初始化和赋值

对复杂数据类型的对象，初始化操作也可能很复杂

- 初始化操作需要把对象设置到合法的初始状态，保证程序安全
- 在定义/使用对象处反复写细节，很容易出错（赋值的情况类似）
- 方便使用的抽象很有价值

C++ 语言引入了程序员定义的初始化和赋值的概念

解决这类问题的一个很好的想法（需要重载功能的支持）

初始化和赋值是意义不同的操作

- 在赋值时应认为被赋值对象已经有值，因此可能需要对原值做适当处理
- 而初始化总是对新建的对象进行
- C++ 特别细致地考虑了初始化与赋值之间的差异，允许程序员为类的这两个操作定义不同的动作，很有道理

赋值

赋值是常规语言里最重要最基本的操作，但赋值的意义是什么？

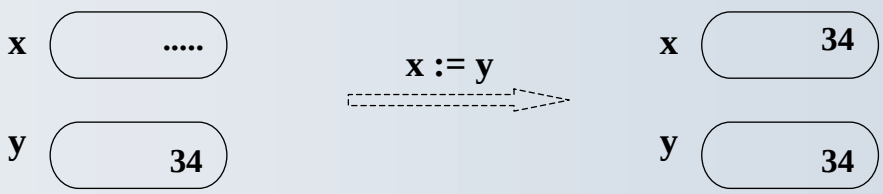
赋值改变变量的值约束。在 **x := ...y...** 中

- 左边（一般而言是表达式） 指称被赋值对象，通常确定一个位置。允许放在赋值符左边的表达式称为左值表达式
- 赋值符右边的表达式表示一个可以赋的值，这种值称为右值

有些表达式同时可以表示左值和右值，放在赋值符号左边时取其左值，放在右边就取其右值（例如变量名 **x**，表达式 **a[i]**，**c.m** 等）

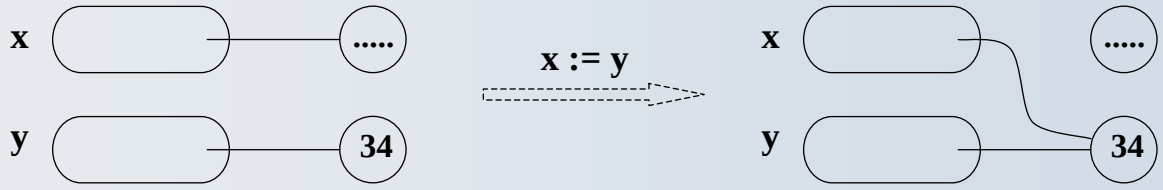
在值语义（值模型）和引用语义（引用模型）里，赋值的意义不同

值语义模型下的赋值的效果是值的复制（拷贝）



赋值

引用语义下赋值是引用的复制，结果是值的共享

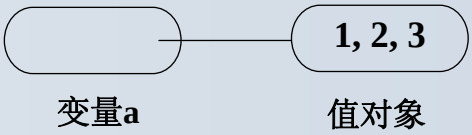


注意这里有两个现象：

- 变量的值是另一对象，赋值导致不同变量以同一对象为值
- 赋值使 x 原来引用的对象（值对象）现在丧失了来自 x 的引用

在采用引用语义的语言里，如果变量出现在表达式里，在需要使用被引用对象的值时，（编译后的）程序将自动做间接操作（dereference）

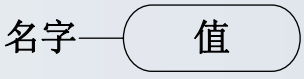
例：如果 a 是 Java 的数组变量（a 的值是数组），可以直接写 a[2]。想想 a[2] 的意思



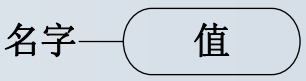
变量的语义模型

不同语言采用了不同的变量语义模型。例如：

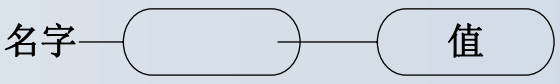
C：彻底的值语义，变量的所有属性都是静态的，只供编译时使用



Java：基本类型的变量采用值语义，无动态属性信息

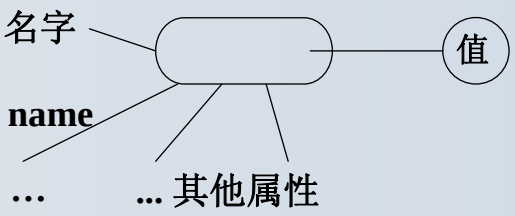


对象类型变量采用引用语义，值对象约束了部分属性信息



Perl 等一些脚本语言：引用语义，值对象包含所有类型信息

Lisp：类似变量的实体称为符号，属性关联关系见图。值对象本身有自己的属性约束



引用变量和引用参数

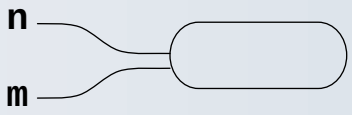
一些语言提供了引用变量，例如 C++：

```
int n = 1, &m = n;
```

许多语言里的函数/过程可以有引用参数，例如：

```
int fun (int &n, …) { … }                    C++
procedure proc (var n : integer, …) …        Pascal
```

引用变量定义应看作给一个对象的另一个名字：



使用 m 等同于使用 n

n 和 m 是同一对象的不同名字

没有对引用变量本身的操作，只能通过它去操作被引用的对象

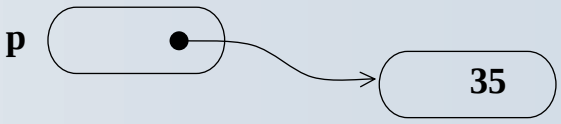
引用参数（变量参数）就是在函数里为实参（必须是程序对象）建立一个局部的名字，通过这个名字可以直接访问实参对象（可能是个变量）

指针变量

指针变量是以变量（对象）标识为值的变量（由于常以变量的内存地址作为其标识，所以常说指针变量以“地址”为值。也有其他情况）

引进指针变量，使地址变成了可以操作的数据（可用于赋值等）

```
int *p = new int(35);
// C++的写法
```



理解指针变量，一个关键问题是分清是对指针变量本身的操作还是对被指对象的操作。指针变量的 dereference （间接，取被指）需要在程序里明确描述，以区分两种不同操作（两种情况都可能作为左值或右值）

C 语言用 p 和 *p 区分这两种情况，Pascal 用 p 和 p^ 区分两种情况

指针很像引用，但其本身也可以操作。值语义的语言里用“指针” 模拟引用语义，支持动态存储管理，建立复杂数据结构。早期语言（Fortran, Algol 等）没有指针。采用引用语义的语言可以没有指针（如 Java）

别名

如果与一个对象约束的名字多于一个，就是出现了别名（alias）

出现别名时，程序正文中看起来不同的名字实际引用着同一个对象

别名使程序理解更加困难，人或系统（如编译器）更难推断程序的行为（难做分析和优化），也大大增加了为程序建立严格语义理论的难度

一些情况：

局部参数与全局变量成为别名

```
int m;
void f(int &n) {
  n++;
  m += n*n;
}
```

```
.. m = 3;
   f(m);
...
```

不同的引用参数相互成为别名

```
void g(int &a, int &b) {
  b = a + 3;
  a += 5;
}
... g(m, m); ...
```

一些语言设计者认为别名是万恶之源，Euclid 语言（1977-1978年）的主要设计目标之一是通过静态检查禁止别名

别名分析在编译中有重要作用

别名

一类重要“别名”问题是数组元素引用，如

```
a[i] = a[j] + a[k];
```

其中a[i]、a[j]、a[k]也可以看作对象的名字，它们是否表示同一个元素的问题完全是动态确定的，仅用静态检查不可能完全确定

广义的，别名是源程序里的两个不同对象描述，在运行中某个时刻实际指称同一个对象，或者说，同一对象可以通过不同的描述形式访问

- C++ 的引用变量就是引进别名
- 可以认为指针也是引进别名的机制，例如

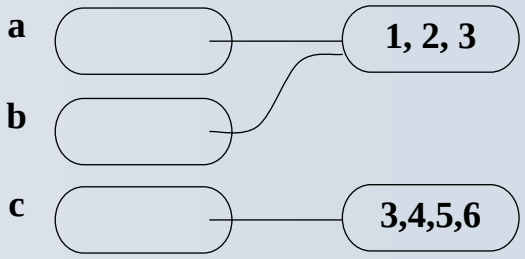
```
int *p = new int(3);
int *q = p;
... ...
*q = 4;
cout << *p;
```

这里看不到赋值会影响与 p 有关的环境情况

别名

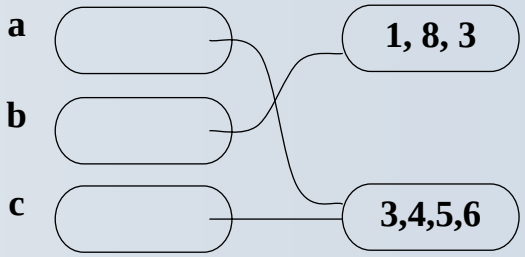
引用语义中的赋值结果是值共享，导致别名：

```
int[] a = {1,2,3};
int[] b = a;
int[] c = {3,4,5,6};
```



赋值 a[1] = 8; 将使 b[1] 的值也变成 8

赋值 a = c; 导致 a 引用 c 的值，b 不变

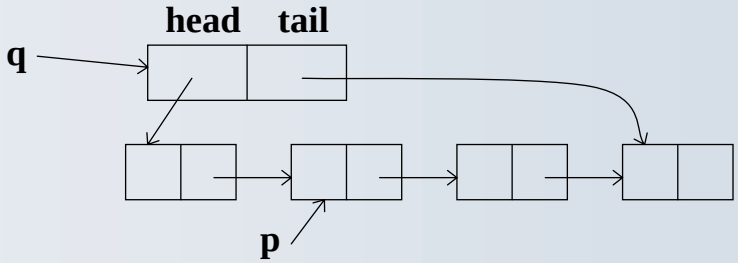


在引用语义中，如果需要避免值共享，就必须做值对象本身的拷贝。例如在 Java 里需要写 `a = (int[])c.clone();`

别名

由指针而产生的值共享（和别名）的情况很常见，而且很有用

很多时候需要值共享，例如建立链接队列，遍历一个链表（指针值共享）



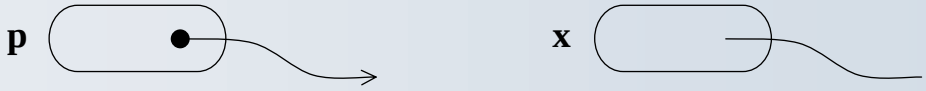
指针值共享是别名的一种典型情况，在采用值语义的语言里不可避免。在采用引用语义的语言里，复杂数据结构里也广泛存在别名的情况

问题：当从一个指针（或引用）找到一个对象时，一般无法知道有没有指向这个对象的其他指针（除非程序员头脑中有非常清晰的编程模型）

在操作复杂数据结构时，指针间的链接关系可能导致非常复杂的别名局面，这可能给存储管理带来很大困难。下面会看到这种情况的影响

悬空引用

悬空引用（**dangling reference**）：程序执行中的某个时刻，处于活动状态的指针变量或引用变量没有引用到合法的对象



出现悬空引用的情况：

```
T *p;
...
p = (T*)malloc(sizeof(T));
...
free(p);
... *p ... // 危险!
{
  T n = ...;
  p = &n;  // p生存期比n长
}
... *p ... // 危险!
```

```
T* fun(...) {
  T n;
  ...
  return &n;
}
... { ...
  T *p = fun(...);
  .. *p .. // 危险!
}
```

出现后两种情况，都因为可以取地址

悬空引用

悬空引用是程序中的极端危险状态

- 从悬空引用出发间接读取，得到的是非法数据
- 对悬空引用间接赋值，可能造成无法预计的破坏

定义后没有初始化的指针变量也处于悬空状态，如 C：

- 全局指针自动设置为空，间接访问这种指针通常会引发操作系统异常
- 局部指针不自动初始化，处于非法状态，间接访问可能引发操作系统异常，或导致难检查的错误（如修改不明数据，甚至系统数据结构）

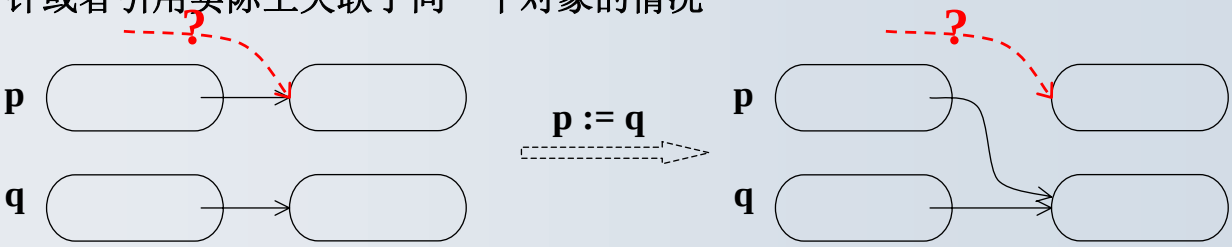
这也是一些语言提供变量自动初始化机制的重要原因

通过语言设计可以防止一些悬空引用（不能完全避免）。例如：

- 多数语言里没有取地址操作，因此不存在由于生存期导致的悬空引用
- 禁止把生存期短的对象的地址赋给生存期长的指针，可减少悬空引用

悬空引用与废料（垃圾）

产生悬空引用的最大问题是复杂数据结构里的值共享：其中可能出现多个指针或者引用实际上关联于同一个对象的情况



给 p 赋值前能不能释放 p 原来指向的对象？

- 如果存在引用同一对象的其他指针，释放就会造成悬空的引用
- 如果不存在引用同一对象的其他指针，不释放就会造成一块实际上已经无法在程序里使用的存储丢失了（产生了废料） （两难）

产生废料的结果是可能使长期运行的系统由于空间“耗尽”而被迫终止

可采用一些编程规则管理存储，但这仍是复杂系统实现中的巨大负担

废料

为防止悬空引用，一些语言根本不提供 **delete/release/free** 一类操作

如果没有存储释放操作，悬空引用问题就只剩下“空指针间接”，而“空指针”是明确的状态，可在程序里检测（非空指针悬空无法检测）

废料（**garbage**，垃圾）：丧失了引用途径，已经不可能在程序里访问，但仍然占据着存储，没有释放的“死对象”

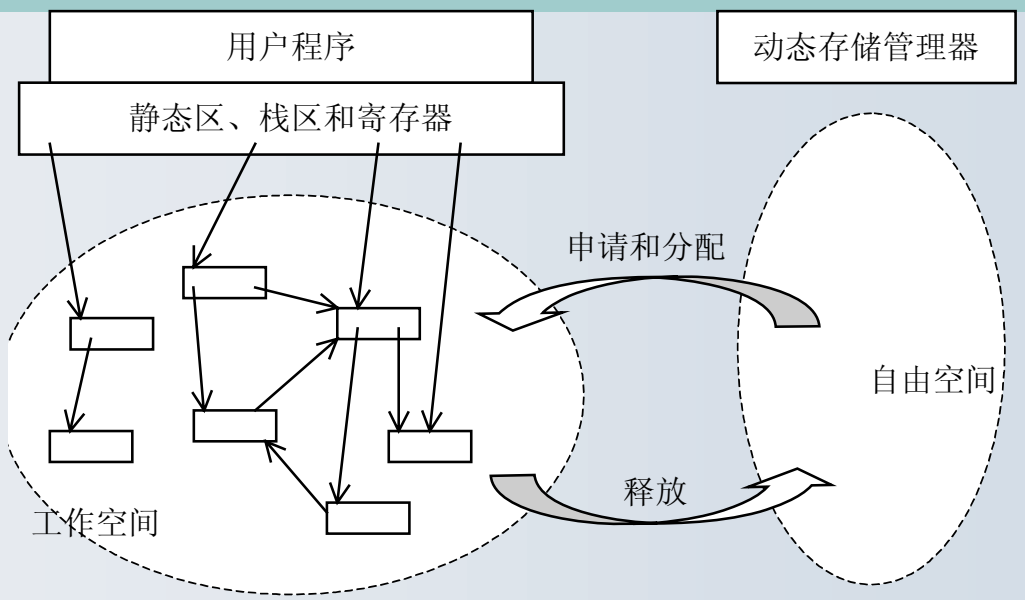
在基于值语义并有指针机制的语言里，由于程序员工作不当或疏忽，或由于情况复杂不能确认操作正确而不敢释放，都可能产生废料

在引用语义的语言里所有赋值都是修改引用，要求程序员去确认某个操作将使对象丢掉所有引用，对编程是巨大负担，实际上根本做不到

人们开发了一些防止废料产生的技术：

- C++ 把堆对象关联于存在期明确的对象（栈对象），基于后者，通过自动执行的销毁动作（析构函数）释放动态分配的对象
- 引用计数技术，运行中维护对每个堆对象的引用数，计数为 0 时释放

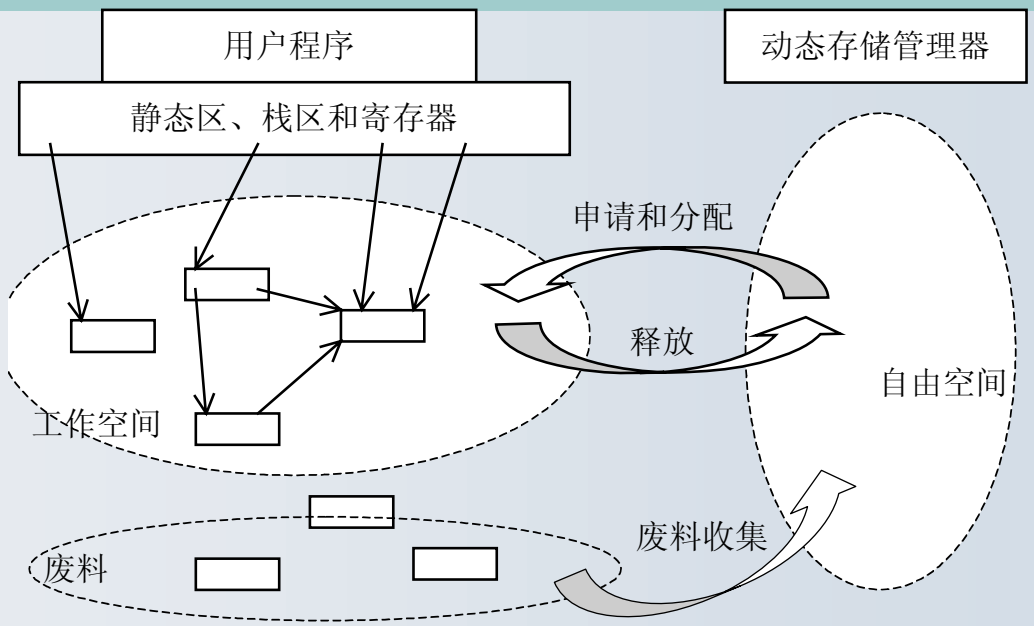
堆区的分配与释放（理想状态）



程序运行中，堆区分为自由空间（未使用的空闲空间）和工作空间两部分

随着存储申请和释放，存储块在两个区之间移动（对象创建和销毁）。理想情况是完美的释放：工作空间中每个对象都存在访问路径，无废料

废料和废料收集



随着程序运行，有些堆对象可能丧失所有的访问路径，成为废料

废料收集（garbage collection，GC，垃圾回收）是程序运行环境里提供一种自动机制，它能自动识别废料，并把它们送回自由空间

废料收集（GC）

废料收集技术起源于 Lisp，最早由 McCarthy 等为实现 Lisp 而开发

废料收集的主要技术：

- 标记-清扫：通过引用关系标出工作空间中全部活对象，收集其余对象
- 复制式：把活对象及其连接关系复制到另一存储区，原区全部收回
- 分代式：复制式的改进，利用对象的生存特性提高GC效率

GC 一直受到实际程序员的排斥，认为其时空开销太大，可能导致程序执行中无法预期（时刻/长短）的停顿。典型：C++ 设计者很反对废料收集

近年来，由于：

- 处理器速度提高，内存扩大，废料收集的开销已经不是大问题
- 在系统开发中完美地管理存储，已经成为程序员最大的负担之一
- OO 的广泛应用，没有废料收集，开发 OO 程序的代价会大大提高

最新的程序语言都提供了自动废料收集（Java，C#，各种新脚本语言）

变量的值模型和引用模型（总结）

对变量的值模型和引用模型的总结

- 值保存在变量的存储区里，实现简单，易使用，易理解
- 变量的值实际用另一个值对象表示，实现复杂，需要复杂的存储管理技术的支持；概念的理解也比较困难
- 值可以直接访问，效率高（因此被大多数常规语言采用）
- 需要多做一次间接访问，效率有所降低
- 赋值是值的拷贝，语义清晰，易于理解；但对于大对象可能费时
- 赋值是引用共享，语义较复杂，编程要当心；只修改引用，效率高
- 存储管理方便，可根据变量的作用域，利用栈机制统一进行管理，存储管理简单，管理开销小
- 由于值对象的创建和销毁，复杂的相互引用关系，必须有堆存储管理的支持和废料收集，管理复杂，开销大

变量的值模型和引用模型

- 需要静态确定变量类型，静态完成存储的分配或安排（如自动变量）。需要变量的类型声明和静态处理。带来高的执行效率；但难支持动态确定大小的数组和其他具有动态性质的数据结构，包括字符串等
- 变量可以无类型（通用的变量），许多特征可以不必（不）静态确定。可能带来一些运行时开销；可自然支持动态改变大小的数组和字符串等
- 需要另提供动态存储分配和指针概念，以支持动态数据结构、OO 等高级技术。指针是比引用更危险更难用的机制，是程序错误的重要根源
- 能方便自然地支持各种高级程序设计技术，包括动态数据结构，面向对象的程序设计等等，不需要另外的指针概念
- 在值模型语言的实现里，对某些特殊数据对象，需要在内部采用引用模式实现，例如，动态数组/字符串等。（这是不得已而为之，必须的）
- 在引用模型语言的实现里，为提高效率，基本类型（如各种数值）常采用值方式实现。（只是为了提高效率，不是必须的）