基本类型

下面考察一些最基本的类型，考虑它们的一些常见问题

不同语言可能提供不同的基本类型集合

布尔类型（逻辑类型）

用于程序控制，只有“真”和“假”两个值

通常采用一个字节表示，用 **1** 表示“真”，用 **0** 表示“假”

原则上可以只用一个二进制位，但字节是计算机里最小的可寻址单元

不同语言可能采用不同形式的逻辑字面量

C 是少有的无逻辑类型的语言

任何基本类型值都可以用作逻辑值参与程序控制

逻辑运算用 **int** 值 **1** 和 **0** 表示真和假（**C++** 增加了 **bool** 类型）

整数类型

- 一般语言里的整数采用定长的二进制表示方式实现。这样，一个整数类型就是有穷的一组连续整数值的集合
- 多数语言不规定整数类型的表示范围。有的提出了最低要求，把问题留给实现，以免给语言的实现带来麻烦
  - 通常要求以程序可检查的方式提供实现的具体信息
  - 如最大和最小整数，例如 **C** 标准库的 **limits.h**
- 具体实现通常采用所在运行平台上效率最高的编码形式实现整数，以得到最高的效率。不具体规定有助于适应各种（新）实现环境的需要

  这种做法使整数类型更像一个抽象类型

  实现的细节和方式由具体实现自由选择，也会损害程序的可移植性
- 有的语言定义了多种整数类型，有些语言只有一种整数类型

  有些语言规定了整数类型的具体表示方式，有些只有原则性的规定

基本类型：整数

- Java 是一个极端
  - 规定了所有数值类型的编码长度（short 、int 和 long 分别是16/32/64位有符号整数）
  - 这种方式提供了另一种“抽象”，目标是程序的平台无关性，把具体平台的具体特征全部抽象掉，有利于程序在不同平台之间的移植
  - 缺点：在某些机器上可能难以有效实现
- C 提供了许多整数类型（不同表示范围，有符号或无符号）
  - 为支持系统程序设计中对于各种类型的细致选择
  - 带来选择的困难，一般应采用最常用类型 int 和 double
- C99 不仅支持抽象的 short、int、long，还（通过标准库）支持一组扩展的规定了具体长度和表示方式的整数类型
  - 这种扩充的目的是使系统程序员有更多的选择。这也是多年系统程序设计实践提出的需求（这种抽象与 Java 类似）

基本类型：子界

一些语言提供了定义整数或枚举的子界类型的功能（如 Pascal、Ada ），支持定义所需范围的类型（在基本类型的表示范围内）。如

```
type Testscore = 0..100          Pascal整数的子界
     Workdays = mon..fri         枚举类型的子界
```

子界通常采用与整数同样的表示方式（也可能用更优化的表示方式）

引入子界的目的是支持应用所需要的数值限制，在语言层内部实现用户所需的检查（数据的值超出界限变成了“类型错误”）

子界类型的问题也在这里，其变量的赋值合法性需要在运行时动态检查

例如 n 和 m 是两个 Testscore 类型的变量，

```
n := n + m;
n := n + 120;      120是整型文字量（可能静态判定错误）
```

通常需要做动态的“类型”检查，防止值越界。（在这里，是否属于本类型的问题要检查数据的值，不可能静态处理）

基本类型：字符和字符串

字符类型和字符串类型：

- 一些语言把字符类型作为基本类型，用一维字符数组实现字符串

  以编译方式的实现作为目标的语言常常采用这种方式，因为不同字符串的大小不统一

- 另一些语言以字符串为基本数据类型，字符看作一个字符的串

  Lisp、ML 和各种脚本语言采用这种设计

- 字符串的问题后面讨论

- 字符类型通常采用某种标准的编码字符集实现

- 多数语言没规定具体的字符集

  如，说 **C** 语言用**ASC II**字符集是不对的。具体 **C** 系统完全可能采用其他字符集（尤其是大型机上的 **C** 系统）

  有些语言规定了字符集，例如 **Java**

字符类型

- 字符集的大小
  - 一些语言采用 **128** 个字符或者 **256** 个字符的字符集
  - **Java** 规定采用 **Unicode** 字符集，这可能是未来语言的趋势。但采用 **Unicode** 字符集也带来许多问题，如处理效率，可读性等
- 在采用小字符集的语言里，字符类型的对象用一个字节实现

  如果采用大字符集，就需要用几个字节来实现一个字符对象
- 字符类型的对象以字符的编码为值
  - 通常把字符类型看作一种有序类型，基于编码值序提供比较操作

    不同字符集，可能导致不同的编码和序关系
  - 一些语言里提供了 **succ/pred** 操作，可以从一个字符找到后一/前一字符

    在 **C** 语言里没有专门操作，但把字符类型看作一种整数类型，因此可以借用整数的加减运算

基本类型：字符

与字符集关系密切的一个问题是国际化和本地化：

- 随着应用发展，字符数据变得越来越重要（软件需要与人打交道），自然语言的多样性也反应到软件、程序设计和语言里
- 网络通讯及由此带来的世界性信息共享，对人生活中的影响越来越大。这种情况也反映到程序语言的设计中

程序设计语言应能处理多种自然语言的文字。这里的问题：

- 字符集，例如 **Unicode**，可能还不足以满足需要
- 字符序和字符串序，不同的自然语言习惯会引出复杂的排序问题（想想中文的情况，有多少中常见的文字排序方法）
- 大小写，一些语言里不区分大小写
- 读入与输出顺序（从左向右，从右向左，...），等等

■ 在新语言的设计和标准化中，在这些方面问题有许多考虑

基本类型：浮点数

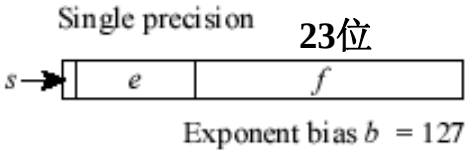
浮点数类型

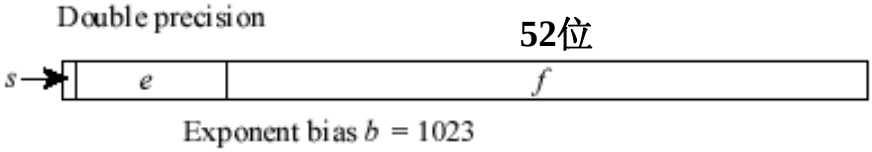
1985 年 **IEEE** 推出了硬件的二进制浮点算术标准，此后的处理器大都采用符合标准的浮点数表示和算术语义

程序语言常承认（默认）这一标准

有关浮点数算术有非常复杂的语义

Pragmatics 有一小节讨论。也有专著





	e	f	Value
Zero	0	0	± 0
Infinity	$2b+1$	0	$\pm\infty$
Normalized	$1 \le e \le 2b$	<any>	$\pm 1.f \times 2^{e-b}$
Denormalized	0	$\neq 0$	$\pm 0.f \times 2^{-b+1}$
NaN	$2b+1$	$\neq 0$	NaN

无穷大，规范数，非规范数，不是数

基本类型：其他数值类型

一些语言提供了定点实数类型：

- 数值包含固定长度的数字位数，具有固定的（默认的）小数点位置
- 通常采用特殊计算规则，一些计算机硬件里有专门的编码形式（称为二-十进制数，二进制编码的十进制数，BCD 数）和一套专门计算指令
- 主要用于商务应用和一般性事务处理（如金融等）
- 例：Ada 提供了特殊的定点类型声明方式，可以声明具有任何精度位数，小数点位置的定点实数

有的语言提供了另一些数值类型，常见的：

- 复数类型。Fortran、C99 都提供了复数类型，C++ 等语言通过库提供
- 有理数类型。 例如 Common Lisp、Scheme
- 大整数类型（支持任意长度的整数及其运算的类型）。Common Lisp、Scheme 都有这种类型，Java 通过库提供了大整数类型

多字节数据的表示：字节序

存储一个基本类型的值可能需要多个字节，如何排列这些字节？

两种字节序（排列方法）：大尾端（big-endian）和小尾端（little-endian）

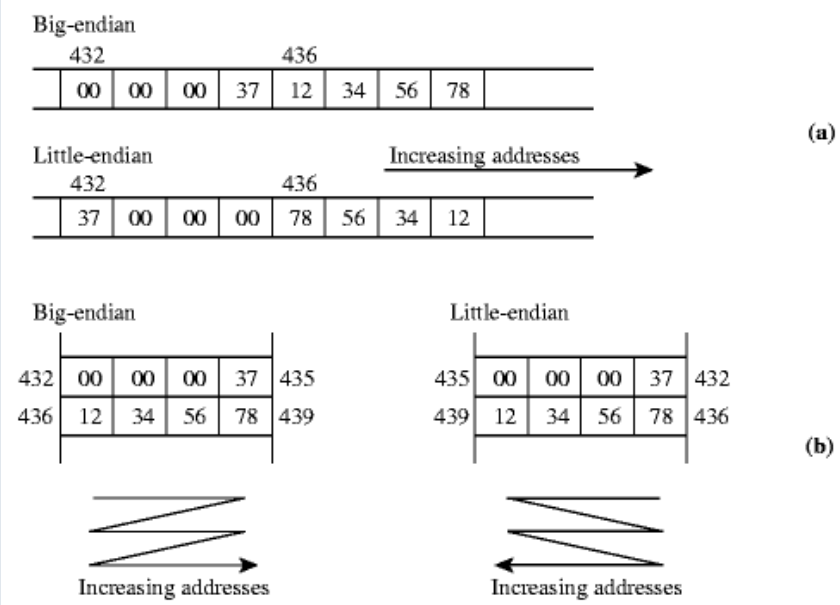
两种字节序各有优点

BE：多字节数据，看起来比较自然

LE：能容忍数据的大小变化

例：X86用LE，用IDE的存储监视窗口时，如选择的数据长度不合适，会看到很混乱的东西

问题：直接用字节流传输大数据，如果字节序不同将无法恢复



设单元432和436存的数是 0x37 和 0x12345678

类型构造

语言需要数据描述的扩充机制，以允许程序员定义自己的类型

定义新类型，必须描述该类型是什么。完整的类型定义包括：

- 定义与之相关的一组属性（数组的元素类型，结构的成员类型和名字）
- 定义可能取值集合（可能是隐含的）
- 定义与之关联的一组合法操作
- 为被定义的类型命名

许多语言的类型机制不完全，可能缺少上述要求中的一些部分。例如：

Pascal 等早期语言都没有提供为自定义类型提供操作集合的机制

C 语言用 **typedef** 定义的名字并不是真正的新类型

本章只讨论类型的基本描述问题

有关相关操作约束和类型封装问题是后面章节的内容

枚举

定义新类型的最简单机制是枚举，由 **Pascal** 引进（称为“纯量类型”），当时的主要目的是提高程序可读性，使编译程序可以帮助检查一些错误

- 枚举是有限个（有序的）标识符表示的值构成的用户定义类型
- 定义时列举出所有属于这个类型的符号常量（枚举符）
- 通常不允许多个枚举类型的枚举符（枚举常量名）相互冲突

枚举类型通常用从 **0** 开始的一段连续整数值实现，可能根据类型中的枚举值个数选用较小的存储方式（例如，用一个字节可以表示 **256** 个不同值）

一些语言仅把枚举看作定义一组符号常量的方便机制；有些语言允许用枚举机制定义真正的类型（如 **C++**，可以为枚举类型定义成员函数）

枚举的基本操作是相等/不等和顺序比较。有些语言提供其他通用操作。如 **Pascal** 有 **succ**、**pred** 等。**C** 枚举等同于整型，允许做任何整数运算

C++ 没有为枚举定义专门的通用操作（和 **C** 一样），但允许用户为具体的枚举类型定义所需的任何具体操作

枚举和其他

一个枚举类型包含哪些值？不同语言有不同的定义

- 多数语言里，枚举类型只包括枚举定义时列出的那些值。例如 **Pascal**
- **C/C++** 里，各枚举常量可以独立给值，枚举变量的取值范围另外规定
  - **C** 语言的枚举符是 **int** 常量，一个枚举类型等同于某个整型
  - **C++** 规定：如果枚举类型 **T** 的所有值非负，**T** 类型变量的取值范围是 **[0, 2^k-1]**，**k** 是使这个范围能包含 **T** 中所有值的最小整数；如有枚举中有负值，**T** 取值范围为 **[-2^k, 2^k-1]**，其中包含 **T** 所有值的补码表示

原因：程序设计实践的需要。例（C++）：

```
enum flag { open = 1, good = 2, eof = 4 };
flag f1 = open | good;
```

通常把整数、字符、布尔类型和枚举、子界统称为离散类型或序数类型，因为其值可数且可排序；把序数类型和浮点数类型统称为标量类型

复合类型

复合类型是最重要的用户定义类型，其对象具有内部结构，以其他类型的对象作为成分，这些情况带来许多新问题

- 需要描述复合对象的结构
  - 成员数目固定还是可变，可变时有无最大限制
  - 是否要求成员为特定类型，是否要求所有成员具有统一类型
  - 如果允许不同类型的成员，对成员类型有无限制（能否为任何类型？**OO**语言里经常出现的一种限制是同属一个基类）。

复合数据对象的常见操作：

- 创建，有没有提供值和初始化的描述方式
- 整体操作：能否整体赋值
  - 能否比较相等和不等
  - 有没有动态构造复合值（聚集值）的方式
- 成员访问，选择、提取和修改（分两步：找到整体对象，再找到成员）

复合类型

- 成员访问方式：
  - 按顺序编号（数组）
  - 按名字（结构/记录）
  - 其他方式，如默认方式（如堆栈、队列等等）
- 复合数据对象的实现。要考虑数据成员在实现中的布局或其他安排
  - 成员是共享整体对象的存储区，还是另行分配
- 此外，实现中还要考虑对象是否需要携带有关类型的结构信息（内情描述字，**dope-vector**，或 **doper**）？如何携带（对象独立携带？…）
- 成员组织方式，顺序或链接，或其他方式、混合方式

计算机硬件没有对复杂数据类型的专门支持，可用机制只有基址寄存器、偏移量寻址、变址（带比例缩放的变址）、下标寄存器

复杂情况只能通过软件模拟（如类型描述字处理、复杂的存储管理等）

复合类型

实现中的新问题

- 成员访问的高效实现
  - 如果成员在对象内分配，就可以采用“基址＋偏移量”的方式
  - 如果基址和偏移量都可以静态确定，就可以采用直接寻址的方式
- 生存期管理，成员有无独立的生存期，如何管理和销毁等
  - 如果成员在对象之外分配，就可能有独立生存期，需要另行管理
- 是否允许类型的某些属性在运行时动态确定
  - 如果允许，可能就无法在静态区和栈上实现了
- 有效利用存储
- 对整个语言实现中的存储管理机制的影响
  - 如果语言允许具有复杂行为的复合数据对象，实现就必须提供复杂的存储管理系统，才能支持程序的运行

结构/记录

记录（**record**，或结构，**structure**）是一类复合数据对象，具有固定数目的成员，成员类型可以不同。各成员分别命名，通过成员名访问

- 成员也称为域（**field**）或者成分，成员本身也可以是复合对象
- 需要描述各成员的类型（它们可能类型不同）
- 记录的主要操作是成员访问，通常用 **r.a** 的形式
- 一些语言允许记录的整体赋值（**C**、**Pascal** 都允许）
- 一些语言允许比较记录值（ **C**、**Pascal** 都不允许）
- 一些语言允许给函数传递记录值，或者允许函数返回记录值（**C** 语言允许两者，**Pascal** 只允许前者）
- 一些语言里记录成分的描述顺序有意义（例如，**C** 规定实现中记录的成员按照描述的顺序存储），另一些语言认为成分的描述顺序并不重要，允许实现根据需要任意重排（如 **ML**）
- 是否允许嵌套的记录声明，不同语言也有不同规定

结构/记录：实现

记录的实现：

- 通常采用连续存储块，把成员排列其中

  可能按定义顺序排列，也可能允许编译器根据需要

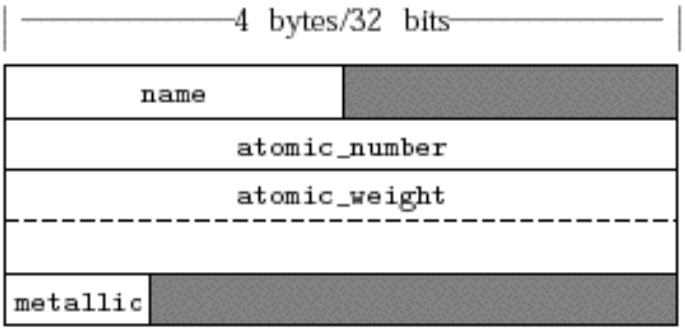
  任意安排
- 记录的实现中通常没有描述字（成员依自身情况可有描述字）
- 编译时静态确定记录成员的大小和整个记录的存储布局，静态确定任一成员 **a** 在记录 **r** 中的相对位置（偏移量，**offset**）
  - 通常不允许运行时才能确定大小的成员，否则就需要更复杂的分配和访问机制的支持
- 成员访问 **r.a** 编译为基于 **r** 的起始位置加上偏移量的位移寻址

  如果记录 **r** 在静态区分配，其成员的位置也都可以静态确定
- 如果语言不要求保持成员顺序，编译器可能做些优化。如根据需要调整成员的布局顺序，最大限度利用存储。也可能为其他需要调整成员顺序

结构/记录：实现

表示化学元素的记录：

```
typedef struct elem {
  char name[2];
  int atomic_number;
  double atomic_weight;
  char metallic;
} xstruct;
```



实现中的一个重要问题是对齐（**alignment**），例如 **int** 和 **double** 需要放在模**4**或者模**8**为**0**的地址，整个记录需要放在具有最大对齐要求的数据类型可以存放的地址（有些编译器提供对齐选项）

数据对齐源自硬件的需要。造成一些情况：

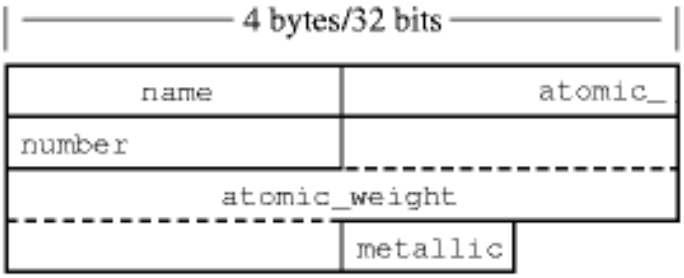
- 对齐问题可能造成记录存储中的空洞，一个成员未必紧跟另一成员
- 记录的存储量未必是成员的储藏量之和，造成存储空间的浪费

记录/结构：对齐和压缩

一些语言允许定义压缩的记录

带来的问题：访问成员需要执行多条指令，还需先在寄存器里拼接起一个完整元素，而后实际操作，存储操作也类似

（时间和空间交换）



实践中节约存储的一种可行方法：按对齐要求从大到小排列记录成分，可能得到最少的空洞（条件：对齐要求都是2的幂。请证明这一结论）

存储空洞带来的另一问题是使记录之间的比较难以进行：

- 不能实现为对存储块的按位比较（或者按单元比较），存储空洞里的垃圾数据可能使比较结果无效
- 对每个记录分别实现特殊的按成员的比较操作，效率可能低，而且会造成代码膨胀（未必都需要用）

记录/结构：成员访问

记录成员访问：从记录对象出发，或从指向记录的指针出发

- 一些语言提供了两种操作符，如 C 分别用 `.` 和 `->`
- 一些语言对指向记录的指针自动间接，可直接从指针访问被指记录的成员。如 Ada 里指针只能指向记录，可直接写 `p.mem`

为方便深度嵌套的记录成员访问，一些语言提供了 with 语句

```
ruby.chemical_composition.elememts[1].name:='AL';
ruby.chemical_composition.elememts[1].atomic_number:=3;
...

with ruby.chemical_composition.elememts[1] do begin
    name := 'AL';
    atomic_number := 3;
    atomic_weight := 26.98154;
    metallic := true;
end
```

Pascal 的 `with` 语句

记录/结构：成员访问

提供 with 语句带来一些问题：

- 作用域：打开记录使局部成员名变为可见，可能遮蔽原来可见的东西
- 难提供同时打开多个同类型的记录的控制结构
- 长的或者嵌套的 with 语句使成员的归属变得不清晰

后来有些语言采用为局部结构引进短别名的方式，例如 Modula 等，可缓解上述问题。在 C 里很容易用指针模拟这种特征（而且无上述问题）：

```
{
  xstruct *p = &ruby.chemical_composition.elememts[1];
  p->name := 'AL';
  p->atomic_number := 3;
  p->atomic_weight := 26.98154;
  p->metallic := true;
}
```

C++ 里还可以通过引用变量解决这种问题

记录/结构：变体记录

记录的一个特殊情况是变体记录，这是一种记录成分，用于支持对一些具有多种不同成分和结构的数据的统一处理

面向对象机制的继承机制可以作为变体记录的替代品，而且具有更好的类型安全性。现在变体记录的重要性已经大大下降

例：定义在叶结点里保存实数的二叉树，非叶结点里只有指针（Pascal）

```
type NodeType = (branch, leaf);
type Link = ^Node;
type Node = record
      case tag : NodeType of
        branch: (left, right : Link);
        leaf : (data : real)
     end
```

这一记录有两种可能布局，依赖于变体标志域 tag 的值

Pascal 的贡献是把变体集成在记录里面，后来许多语言采用这种方式

记录/结构：变体记录

变体记录通过 case 语句使用，例如程序里可以写：

```
case a.tag of
  branch: (... a.left ... a.right ...);
  leaf: (... a.data ...)
end
```

变体的实现：

- 为几个不同变体在记录中分配一块公共空间，在这块公共空间里为不同变体的成分安排好布局方式，静态计算出偏移量

  C 的 union 是一种无标志变体（对多种不同数据的简单的统一包装）
- 对带标志变体（如 Pascal 变体记录）
  - 访问时检查标志，根据标志确定访问方式
- 对无标志变体（如 C 的 union）
  - 直接按偏移量访问。数据安全由程序员负责

记录/结构：变体记录

变体记录的主要问题是安全性（**safety**）

问题：能否防止错误的使用

Pascal 把变体标志域当作独立的成分，看作一个普通的记录域

- 可独立检查，可以（而且必须）单独赋值
- 对于一个合法的变体记录状态，一旦给其标志域域赋了另一个值，该变体中的其他部分就失去了合法值（相当于没有初始化），这个变体就处于一种不安全的状态。允许直接给标志域赋值非常危险
- **Pascal** 的变体标志域还是可选的（允许描述自由变体），更危险

Pascal 的变体记录是其设计中的一个缺陷缺陷

人们在后来的一些语言（**Modula-1, 2** 和 **Ada** 等）里进一步研究了如何提供安全有效的变体记录的问题

有关情况见参考书

记录/结构：变体记录

C 的 **union** 是无标志变体，简单使用危险性很大，正确使用完全靠程序员

```
union data {
    char c;
    int n;
    double d;
} a, b;
```

对于程序中出现的 **a, b**，没有提供任何语言机制去判断它们当时保存的是哪种值（类型是什么，提供哪个成分？）

通过 **struct** 和 **union** 的结合，加上一套编程约定的方法，可以模拟 **Pascal** 的带标志变体记录

这样做时，还是存在与 **Pascal** 的变体记录类似的不安全性

记录/结构：变体记录

在C 语言里模拟带区分的变体

```
struct node nd;

switch (nd.tag) {
case branch:
  .. nd.dt.br.left ..
  .. nd.dt.br.right ..
case leaf:
  .. nd.dt.data ...
}
```

```
enum dist { branch, leaf };
struct node {
  enum dist tag;
  union {
    struct {
      struct node *left,
                  *right;
    } br;
    double data;
  } dt;
};
```

变体是一种很不安全的编程机制

由于 OO 中继承概念的出现，在提供了 OO 机制的语言里，完全能（也应该）用继承机制取代变体记录

Java 取消了变体记录的概念（取消了 union），也是基于安全性考虑

数组

数组：

- 元素类型相同的复合数据对象（数组变量）或者类型（数组类型），元素的个数任意，采用下标方式根据排列位置访问各个元素
- 在不少语言里数组不是真正的类型构造符，而只是值构造符，可以用于定义变量或者其他复合数据的成分，但不能用于定义类型。例如 C/C++ 里就是这样（typedef 不是真正的类型定义）
- 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 等允许用枚举类型作为下标）
  - 维数和每个维的上下界（固定下界时需要有元素个数）
- 不同语言里，数组定义的形式各异：

Pascal	**v : array[-2..10] of real**
C	**double v[13];**
Fortran	**DIMENSION V(13)**

- 数组可以看作从下标类型到元素类型的有限函数