数组

- 数组的属性：
  - 成员类型，有些语言允许非整数的下标类型（例如，**Pascal**、**Ada** 等允许以枚举类型作为下标）
  - 维数和每个维的上下界（固定下界时需要有元素个数）
- 可以将一个数组看作是一个从下标类型到元素类型的有限函数
- 数组有一维的和多维的
  - 一维数组只有一个下标，多维数组有多个下标

    有些语言要求实现至少必须支持一定维数的多维数组，有些语言或实现对最大的维数有限制
  - 一些语言只支持一维数组，把多维数组看成以数组作为元素的数组

    如 **C** 和 **C++**
  - 多数语言里直接他提供多维数组的概念

数组

数组操作：

- 成员访问：通常见用 **[...]** 和 **(...)**
- 其他操作的情况：
  - 能否整体赋值（**C** 不允许，**Pascal** 允许）
  - 能否比较相等（大多数语言都不允许，**Ada** 允许）
  - 能否整体传入子程序或者从函数返回（一些语言允许将数组值传给子程序，一些不允许；多数语言不允许返回数组）
  - 有没有数组值（称为聚集值）的描述方式（在程序里直接写数组值）
  - 程序里能否访问数组的属性（如 **Ada** 可访问长度、下标上下界等）

C：只有成员访问操作，不允许整体赋值和比较，只能描述数组初值

Java：可赋值（是引用赋值，不是数组内容拷贝，可以通过 **clone** 拷贝），数组对象提供可访问属性 **length**

数组：存储表示

采用元素等距连续排列的方式（因为元素的类型统一），访问效率高

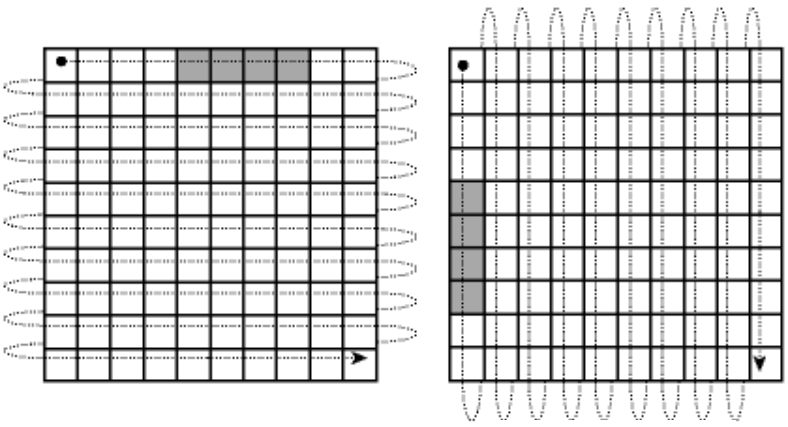
- 有些语言提供压缩排列方式，以提高存储利用率
- 许多语言里的数组有动态属性（内情描述，可以关联于具体数组，或依附于数组类型），以支持运行中的下标检查（下标越界是动态运行错）。动态属性包括：维数，各维下界和上界值，元素大小等

多维数组的元素存储：

可采用行优先或列优先元素排列方式

Fortran 采用列优先方式

多数语言用行优先方式



行优先序　　列优先序

数组：操作

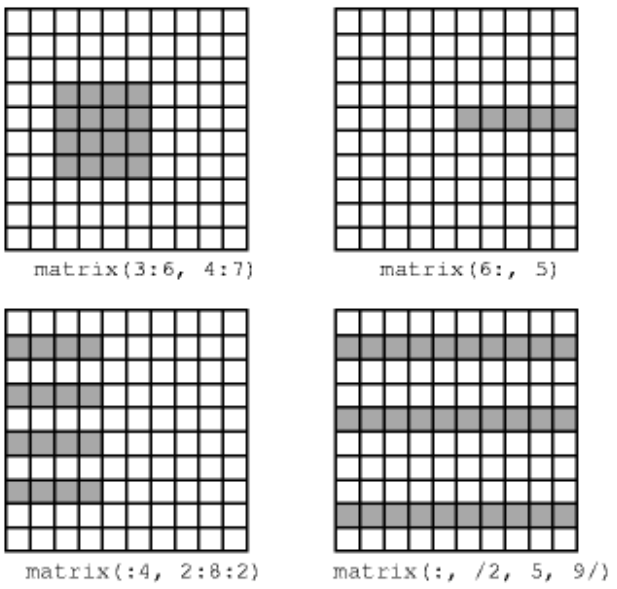
最基本操作是元素访问

另一类操作称为数组切片（**slicing**）：取出数组里的一部分元素（结果也是一个数组）

右图是 **Fortran 90** 提供的一些数组切片操作实例（**Fortran** 的数组下标从 **1** 开始（**2:8:2**是一种循环描述）

Ada 允许对一维数组切片

C++ 标准库的 **<valarray>** 库提供了很多“数组”操作功能（包括切片）



有些语言提供了其他操作。**Ada** 允许按字典序比较离散类型的一维数组（把这种数组看作元素的串）

APL 语言提供了很多数组运算，后来的语言在这方面功能都参考了 **APL**

数组：形状

数组的维数和各维上下界称为数组的形状（**shape**，**Fortran** 的术语）

数组对象与形状的关联，存在许多可能性。实例（用 **C** 的形式）：

```
int a[100];
int f(...) { int b[100]; ... }
int g(int n) { int M[n]; ... }
```

等等

1. 全局生存期，静态形状：程序加工时确定数组的大小和形状，运行前在静态数据区完成数组的创建，静态确定分配的位置

   基于固定位置（如数组首地址）和动态算出的偏移量访问数组元素

2. 局部生存期，静态形状：程序加工（编译时）时计算出数组的大小和形状（静态确定），在子程序的帧里为数组安排好位置

   运行进入子程序之时分配子程序的栈帧，其中包含着为这种数组分配的空间。通过相对于帧指针的偏移量访问元素

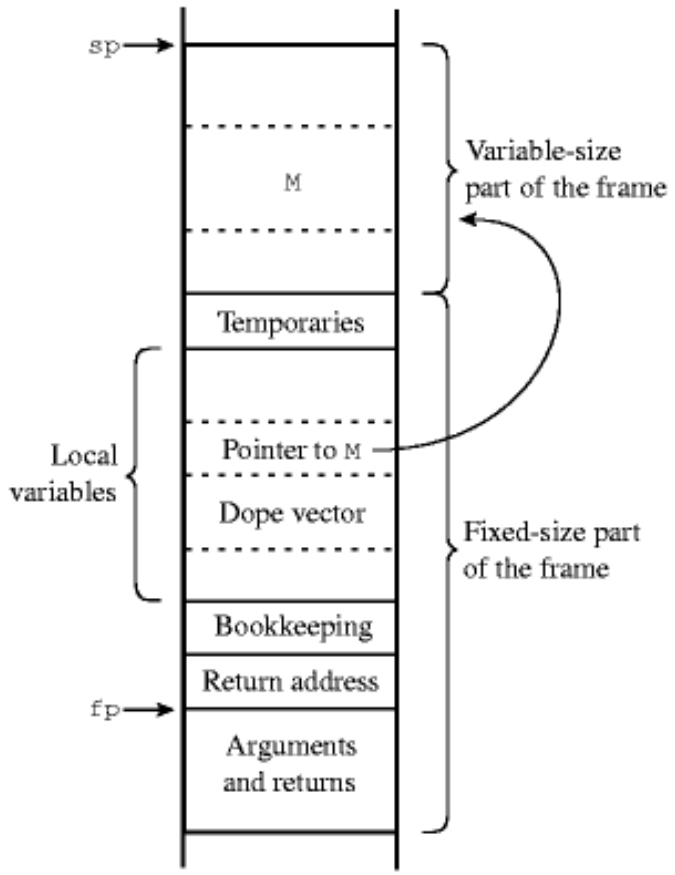
数组：形状

3. 局部生存期，创建时形状（一类动态数组，如 **Ada**，**C99**）

   运行中在创建时动态算出数组的大小，在当时的栈帧之上为其另行分配，退出子程序时与子程序的帧同时撤销

   通过多一层间接的方式访问数组元素（类似于引用）

```
int g(int n) {
    int M[n], M1[2*n];
    ... M[i] ...
    ... M1[j] ...
}
// C99 允许
```



数组：形状

4. 任意生存期，创建时形状：允许在运行中的任何时刻创建，创建时确定数组形状，创建后形状保持不变。需要在堆中为数组对象分配空间

   通过引用（变量）掌握和使用数组（访问元素）

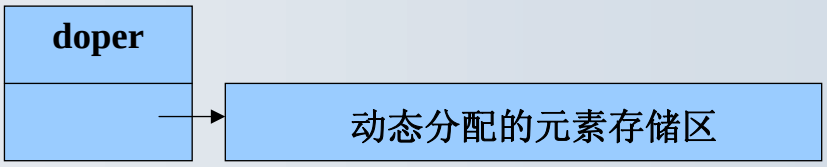
   例如 Java 数组对象

   存储布局：

5. 任意生存期，动态形状：不但动态创建，而且可以存在期间而根据操作的需要而改变形状。需要在堆中分配，元素存储区需另行分配，操作中检查数组大小，需要存放更多元素时动态替换一块元素存储区

   典型例子如 C++ 标准模板库的 <vector>和<string>（看作数组）

   存储布局：



数组：地址计算

假定有数组声明：A : array[$L_1..U_1$, $L_2..U_2$, $L_3..U_3$] of elem_type

令：

$$\begin{aligned} S_3 &= \text{size of elem\_type} \\ S_2 &= (U_3 - L_3 + 1) \times S_3 \\ S_1 &= (U_2 - L_2 + 1) \times S_2 \end{aligned}$$

A[i, j, k] 的地址是（用 &A 表示 A 的起始地址）：

$$\begin{aligned} & \&A + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3 \\ = \ & \&A - [L_1 \times S_1 + L_2 \times S_2 + L_3 \times S_3] + (i \times S_1 + j \times S_2 + k \times S_3) \end{aligned}$$

- 静态数组的 &A 静态已知；局部数组的 &A 在帧里的偏移量已知
- 对于静态确定了形状的数组，方括号内表达式可以静态计算；如果形状可动态变化，这部分就需要动态计算（对 C 语言，由于下标总从 0 开始，这部分的值永远是0，即使是动态形状，也没有运行时开销）
- 圆括号内的部分通常需要动态计算，除非访问用的下标是静态表达式

$\&A - [L_1 \times S_1 + L_2 \times S_2 + L_3 \times S_3]$ 称为虚拟 0 点，通常把它和上面计算出的几个 S 值作为数组属性，以提高执行效率

数组：实例

C 数组：

- 无动态属性（无内情描述），只存储数组元素，达到最小化

  带来的优点：数组的一部分可以当作数组使用
- 数组元素都从 0 开始编号，元素的位置计算方便，访问效率高
- 通过数组与指针关系和传递数组起始地址的方式支持“通用”子程序
- C99 提供了具有局部作用域和创建时形状的“动态数组”
- 通过动态存储分配，可模拟任意生存期（创建或动态形状）的数组
- 无法检查（不检查）越界，安全性差。是许多系统的缓冲区溢出漏洞根源

Java 数组：

- 任意生存期，创建时形状（所有数组对象都是动态分配），下标从0开始
- 通过可访问的长度属性 length 支持处理数组的“通用”子程序
- 动态越界检查，越界时引发异常

数组：实例

Ada 提供了类型安全的数组，允许定义通用过程。

数组类型和变量定义：

```
type vector is array (integer range <>) of float;
vec1 : vector(1..100);
vec2 : vector(20..45); -- vec1,vec2 都是 vector 类型变量

function mean (a : vector) return float is
  temp : float;
begin
  temp := 0;
  for i in a'range loop temp := temp + a(i); end loop;
  return temp / a'length;
end mean;
```

- 允许定义的数组类型时不刻画下标范围
- 允许程序访问数组属性（a'range, a'first, a'last），以这种方式定义类型安全的“通用”子程序

字符串

一些语言里的字符串是基本对象；一些语言里的字符串是字符的数组，常为这种数组提供一些其他数组不能做的操作（包括动态改变大小等）：

- 使用特别广泛，可能有些特殊操作服务于应用需要
- 特点：一维，其中不包含对象引用，其实现可以（且应该）特殊处理

语言中的字符串大小可以是：

- 默认固定长度限制（可能是具有静态形状）。每个字符串的字符都不超过该限制（老的 **Fortran**）
- 静态长度约束，声明中包含字符串长度（与数组一样）。如 **Pascal**、**Ada** 里的字符串（这种字符串可以在静态区或栈帧里创建，易于管理）

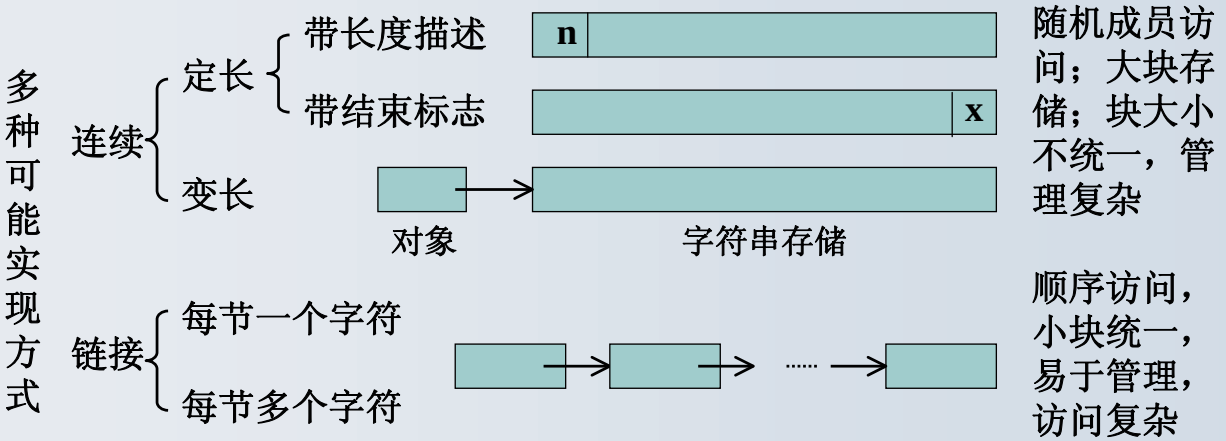
  var name : packed array [1..20] of char;
- 创建时长度约束，或者允许动态改变长度和运算中的动态创建。这样的字符串使用方便，但需要在堆中分配和复杂的存储管理技术支持（脚本语言，**Lisp** 和 **Smalltalk** 等语言和许多字符串库采用这种方式）

字符串

C 以字符数组作为实现字符串的基础，通过指针与数组的关系、空字符结束标志等技术，同时支持静态约束长度的字符数组和动态分配的存放字符序列的存储块，都可用于表示字符串，两种结构使用方式统一

主要字符串操作：比较（按字典序），拼接，子串选择和子串匹配等

早期著名的字符串处理语言 **Snobol** 提供大量字符串操作，是后来一切字符串机制设计的蓝本。脚本语言通常提供了强大的字符串处理机制



表

表（list）是 Lisp 语言的基本构造对象，即数据结构里讨论的“广义表”

许多语言以这种对象作为数据表示或数据表示的基础，包括 ML、Prolog、Smalltalk，以及各种符号计算系统（如 Maple、Mathematica 等）

抽象描述：表是 $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ 其中 α_i 可以是基本数据对象或表

基本操作：

$$
\begin{array}{lcll}
head((\alpha_1, \alpha_2, \ldots, \alpha_n)) & = & \alpha_1 & car \\
tail((\alpha_1, \alpha_2, \ldots, \alpha_n)) & = & (\alpha_2, \ldots, \alpha_n) & cdr \\
cons(\alpha, (\alpha_1, \ldots, \alpha_n)) & = & (\alpha, \alpha_1, \ldots, \alpha_n) & cons
\end{array}
$$

Lisp 函数名

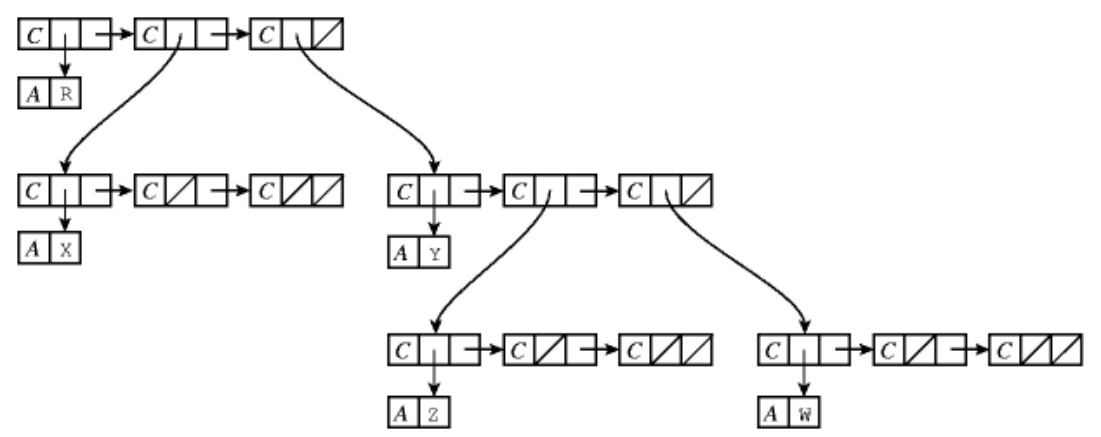
通过这些基本操作，可以构造或解剖任意复杂的表结构

- 一个表的元素可以是任意的基本数据对象或者表，同一层的元素可以是任意的基本数据对象，也可以是任意形式的表
- 表是一种递归数据结构，其成分具有同样的结构
- 利用表结构，用表头原子作为标志，可以模拟各种复杂的数据结构

表

表的基本实现方式：采用链接结构和单元标志。例：

(R, (X, nil, nil), (Y, (Z, nil, nil), (W, nil, nil)))



标志 A 表示原子，标志 C 表示 cons 单元。cons 单元是构造表的基本单元

在支持表结构的语言里，都提供了一组多态性表操作

指针

指针是一种间接的组合手段，用于在程序设计的层次上实现“引用”

常规语言通常通过指针和动态存储分配支持构造复杂的数据对象，如大小可动态变化、成分数目可动态变化、结构可动态变化的对象

有关基本机制包括：

- 定义指针类型（以被指类型相互区分）及指针对象的机制
  - 指针类型（access 类型，访问类型）是一类简单类型
  - 指针类型的对象里存储对其他对象的引用（常用地址，可用其他方式）
- 动态创建对象的机制，为对象分配空间并返回其引用
  - 这种对象没有名字，只能通过指针引用
  - 创建可在任何时候进行，对象的某些属性可在创建时确定（如大小）
  - 生存期与程序结构无关，不自动销毁
- dereference 操作（间接），由指针取得被指对象

指针

- 对不再使用的动态对象的释放机制（free/delete/release），或对不可能再用的动态对象的自动回收机制（废料收集）
- 在大部分语言（例如 Pascal）里，指针只能指向堆中分配的数据对象（数据对象严格划分为两个世界，指针只能指到堆里）

  少数语言（例如 C）里指针可以指向堆对象，也可以指向栈对象或静态对象（为此需要提供取得引用的机制，&）
  - 可以构造出三种存储区的不同对象间的复杂引用关系
  - 有实用价值，也是很危险的机制，带来更多产生悬空引用的可能性
- 一些语言（如 C、Ada 95）提供了指向函数（子程序）的指针
  - C 发明了函数指针，原来是作为函数的函数参数的替代品。后来认识到这种机制的重要性，通过它可以建立数据与计算功能（函数）之间的灵活联系，基于它开发了许多高级编程技术（包括 OO）
  - Ada 95 对 Ada 83 的一个重要扩充（认识到其重要性）

指针

指针的定义，有几种情况：

- 指针有类型，只能引用特定类型的对象。这将使与指针有关的类型问题可以静态分析和处理。常规语言多采用这种方式
- 指针可以引用任何类型的对象。这种情况下，要正确访问被指对象
  - 要求程序员正确处理。例如对于 C 语言里的 void 指针，必须先明确转换到某个类型的指针后才能间接访问被指对象
  - 要求被指数据对象带有类型，通过运行中的动态检查保证正确处理
- OO 语言里的指针可指向一个类型的所有子类型的对象。带来很大灵活性，支持多态性和许多程序技术，也产生了一些“动态类型”问题

许多语言提供了特殊操作（如 new）完成动态对象创建，以类型为“参数”，可以支持类型检查、复杂的对象创建动作（包括用户定义动作）

C 通过标准库的一组普通函数支持动态对象，没有引进新的功能，因此也无法利用语言的类型功能，不能支持类型检查（一切交给程序员）

指针：实现

指针的实现很简单，只需解决引用的表示问题：

- 用实际地址（绝对地址）：优点是寻址效率高。
- 用虚拟地址（相对地址）：提供更多灵活性，访问时需要从相对地址计算出绝对地址。由于有硬件支持，效率不会降低
  - 采用这种方式，移动对象的位置时比较容易维护指针关联（只需修改基地址，例如整体写入文件，以后重新装入另一块存储）
- 统一管理所有的动态对象：多种不同对象混在一起（大小和使用方式都不统一），管理中需要处理各种问题（碎片、大小等），不利于废料收集
- 在一个堆区里只处理同样大小的对象，创建和回收比较方便
  - 如果按类型分开存放不同对象，存储位置就可以作为类型的信息
  - 管理多个存储块增加了工作量，可能造成整体的空间损失

指针是为了模拟引用，但它比引用更危险。采用引用语义的语言里不再需要指针，因为变量本身就是引用

指针：C 语言

C 的指针很特殊，它承担的任务特别多，也特别容易用错

- C 指针可看作是地址概念在高级语言层面的简单而直接的反映
- 同时提供了有类型的指针和无类型的指针（void 指针）
- 允许任意指针之间的类型转换（包括有类型与无类型指针之间，不同类型的有类型指针之间），以及指针和整数之间的转换
  - 指针转换后使用的正确性完全由程序员保证
- 数组名访问被自动转换为指向数组起始位置的指针值，主要是为了支持处理数组的函数，“指针运算”的首要目的也在于此
  - 各种指针运算使程序可以很方便地操作数组元素
  - 通过数组与指针的关系，允许把数组的一部分当做数组使用
- 提供取地址操作，使指针可以引用静态对象和堆对象
  - 程序员可利用这种概念，按统一的编程模型处理所有对象。以指针为参数的函数可处理所有类型合适的对象（无论它在哪个存储区）

指针：C 语言

- 动态存储分配完全借助于指针的概念，用普通的函数实现
  - 要求程序中显式计算分配申请的大小，自己保证类型转换的正确性
  - 利用通用指针和具体类型的指针之间的转换
  - 不需要特殊内部操作，但带来了引进类型错误的危险性
- 函数指针使人可以把函数约束到变量（或其他数据结构）
  - 基本功能是了为实现函数的函数参数，减少概念（不引进新参数类）
  - 重要性：是常规语言里建立子程序与数据间关系的一种自然机制，可用于支持“数据驱动的程序设计”，也是 OO 的技术基础
  - 老 C 语言允许不完全函数原型（和函数指针声明），容易写出带有编译器无法检查的类型错误的程序。必须特别注意，一定不要用

C 指针是 C 语言里最灵活，功能最强大的机制，也是最难用好的机制

问题：废料（存储泄漏）和悬空引用，指针的错误使用（类型等）

数据类型与存储

数据类型的实现需要处理一些存储管理问题

- 数组类型，需要记录元素大小、各维的上下界等信息记入符号表
- 记录类型，需要算出各成员的偏移量，记入符号表
- 静态对象根据类型确定所需空间，在静态区分配，把位置记入符号表
- 局部对象计算出所需空间量，在子程序的堆栈帧里安排位置，把以帧指针为基址的偏移量记入符号表
- 对静态对象的成员访问，根据符号表编译成绝对地址访问（对于记录/结构）或者相对于虚拟0点的地址计算（对于数组）
- 对局部对象的成员访问，编译为相对帧指针的偏移量或者地址计算

动态分配对象，（new操作）根据类型确定所需空间量从堆里分配

从指针出发的成员访问编译为相对于指针值的偏移量寻址

通常把动态存储块的大小存放在块中某个不可见位置，释放时利用它

数据类型和存储管理

复合数据对象（特别是记录）的存在，使悬空引用和废料问题更难解决

- 不仅静态区和栈里里可能有指针，任何动态对象的内部都可能有指针，要识别所有的活对象，就需要追溯所有复合对象里的指针（引用）
- 问题：一个记录里的哪些域是指针，需要继续追溯检查？

要进行这一工作，必须知道各种记录类型的布局。为此：

- 编译器必须为每个记录类型生成一个布局描述。这是一个表格，其中列出记录里所有指针的偏移量（例如 Java）
- 对每个对象，都必须能找到相应类型的布局表格（以便废料收集器或其他管理程序知道该如何工作下去），通常在对象里放一个隐式指针

对静态区和堆栈帧，也可能需要类似表格指明其中的指针位置

《Pragmatics》第7章介绍了两种防止悬空引用的技术，以及若干废料收集方法的简单情况，可以自己读一读

其他类型机制

一些语言还提供了其他复合类型的构造机制

集合（例如**Pascal**）：

可以定义任何类型的集合，提供元素关系判断，和各种基本的集合操作

目前大部分都没有提供这种功能，有些通过库功能提供集合

集合语言 **SETL**，人们在那里研究了各种集合操作的高效实现和应用

递归数据类型（特别是函数式语言）

可以声明递归类型，方式与声明枚举相似。例如在 **ML** 里

```
datatype  ctree = empty | node of string * ctree * ctree;
```

定义了一种树类型（递归）**ctree**：或者为空树，或者为有一个结点（根），这种树包含一个 **string** 值和两棵 **ctree** 子树

多态性

多态性（**polymorphism**）指一个事物与多个类型相关（有不同看法）

类型是程序语言中最重要的概念之一，但也应该允许程序里出现超越类型的描述，如定义多态性对象，允许各种多态性描述。例：

- 数组或者表的排序程序，其算法过程与数组的元素类型无关
- 栈和队列数据结构，有关的操作与其中元素类型无关；等等

人们希望在常规语言中支持多态性，希望泛型子程序一类概念也可以出现在常规的静态类型语言里。因为这样可以大大提高语言的表述能力（一个描述，多处/多种使用）；同时又希望支持静态类型检查，保证类型安全性

- **1980**年代对类型的研究，逐渐形成了比较清晰的多态性概念
- 后来的语言都（或多或少地）支持多态性概念，设法支持泛型程序设计
- 语言的多态性程序设计机制留待后面章节讨论
- 现在大致介绍一下多态性的概念，及其对类型检查等的影响

多态性

侠义看法：程序里的一个对象与多个类型相关，例：

- 多态变量：不同时刻可能具有不同类型的值
- 多态子程序：可以处理不同类型的参数

一些“静态类型”的 OO 语言（如 C++、Java）允许“受限的”多态性，一个对象（引用，指针）只能引用某个类型的子类型的对象，子程序只能以某个类型的子类型的对象为参数（后面还要讨论）

广义些的看法：程序里一个描述（例如一个名字）与多个类型相关

- 运算符或者函数/过程名的重载（称为 ad hoc 多态性，实质多态性）
- 多态性的文字量：在不同环境里表示不同类型的值
- OO语言里特别重要的由于动态约束而产生的多态性
- 类型模板（类模板，template、generic）和子程序模板定义，虽然它们都只是静态描述，但也能表示多个不同的实例（类型、子程序等）

多态性

多态性给类型检查带来问题，要求静态处理时做更多的类型解析工作：

- 语言本身的运算符重载是绝大多数语言里都存在的现象
- 如果允许用户定义的运算符重载或子程序重载，语言必须有一套运算符解析规则。如果还有类型强制，要正确确定应进行的强制转换，选出使用的子程序，规则可能变得很复杂

例：C++ 的重载解析规则（如何确定运算符或函数名的指称）：

1. 准确匹配，不转换或平凡转换（如数组/函数名到指针，T 到 const T）
2. 包含提升的匹配（整数提升，flaot 到 double 的提升，需要执行强制）
3. 使用语言的标准转换的匹配（需要执行强制）
4. 使用用户定义转换的匹配（调用用户定义强制）
5. 利用函数声明中的 ... 的匹配

由于存在函数模板和自动实例化，进一步增加了类型解析的工作和困难

类型推理

语言中某些结构有类型，但没有明确给出类型，需要通过相关结构的信息去推断，确定相应类型的这个过程就是类型推理（**type inference**）

在许多环境中需要做类型推理：

- 处理表达式过程中，需要从运算对象类型和运算符性质，推断运算结果的类型，以便完成更大表达式的类型检查（及生成可能的强制）
- 一个文字量出现在一个环境里，但它可能代表不同类型的值，例如
  - 许多语言允许定义整型的子界
  - 有些语言允许不同枚举类型的枚举符重叠（如 **Ada**）
  - **C** 里的 **0** 代表一个整数值，也代表空指针值
- **C++** 的函数模板和自动实例化机制。当某模板函数被用在一个环境里，系统需要根据环境的类型要求，自动推断出应该如何做实例化，如果需要，就生成一个针对特定参数类型的模板实例

多态性和类型推理

有些语言提出了更复杂的类型推理要求，用以支持一些特殊的功能，特别是用于支持类型清晰的多态性子程序

Lisp 语言里的符号原子（类似变量）具有动态类型，可以约束于任何类型的值，因此具有多态性；其中的子程序（函数）可以处理任意类型的对象（例如取一个表的头部），因此都是多态子程序

```
(define length (lst)
  (cond ((null lst) 0)
        (#t (+ 1 (length (cdr lst))))))
(define include (a lst)
  (cond ((equal a (car lst)) #t)
        (#t (include a (cad lst)))))
```

这些函数都可以对任意类型的元素和表使用（泛型函数，**generic**）

这种多态性是“无类型多态性”，无法静态检查类型（只能做动态检查）

类型推理

研究者在 ML 语言里仔细研究了类型推理问题，弄清了在强类型语言里通过静态类型推理可以得到哪些信息，如何通过类型推理支持多态性子程序及其类型检查，并支持处理复杂数据对象的多态性子程序

在 ML 里定义函数时不描述参数和返回值类型，可能通过函数的内部结构推断这些类型（例如 x 参与加法且另一对象是1，就推断 x 为整型等等）

简单实例：

```
fun pow(m, n) =
  if n = 0 then 1
  else m * pow(m, n-1);

fun append(l1, l2) =
  if l1 = nil then l2
  else hd(l1)::append(tl(l1),l2);

pow : int * int -> int
append : 'a list * 'a list -> 'a list
```


数据类型

- 类型机制及其作用
- 类型检查
- 类型等价和类型相容
- 类型转换和强制
- 基本类型
- 用户定义类型
- 数据类型与存储管理
- 多态性
- 类型推理