程序设计语言原理

Principle of Programming Languages

裘宗燕

北京大学数学学院

2012.2～2012.6

7. 程序组织：模块

- 模块
- 历史发展
- 作用域和访问控制
- 接口与实现
- 数据抽象
- 在 C 语言里模拟数据抽象和模块
- 泛型模块（generic，类属）

程序组织问题

在结构简单的高级语言里，子程序是组织程序的最大单位。随着应用系统的规模越来越大，仅仅基于子程序的组织结构已无法满足实际开发的需要

随着程序变得更大更复杂，程序的组织结构变得越来越重要，这就要求在语言层面提供更高层的，更强大的支持软件系统组织的语言结构和机制

软件的组织（结构）通常并不改变语义，只是为了更好地支持：

- 程序的开发、维护、修改和升级活动
- 对程序开发工作的管理
- 开发和维护大规模、长生存周期的软件系统

程序组织的最基本手段是信息封装和屏蔽，需要能方便地：

- 把一个软件系统的程序划分为一些部分
- 定义不同程序部分之间的信息交流方式

程序组织：需求

在程序开发实践中，人们早就认识到程序组织方面的各种需求，例如：

- 经常需要把一批子程序组织在一起
  - 因为它们相互有关联，共同提供了程序所需要的某方面功能
  - 这样一组功能可能为软件系统提供某方面的一套服务
- 经常需要把一组子程序和与之相关一批数据组织到一起
  - 因为这些程序和数据具有内在联系，其中的数据对象记录了这批操作的基本状态和变化，形成一种 “局部”计算状态
  - 这样的一组子程序和局部状态，可能构成大系统里的一个子系统
- 经常需要把一个用户定义的数据类型和一组相关操作组织起来，它们共同形成了一种抽象的数据类型，提供了完整的数据表示或结构

这些都要求有子程序层次之上的程序组织机制的支持

没有语言支持，就只能通过编程约定，不能提供足够的保护和自动检查

早期机制：Fortran

程序语言中早期的组织机制是 **Fortran** 的分别编译和连接

- **Fortran** 程序由主程序和一组子程序组成，单层结构（平坦结构）
- 子程序名均名为全局的。一个子程序提供一个局部作用域，可以作为独立的程序部分进行开发，独立编译为“可再定位”目标代码块
- 子程序参数全部为引用参数（无论标量还是数组）
- 允许子程序库（对后来的发展非常重要）
- 提供共用区（**COMMON**区）作为子程序之外的全局数据
  - **COMMON**区名具有全局性，支持在不同程序块之间的数据共享
  - 不同程序块可以对同一个**COMMON**区定义自己的观点
- 程序连接阶段解决子程序调用与定义的对接
  - 对子程序的调用方式不做任何检查
  - 连接没有任何安全保证，是许多程序错误的根源

底层：目标文件和连接

连接是构造大型程序的一个重要加工阶段

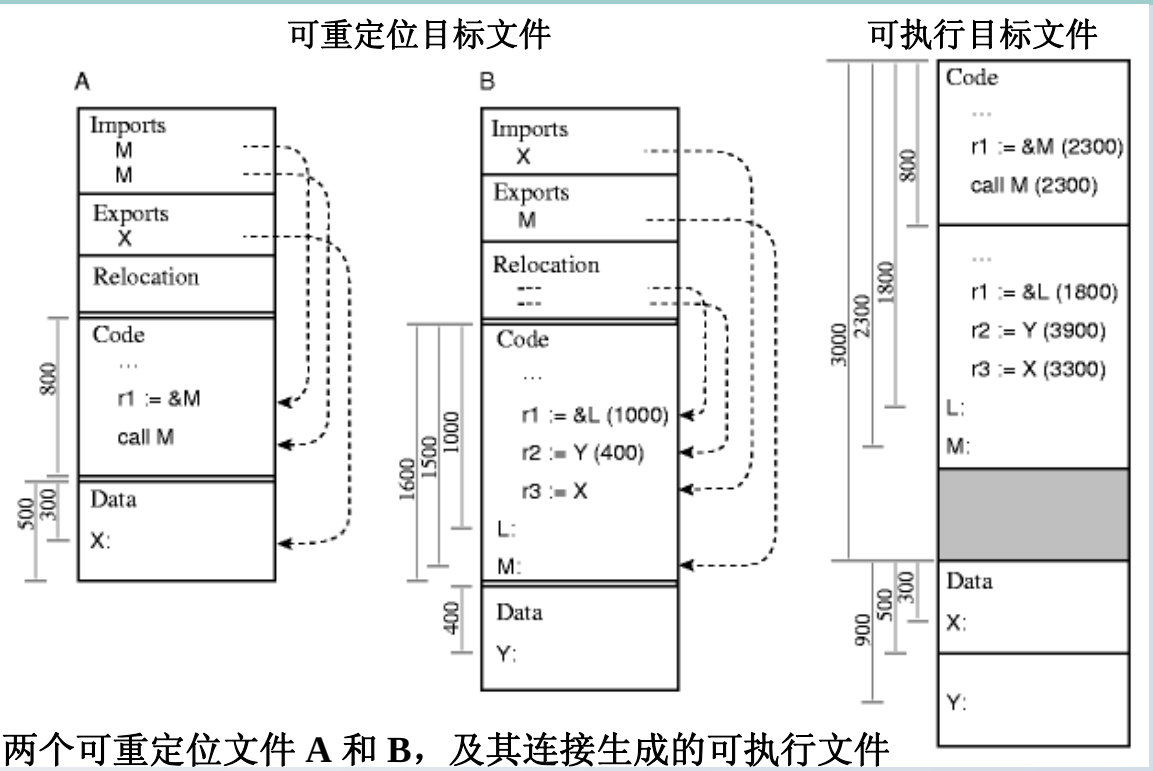
- 编译器送给连接程序的代码形式称为“可重定位”目标文件
- 连接器的工作是将多个目标文件连接成为一个“可执行文件”

可重定位目标文件里包含：

- 再定位表（**relocation table**）：标明引用了本文件里的一些具体位置的指令，本文件具体定位时必须修改这些地方的位置信息
- 导入表（**import table**）：标明文件里所有引用了未知位置（通过名字）的指令。例如，引用了应该另有定义的子程序或者变量
- 导出表（**export table**）：是本文件里定义的所有公用名字及其定义位置的对照表，用于支持其他文件中的引用

导入导出表里的名字被称为外部符号（**external symbol**），一个文件里只有这些外部符号与其他目标文件有关，非外部的名字与其他文件无关

底层：连接示例



两个可重定位文件 **A** 和 **B**，及其连接生成的可执行文件

底层：连接

连接中要维持一个定义表（一个字典），记录当时已知的所有外部符号：

- 如果已知一个符号的定义位置，表中记录该定义位置
- 如果尚不知位置，则由此拉出一条引用该外部符号的代码位置的链

连接过程就是反复读入一个个目标文件，在可执行文件里为读入文件里的代码和数据安排位置，并修改文件里所有需要重定位的指令，其中：

- 逐个地用目标文件的导出表里的项检索当前的定义表
  - 如果表里有相应项：如果表中无位置信息则加入位置定义，并追溯使用链进行修改；若已有位置定义，就报一个“重复定义”错误
  - 如果表里没有相应的项，加入一个新项
- 逐个地用目标文件的导入表里的项检索当前的定义表
  - 如果在表中找到了相应的项：如果找到相应的位置定义，就在文件里做相应的修改；如果找不到无位置定义就把本项链入
  - 如果没找到，就加入一个新的外部符号项，并把本引用位置链入

后续试验：C

C 语言采用与 Fortran 类似的加工模式，提供了另外一些功能

- 允许在任何程序文件里定义子程序和全局数据（外部符号）
- 支持程序块的信息局部化。提供 static 修饰符，凡是加了这种修饰符的符号（变量或函数名）都不作为外部符号，不加入导出表
- 提供外部声明，用于描述跨程序块类型信息，提高连接的类型安全性

C 语言支持以分块的方式编程，允许把一个程序的代码分为多个源文件

基本程序组织单元是“翻译单位”（translation unit，编译单位），C 预处理程序处理一个 .c 源文件产生一个翻译单位，其中：

- 所有的 #include 都已经用拷贝相应文件的内容取代
- 所有的宏都已展开
- 所有条件编译命令都已处理，应该丢弃的程序段都已丢弃

一个翻译单位就是编译器看到的一个文件单位

后续试验：C

一个翻译单位经过编译，最终产生一个目标文件

上面所说的信息局部化，就是基于编译单位的局部化

C 语言并不能保证跨模块的类型安全性：

- 语言本身并不强制要求跨模块信息的类型一致性
  - 函数原型声明可选。遇到调用本翻译单位中没有声明的函数时采用默认翻译方式（认为返回值是 int，参数不转换）
  - 对不同编译单位中同一外部变量的类型、同一函数的声明，编译器和连接器都不做任何一致性检查
- 保留了原 C 语言的函数原型、指向函数的指针类型说明形式，其中不需要提供参数信息（个数和类型）
- 所有正确做法都是“编程习惯”和“规范”而不是语言的强制性要求
- 不保证编译器和连接器对不同模块有统一的观点

这样规定，一方面是为了向后兼容，也受到语言结构的限制

后续试验：C

ANSI C/C99/C++ 都在加强连接中对于类型的控制：

- **ANSI C 增加了函数原型的类型参数描述，鼓励程序员写完整原型**
- **C99 强调程序员应提供函数原型，以及完整的参数描述（未强制要求）**
  - 建议函数头部和原型必须提供返回值类型
  - 明确规定原型中不给出参数描述是过时的贬斥写法
  - 扩充了数组参数的描述方式，以提倡更安全的使用方式
- **C++ 强制性地要求函数原型**

为能用 C 语言开发大型程序，人们总结出了许多良好编程技术，包括基于头文件/程序文件的信息分配等。通过这些技术，以适当的组织方式支持程序分块开发和类型一致的编译、连接

但是，C 语言本身并不包含任何支持程序组织的结构，也没有进一步的信息屏蔽、局部化机制。语言并不禁止不正确或不合适的使用

模块和模块化

David L. Parnas 的论文：On The Criteria To Be Used In Decomposeing System into Modules (CACM, 1972) 被认为是程序模块化的经典论文

随着计算机开发和应用的发展，人们逐步理解了模块概念和程序模块化的基本概念和技术，逐步形成了有关程序模块化的一些基本认识：

- 一个程序应该由一组尽可能相互独立的模块构成
- 每个模块实现整个系统的一部分逻辑功能，其协作实现整个系统的工作
- 每个模块都应具有定义良好的清晰接口，不同的模块之间只通过接口交流信息，局部的实现细节信息应尽可能地屏蔽

模块可以看作对程序中一些部分的分隔和“包装”，把程序里的一下部分独立出来，形成一个个局部实体，将它们与它们的外部环境清晰地隔离

虽然可以通过人工方式模拟模块（例如在 C 语言里，通过编程规范），但要很好支持模块化程序（和系统）设计，就需要语言提供新的结构

模块化的意义

模块化有着多方面的意义：

- 模块划分形成软件的结构划分：把一个复杂软件分割为一些相对独立、相互关系和信息交流都很清晰的部分，有利于控制和克服复杂性
- 开发过程本身需要模块化的支持：
  - 复杂软件系统包含大量代码，模块可作为程序开发的基本单位，支持多人参与的协作开发，支持各部分的独立开发、测试和系统集成
  - 模块可以限制程序错误的影响范围，在开发和运行中都很有意义
- 复杂软件有较长生存周期，需要根据新认识、新环境和新需求等不断演化。模块化为局部维护和修改提供屏蔽，有利于系统的演化
- 有典型逻辑意义的独立模块，可能用于其他软件系统（作为重用单位）
  - 减少软件开发的工作
  - 重复使用经过良好调整、反复测试和长期实践检验的程序部分（功能组件），可能提高软件执行效率、质量和系统的可靠性

语言里的模块结构：特点

支持模块化程序设计的语言，通常提供一种独立的模块结构：

module 模块名 begin ... ... end

具体语言的定义形式和作用差别很大（下面讨论）

模块结构形成对它所在的作用域的一个静态划分，作为一种作用域单位

一个模块是包围着一组相关定义的一个外围作用域，封装起这些定义

位于一个模块内的程序对象可以按常规作用域规则相互访问

在模块之外不能简单地访问模块内部的定义

模块里定义的东西在本模块定义域之外不直接可见

模块结构的一个重要作用是访问控制

模块结构的一个作用是避免过多的名字“污染”全局环境

作为一种信息屏蔽机制，有利于程序的开发、理解和维护等

模块结构：特点

- 模块结构并不改变其内部定义对象的生存期

  模块通常出现在全局作用域里，其中的变量具有静态生存期

  与子程序不同，有初始化问题，但在程序执行中一直存在

- 模块与程序的执行和控制流无关。程序员可根据需要将任意一组相关对象的定义包装在一个模块里，自由定义模块的层次结构，组织程序，利用模块机制，得到名字的局部化

模块与外部的信息交流的一个问题是：是否允许定义在外部的信息进入模块（从模块内能否看到模块外部定义的变量、子程序、类型等）

- 自动继承（与过程/函数类似），自动将模块外部作为外围环境
- 选择性继承（如，继承语言的预定义环境，不继承用户定义环境）
- 完全隔离，都不继承，所有使用都必须明确描述来源
- 可以通过特殊的描述形式使用

自动/不自动继承外围定义的作用域分别称为开作用域/闭作用域

作用域规则

不同语言的模块机制采用了不同的设计选择，主要有两个方面

根据模块内部能否使用外部信息，可把模块分为“开模块”和“闭模块”

- 开作用域模块自动继承外围作用域的所有定义
- 闭作用域模块不自动继承外围作用域的定义
  - 通常自动继承语言的预定义机制，如基本类型/内部操作。某些语言称它们为“渗透性机制”，用户定义的东西不是渗透性的
  - 具体模块需要清晰地定义从外部继承哪些东西

根据模块对内部定义的封装程度（对内部信息的保护力度）

- 弱封装模块的内部定义（原则上说）都可以访问（和使用）。但通常不能直接访问，需要借助于模块名，采用特殊的写法
- 强封装模块要求在定义中清晰描述其为外部提供的功能，只有这些功能是外部可用的，其余留作内部私用，外部根本不能使用

作用域规则

C++ 的 namespace 是一种典型的弱封装的开作用域模块

1. 其外围作用域里的定义在 namespace 里自动可见，遵循普通作用域规则
2. namspace 内部的任何定义都可以通过模块名和 :: 访问

简单示例：

```
namespace A {
   ... ...
   int f(int n) { ... ... }
}

... m = A::f(...); ...
```

C++ namespace 还是一种“开放性”模块，不必定义为一个整体

整个程序里具有同样名字的 namespace 都属于同一个模块

这种模块的唯一作用是为程序里的定义（及其名字）提供一种分层划分

作用域和封装

一些语言设计师认为模块定义应该是一种封闭的强封装作用域

下面是一种常见的模块设计（Modula/Ada 等都与此类似）：

- 模块内部定义的东西均默认为不允许外部访问的，除非明确说明为提供给外部使用的类型、变量和子程序等，外部才能访问
  - 模块提供一种“导出”描述，用一个导出（export）表声明本模块提供给外部的所有名字（功能）
- 模块里不允许直接访问任何外部定义（外部定义在模块内不可见）
  - 如果一个模块里需要使用外围有定义的某些功能，就都必须明确说明。通常提供一个导入(import)表
  - 可导入另一模块，或者导入另一模块里的某些具体定义
  - 通常区分“渗透性功能”和“非渗透性功能”，“渗透性功能”自动渗透进入模块，以方便程序员（如默认外部环境定义的基本类型及相关操作。由语言明确定义。过多的引入会给用户带来负担）

作用域规则

为方便使用，有些语言提供了打开机制。在模块（或其他作用域）里写

using M （或者其他类似语法形式）

把模块 **M** 导出的名字（对弱封装模块，就是把模块定义的所有名字）都直接放入写这个语句的局部环境里，使之成为可以直接使用的功能，相当于在当时的局部作用域里打开了模块 **M**

这种方式的问题有几个问题（与 **with** 语句的情况类似）：

- 打开模块作用域可能造成名字屏蔽（如程序里用的外部名字被屏蔽，将在无意中改变程序的语义），或与局部作用域的名字冲突

  打开弱封装模块的问题更严重，打开模块可能引进许多名字，不仔细查看模块体，无法弄清到底引进了多少新名字

- 如果几个模块里有同名对象，同时打开它们就会引起名字冲突

  一些语言引进了更细致的名字引入机制。例如 **C++** 允许逐个引入模块里的名字，还可以进一步通过 **typedef** 等方式控制可见性

模块结构

模块的常见形式：

```
module M
  export T1, prog2, func3, v4, x5;
  import M1, M2.prog1, M2.func3;
  using M3;
  ... ... // 各种内部定义
begin
  ... ... // 模块的初始化代码段
end module
```

接口定义

内部实现

模块内部的定义可能包括：

- 类型定义，子程序定义（供内部或外部使用）
- 数据定义（形成模块的局部状态）

初始化代码段用于建立模块的合法初始状态

接口与实现

提供模块机制是为了信息隐蔽

- 如果把整个模块的所有实现代码都与接口部分放在一起，提供给使用方，实际上也形成了另一种信息暴露
- 许多因素可能导致人们不希望暴露模块的实现细节，例如为了商业利益和竞争，为了信息安全性等等
- 如果使用一个模块时确实需要参照其全部代码，那么该模块实现的任何细微变化，就都会影响所有的使用代码
- 如果模块的使用与模块的实现之间有紧密的耦合关系，两方面都难以独立地修改，严重影响软件的维护和演化

应看到，使用模块 **M** 的程序部分，常常不需要参考 **M** 的实现细节

- 使用处只需要知道 **M** 的 **export** 表里提供的各种类型、对象和子程序的与正确使用有关的特征（原型，类型）
- 模块内部使用的功能，以及 **export** 表里的子程序的实现，都与使用无关

接口与实现

基于这种认识，人们提出把模块分为接口和实现两个部分

- 接口部分只提供使用该模块所必须的信息（绝不多提供）
  - 供外部使用的类型（可能需要包含类型的布局）
  - **export** 子程序的类型特征（原型描述：包括参数和返回值类型）
  - **export** 变量的名字和类型
  - 数组问题（是否提供形状信息？数组类型是否包括形状？）
- 具体实现方面的信息都放在模块的实现部分，包括
  - 接口子程序的实现，一些类型的细节信息
  - 内部使用的所有子程序、类型与变量等
- 编译程序可以根据模块的接口信息，确定使用模块的代码是否符合模块的要求（类型检查）

一些语言采用了这种设计，明确把模块接口和实现分为两个独立实体

接口与实现

模块接口实现了一种隔离：

- 只要使用方遵守模块的接口约定，其程序代码就可以自由地修改，而不会对模块界面及其实现有任何影响
- 只要模块的接口不变，其实现部分的代码可以自由地修改更新。但更新后的模块需要重新编译，系统需重新连接以使用新的模块目标文件
- 如果模块的接口改变了，模块实现部分和使用模块的程序都需要修改，两边都需要重新编译，重新连接

模块接口和实现分离的一个难点是数据类型定义的划分。问题是在接口里提供什么信息。这里的关键问题：

- 如果使用处需要创建相应的对象，至少需要知道对象大小的信息
- 如果一个程序库必须提供各种关键数据类型的详细布局信息，实际上就暴露库程序的实现细节。这种情况可能成为安全隐患

这方面问题是强封装模块机制的设计难点，但可能通过程序技术处理

模块：接口与实现

Ada 模块（称为package）接口示例：

```
package BSTREE is      -- 二叉树模块
    type BSTREE is private;
    function HAS(i : ITEM, P : BSTREE) return boolean;
    procedure INSERT(i : ITEM, P : BSTREE);
private
    type BSTPRT;
    type BSTREE is record
        data : ITEM;
        left, right : BSTPRT;
    end record;
    type BSTPRT is access BSTREE;
end package;
```

用户不能访问 private 部分的定义（但可以看见数据布局）

模块体（实现部分）另外定义。可以根据需要确定接口和实现的划分

模块：接口与实现

Ada 的模块机制支持对接口子程序和内部实现子程序做很好的划分：

- 包接口文件里只包含接口子程序的原型
- 内部子程序在包接口文件里没有任何体现，有利于实现封装

在 C++ 里定义类 T 时，写类 T 的头文件时：

- 需要包含类 T 的完整定义，不能采用不完全的声明形式
- 需要把 T 的布局描述放入头文件，是因为使用处需要定义类 T 的变量，必须知道类 T 的大小，否则编译无法完成存储分配
- 只与内部实现有关的子程序，其原型也必须放在类定义里（放入头文件里），因为 C++ 没有真正的接口和实现的隔离
- 一个类里内部子程序任何增删或原型修改，都导致重新编译所有使用该类的程序部分。如果一个类包含许多实现细节，原本与使用无关的修改也经常会导致大量无关程序的重新编译（令人恼火）

这些情况对开发效率有很大影响

基于模块的数据抽象

在软件开发的理论和实践中，人们逐渐认识到数据抽象的价值：

- 一个数据抽象就是一种封装，它提供了一个抽象的数据类型
- 数据抽象为所定义的抽象类型提供了一组操作，这种抽象类型的用户只能通过这组操作使用所定义的抽象类型
- 数据抽象使用户可以像使用内部类型一样去使用用户定义类型
- 数据抽象内部定义了该数据类型的对象的具体实现的表示方式
  - 这一表示方式对于外部是隐藏的，不可见的
  - 只要能够支持所需的操作，可以采用任何实现方式
  - 仅仅修改实际实现方式并不改变抽象数据类型
  - 支持数据类型实现方式的演化
- 数据抽象还需要为抽象数据类型的变量的初始化和销毁提供相应的机制（显式的操作或者隐式的自动动作）

基于模块的数据抽象

人们发现，软件系统中的许多部分实际上是需要定义一些不同的类型

数据抽象是克服软件复杂性的一种有力武器

有关数据抽象的认识导致产生了基于数据抽象的程序设计方法学

人们发现，需要用模块机制定义的许多东西实际上就是数据抽象

这一认识也导致人们提出了一些有关语言中的模块结构的新想法

在不同的语言里通过模块机制实现数据抽象的方式有两种：

1. 模块作为其内部实现的一种或多种数据抽象的管理器
2. 模块本身作为抽象数据类型

模块和数据抽象

模块作为抽象数据类型的管理器，相关开发工作：

- 在模块里定义所需的类型以及与之相关的操作，并实现这些操作
- 导出模块提供给用户使用的类型和接口操作（如前面例子**BSTREE**）
  - 用于内部实现的类型和操作不导出
  - 通常需要导出一对 **create/destroy** 函数，其功能是创建或销毁本类型的数据对象（此方式通常用于动态分配的对象）
  - 还要/或者提供一对 **init/finalize** 操作，对定义好的变量做初始化或者终止处理（此方式通常用于值模式的变量）
- 完全可以用一个模块管理多个类型及其相关操作

缺点：用户定义抽象数据类型的对象时，需要显式调用操作创建/销毁对象，这可能变成程序员的负担，未执行这些操作可能带来不良的后果（如存储流失）或者严重的动态错误（如非法访问）

模块和数据抽象

定义堆栈类型 stack 的模块（接口）

```
module STM is
   type stack;
   proc initstack(stack s);
   proc push(stack s, int x);
   ...
   proc fanilizestack(stack s);
end module
```

使用：

```
using STM;
stack st1, st2;
initstack(st1); initstack(st2);
push(st1, 3); push(st2, 5);
... ... /* 使用这两个栈 */
finalizestack(st1); fanilizestack(st2);
```


模块作为数据抽象

将模块本身作为类型，用模块名作为类型名去定义相关变量等：

b : BSTREE(...); // 假定这里模块就是类型

操作都需要通过模块（实例）的名字调用：

b.INSERT(...)
if b.HAS(...) then ...

概念上看，好像每个模块（实例）有自己的一套独立操作。实际上，同一模块（类型）的所有实例可以共享一套操作

- 模块实例的创建和销毁可能作为嵌入语言的一些机制自动完成，或者通过调用模块提供的适当子程序完成
- 对模块中所有局部子程序的调用，总是从某个模块实例（调用实例）出发的，该实例总是这些子程序的一个参数。在子程序里可以直接访问
- 为实现对调用实例的访问，局部子程序的实现中自动创建一个指向调用实例的指针（**this**，作为子程序运行环境的一部分）

模块作为数据抽象

```
module stack is
   proc initstack(...);
   proc push(int x);
   proc pop(int &x);
   ...
   proc fanilizestack();
end module
```

使用：

```
stack st1, st2;
st1.initstack(); st2.initstack();
st1.push(3); st2.push(5);
... ...
st1.finalizestack(); st2.fanilizestack();
```

语言可以定义自动执行的构造函数和析构函数机制。支持数据抽象的语言应有这方面的自动支持（如C++），以防程序员忘记初始化和结束处理

数据抽象

如果用模块实现抽象类型，在接口上如何提供类型信息？两种方式：

- 完整的类型构造（布局）
- 只提供类型名，希望隐藏类型的实现细节

分别称为“透明类型”和“非透明类型”（opaque类型，隐晦类型）

问题：模块 A 导出类型 T，模块 B 导入类型 T 并定义了 T 类型的变量 n，编译程序必须知道怎样给 n 分配空间

- 许多语言要求以透明方式提供类型的布局，以便编译程序参考
- 一些语言提供“隐晦”方式

  实际上要采用引用语义，要求另外提供创建 T 类型程序对象的子程序

在实践中，可以用指针模拟隐晦方式的类型定义

后面讨论这方面的技术

数据抽象：操作

对抽象类型的对象，除调用模块所提供的功能之外，还能做什么？

模块提供的操作：

- 构造对象初始状态，对象销毁前的临终处理（通过子程序提供这些功能）
- 模块接口提供的各种相关操作（访问、修改状态、相互操作等）

问题：能否做一些一般性的对所有对象都能做的事情？如

是否允许赋值？是否允许比较两个抽象类型对象的相等和不等？

这些问题的回答牵涉到模块对抽象类型的保护程度

- 通过赋值可能得到一些对象的内部状态信息，通过判断相等或不等，也可能泄漏出对象内部的实现方式信息等
- 不同语言对抽象类型有不同的规定
- 有些语言允许用户自己选择不同的保护方式

数据抽象：操作

Ada 对结构类型提供了很多默认操作，包括抽象类型的对象赋值和比较

- 如果把抽象类型定义为 **private**，那么就只允许做赋值、相等/不等比较
- 如果把抽象类型声明为 **limited private**，那么就不允许赋值，也不允许做相等或者不等比较（除非模块接口提供特定实现）

C++ 与数据抽象相关的概念是类，采用的方式是：

- 默认赋值是按二进制位的对象复制，默认的情况是不允许对象比较（继承了 **C** 里的规定，不允许做两个结构的比较）
- 允许类定义中包含自定义的赋值和对象比较函数（作为成员函数）
- 可以通过技术禁止赋值（把赋值操作定义为 **private** 成员函数）

Java 对用户定义类型采用引用语义，问题就很不一样了

- 赋值是引用赋值，比较是引用相等，总是允许
- **Object** 定义了 **clone** 方法，允许用户覆盖它，以实现所需的复制行为

数据抽象

通过模块提供的数据抽象可以带来多方面利益：

- 独立的数据抽象大大减少了程序员需要同时考虑的细节，减少思考的负担
- 防止程序员以不适当的方式使用部件，限制了各个部分的作用范围，这些都能起到限制故障影响范围的作用，有利于程序开发和维护
- 作为一种重要的程序部件层次，支持好的程序结构。使各部件分离，允许修改部件内部实现而不必修改使用它们的外部代码
- 逐层构造的数据抽象，形成系统的一种分层构造
- 可以作为一种库单元，提供给其他程序使用

模块可支持任意的数据封装，用于开发满足各种需要的程序单元

- 人们希望模块能成为支持软件重用的基础设施（软件重用对提高程序质量，降低开发代价很有价值）
- 但模块作为重用单元时，不能提供足够的灵活性。完成的模块是一个“固体”，只能用于其设计所确定的工作，不能很方便地调整功能

模块：重用

实践中人们发现程序开发中的一些常见情况

- 遇到的新问题常常与解决过的问题类似，但又不完全相同的
- 如果新的需要与库里的某模块的功能不同（即使差异很小），这个模块就不能以 “现有”形式重用
- 如果想去“重用”这个模块（不另行开发），程序员就只能做源代码拷贝，深入研究后设法修改其代码
  - 从实际价值看，一旦要修改代码，所谓重用的意义就不大了
  - 修改代码的代价很大，修改后，代码的可读性、易理解性和易维护性都会恶化，可能危害整个系统。模块继续重用的可能性变得更小
- 通过模块定义的抽象数据类型是独立的，相互间没有关系。而实际中常需要一些相互有关的类型（变体和联合就是为迎合这方面的需要）
- 如何在抽象数据类型的框架中提供这种功能，也是需要解决的问题
- 面向对象（OO）的机制在这些方面都能提供帮助（下一章讨论）

在 C 语言里模拟模块和数据封装

C 语言没有提供数据抽象机制，只能通过程序技术模拟。基本方法：

- 用一对头文件（.h 文件）/程序文件（.c 文件），用头文件模拟模块接口，在程序文件里给出所有实现细节
- 头文件中给出“抽象类型”的定义和相关操作原型
  - 可采用透明类型方式，通过程序技术也可以实现非透明类型
  - 给出数据对象的初始化和终结操作的原型
  - 给出数据类型的所有导出操作（提供给外部的操作）的原型
- 程序文件里给出所有操作的完整实现定义
  - 程序文件开头用 #include 命令做文件包含，引进头文件给出的定义，以检查两者的一致性
  - 所有导出操作定义为一般函数
  - 所有内部使用的辅助函数定义为 static 函数

模块：C 模拟

使用“抽象数据类型”必须遵守一套完整约定：

- 定义对象必须首先调用初始化操作，保证使用前对象已处于合法状态
- 不再使用的对象必须调用抽象类型提供的终结操作，以免存储流失等

不遵守约定的使用可能导致运行中的严重后果

模块的提供方式有两种：

- 提供头文件和程序文件的源文件
- 提供头文件的源文件，以及由程序文件编译后生成的目标文件

使用一个“模块”的每个程序文件都用 #include 包含“模块”的头文件

- 如果有所需“模块”的源文件，就把这个“模块”的程序文件加入所开发的项目，与自己开发的其他部分一起管理，用同样的编译和连接过程
- 如果只有程序文件的目标文件，就在连接时将相关目标文件连入

C 模拟：导出透明类型

```
/* stack1.h */
typedef struct stack {
   int num, capacity;
   int *elems;
} stack;

int init_st(stack* st, int num);
void delete_st(stack *st);
int pop_st(stack *st, int* np);
int push_st(stack *st, int n);
```

这个大家都比较熟悉

但如何实现非透明类型？

```
/* stack1.c */
#include <stddef.h>
#include "stack1.h"

int init_st(stack* st, int num) {
   st->elems =
        (int*)malloc(sizeof(int)*num);
   st->num = 0; st->capacity = num;
   return 1;
}

int pop_st(stack *st, int* np) {  … … }
void delete_st(stack *st) { … … }
int push_st(stack *st, int n) { … …}
```

为简单起见，这里没考虑分配失败

C 模拟：导出非透明类型

实际上就是增加一层间接

操作也需要适当修改

```
/* stack2.h */
typedef struct st_imp* stack;

stack init_st(int num);
void delete_st(stack st);
int push_st(stack st, int n);
int pop_st(stack st, int* np);
```

靠指针实现一层间接，将具体实现隐藏在指针背后

由于指针大小相同，因此不知道被指类型也可以实现

C 允许这样声明“非透明类型”

使用时不能对被指类型间接

```
#include <stddef.h>
#include "stack2.h"
typedef struct st_imp {
   int num, capacity;
   int *elems;
} st_imp;

stack init_st(int num) {
   stack st = (st_imp*)malloc(sizeof(st_imp));
   st->elems = (int*)malloc(sizeof(int)*num);
   st->num = 0; st->capacity = num;
   return st;
}

void delete_st(stack st) { … … }
int push_st(stack st, int n) { … …}
int pop_st(stack st, int* np) {  … … }
```

模块：C 模拟

这种 C 语言模拟，实际上也反应了其他语言中模块实现的一些基本技术

如果提供模块机制的语言采用非透明类型：

- 其导出类型下面只能是一个引用，因此提供了更好的保护
- 抽象类型的对象只能通过动态存储管理在堆中分配
- 所有使用都需要多通过一层间接，可能损失效率

语言采用透明类型：

- 允许定义局部或者静态的抽象类型对象（静态对象或栈对象）
- 可能提高操作效率
- 将内部表示暴露给用户，只能提供较低程度的保护

某些语言既允许透明导出类型，也允许非透明导出类型。两种不同东西的实现方式不一样。C/C++ 实际上是把相关的选择交给程序员处理

泛型功能（generic，类属）

模块也有类型约束的问题

作为一种抽象，同样可以考虑给模块引入类型参数，使之成为“泛型模块”

实际中很需要这种东西，例如：

- 对于不同的元素类型，各种基本容器（栈、队列、表等等）的接口和实现都一样，只是其中的元素类型不同
- 一个复杂系统里可能用到许多不同元素类型的栈、队列等等。针对每个元素类型写一个实现，太繁琐，也影响程序的维护和演化
- 没有泛型类型，各种抽象数据类型库的定义和使用都将受到极大限制。我们实际上无法定义真正有用的类型库

有关泛型模块的讨论推迟到下一章，在讨论了基本的面向对象机制之后进行。模块的泛型问题与类的泛型问题类似，只是更简单一些