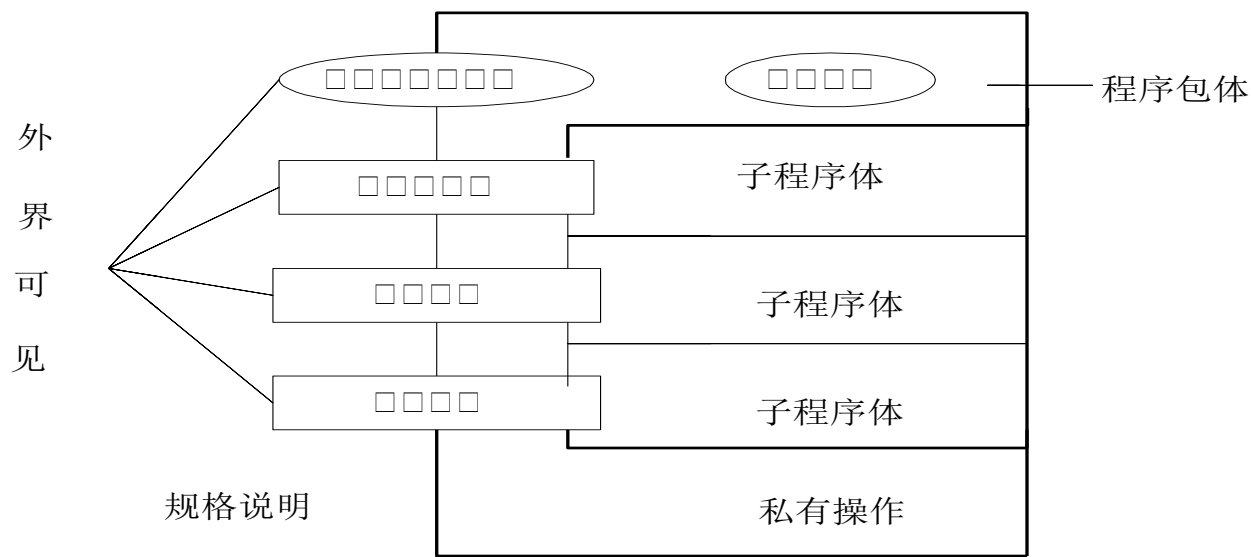
sortInt(list, listLen)

# 续

- 将相关的数据和操作封装成大模块（若干类型，若干过程/函数）结构上形成包package或模块 Modula
- 包是可分别编译。随时连接软件资源，是解决复杂系统的有力手段
- 包的规格说明和包体显式分开。语义上正好是“做什么”，“怎么做”



## 3.6.1 模块和包



- 规格说明和体在表示结构上的分离。有利于修改，维护
- 封装实现数据隐藏，有利于安全
- 规格说明是程序包的抽象，有利于复杂系统简化

- 模块（包）封装数据与操作，它有可控界面，外界不能操纵私有数据引出公有（**public**包的使用者可见）、私有（**private**本包所有操作可访问，包外不可见）、保护（**protected**,包外不可见，但本包的子包可见）概念
- 包只是以封装手段,可有/可没有逻辑语义
  - 只有数据无操作，数据块**BLOCK DATA**（**FORTRAN**）
  - 只有操作无共享数据如函数包，数学库
  - 有数据有操作，一口对外可模拟自动机
  - 有数据有操作，模拟客观世界对象—增加程序表达能力
  - 封装的包可实现复杂的数据类型**ADT**

## Ada 的复数程序包

```
package COMPLEX is
type NUMBER is record
    REAL_PART:FLOAT;
    IMAG_PART:FLOAT;
end record;
function "+"(A, B: in NUMBER) return NUMBER;
function "-"(A, B: in NUMBER) return NUMBER;
function "*" (A, B: in NUMBER) return NUMBER;
end COMPLEX;
package body COMPLEX is
    <声明私有数据/操作>
    <实现规格说明中定义的每一个过程/函数>
    <实现本包私有过程/函数>
endCOMPLEX;
```

有了这个程序包我们可以编出复数应用程序：

```
with COMPLEX;  
use COMPLEX;  
procedure MAIN is  
  COMP_1: NUMBER := (1.0, 2.0); --1+2i  
  COMP_2: NUMBER :=(3.0, 4.0);  -- 3+4i  
  W, X, Z: NUMBER;  
begin  
  W := COMP_1 + COMP_2;          --W = 4+6i  
  X := COMP_2 - COMP_1;          --X = 2+6i  
  Z := COMP_1 * COMP_2;          --Z = -5+10i  
end MAIN;
```



## 3.6.2 抽象数据类型

### •数据抽象

数据抽象是抽象数据类型的方法学

定义一组数据集V，以及其上的操作集Op，构成  
ADT (Abstract Data Type)

$T = (V, Op)$

- 什么和怎么做分开(规格说明和体)
- 实现数据隐藏（体中声明的数据和操作外界不可见）
- 分别开发分别编译（可做大程序）
- 简化复杂性便于调试（利用抽象实现分治）
- 构造新类型，计算直观方便（面向对象的基础）

# 抽象数据类型的作用

- [1] 数据抽象将数据类型的使用和实现分开，先决定数据是如何使用的（外部性态）而不是它的实现内部构造。提高了程序开发的抽象层次，较易控制大型系统的全局。
- [2] 数据抽象简化了程序正确性问题。即如果程序不正确，改起来方便。只要规格说明不变，只改变错误程序的体，不影响使用该资源的应用程序。增加了软件的可维护性。
- [3] 利用抽象数据类型可以准确地构造新类型。
- [4] 数据抽象能实现数据隐藏，即非规格说明界面上的数据，外部用户是不可见的。增加软件安全性,可靠性。
- [5] 作为软件设计技术，模块性利于软件开发，应用程序和模块实现的开发可独立进行。

# 用户自定义类型的抽象数据类型

- 应提供和语言定义的类型相同的特性，包括：
  - 类型定义，允许程序单元声明此类型的变量，而隐藏该类型对象的表示形式；
  - 在该类型对象上的操作集合。
- 抽象数据类型满足下面两个条件：
  - 此类型对象的表示方式对于使用它的程序单元来说是隐藏的，因此只有在类型定义中提供的操作才能直接操作这些对象。
  - 把类型的声明和在该类型上的操作协议包含在一个语法单元中。类型接口不依赖对象的表述方式或操作的实现方式，另外，其它程序单元可以创建所定义类型的变量。

## •构造新类型

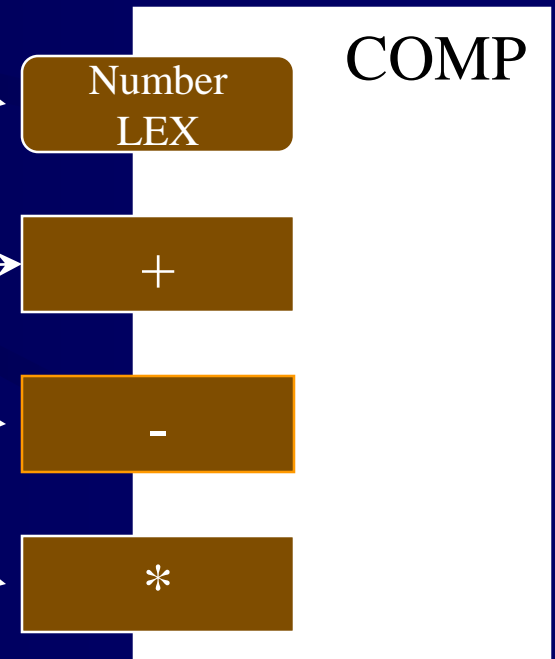
Switch COMPLEY use COMPLEY;  
Procedure MAIN is

C1:NUMBER is =(1.0,2.0);  
C2:NUMBER is =(3.0,4.0);  
W, X, Z: NUMBER;

begin

W:=C1+C2;  
X :=C2-C1;  
Z :=C1\*C2;

end MAIN



## • 构造函数和析构函数

变量—类型  $V : \text{integer}$ ; 以类型指明程序对象

变量—抽象数据类型, 声明时要指明如何构造

$C : \text{NUMBER} := (1.0, 2.0)$  较简单可用赋初值办法, 复杂在运行中由构造函数 (constructor) 完成

$C = \text{ADT\_Name}(\text{<构造参数>}) \{ \text{<构造操作即函数体>} \}$

构造函数 (constructor) 和 ADT 同名

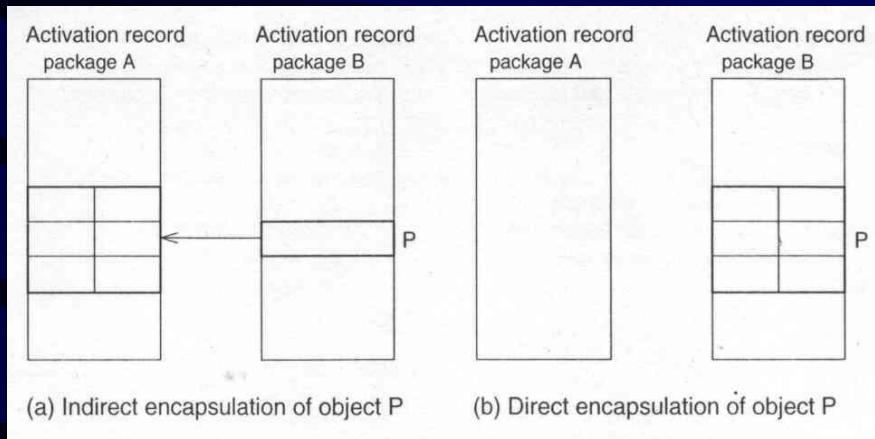
不再使用的程序对象, 用析构函数 (destructor) 显式删除  
一般形式是:

$\text{ADT\_Name}() \{ \dots \}$  和 ADT 同名

$\sim \text{ADT\_Name}() \{ \dots \}$

# 抽象数据类型：实现

- 实现封装数据对象的两个模型。

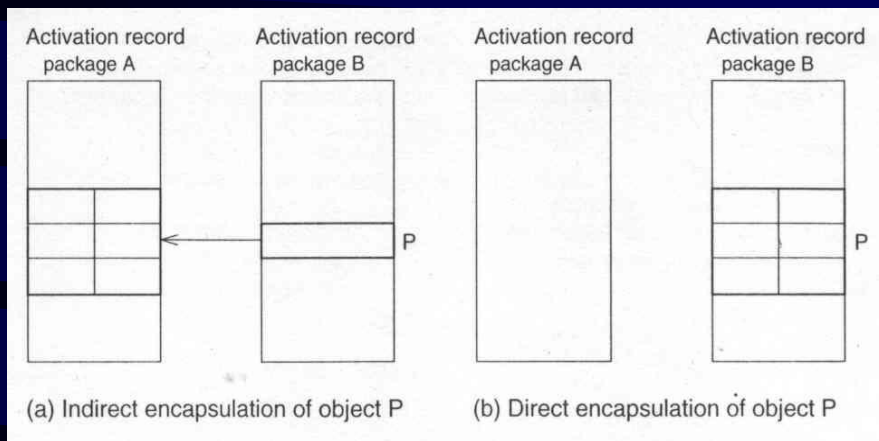


在间接封装中，ADT的实现独立于其使用，A的内部修改不影响B。但运行时间开销大。

- 间接封装（图（a））
  - 抽象数据类型的结构包A中不仅有对象P的定义，对象P的实际存储在A的激活记录中维护。包B中声明和使用对象P，运行时激活记录必须包含一个到实际数据存储的指针。
- 直接封装（图（b））

# 抽象数据类型：实现问题

- 实现封装数据对象的两个模型。



直接封装中：和上面情形相反，对P的访问会省时间，但如抽象对象的表示改变，所有它的使用的实例需重编译。时间花销在编译过程中。

- 间接封装（图（a））
- 直接封装（图（b））
  - 对象P的实际存储在B的激活记录中维护。

# 抽象数据类型：实现问题

- Ada使用？
  - 直接封装模型。因此翻译抽象数据对象的使用将需要对象表示的详细细节，即需知道包规约中的私有部分。
- 直接封装和间接封装可以用在支持封装的任何程序中，而不管其在语言中实现的封装模型是什么。



# • C语言以文件实现抽象数据类型

## C语言4种文件

--头文件 “modules.h” 定义宏和类型声明

--主模块 “name.c” 给出新类型的数据和操作定义

--其它模块 “other.c” 实现头文件中声明

--通过makefile指明各文件关系

make SOMETYPE

SOMETYPE: name.o other.o //连续两模块的目标码  
文件取名SOMETYPE

CC\_o SOMETYPE name.o other.o //编译并连成为  
SOMETYPE的目标文件

name.o : name.c modules.h //两源文件连成为name.o

CC\_o name.o name.c //源文件编译后形成目标文件

other.o : other.c modules.h

CC\_o other.o other.c //SOMETYPE 如预定义的了

### 3.6.3 类属

函数是表达式集的抽象、过程是命令集的抽象，类属(generic)是声明集的抽象。

即声明的变量、类型、子程序都是参数化的。

Ada类属子程序：

```
generic
  type ELEMENT is private; --类属类型参数
procedure EXCHANGE (FIRST, SECOND: in out ELEMENT);
  TEMP: ELEMENT;           --程序中用类属形参
begin
  TEMP := FIRST;
  FIRST := SECOND;
  SECOND := FIRST;
end EXCHANGE;
```

关键字procedure换成package即为类属包

参数化类型与实际类型结合。由类属设例指明：

```
procedure INT_EXCHANGE is new EXCHANGE (Integer);
```

实在的Integer与ELEMENT匹配，即在procedure中任何出现ELEMENT的地方以Integer代，等于有了整数的 EXCHANGE 如同模板，可以写出多个过程：

```
procedure CHAR_EXCHANGE is new EXCHANGE ( Character);  
procedure SMALLINT_EXCHANGE is new EXCHANGE  
(MY_INT);
```

# 类属定义一般形式

generic

<类属值> | <类属变量> | <类属类型> | <类属子程序>

package GENERIC\_PKG is --用类属参数的包

.

.

.

end GENERIC\_PKG;

类属实例化是在有了实在参数以后作实例声明：

package INS\_PKG is new GENERIC\_PKG(<实参变元>);

类属形实参数的个数、次序要匹配。

## • 类属参数

原则上声明中的所有程序对象都可以参数化，其实现机制如同子程序中参数结合，类属值一般是复制机制，类属对象(变量)、类型、子程序用引用机制。取决于实现。

类属声明是 `in` 模式变量 可换值

`inout` 模式变量 可换变量

类属类型 `private` 私有 可以和任何类型匹配

`<>` 枚举 可以和任何枚举类型匹配

`range <>` 整 可以和任何整型匹配

`delta <>` 定点 可以和任何定点类型匹配

`digits <>` 浮点 可以和任何浮点类型匹配

类属子程序 `with procedure(<参数化类型>)`

可以和任何过程匹配，同时类型参数例化

## · Ada的类属程序包的例子

程序包中封装一分类和归并程序。被分数据按数组类型设计，它的元素类型，按升序还是降序，数组下标，都参数化：

```
generic
```

```
type ITEM is private          --数组元素类型参数化
```

```
type SEQUENCE is array (Integer range < > ) of ITEM;
```

```
    --数组类型参数化。ITEM马上就用了
```

```
with function PRECEDES (x, y:ITEM ) return Boolean;
```

```
    --参数化操作未定升降序
```

```
package SORTING is
```

```
    procedure SORT (S: in SEQUENCE);
```

```
    procedure MERGE (S1, S2: in SEQUENCE;
```

```
        S:      out SEQUENCE);
```

```
end SORTING;
```

有两个实在函数：

```
function "<=" (X, Y: Float) return Boolean;
```

```
function ">=" (X, Y: Float) return Boolean;
```

那么就可以作以下设例声明：

```
type FLPAT_SEQUENCE is array (Integer range < >) of Float;
```

```
package ASCENDING is new SORTING(Float, FLPAT_SEQUENCE,  
"<=");
```

```
package DESCENDING is new SORTING(Float, FLPAT_SEQUENCE,  
">=");
```

这样， 就有了一个升序， 一个降序的两个程序包。

## • 类属动态实现问题

类属程序是抽象程序，可以编译，但不可执行，每次执行的是实例程序。其编译执行如同C语言之宏替换，也就是参数静态束定于实参，强类型语言可以做到。

类属程序只有一个，运行时动态束定就不需要显式设例。因此，动态设例的程序表达能力更强，程序易于扩充。



# 类属抽象数据类型

- 类属抽象类型定义允许类型的一个属性被分离地规约，从而给出一个基类型定义。使用该属性为参数，进而可从同一个基类型导出几个特殊类型。
- 它们的结构和基类型类似，但参数可影响抽象类型定义中操作的定义以及类型本身的定义。参数可以是类型名或值。

# 类属抽象类型定义的实例化

- 一个类属包定义表示了一个模板，可用于创建特殊的抽象数据类型。这个创建过程称为实例化，通过一组参数的代入。

- 例：前图类属栈类型定义的实例化：

```
package IntStackType is
```

```
    new AnyStackType(elem=> integer);
```

```
package SetStackType is
```

```
    new AnyStackType(elem=> Section);
```

- 不同大小的整数栈的声明：

```
    Stk1: IntStackType.Stack(100);
```

```
    NewStk: IntStackType.Stack(20);
```

# 类属抽象类型定义的实例化

- 类属类型AnyStackType可以用不同的参数值多次实例化，每次实例化均产生包中类型名Stack的另一个定义。这样当栈在声明中被引用时，有可能是含混的。
  - Ada中需要将包名放在类型名前作前缀，如：  
IntStackType.stack或SetStackType.stack
  - 在C++中，用模板来定义类属类：  
template <class **type\_name**> class classname  
class\_definition

# 类属抽象数据类型：实现

- 类属抽象数据类型通常有直接的实现。实例化时，必须给出参数，编译器使用类属定义为模板，插入参数值，然后编译该定义。
- 在程序执行过程中，只有数据对象和子程序出现。包定义仅仅作作为限制数据对象和子程序可见性的设备，包本身并不出现在运行时。
- 如果类属类型定义被多次实例化，则该直接实现可能过于低效，需要考虑避免产生过多的子程序拷贝及重复编译。

# 第三章小结

要点:

- 计算实现的模型: 冯·诺依曼原理——强制改变内存中的值。不易证明其正确。
- 组织程序的范型即: 算法过程+数据结构 (计算控制+计算对象)

## 3.1 计算对象表示—值与类型:

- 名值分离、值的解释、头等程序对象、V的OP, 如何分类

## 3.2 计算对象实现—存储:

- 存储的内容、分配/除配/回收等、栈/堆/堆栈帧、动静态存储、指针/引用与悬挂引用

## 3.3 计算对象连接—束定

- 定义/声明、作用域与生命期

原罪: 名值分离导致的多对多

目标: 人和机器的无二义理解

难点: 类型系统与动态环境下的值语义

## 3.4 计算组织-程序控制

- 顺序+goto → 顺序+分支/选择/条件+迭代/循环
- 分支设计问题：嵌套与多分支
- 迭代/循环：
  - 显式——无条件/有条件/计数：循环快的语义等价
  - 隐式——遍历具有聚集特性的数据结构（迭代器）
- 关于goto的讨论
  - goto问题解决了么？
  - 循环中的隐式goto及其规则
  - 迭代器中能否引入变形goto？
  - Dijkstra的观点：
    - 结构一致准则：表示逻辑与执行逻辑相符
    - 卫式/guard：进一步理解控制，确保控制的正确性，满足条件的多分支如何执行。控制的语义描述。
- 异常控制：可控/显式、不可控/隐式
  - 更好地用户体验

## 3.5 函数和过程

- 过程与函数抽象的异同
- 接口/界面（功能语义）与功能实现体
- 界面的规格：形实参数的匹配
- 无参、无返回与函数副作用
- 参数的传递机制与指针参数
- 高阶函数

## 3.6 抽象与封装

- 抽象：方法/数据的特征规约，望名知义
- 封装：分清你我，明确变量的作用域
- 以抽象数据类型构建类型系统
- 类属：声明集的抽象，同样的方法不同的类型