2.1. 类型的概念

2.1.4 类型系统

- 系统：一组相互关联的部件，它们协同工作以实现一个共同的目标。
  - “我们都是来自五湖四海，为了一个共同的革命目标，走到一起来了。”– 毛泽东
  - 在一个软件系统中，不同的类型之间存在着关联，它们的实例在运行时协同工作，以达到设计目标。
- 类型系统：一组相互关联的类型，它们的实例协同工作以实现一个共同的目标。

2.1. 类型的概念

2.1.4 类型系统

- 类型系统可以是开放的，也可能是封闭的。
  - 在Pascal之前的程序设计语言的类型系统是封闭的，程序员能够使用的，只能是基本类型。
- 程序员总是希望他们能够在基本类型以外，定义自己所需要的新的类型，这就要求程序设计语言的类型系统是开放的。
  - 开放的类型系统要提供相应的机制（如类型构造符）以允许程序员定义新的类型（自定义类型），允许对变量的值限制于某种类型加以说明，对程序中自定义类型的表达式能进行类型检验。
  - 每个自定义类型都可以递归地追溯到某个（些）基本类型，因而产生了类型之间的关联。

2.1. 类型的概念

2.1.4 类型系统

- 从类型系统的角度来看，
  - 一个程序设计语言L，它所有基本类型构成了一个类型系统 T_L;
  - 一个用 L 写的程序 P, 它所使用的类型既有 T_L中的类型，也可能有（用枚举、构造或限定的方式定义的）自定义类型，因而构成了另一个类型系统 T_P。
  - 显然， T_P包含了 T_L。
- 从这个意义上讲，我们写的任何一个程序，事实上都是在构造这个程序的类型系统，然后利用它来达到设计这个程序的目的。

2.1. 类型的概念

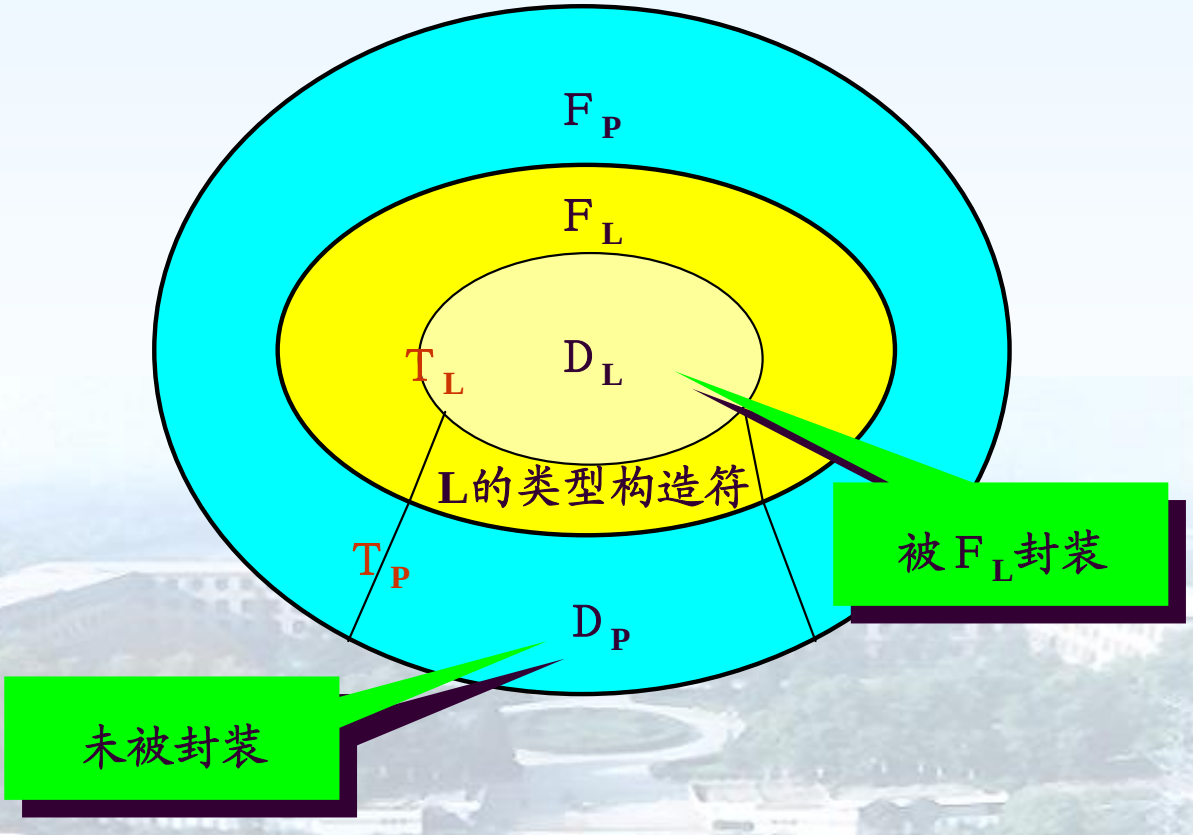
2.1.4 类型系统

- T_L和T_P中的每个类型都有自己的值集和操作集。令
  - D_L表示 L中基本类型之值集定义的集合（由定义了一组值集的那些操作所构成的集合）；
  - D_P表示 P 中自定义类型之值集定义的集合;
  - F_L表示 L中基本类型之操作集的集合;
  - F_P表示 P 中自定义类型之操作集以及扩充了基本类型之操作集的那些操作所构成的集合。

2.1. 类型的概念

2.1.4 类型系统

- 如果 L 是一种不支持数据抽象的语言，则 P 的类型系统的特征是:



2.1. 类型的概念

2.1.4 类型系统

- 这样的特征所导致的后果是什么？看一个例子:

2.1. 类型的概念

现有两种实体：“矩形”（Rectangle）和“旗杆”（Flagpole）。其中，“矩形”的属性为长度与宽度，“旗杆”的属性为高度（指地面以上）和深度（指地面以下），且均为整数。若有：

```
struct Rectangle { int length, width; };
struct Flagpole  { int height, depth; };

int rectangleArea  (int x, int y) { return (x >= 0 && y >= 0) ? x*y : 0; }
int flagpoleLength (int x, int y) { return (x >= 0 && y >= 0) ? x+y : 0; }
void f()
{
  struct Rectangle rect;
  struct Flagpole  flgp;
  rect.length = 20;  rect.width  = 5;  flgp.height = 20;  flgp.depth  = 5;



}
```

试续写函数 f，以各举一例说明，对函数rectangleArea和flagpoleLength的调用都可能出现语法上合法，但结果与函数的语义不一致的情况，并指出其原因。

2.1. 类型的概念

现有两种实体：“矩形”（Rectangle）和“旗杆”（Flagpole）。其中，“矩形”的属性为长度与宽度，“旗杆”的属性为高度（指地面以上）和深度（指地面以下），且均为整数。若有：

```
struct Rectangle { int length, width; };
struct Flagpole  { int height, depth; };

int rectangleArea  (int x, int y) { return (x >= 0 && y >= 0) ? x*y : 0; }
int flagpoleLength (int x, int y) { return (x >= 0 && y >= 0) ? x+y : 0; }
void f()
{
  struct Rectangle rect;
  struct Flagpole  flgp;
  rect.length = 20;  rect.width  = 5;  flgp.height = 20;  flgp.depth  = 5;
  int area = rectangleArea(flgp.height, flgp.depth);
  int length = flagpoleLength(rect.length, rect.width);

}
```

原因：从语义看是自定义类型的操作，而从语法上看是基本类型的操作。

试续写函数 f，以各举一例说明，对函数rectangleArea和flagpoleLength的调用都可能出现语法上合法，但结果与函数的语义不一致的情况，并指出其原因。

2.1. 类型的概念

2.1.5 类型化

- 类型化：用类型系统来检查、管理和控制对应的程序。
- 类型化是程序设计语言的一种内在属性，是支持应用程序中的类型和类型系统的。没有这种支持（例如没有类型定义机制和类型检查机制），应用程序中就不可能有类型。

2.1. 类型的概念

2.1.5 类型化

- 从类型化特征来看，目前的程序设计语言主要有两类：
  - 静态类型化语言：它的每一表达式的类型在静态程序分析（例如编译）时是确定的。在所有的程序构造中，每个类型在编译时是已知的，所有的类型参数都是编译时的参数。
  - 强类型化语言：它的每个表达式保证是类型一致的。即允许有些表达式的类型本身是静态未知的，但可以用引入运行时的类型检验（或推理）来做到。
- 静态类型化语言一定是强类型化语言，而逆命题则不一定成立。

2.1. 类型的概念

2.1.5 类型化

例如，Pascal中的变体记录类型：

```
TYPE Gender = (Masculine, Feminine);
     Name = record
              case Kind : Gender of
                Masuline : (He : MasulineName);
                Feminine : (She: FeminineName);
            end
```

在编译时,类型 Name 的一个分量名及其类型是不知道的，只知道界，就是该分量名将取自{ He, She }，对应的类型将是MasulineName或者FeminineName。对于

```
var x : Name;
f(x.He);     { procedure f(n) var n : Name; ... }
... ...
f(x.She);
```

我们看到，两次调用同一个过程 f 的实参的类型是不一样的，分别为 MasulineName 和 FeminineName，静态无法判定这样的调用是否合法，要由语言的运行时类型检验机制依据当时的 x.Kind 的值（Masuline 或 Feminine）在运行时才能作出判定，而f(x)方式的调用则是静态非法的。因此， Pascal不是静态类型化而是强类型化的。

2.1. 类型的概念

2.1.6 类型系统的作用

- 抽象作用（分类地表示客观世界中的事物）：
  - 允许通过对数据（或对象）的命名和对数据特定性质的认定，以组织和表达一个问题的求解。一个类型系统使得人们可以把自己的注意力集中于某种（些）类型的事物上。
- 限制作用（防止基本逻辑单元之间交互作用的不一致性）：
  - 可以帮助人们发现错误。当一个操作接收的值不属于规定的类型，编译程序就会检测出类型错误。

2.2. 程序设计语言中的类型

基于类型系统的观念来看程序设计

- 重申：程序设计的过程，实质上是针对应用的特征，构造对应程序的类型系统的过程。
- 但是，构造类型系统，不应当简单地理解成将所有的自定义数据结构冠以类型名称，加上一堆无组织的函数和过程就行了。因为程序设计语言本身并不能保证自定义类型的值集与操作集的确定是合理的，也一般不能保证操作的结果是封闭在值集中的。

2.2. 程序设计语言中的类型

突出的问题在于类型中操作集的设计

- 哪些操作应当属于同一个类型？
- 这些操作中，哪些是关于值集中单个元素的操作，哪些是关于多个元素的操作？
- 这些操作是否对值集的所有元素是一致的？
- 这些操作中，哪些是在设计该类型时就必须考虑的，哪些是可以在以后再加入该类型，而又与以前的操作是相容的？

在面向对象程序设计中，这些是不能回避的问题。

2.2. 程序设计语言中的类型

How? 向程序设计语言学习类型系统的定义方法

- 任何一个商品化的、哪怕是不起眼的程序设计语言，都是一个精巧的软件范例，都值得我们认真学习。
- 学习程序设计语言中的类型系统，是因为在面向对象程序设计中采用的观念是类似的:
  - 对于一个类型，在设计时首先考虑的是它应该有什么操作，而不是现在需要有什么操作。

2.2. 程序设计语言中的类型

例：C++语言中与类型相关的成分

- 在用C++语言写的任何一个程序中，所出现的每一个
  - 标识符（*identifier*），如变量（*variables*）、常量（*constants*）、函数（*functions*）等，
  - 字面值（*literals*），
  - 运算符/操作符（*operators*），
  - 以及以上元素的合法组合，如表达式（*expressions*）等，

  都有特定的类型。
- 在一个程序中，每个类型有着不同的名称。

2.2. 程序设计语言中的类型

程序设计语言中类型的作用

- 决定变量的存储空间的大小;
- 决定变量或常量的合法取值范围（值集）;
- 决定合法的（一般是用运算符或函数表示的）操作的范围（操作集）;
- 区分名字相同、类型不同的符号（变量、函数、运算符）;
- 决定将一种类型的值能否和如何转换成另一种类型的值;
- 为编译程序提供依据，令其检查出程序中的一部分错误。

2.2. 程序设计语言中的类型

C++语言中的声明机制

- Before a name (identifier) can be used in a C++ program, it must be declared. That is, its type must be specified to inform the compiler to what kind of entity the name refers.

  （在能够在一个C++程序中使用一个名字（标识符）之前，必须先声明这个名字。这就是说，必须指明这个名字的类型，以通知编译程序这个名字所指的是哪一种实体。）

2.2. 程序设计语言中的类型

C++语言中声明机制的特征

- 声明分为定义声明和非定义声明两种。
- 定义声明为所定义的名字进行的动作是:
  - 确定类型;
  - 分配存储空间;
  - 绑定（*binding*，由编译程序将一个名字与它对应的存储空间关联在一起的动作）;
  - 按照语言既定的省缺值（*default value*）或者声明中给出的初始化动作（如果有的话）设置对应变量的初值（*initial value*）。
- 这些动作统称为对应类型的一次实例（*instance*）生成。
- 定义声明也可以用来定义新的类型或者新的类型名。

2.2. 程序设计语言中的类型

C++语言中声明机制的特征

- 非定义声明是为那些还没有定义声明（或者在其他地方已定义声明）却需要在这里使用的名字，进行“先承认再核实”的声明。
- 用户定义的所有名字都必须有对应的定义声明。

2.2. 程序设计语言中的类型

定义声明

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

2.2. 程序设计语言中的类型

非定义声明

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

2.2. 程序设计语言中的类型

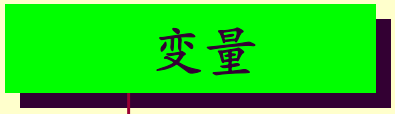
例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

2.2. 程序设计语言中的类型

例：




```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

2.2. 程序设计语言中的类型

例：

字面值

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

2.2. 程序设计语言中的类型

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

变量初始化

2.2. 程序设计语言中的类型

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

函数名

2.2. 程序设计语言中的类型

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```

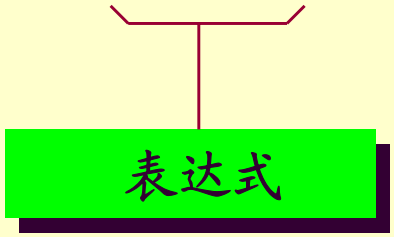
运算符

2.2. 程序设计语言中的类型

例：

```
// 符号的定义（声明）：
float x;
int   y = 7;
float f(int);

// 符号的使用：
x = y + f(2);
```



2.2. 程序设计语言中的类型

例：C++语言中基本类型的种类

- C++ has a set of fundamental types corresponding to the most common basic storage units of a computer and the most common ways of using them to hold data:
  - Boolean（布尔）type
  - Character（字符）types
  - Integer（整数）types
  - Floating-point（浮点数）types
  - Void type

2.2. 程序设计语言中的类型

例：C++语言中自定义类型的种类

- A user can define:
  - Enumeration（枚举）types
- From these types（指基本类型和已有的自定义类型）, we can construct other types:
  - Pointer（指针）types
  - Array（数组）types
  - Reference（引用）types
  - Data structure（结构）and classes（类）

2.2. 程序设计语言中的类型

2.2.1 可枚举类型

- 可枚举类型由元素有限的值集构成。
- 值集的描述:
  - 枚举法
  - 限定法

2.2. 程序设计语言中的类型

2.2.1 可枚举类型

- 可枚举类型由元素有限的值集构成。
- 值集的描述:
  - 枚举法
  - 限定法

C语言:

```
enum keyword { ASM, AUTO, BREAK };
```

Pascal语言:

```
type month =
   (Jan, Feb, Mar, Apr, May, Jun,
    Jul, Aug, Sep, Oct, Nov, Dec);
```

2.2. 程序设计语言中的类型

2.2.1 可枚举类型

- 可枚举类型由元素有限的值集构成。
- 值集的描述:
  - 枚举法
  - 限定法

```
Pascal语言:
type summer_month = (Jun,..., Aug);
```

要求大值集上存在全序

2.2. 程序设计语言中的类型

2.2.1 可枚举类型

- 可枚举类型的值集的元素数量是有限的。
- 应当支持的基本操作:
  - 相等（如: ==: bool × bool→bool）
  - 赋值（如: =: bool→bool）
  - 若值集上存在全序，还应支持:
    - 前趋（如: pred: month→month）
    - 后继（如: succ: month→month）

2.2. 程序设计语言中的类型

例：C++语言中的可枚举类型Enumeration

- **Enumeration**是用户以枚举值集的方式定义的类型。
- **Enumeration**值集中的每个值都是一个已命名的整数常量（*named integer constant*），这个名字就是**Enumeration**类型的一个字面值。

例：

```
// 若不指明对应的整数值，从0开始递增
enum keyword { ASM, AUTO, BREAK };

// 也可以指明对应的整数值
enum summer_month { Jun = 6, Jul = 7, Aug = 8 };
```

2.2. 程序设计语言中的类型

例：C++语言中的可枚举类型Enumeration

- Enumeration类型的主要用途，是以易于理解、易于控制的方式表示程序中需要区分的状态：

```
/* 既不容易理解、也不容易控制的
   状态表示 */
void getAJob( int p )
{
  switch (p) {
    case 0:  // 0代表程序员
      // 程序员求职处理
      break;
    case 1:  // 1代表软件工程师
      // 软件工程师求职处理
      break;
  }
}
```

```
enum Position
  { programmer, software_engineer };
void getAJob( Position p )
{
  switch (p) {
    case programmer:
      //程序员求职处理
      break;
    case software_engineer:
      //软件工程师求职处理
      break;
  }
}
```

2.2. 程序设计语言中的类型

2.2.2 布尔类型

- 值集的描述：枚举法。
  - $\mathbf{V_{bool}}$ **= { true, false }**
- 应当支持的基本操作：
  - **and: bool × bool → bool** 语义：x and y = if x then y else false
  - **or : bool × bool → bool** 语义：x or y = if x then true else y
  - **not: bool → bool** 语义：not x = if x then false else true
  - **imp: bool × bool → bool** 语义：x imp y = if x then y else true
  - **equiv: bool × bool → bool** 语义：x equiv y = if x then y else not y

2.2. 程序设计语言中的类型

2.2.3 字符类型与数类型

- 值集的描述：限定法或构造法。
- 应当支持的基本操作：
  - 赋值
  - 等价操作与关系操作（如：==、!=、>、>=、<、<= ）
  - 常规整数运算和实数运算操作

2.2. 程序设计语言中的类型

例：C++语言中的字符类型

- 字符类型的名称：char（unsigned char）； signed char; wchar_t。
- char类型和signed char类型的值集：ASCII字符集（*ISO 646*）。
- wchar_t类型的值集：Unicode字符集（*ISO 10646*）。
- char类型值的存储空间（一般是1个字节）。
- 字符类型的值可以承受算术运算和逻辑运算。这时，字符类型的每个值被转换成对应的一个整数（char类型为0~255，signed char类型为-127~127）。
- 字符字面值的类型为char，是用单引号括起来的单个字符，或者是用单引号括起来的、前加转义符\的单个字符。

2.2. 程序设计语言中的类型

例：C++语言中的字符类型

例：

```
char  c;
unsigned char tab = '\t';
bool  b;

c = 'A';
b = c == 'A';              // true
b = c == tab;              // false
b = (c + 1) == 'B';        // true
b = c > 'b';               // false
```

2.2. 程序设计语言中的类型

例：C++语言中的整数类型

- 整数类型的名称：**int（signed int）； unsigned int； short（signed short）； unsigned short； long（signed long）； unsigned long。**
- 整数类型的值集：整数集合的子集。
- 整数类型值的存储空间与平台（*platform*，通常指特定的硬件和操作系统）相关，典型的存储空间分配量是：**short—2 字节； int—4 字节； long—8 字节。**
- 整数类型的值可以承受算术运算和逻辑运算。
- 整数字面值是由+、-、数字组成的串，前加**0**为八进制，前加**0x**为十六进制；后加**U**为无符号整数字面值，后加**L**为长整数字面值。

2.2. 程序设计语言中的类型

例：C++语言中的整数类型

例：

```
int   i, j, k;
long  n = 123456789L;
unsigned short s;

i = 0x53;      // i的值为83
j = i - 010;   // j的值为83-8=75
n = n - i + j;
s = 1;         // s的值为0000000000000001（二进制）
s = s << 3;    // 左移三位后s为0000000000001000
k = s;         // k的值为8
```

2.2. 程序设计语言中的类型

例：C++语言中的浮点数类型

- 浮点数类型的名称：**float**；**double**；**long double**。
- 浮点数类型的值集：实数集合的子集。
- 浮点数类型值的存储空间与平台相关。
- 浮点数类型的值可以承受算术运算和逻辑运算。
  - 应特别注意误差。例如，在表达式中直接判两个浮点数类型的值相等或不相等是不可靠的，一般应判二者之差的绝对值小于或大于指定的误差值。
- 浮点数字面值是由+、-、小数点、数字和 **e**（表示指数）组成的串，后加 **f** 或 **F** 为**float**类型，后加 **l** 或 **L** 为**long double**类型，后面什么也不加为**double**类型。

2.2. 程序设计语言中的类型

例：C++语言中的浮点数类型

例：

```
long  n = 123456789L;
float f;
double d;
long double d2;

f = 1234567.89f;
d2 = f / 1.23e5L;   // 1234567.89/123000.0
d = n + 1000000L;   // d的小数部分未必为0
f = f + 0.0000001; // 精度的限制可能使f的值不变
```

2.2. 程序设计语言中的类型

2.2.4 void类型

- 值集的描述：ϕ。
- **void**类型用来表示不返回值的函数（过程），以及构造出不限制所指对象类型的指针类型（**void** *），来指向类型不限或类型不可预知的对象。
- 应当支持的基本操作:
  - 指针赋值。

2.2. 程序设计语言中的类型

2.2.5 结构化类型之一：数组

- 结构化类型是指把多个数据组织在一起、当做一个单元对待的类型。
- 数组：变量的有序集合。
- 若数组中所有元素的类型相同，叫做同质数组，否则称为异质数组。常用的语言中只允许使用同质数组。
- 一数组的所有元素各取一值所构成的一个多重集，构成了该数组的一个值。

2.2. 程序设计语言中的类型

2.2.5 结构化类型之一：数组

- 一个数组类型 a 由元素类型 t 和索引类型 i 组成，其中：
  - t 是任意类型，
  - i 的值集上须存在全序，通常是整数类型或可枚举类型。
- 数组类型 a 的值集：$V_a = \underbrace{V_t \times V_t \times \ \ldots \ \times V_t}_{|V_i|}$
- 数组类型 a 应当支持的基本操作：
  - 取数组元素：[ ]: a × i→t

2.2. 程序设计语言中的类型

例：C++语言中的数组类型

- For a type t, t[*size*] is the type “array of *size* elements of type t.” The elements are indexed from 0 to *size*-1.
- 数组类型和数组类型的变量（后者通常简称为数组），一般在定义声明中是同时声明的。

例：

```
float v[3];       //      float[3] v;
char* a[32];      //    (char*)[32] a;

int   d2[10][20]; // (int[10])[20] d2;
```

2.2. 程序设计语言中的类型

例：C++语言中的数组类型

- For a type t, t[*size*] is the type “array of *size* elements of type t.” The elements are indexed from 0 to *size*-1.
- 数组类型和数组类型的变量（后者通常简称为数组），一般在定义声明中是同时声明的。

例：

```
float v[3];       //      float[3] v;
char* a[32];      //    (char*)[32] a;

int   d2[10][20]; // (int[10])[20] d2;
```

声明数组类型

声明数组变量

2.2. 程序设计语言中的类型

2.2.6 结构化类型之二：记录

- 记录也是多个值的组合，它的组成分量叫做域（*fields*），可以通过域的名字显式地访问。记录中的域可以有不同的类型。
- 记录中各域的类型都可以追溯到某个或某些基本类型。
- 记录可以看成是用域名作为索引的异质数组。
- 记录类型的值集由各域之类型的值集的笛卡儿积构成。
  - 记录类型的一个值是由该记录类型各域的值组成的一个N元组（*N-tuple*，N是域的个数）。

2.2. 程序设计语言中的类型

例：C++语言中的记录类型－结构

- A struct is an aggregate of elements of the (nearly) arbitrary types.（其元素是异质的（*heterogeneous*））
- 对比：An array is an aggregate of elements of the same type.（其元素是同质的（*homogeneous*））
- 构成一个结构的元素（域）称为其成员（*members*）。
- 结构被用来定义有多种类型不同的属性（*attributes*）的客观事物（这样的事物是大量存在的）。
- C++中的结构实际上是特殊的类。

2.2. 程序设计语言中的类型

例：C++语言中的记录类型－结构

```
例：
struct Pair {
  string name; // 名字
  double val;  // 同名字的数量或其他
};             // 这里的分号（;）不能缺少

struct address { // 定义某人的地址
  char* name;       // 某人的姓名，假定最长为20个字符
  int   number;     // 门牌号码，假定int为4字节
  char* street;     // 街名，假定最长为20个字符
  char* town;       // 城市名，假定最长为14个字符
  char  state[2];   // 州/省缩写名
  int   zip;        // 邮政编码，假定int为4字节
};                  // 这里的分号（;）不能缺少
```

2.2. 程序设计语言中的类型

例：C++语言中的记录类型－结构

- 结构的每个成员都有相应的类型。
- 一个结构看上去只是定义了一种事物的一个实例的各个属性（用成员表示），但根据各成员类型的值集可以构造出这个结构的值集（这是一种典型的抽象）。
- 例如，结构**address**的值集 $\mathbf{V}_{\mathbf{address}}$ 是这样构造出来的：

2.2. 程序设

例：C++语

```
struct address {
  char* name;       // 某人的姓名，假定最长为20个字符
  int   number;     // 门牌号码，假定int为4字节
  char* street;     // 街名，假定最长为20个字符
  char* town;       // 城市名，假定最长为14个字符
  char  state[2];   // 州/省缩写名
  int   zip;        // 邮政编码，假定int为4字节
};
```

- 结构的每个
- 一个结构看
性（用成员表示），但根据各成员类型的值集可以构造出这个结构的值集（这是一种典型的抽象）。
- 例如，结构address的值集 $V_{address}$ 是这样构造出来的：

$$V_{address} = V_{(name)} \times V_{(number)} \times V_{(street)} \times V_{(town)} \times V_{(state)} \times V_{(zip)}$$

$$= V_{char[21]} \times V_{int} \times V_{char[21]} \times V_{char[15]} \times V_{char[2]} \times V_{int}$$

$$|V_{address}| = 256^{20} * (2^{31}-1) * 256^{20} * 256^{14} * 256^{2} * (2^{31}-1)$$

$$\approx 2^{510} \approx 10^{155}$$

2.2. 程序设计语言中的类型

例：C++语言中的记录类型－结构

- C++提供了取结构之指定成员值的两个操作（设 s 为任意一结构，n 为 s 的成员名所构成的集合，$t_{s,n}$ 为在给定 n 一个元素后对应的 s 成员的类型）：
  - member selection（using an object）.　.：$s \times n \rightarrow t_{s,n}$
    - 返回第一操作数中名字与第二操作数相同的成员的值
  - member selection（using a pointer）.　->：$s^* \times n \rightarrow t_{s,n}$
    - 返回第一操作数所指向的那个对象中，名字与第二操作数相同的成员的值。

2.2. 程序设计语言中的类型

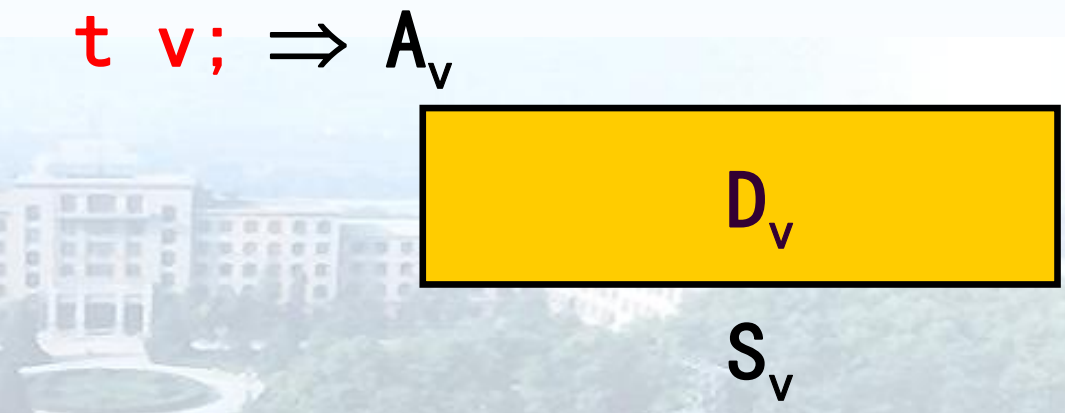
2.2.7 指针类型

- 指针类型：其值是存储对应数据的地址的类型。
- 指针只是表示被访问的对象，而不代表那个对象。
- 一个指针类型必然与另一个类型 t 相关（用 t 构造而成），故该指针类型被记为 t*。t 本身也可以是另一个指针类型。
- 指针类型 t* 的值集，是所有存储类型 t 的值的那些合法地址构成的集合。
  - 这便是C/C++的初学者对指针类型产生恐惧的根本原因：所有指针类型的值的结构是相同的，C/C++在编译时又不进行指针值的合法性判断，若运行时指针类型的值失控，就……“杯具了”

2.2. 程序设计语言中的类型

指针与指针类型

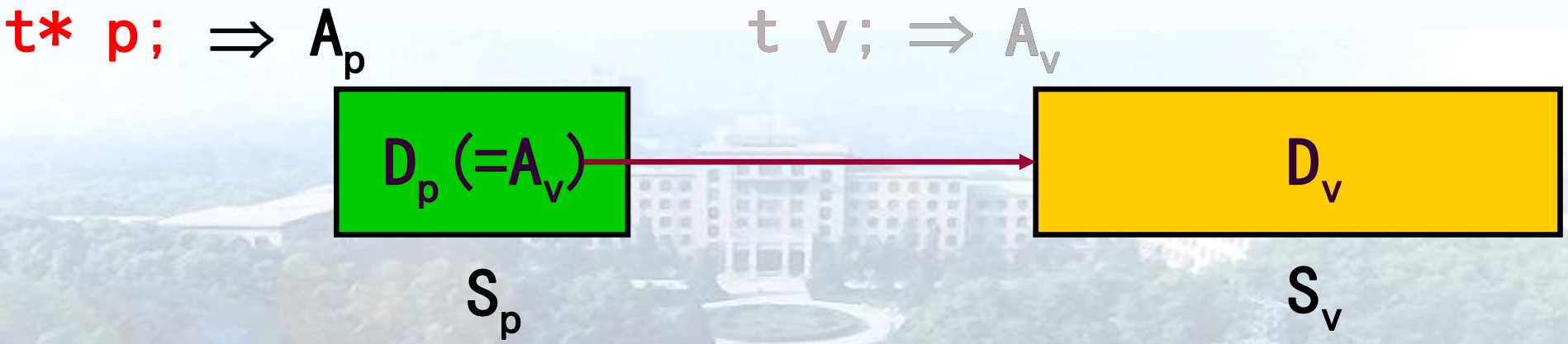
- 类型为 t 的变量 v 对应着一块由 t 决定大小的存储空间 S_v；在编译程序绑定后，用 v 的名字可找到 S_v 的起始地址 A_v，进而访问到 S_v 中所存储的 v 的值 D_v。



2.2. 程序设计语言中的类型

指针与指针类型

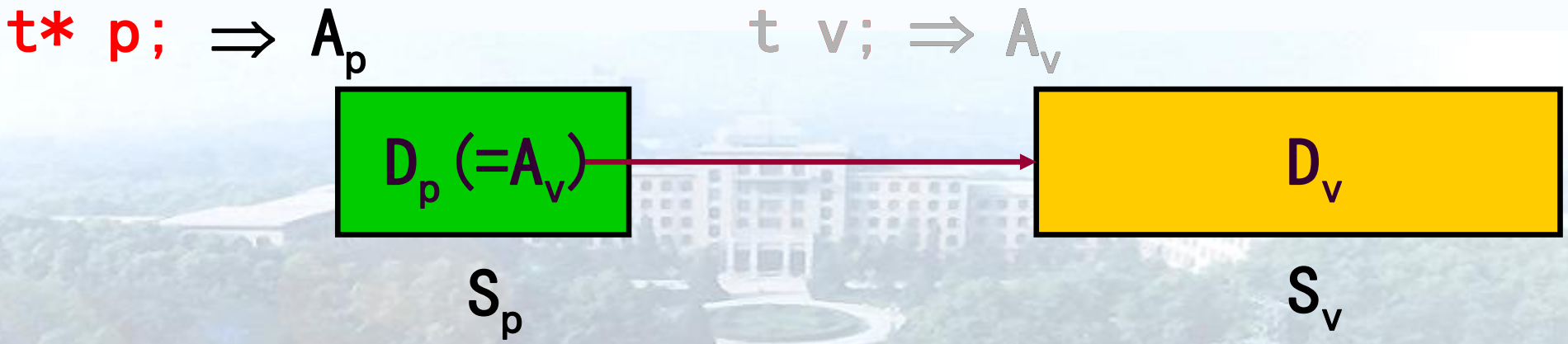
- 焦点是设法得到A_v，并知道S_v的长度（Why?）。
- 如果把 A_v 存储在另一个变量 p 中，用 p 的名字可找到对应空间 S_p 的起始地址 A_p，进而可访问到所存储的 p 的值 $D_p = A_v$，再通过 A_v，就可在不知道 v 的名字的情况下通过 p 间接访问到 D_v。



2.2. 程序设计语言中的类型

指针与指针类型

- p 称为指针变量，指针是指针变量和指针常量的统称。p 的类型是 t*。t* 是一个指针类型，还表明指针所指向的空间（如例中的 S_v）的合法有效长度为 sizeof(t)。可使用的指针类型字面值是 0（通常被定义成 NULL）。



2.2. 程序设计语言中的类型

指针的用途

- 用途 1：有时，需要用同一存储空间 S_v 分时存储多个同类型的值（过去的主要目的是节省存储空间，现在的主要目的是保证这些值之间的互斥），这时可用指针。

2.2. 程序设计语言中的类型

指针的用途

- 用途 1：有时，需要用同一存储空间 S_v 分时存储多个同类型的值（过去的主要目的是节省存储空间，现在的主要目的是保证这些值之间的互斥），这时可用指针。

2.2. 程序设计语言中的类型

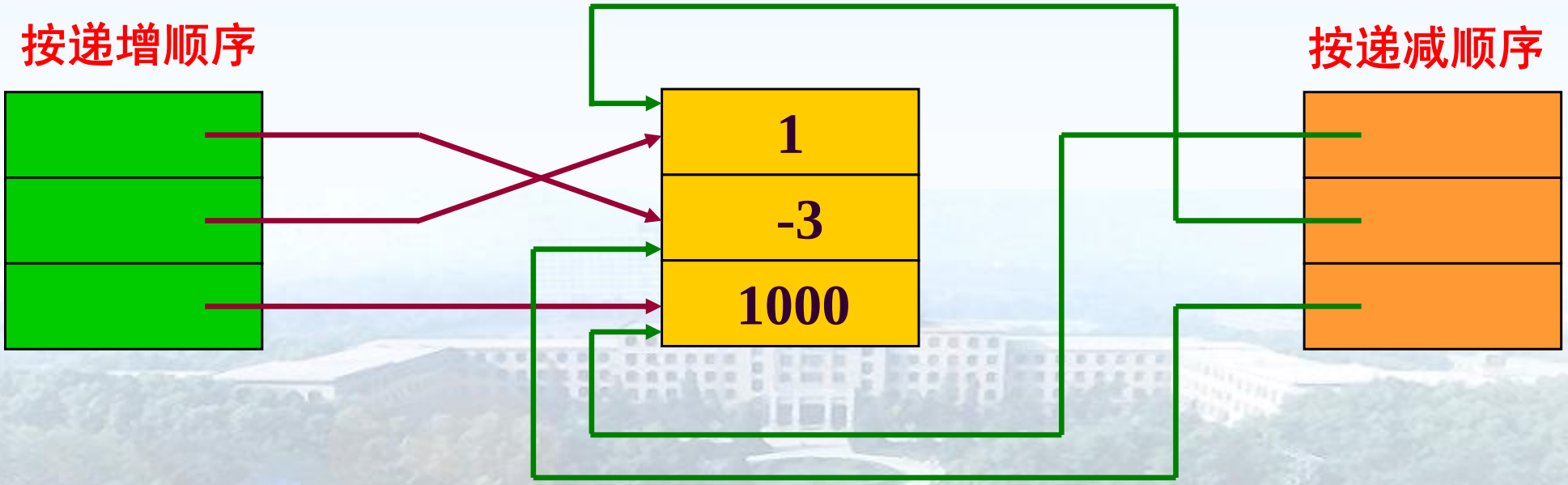
指针的用途

- 用途 1：有时，需要用同一存储空间 S_v 分时存储多个同类型的值（过去的主要目的是节省存储空间，现在的主要目的是保证这些值之间的互斥），这时可用指针。

2.2. 程序设计语言中的类型

指针的用途

- 用途 2：如果需要同时对一组同类型的数据（通常用数组来表示）进行多个侧面/角度的组织，以有效支持多种不同性质的操作，这时可用多个指针数组来实现。



2.2. 程序设计语言中的类型

指针的用途

- 用途 3：对于连续存储着类型为 t 的许多个值（例如数组），当需要依次进行某种处理时，可以不需要知道数组下标的情况下，用改变一个指针变量的值的方式依次进行访问。

2.2. 程序设计语言中的类型

指针的用途

- 用途 3：对于连续存储着类型为 t 的许多个值（例如数组），当需要依次进行某种处理时，可以不需要知道数组下标的情况下，用改变一个指针变量的值的方式依次进行访问。

2.2. 程序设计语言中的类型

指针的用途

- 用途 3：对于连续存储着类型为 t 的许多个值（例如数组），当需要依次进行某种处理时，可以不需要知道数组下标的情况下，用改变一个指针变量的值的方式依次进行访问。

2.2. 程序设计语言中的类型

C/C++中与指针有关的声明

例：

```
// 定义声明（与数据相关的指针）
char*  pc;              // pc:  指向char值的指针
int*   pi;              // pi:  指向int值的指针
char** ppc;             // ppc: 指向“指向char值的指针”的指针
int*   ap[15];          // ap:  由15个指向int值的指针构成的数组

// 定义声明（与函数相关的指针）
int    (*fp)(char*); /* fp:  指向参数为char*类型、返回值为int类
                            型的一类函数的代码的指针 */

// 非定义声明
int*   f(char*);  // f: 参数为char*类型、返回值为int*类型的函数
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **address of.** **&：t→ t***
  - 求出自变量的地址并返回之。

例：

```
char   c = 'a';
char* p = &c;
```

- **dereferencing.** *** ：t*→ t**
  - 用自变量的值作为地址，求出该地址中存储的值并返回之。

例：

```
char   c = 'a';
char* p = &c;
char   c2 = *p;
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **post increment.　++：t*→t***
  - 先返回指针值，再自增 sizeof(t)。
- **pre increment.　++：t*→t***
  - 指针值先自增 sizeof(t)，再返回之。

例：

```
int  v[10], i, j,
     *p = v, *q;

q = p++;
i = *q;  // i == v[0]
j = *p;  // j == v[1]
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **post increment.** **++：t*→t***
  - 先返回指针值，再自增 sizeof(t)。
- **pre increment.** **++：t*→t***
  - 指针值先自增 sizeof(t)，再返回之。

例：

```
int  v[10], i, j,
     *p = v, *q;

q = ++p;
i = *q;  // i == v[1]
j = *p;  // j == v[1]
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **post decrement.** **--： t*→t***
  - 先返回指针值，再自减 sizeof(t)。
- **pre decrement.** **--： t*→t***
  - 指针值先自减 sizeof(t)，再返回之。

例：

```
int  v[10], i, j,
     *p = &v[9], *q;

q = p--;
i = *q;  // i == v[9]
j = *p;  // j == v[8]
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **post decrement.** **--：t*→t***
  - 先返回指针值，再自减sizeof(t)。
- **pre decrement.** **--：t*→t***
  - 指针值先自减 sizeof(t)，再返回之。

例：

```
int  v[10], i, j,
     *p = &v[9], *q;

q = --p;
i = *q;  // i == v[8]
j = *p;  // j == v[8]
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **add(plus).** **+ : t* × int→t***
  - 返回: <第一操作数当前值 > + <sizeof(t)与第二操作数之积>
- **substract(minus).** **- : t* × int→t***
  - 返回: <第一操作数当前值 > - <sizeof(t)与第二操作数之积>

例:

```
int  v[10], i, j,
     *p = &v[5], *q;

q = p + 3;
i = *q;  // i == v[8]
j = *p;  // j == v[5]
```

2.2. 程序设计语言中的类型

C/C++中指针类型的操作

- **add(plus).** **+ : t* × int→t***
  - 返回: <第一操作数当前值 > + <sizeof(t)与第二操作数之积>
- **substract(minus).** **- : t* × int→t***
  - 返回: <第一操作数当前值 > - <sizeof(t)与第二操作数之积>

例:

```
int  v[10], i, j,
     *p = &v[5], *q;

q = p - 3;
i = *q;  // i == v[2]
j = *p;  // j == v[5]
```

2.2. 程序设计语言中的类型

使用C/C++指针类型必须注意的问题

- Note that most C++ implementations offer no range checking for arrays.
- 对指针和数组下标进行的操作都可能造成数组越界，要训练自己将控制数组越界作为“本能”。
- 两个指针直接相减在语法上也是允许的（但必须相对于同一个数组）；两个指针直接相加是不允许的。
- Complicated pointer arithmetic is usually unnecessary and often best avoided.

2.2. 程序设计语言中的类型

2.2.8 联合类型与变体记录

- 当语言允许一个变量能说明它的合法的值是属于两个或更多个类型时，则这个变量的类型是联合类型（例如C/C++中的union）。
- 联合类型的作用是保证值的互斥，同时也可以节省空间。
- 组成一个联合类型的各个类型的量在编译时的类型检验中是无效的，因此若不加控制，可能引起运行时的类型错误。

2.2. 程序设计语言中的类型

2.2.8 联合类型与变体记录

- 因此，需要有某种形式的运行时的类型检验，这就要在运行时保持某种信息以说明在一个联合类型的实例中当前所存贮的值的类型。这一信息叫做区分标志。
  - **Pascal** 和 **Ada** 都支持有区分标志的联合类型，取名为变体记录。
  - 变体记录分为两个部分，第一部分是标志，第二部分是变体，即记录中可以选用的不同类型，并用 **case** 语句来实现选择。

要点与引伸

- 用不是某个类型的操作集中的操作来直接作用于该类型的值集，会产生难以发现和纠正的错误。但是，不支持数据抽象的语言还不能从根本上约束这种行为。
- 在程序设计中，经常要利用和建立类型的值集中元素的有序性。一些看上去无序的值，语言在实现时也需要使之有序；
- 自定义类型的数据结构定义，通常是在声明该类型值集中一个元素的特征，而不是该值集的集合特征；

要点与引伸（续）

- 类型中的操作实际上有两大类，一类是值集上的元素操作，另一类是值集上的集合操作。当值集元素也是集合时，要注意区分;
- 指针的值集是合法地址的集合，而不是所有地址的集合。由于语言可能不控制指针类型的值集（如C、C++），所以由指针引起的越界错误，就是因为指针当前值不在其值集内。
- 语言中一些司空见惯的操作（如记录类型上的 . 操作）实际上是很复杂的，不妨想象一下它们的实现。

下一次课的内容

- 类型系统 – 数据抽象
  - 对第二次课中提出的问题作进一步的讨论
  - 什么是抽象?
  - 数据抽象
  - 抽象数据类型
- 类型系统 – 多态
  - 什么是多态?
  - 多态的种类
  - 参数多态；包含多态；过载多态；强制多态；其他多态