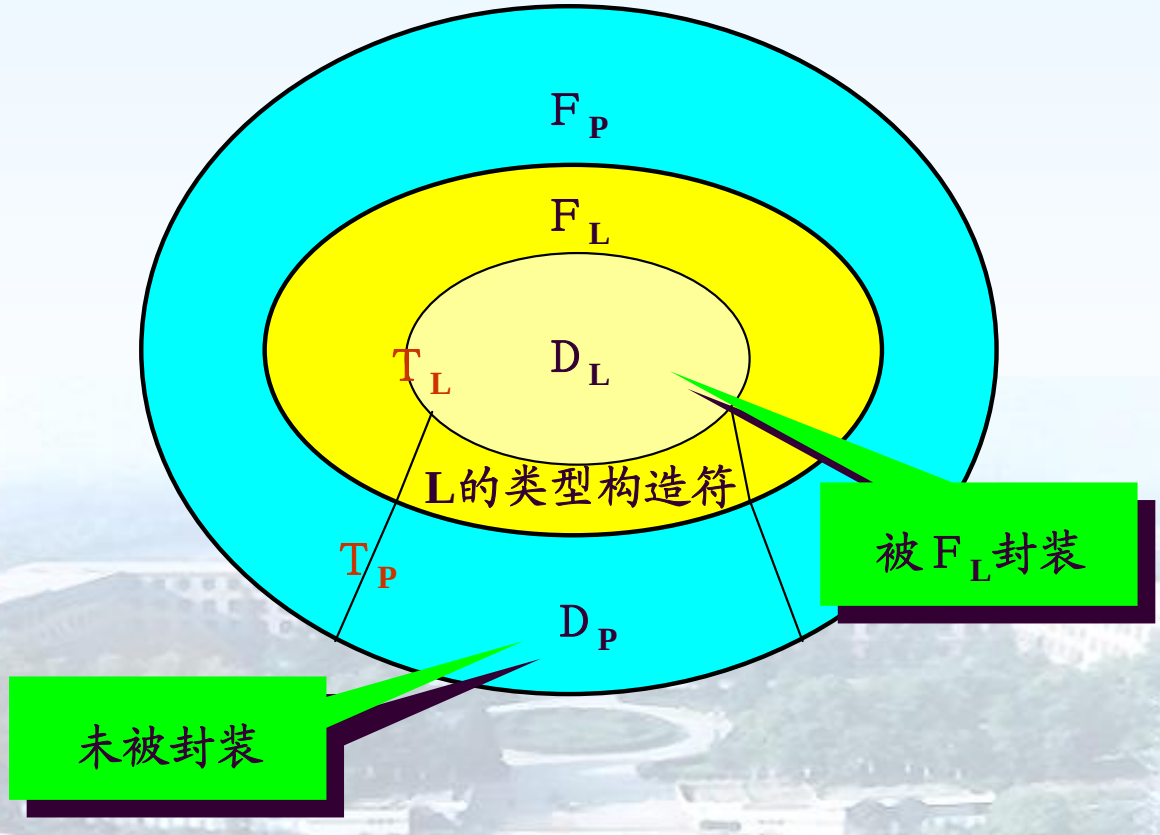
对第二次课中提出的问题作进一步的讨论

回顾

- 用不支持数据抽象的语言 L 实现的程序 P 的类型系统，其特征是：



对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－封装与未封装

- L 的基本类型对外只提供了操作集和类型构造符，而不提供数据结构的定义。因此，语言的使用者一般只可能整体地使用某种基本类型的实例，而不能对其分量直接进行操作。
- P 中自定义操作都直接或间接地基于 L 的基本类型操作集，以便对自定义数据结构中的基本类型分量进行相应的操作。

这似乎是天经地义的。

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－封装与未封装

- P 中的自定义数据结构没有被自定义操作所包围，因此存在着使用基本类型的操作直接对其成员进行操作的可能性。有时，这样的操作在逻辑上是非法的，但编译程序无法检查出来。

仔细想起来是不平等的。

问题在于，L 没有向 P 提供支持，使之能够让自定义类型的操作来封装自定义类型的数据结构。

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－封装与未封装

```
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，
};
struct address addr;
void setAddr()
{
  addr.name = (char *)malloc(21)
  strcpy(addr.name, "George W. B
  // ... ...
}
```

作所包围，因此
员进行操作的可
去的，但编译程

```
extern struct address addr;
void aJoke()
{
  addr.name[0] = '\0';
}
void anOpFromHackers()
{
  strcpy(addr.name, "Bin Ladin");
}
```

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－操作集与值集的对应

- 由于 P 没有在语法上明确给出自定义类型的操作集与值集的对应关系，因而编译程序只能根据特定操作（可能是基本类型的操作）所涉及的类型是否合法，来间接地判定操作与值集的关系，而这样的判断未必符合 P 的类型系统的设计本意。

仔细想起来是不平等的。

问题在于，L 没有向 P 提供支持，使之能够在语法上明确给出自定义类型的操作集与值集的对应关系。

对第二次课中提出的问题作进一步的讨论

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。

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－区分类型的设计者与使用者

- P 中自定义类型的数据结构不得不公开，否则由设计者所进行正常的操作定义就无法进行，如:
  - 在定义结构化类型时，可能既需要对值的整体所进行的操作，也需要对值的分量的操作。
  - 对值的整体的操作不存在统一的语义（例如，两个值相等的条件，不一定都是对应分量值全部相等），因此通常需要由程序员用一组对分量的操作来构成整体的操作。

由于 L 的限制，这样的不平等对 P 是必要的。

问题在于，L 没有向 P 提供支持，使之能够对自定义类型的设计者与使用者实施不同的限制。

对…讨论

L 表…与使用者

- P…设计者所…
  - …行的操…
  - …值相等…通常需…

由…

型…义类…

```
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
};

bool isSameAddr(struct address a, struct address b)
{
  if (strcmp(a.name, b.name)) return false;
  // ... ...
  return true;
}

void changeAddr(struct address* p,
  long n, char* s, char* t, char* st, long z)
{
  p->number = n; strncpy(p->street, s, 20);
  // ... ...
}
```

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－已被占用的标识

- 由于 L 的基本类型上的操作只能扩充、不能被置换，所以 P 中的自定义类型中的操作不能使用基本类型中的操作已经占用的标识（包括操作符和函数名），而只能通过定义新的函数或过程来实现。
  - 这使得在 P 中往往不能用最易于理解的方式来进行命名，以致于降低了 P 的可读性、可理解性。

仔细想起来是不平等的。

问题在于，当编译程序无法判断一个标识的类型时，只好让基本类型独享它。

对第二次课中提出的问题作进一步的讨论

L 和 P 的类型系统的对比－已被占用的标识

例：

```
struct Complex {
  double re, im;
};
struct Complex complexAdd( struct Complex a, struct Complex b )
{
  static struct Complex r;
  r.re = a.re + b.re;
  r.im = a.im + b.im;
  return r;
}
void f( struct Complex a, struct Complex b )
{
  struct Complex c = a + b;
  struct Complex d = complexAdd(a, b);
}
```

对第二次课中提出的问题作进一步的讨论

问题的归纳

- 封装与未封装：L 没有向 P 提供支持，使之能让自定义类型的操作来封装自定义类型的数据结构。
- 操作集与值集的对应：L 没有向 P 提供支持，使之能在语法上明确给出自定义类型的操作集与值集的对应关系。
- 区分类型的设计者与使用者：L 没有向 P 提供支持，使之能对自定义类型的设计者与使用者实施不同的限制。
- 已被占用的标识：当编译程序无法判断一个标识的类型时，只好让基本类型独享它。

在很长一段时间里，人们无奈地接受了这样的不平等。

对第二次课中提出的问题作进一步的讨论

为什么要谋求这种平等?

- 我们希望象程序设计语言控制自己的类型系统那样，来有效地控制大型软件系统的复杂性。
- 尽管数据结构是相对稳定的，但仍然存在改变的可能性，应当尽可能隔离所发生的改变对相关程序的影响。那么，在程序中是不是有能力做到这一点?
- 如果单独采用程序设计技术或风格解决不了上述问题（看来是这样），是不是应当要求程序设计语言来提供有效的支持?

解决的思路：令程序设计语言支持数据抽象。

2.3. 数据抽象

2.3.1 什么是抽象?

- 一个抽象是对一种事物或一个系统的简化描述，它使外界集中注意力于该事物或系统的本质方面，而忽略其细节。

2.3. 数据抽象

2.3.1 什么是抽象?

- **B. Meyer 认为，软件工程师应当掌握 13 个基本原理:**
  - Abstraction 抽象
  - Distinction between specification and implementation 区分规格说明与实现的差异
  - Recursion 递归
  - Information hiding 信息隐蔽
  - Reuse 重用
  - Battling complexity 有效地应对复杂性
  - Scaling up 可伸缩的递增开发
  - Designing for change 适应变化的设计
  - Classification 分类
  - Typing 类型化
  - Contracts 功能抽象
  - Exception handling 异常处理
  - Errors and debugging 排错与调试

2.3. 数据抽象

2.3.1 什么是抽象?

- **B. Meyer 认为，软件工程师应当掌握 13 个基本原理:**
  - Abstraction 抽象
  - Distinction between specification and implementation 区分规格说明与实现的差异
  - Recursion 递归
  - Information hiding 信息隐蔽
  - Reuse 重用
  - Battling complexity 有效地应对复杂性
  - Scaling up 可伸缩的递增开发
  - Designing for change 适应变化的设计
  - Classification 分类
  - Typing 类型化
  - Contracts 功能抽象
  - Exception handling 异常处理
  - Errors and debugging 排错与调试

2.3. 数据抽象

2.3.1 什么是抽象?

- **B. Meyer 认为，软件工程师应当掌握 13 个基本原理:**
  - Abstraction 抽象
  - Distinction between specification and implementation 区分规格说明与实现的差异
  - Recursion 递归
  - Information hiding 信息隐蔽
  - Reuse 重用
  - Battling complexity 有效地应对复杂性
  - Scaling up 可伸缩的递增开发
  - Designing for change 适应变化的设计
  - Classification 分类
  - Typing 类型化
  - Contracts 功能抽象
  - Exception handling 异常处理
  - Errors and debugging 排错与调试

2.3. 数据抽象

2.3.1 什么是抽象?

- **B. Meyer 认为，软件工程师应当掌握 13 个基本原理:**
  - Abstraction 抽象
  - Distinction between specification and implementation 区分规格说明与实现的差异
  - Recursion 递归
  - Information hiding 信息隐蔽
  - Reuse 重用
  - Battling complexity 有效地应对复杂性
  - Scaling up 可伸缩的递增开发
  - Designing for change 适应变化的设计
  - Classification 分类
  - Typing 类型化
  - Contracts 功能抽象
  - Exception handling 异常处理
  - Errors and debugging 排错与调试

2.3. 数据抽象

2.3.1 什么是抽象?

- Why Abstraction?
  - Because abstraction is the key to organizing complexity.
  - Ease of categorization – enables collection of instances of entities into groups.
- 在程序设计语言中基本的抽象有两类:
  - 功能抽象（处理抽象）（*functional/process abstraction*）
  - 数据抽象（*data abstraction*）

2.3. 数据抽象

2.3.1 什么是抽象?

- 功能抽象是通过过程或函数的声明来完成的。过程或函数的声明在程序中提供了一种只说明这个过程或函数具有某种处理能力、但不提供其细节的手段。

```
extern int printf( const char*, ... );

main ()
{
  printf("Hello world!\n");
  return 0;
}
```

2.3. 数据抽象

2.3.1 什么是抽象?

- 数据抽象是通过设计抽象数据对象来完成的。
  （Data Abstraction is achieved through the design of abstract data objects）

下面我们主要来讨论数据抽象。

2.3. 数据抽象

2.3.2 数据抽象

与数据抽象直接相关的两个基本概念:

（1）信息隐蔽（*information hiding*）：

- It is the term used for the central principle in the design of abstractions: Try to hide as much information as possible from the component users.
- Information on a need-to-know basis.

2.3. 数据抽象

2.3.2 数据抽象

与数据抽象直接相关的两个基本概念:

（2）封装（*encapsulation*）:

- The act of grouping together a set of data objects, together with the set of abstract operations, such that only the abstract operations are used to manipulate the data object.

2.3. 数据抽象

2.3.2 数据抽象

- 信息隐蔽与封装之间的关系：
  - 封装蕴涵了信息隐蔽。因为：
    - 封装使得外界只能通过规定的抽象操作来操纵数据对象，这意味着这些数据对象被隐藏了。

2.3. 数据抽象

2.3.2 数据抽象

- 信息隐蔽与封装之间的关系：
  - 信息隐蔽并不必然蕴涵封装。因为：
    - 可以隐藏信息而不必“group data and operations together”。
      - 例：模块和例程库的信息是隐藏的，但不一定集数据和操作为一体，有的只有数据，有的只有操作。
    - 可以既隐藏信息，又允许外界有权直接访问，从而违反了“…only the abstract operations are used to manipulate the data object.”这一条件。
      - 例：**Simula 67**中的 **class**。

2.3. 数据抽象

2.3.2 数据抽象

- 信息隐蔽和封装的支持要求是不同的:
  - Information hiding is a question of program design.
    - Information hiding is possible in any program that is properly designed regardless of language used.
    - 即使语言不支持封装，也可以设法隐蔽信息。
  - Encapsulation is a question of language design.
    - Does the language allow grouping and prohibit access to hidden information?
    - 若语言不支持封装，就无法封装。
- 因此，所谓“可以用非面向对象程序设计语言来实现面向对象程序设计”，是对封装概念认识不清的表现。

2.3. 数据抽象

2.3.2 数据抽象

- 程序设计语言设计人员在高级语言、特别是 **Pascal** 语言的使用过程中积累了经验，也发现了问题，认识到只有结构化程序设计这一方法，不足以对付日益增大的规模，还需要有其他的途径。
- 经过研究和探索，在 **1970** 年代形成了两个新的观念:
  - 数据类型的定义不应当只是值的集合，还必须包括作用于值集的操作集。
  - 对数据的保护不能只依靠程序员或操作系统，而应当在语言本身提供保护的机制（封装）。

2.3. 数据抽象

2.3.2 数据抽象

数据表示即通常所谓的数据结构定义

- 1977年底，产生了“数据抽象”这一新的名词。
  - **Def.1** Data abstraction allows programmers to hide *data representation* details behind a (comparatively) simple set of operations (an interface).
  - **Def.2** Data abstraction is a modularity mechanism in which:
    - a small set of procedures act on data values
    - only these procedures manipulate the data directly

数据抽象是一种基于封装的模块化机制。

2.3. 数据抽象

2.3.2 数据抽象

- 数据抽象最本质的特征，就是把数据类型的使用与它的实现加以分离，它使得程序设计人员能够:
  - 把大的系统分解成多个小的部分，每个部分有一个按所处理的数据而设计的接口;
  - 这些接口是这个部分的说明，是外部可见的，而这个部分的具体实现则是隐蔽的、外部不可见的;
  - 所需的保护措施放在每个接口之中。

2.3. 数据抽象

2.3.3 抽象数据类型（Abstract Data Types，ADT）

- **Def.1** An ADT is the result of encapsulating a set of data objects with a set of operations on those data objects to result in a new data type. The entire definition should be encapsulated in such a way that the user of the type needs to know only the type name and the semantics of the available operations. (a logical grouping and naming).

2.3. 数据抽象

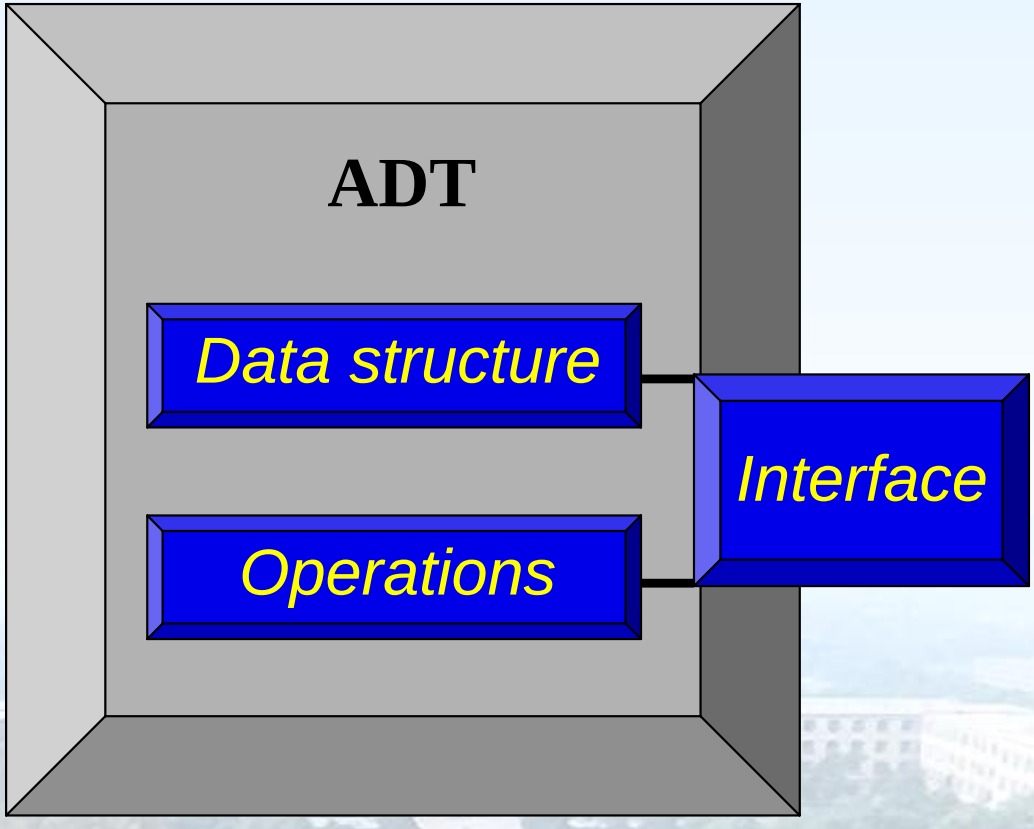
2.3.3 抽象数据类型

- **Def.2** An ADT is a user-defined data type that satisfies the following two conditions:
  - The representation of and operations on objects of the type are defined in a single syntactic unit; also, other units can create objects of the type.
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition.

抽象数据类型是数据抽象的一种类型化实现机制。

2.3. 数据抽象

2.3.3 抽象数据类型



2.3. 数据抽象

2.3.3 抽象数据类型

- 实际上，语言中的基本类型本身就是 **ADT**。例如浮点数类型（**float**，**double**，**long double**）：
  - Information about the storage is hidden from the user.
  - Modification of information is only through the assignment operator – you can't modify the information at the bit-level.
  - Operations on it is only through the predefined arithmetic operations.

2.3. 数据抽象

2.3.3 抽象数据类型

- 但是，并不是所有语言都支持用户自定义 **ADT**（这就是我们前面讨论的不平等问题的关键所在）。
- 首先支持用户自定义 **ADT** 的语言是 **Ada**。
- 目前的面向对象程序设计语言的标志之一，就是允许用户以各语言特定的方式定义自己所需的 **ADT**。

2.3. 数据抽象

2.3.3 抽象数据类型（例：Stack）

- 在设计这个ADT时，我们应当具有关于堆栈的一般知识:
  - 抽象操作 – 堆栈操作的语法:
    - **create : void→Stack**
    - **destroy : Stack→void**
    - **is_empty : Stack→bool**
    - **top : Stack→Element**
    - **push : Stack × Element→Stack**
    - **pop : Stack→Stack**
  - 数据不变量（*data invariant*）– 堆栈核心操作的语义:
    - **pop (push (s, e)) = s**
- We have no idea how a stack is implemented. All we care about is the behavior of the data according to the defined functions.

用Ada语言中的 Package 结构定义这个ADT:

```
package STACK is  -- 接口的定义
  function IS_EMPTY return BOOLEAN;
  function TOP return REAL;
  procedure PUSH(X: REAL);
  function POP return REAL;
end;
```

```
package body STACK is  -- 实现的定义
  -- 数据表示
  MAX: constant := 100;
  S: array(1..MAX) of REAL;
  PTR: INTEGER range 0..MAX;
  -- 操作的定义
  function IS_EMPTY return BOOLEAN is
  -- ... ...
  function TOP return REAL is
  -- ... ...
  procedure PUSH(X: REAL) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  function POP return REAL is
  -- ... ...
  begin   -- 初始化操作
    PTR := 0;
  end STACK;
```

```
-- ADT的使用
-- ... ...
STACK.PUSH(X);
-- ... ...
X := STACK.POP;
-- ... ...
```

用C++语言中的 class 结构定义这个ADT:

```
// STACK.h
define MAX 100
class STACK {
public:          // 接口的定义
  bool   IS_EMPTY();
  double TOP();
  void   PUSH(double X);
  double POP();
  STACK();
private:         // 实现的定义: 数据表示
  double S[MAX];
  int    PTR;
};
```

```
// ADT的使用
include "STACK.h"
// ... ...
STACK s;
// ... ...
s.PUSH(X);
// ... ...
X = s.POP();
// ... ...
```

```
// 实现的定义: 操作的定义
// STACK.cpp
bool STACK::IS_EMPTY()
{ // ... ... }
double STACK::TOP()
{ // ... ... }
void STACK::PUSH(double X)
{
  PTR++;
  S[PTR] = X;
}
double STACK::POP()
{ // ... ... }
STACK::STACK()  // 初始化操作
{
  PTR = -1;
}
```

2.3. 数据抽象

2.3.3 抽象数据类型

- 从 **Ada** 和 **C++** 关于同一 **ADT** 的不同实现中，可以看出 **ADT** 有两种实现模式:
  - Module-as-manager:
    - A module exports an abstract data type
    - Create and destroy operations
    - Object-oriented design is optional (OO as an extension)
    - E.g. Ada, Modula-3, Oberon, CLOS

2.3. 数据抽象

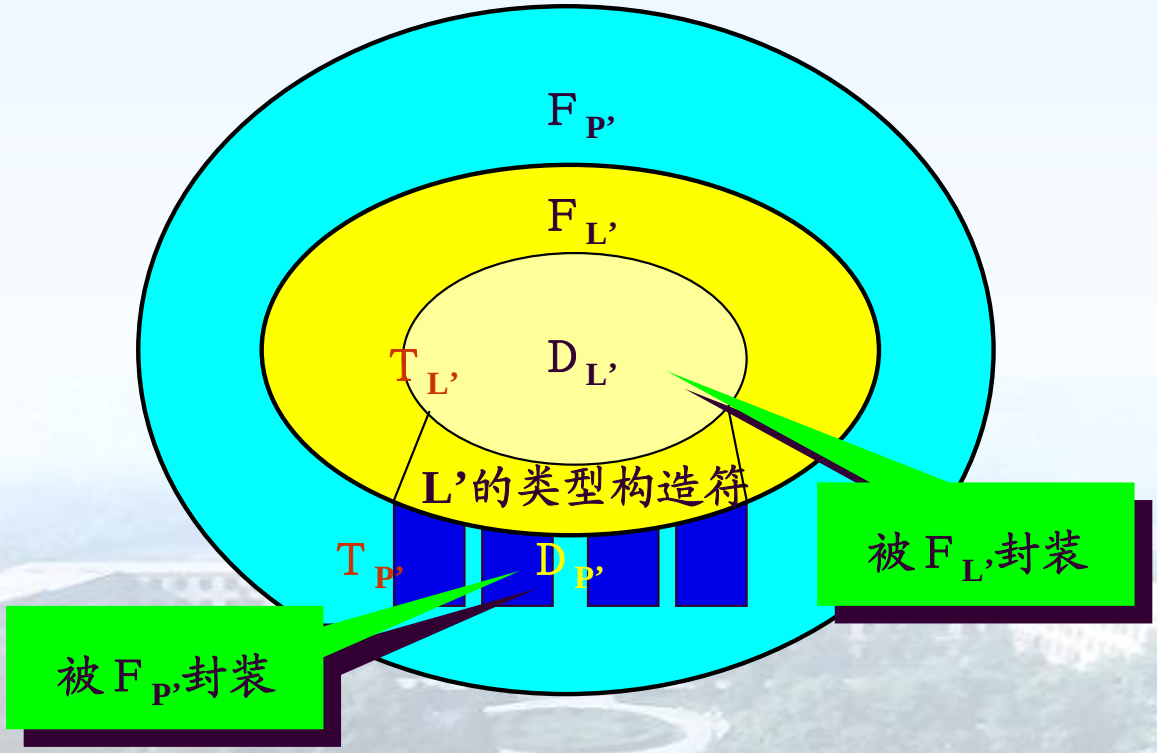
2.3.3 抽象数据类型

- 从 **Ada** 和 **C++** 关于同一 **ADT** 的不同实现中，可以看出 **ADT** 有两种实现模式:
  - Module-as-type :
    - A module is an abstract data type
    - Standardized constructor and destructor syntax
    - Object-oriented design is applied everywhere
    - E.g. Java, Smalltalk, Eiffel, C++

2.3. 数据抽象

2.3.3 抽象数据类型

- 用支持数据抽象的语言 L’ 实现的程序 P’ 的类型系统，则 P’ 的类型系统的特征是:



2.3. 数据抽象

2.3.3 抽象数据类型

- ADT 为程序设计提供了许多好处：
  - 模块化：每个 ADT 自成一个模块，这使得程序的结构化得以保证，同时也使程序的编写简化，使程序易读易懂。
  - 封装性和完整性：允许让什么数据进出受到了接口的控制；不合法的请求不能进入内部。只要实现是正确的，数据结构的完整性就得以保证。
  - 简化了对正确性的检验：由于每个 ADT 是一个单独的模块，可以对它们的正确性分别予以检验，编写主控程序时，对 ADT 只是调用问题。这样，对主控程序的正确性验证就大为简化。ADT 是类型化的，可利用编译检验。
  - 实现部分可以独立更换。

2.3. 数据抽象

2.3.3 抽象数据类型

- ADT 要求程序设计语言具有的能力:
  - 命名：要按照作用域的规则保证名的可见度，设置保护机制以保证隐蔽信息的专用；一个数据只能用一种办法来命名。
  - 类型检验：一般是在编译时进行检验。当类型信息不能或很少可能显式地给出时，需要对表达式的类型进行推理以实现检验。
  - 说明的记法：程序员能够从 ADT 的说明中得到所需要的全部信息， 而且说明和实现必须一致。
  - 分别编译：在一个程序中，每个 ADT 是单独进行编译的， 然后和主程序链接在一起。分别编译保证了 ADT 的封装性，同时为编译优化提供了方便。

2.4 多态

2.4.1 什么是多态（Polymorphism）？

- **Greek:**
  - mono = single
  - poly = many
  - morph = form
- **Monomorphism**（单态）
  - every value belongs to exactly one type
- **Polymorphism**（多态）
  - a value can belong to multiple types

2.4 多态

2.4.1 什么是多态

- 多态是一种普遍存在的现象：
  - H_2O：冰、水、汽；
  - 算术运算：1+1；1 + 0.5；1/2 + 0.5；
  - 对一组不同的事物进行一致的处理：红绿灯对不同类别车辆的统一控制；
  - 将一个集合上的操作施加于其子集：在自然数之间进行整数加（+）运算。

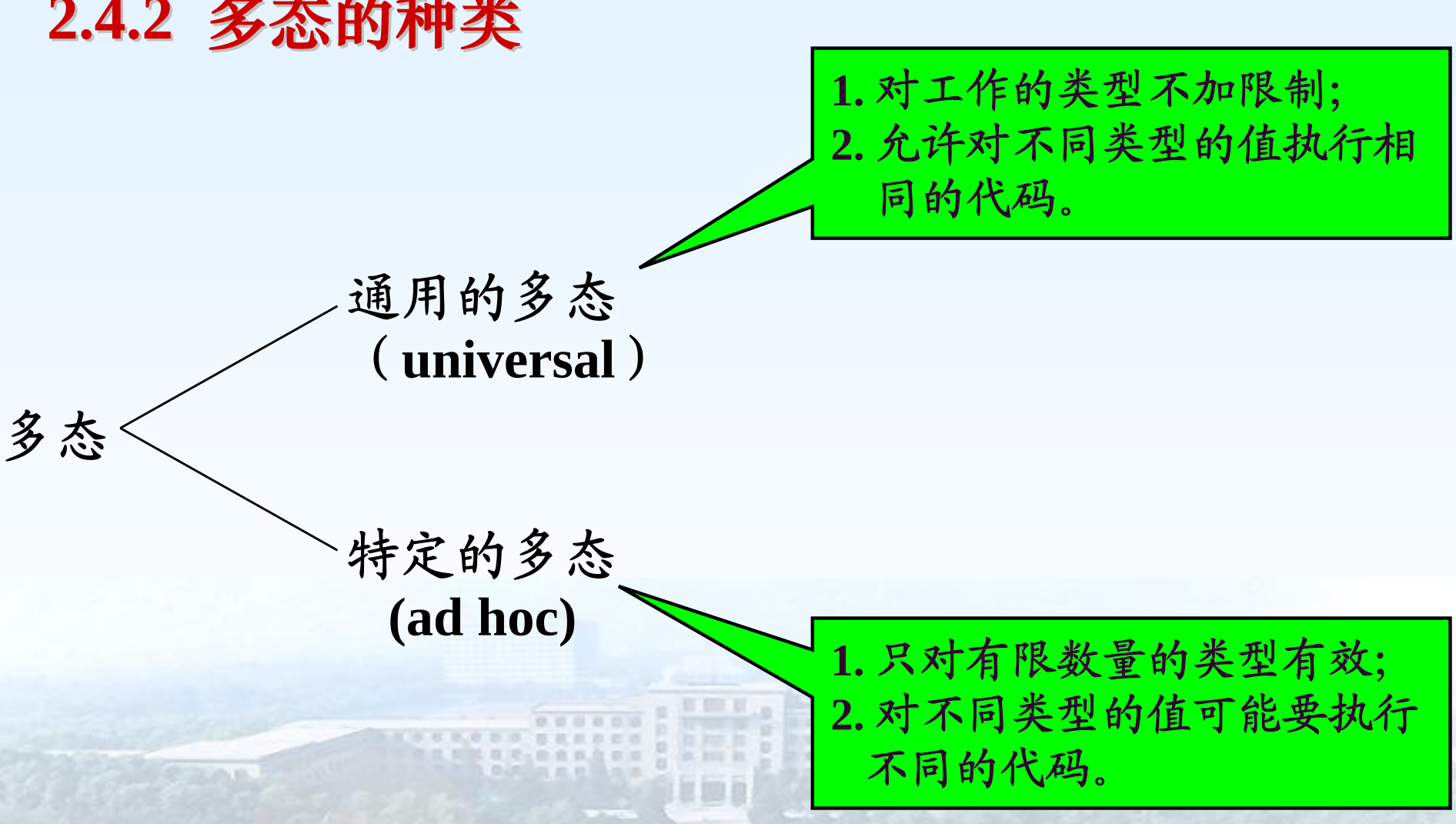
2.4 多态

2.4.1 什么是多态

- 在软件系统中为什么要支持多态?
  - 使程序中的数学运算符合常规的数学运算规则;
  - 使程序提供更强的表达能力;
  - 使得对不同类型的数据有同样的操作语义（程序的重用）;
  - 重用标识的资源，提高程序的可读性和可理解性。
- 软件系统支持多态的前提:
  - 能够静态（编译时）或动态（运行时）地确定类型。
- 复习（第二次课中关于类型的作用的讨论）:
  - 决定将一种类型的值能否和如何转换成另一种类型的值。

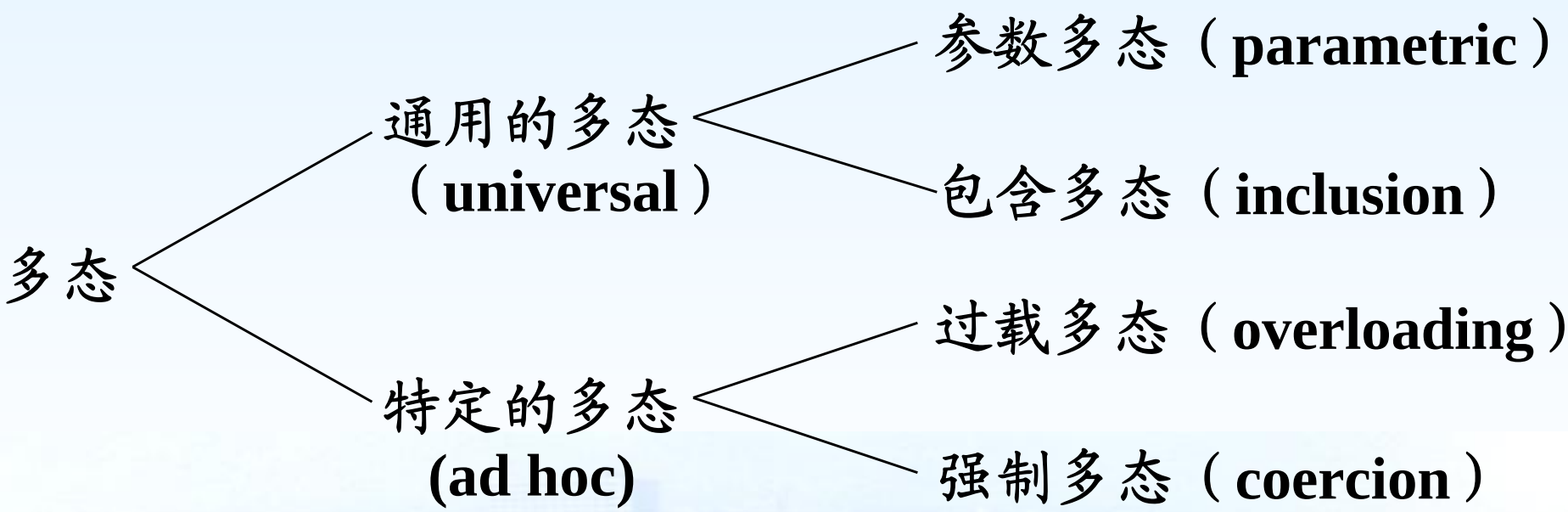
2.4 多态

2.4.2 多态的种类



2.4 多态

2.4.2 多态的种类



2.4 多态

2.4.3 参数多态

- 概念：采用参数化模板，通过给出不同的类型实参，使得一个结构有多种类型。
  - 例：**Ada** 中的 **generic**（类属）。

2.4 多态

```
package STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return REAL;
  procedure PUSH(X: REAL);
  function POP return REAL;
end;

package body STACK is
  MAX: constant := 100;
  S: array(1..MAX) of REAL;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: REAL) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end STACK;
```

```
package I_STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return INTEGER;
  procedure PUSH(X: INTEGER);
  function POP return INTEGER;
end;

package body I_STACK is
  MAX: constant := 200;
  S: array(1..MAX) of INTEGER;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: INTEGER) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end I_STACK;
```

2.4 多态

```
package STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return REAL;
  procedure PUSH(X: REAL);
  function POP return REAL;
end;

package body STACK is
  MAX: constant := 100;
  S: array(1..MAX) of REAL;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: REAL) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end STACK;
```

```
package I_STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return INTEGER;
  procedure PUSH(X: INTEGER);
  function POP return INTEGER;
end;

package body I_STACK is
  MAX: constant := 200;
  S: array(1..MAX) of INTEGER;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: INTEGER) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end I_STACK;
```

2.4 多态

2.4.3 参数多态

- 分析：这两个 packages 的
  - 应用内涵一致：提供的都是堆栈抽象操作；
  - 实现结构相似：采用同种数据结构，实现代码相同；
  - 元素的类型不同，一些用常量表示的细节不同。
- 联想：函数是相似的语句序列的一种涵义明确的抽象，用函数中的语句表示相同的部分，用形参指明不同的部分，用实参体现不同的部分。但是，函数的参数只能是数据或地址，不能是类型。
- 期望：程序设计语言能否提供另一种支持类型参数的抽象？

2.4 多态

```
package STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return REAL;
  procedure PUSH(X: REAL);
  function POP return REAL;
end;

package body STACK is
  MAX: constant := 100;
  S: array(1..MAX) of REAL;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: REAL) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end STACK;
```

```
generic
  MAX: INTEGER;
  type ELEM is private;
package STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return ELEM;
  procedure PUSH(X: ELEM);
  function POP return ELEM;
end;

package body STACK is
  -- MAX: constant := 100;
  S: array(1..MAX) of ELEM;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: ELEM) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end STACK;
```

2.4 多态

```
-- 用 generic 结构产生一个新的
-- package:
declare
  package R_STACK is
            new STACK(100, REAL);
-- ... ...
 R_STACK.PUSH(0.3);
-- ... ...
```

```
-- 用 generic 结构产生另一个新
-- 的 package:
declare
  package I_STACK is
            new STACK(200, INTEGER);
-- ... ...
I_STACK.PUSH(1000);
-- ... ...
```

```
generic
  MAX: INTEGER;
  type ELEM is private;
package STACK is
  function IS_EMPTY return BOOLEAN;
  function TOP return ELEM;
  procedure PUSH(X: ELEM);
  function POP return ELEM;
end;

package body STACK is
  -- MAX: constant := 100;
  S: array(1..MAX) of ELEM;
  PTR: INTEGER range 0..MAX;

  -- ... ...
  procedure PUSH(X: ELEM) is
    begin
      PTR := PTR + 1;
      S(PTR) := X;
    end PUSH;
  -- ... ...
end STACK;
```

2.4 多态

2.4.3 参数多态

- 从 Ada 的 generic 结构可以看出:
  - 对实参所取的类型不加限制;
  - 对不同的实参执行的是相同的代码。
- 类属:
  - 在一个抽象结构中允许以参量形式来表示可变的类型、函数、常数、数据值，在编译时（静态）进行实例化，结果是一个具体的结构（类型、函数等）。
- 对比：类型的实例化（类型→变量）可以静态进行，也可以动态进行，但结果都是一个值。

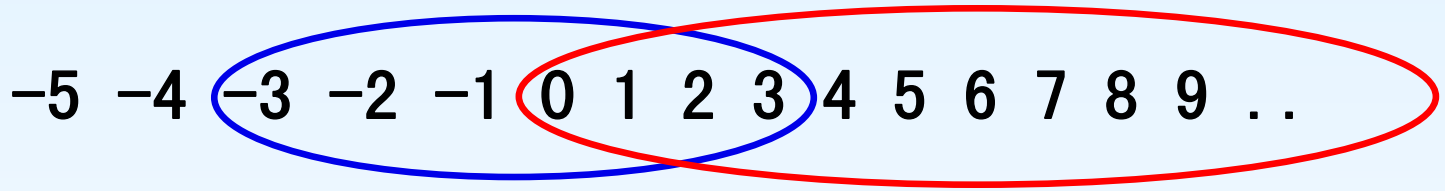
2.4 多态

2.4.4 包含多态

- 概念：同样的操作可用于一个类型及其子类型。
  - 例：**Pascal** 中的子界。
  - 子类型：a value of a subtype can be used everywhere a value of the supertype can be expected.
  - 注意：子类型与将来要介绍的子类是有区别的。
- 包含多态一般需要进行运行时的类型检查。

2.4 多态

2.4.4 包含多态




```
TYPE Natural = 0 .. maxint;
TYPE Small = -3 .. 3;

VAR i : integer;
    n, m : Natural;
    s, t : Small;

    { ... ... }
    s := s + 1;
    t := t - s;
    i := n + m;
    n := i;
```

2.4 多态

2.4.4 包含多态

- 包含多态的操作存在着逆单调（Anti-mornotonic）：
  - 一个类型 t 上的操作，当其定义域缩小成 t 的一个子类型时，其值域应不小于 t。
  - 例如，对于 integer 类型上的操作:

$+ : \quad integer \times integer \rightarrow integer$

$- : \quad integer \times integer \rightarrow integer$

2.4 多态

2.4.4 包含多态

- 包含多态的操作存在着逆单调（Anti-mornotonic）：
  - 一个类型 t 上的操作，当其定义域缩小成 t 的一个子类型时，其值域应不小于 t。
  - 例如，对于 integer 类型上的操作，当其定义域为其子类型时:

    +： Small×Small→Small ✗

    -： Natural×Natural→Natural ✗

2.4 多态

2.4.4 包含多态

- 包含多态的操作存在着逆单调（Anti-mornotonic）：
  - 一个类型 t 上的操作，当其定义域缩小成 t 的一个子类型时，其值域应不小于 t。
  - 例如，对于 integer 类型上的操作，当其定义域为其子类型时：

    +： Small×Small→integer ✓

    -： Natural×Natural→integer ✓

2.4 多态

2.4.5 过载多态

- 概念：同一个名（操作符、函数名）在不同的上下文中有不同的类型。
- 程序设计语言中基本类型的大多数操作符都是过载多态的。如C语言中的

  ==： int×int→int

  ==： double×double→int

  ==： char×char→int

  ……

2.4 多态

2.4.5 过载多态

- 例：有些程序设计语言允许用户自定义过载多态的操作符。如
  - **C++ 语言中的:**

    bool operator == (Complex c1, Complex c2);

    bool operator == (Address a1, Address a2);

  - **Ada 语言中的:**

    function “+” (r : float, i : integer) return float is ..... end;

2.4 多态

2.4.5 过载多态

- 一个过载多态的操作符或函数名，它所对应的通常是不同的实现。例如:

```
bool operator == (Complex c1, Complex c2)
{
  return (c1.re == c2.re && c1.im == c2.im) ?
    true : false;
}
bool operator == (Address a1, Address a2)
{
  if (strcmp(a1.name, a2.name)) return false;
  // ... ...
  return true;
}
```

2.4 多态

2.4.6 强制多态

- 概念：编译程序通过语义操作，把操作对象的类型强行加以变换，以符合函数或操作符的要求。
- 程序设计语言中基本类型的大多数操作符，在发生不同类型的数据进行混合运算时，编译程序一般都会进行强制多态。
- 程序员也可以显式地进行强制多态的操作（**Casting**）。

2.4 多态

2.4.6 强制多态

- 例如:

```
extern void g(double);

void f()
{
  int    i, j;
  double d1, d2, *p;
  // ... ...
  d1 = d2 + i;       // 编译程序将 i 变换成 double 类型
  g(j);              // 编译程序将 j 变换成 double 类型
  p = (double *)&i; // Casting
  // ... ...
}
```

2.4 多态

2.4.6 强制多态

- 在不同类型之间实现强制多态，通常需要执行不同的转换操作。
- 并不是在任意两个类型之间都可以进行强制多态。
  - 强制多态的原则是：将值集较小（即占用存储空间较小）的类型，变换成值集包含了前者（即占用存储空间较大）的类型。反之，应当注意可能发生对值的损伤（特别是在使用 **Casting** 时）。
  - 对指针使用 **Casting** 要十分谨慎。

2.4 多态

2.4.6 强制多态

- 有时，强制多态与过载多态是混合出现的。例如，对于表达式

  1 + 2; 1.0 + 2; 1 + 2.0; 1.0 + 2.0;

  中出现的多态，就会有多种解释:
  - 操作符 + 有四种过载多态;
  - 操作符 + 只有一种： **double×double→double**，要将参与运算的整数强制变换成浮点数;
  - 操作符 + 有两种过载多态： **int×int→int** 和 **double×double→double**，要将混合运算中的整数强制变换成浮点数。

2.4 多态

2.4.7 其他

- 现在的多数文献中都认为，在面向对象程序设计语言中还存在着两种多态:
  - 在有继承关系的类之间存在的多态（但不能完全等同于包含多态）；
  - 通过动态绑定机制，在运行时才确定接收消息的对象类型（如 C++ 语言中的虚拟函数）。

要点与引伸

数据抽象的基本思路:

- 显式地定义值集（或值的数据结构）、操作集以及之间的对应关系;
- 借用以操作的语法说明和语义说明构成的接口，以及操作集与值集间关系的显式定义，来封装值的数据结构和操作的实现细节;
- 明确地分离类型的接口和实现，使得对实现部的绝大多数修改与外界无关。

要点与引伸（续）

- 在类型化机制的约束和支持下，多态使语言提供了更强的表达能力，提供了更多的重用机会。
- 过程和函数用数值或地址作为自变量，类属还允许用类型作为自变量。
- 类属所采用的抽象手段与 **ADT** 相似，也是将相同或相似的、相对稳定的结构（包括数据结构和程序结构）抽象出来，加以确定的表示，不同之处是将抽象层次从关于数据的抽象提高到了关于类型的抽象。

下一次课的内容

- 类型系统－类型的形式化描述简介
  - 类型的数学模型研究
  - 代数模型
  - λ演算
- 布置第一次作业
- 面向对象程序设计－程序设计范型
  - 程序设计范型的概念
  - 典型的程序设计范型
  - 面向对象的基本概念体系
  - 在程序设计时要解决的几个主要问题
- 面向对象程序设计－程序设计语言中的**OOP**机制
  - 类