子程序环境：嵌套子程序

语言允许嵌套子程序定义（子程序里定义局部子程序），实现的问题很多

这是 C 采用平坦结构的根本原因，许多语言采用同样的设计思想

语言允许嵌套子程序，可以支持更多的程序组织方式：

- 子程序的内部功能可以通过内部子程序实现，既能很好完成子程序的功能分解，又不会在全局作用域中引进大量子程序
- 在没有数据抽象和其他高级程序组织功能的情况下，嵌套子程序结构使程序员有了一套简单易用的功能分解和程序组织手段
- 随着其他程序组织方式的出现，嵌套子程序的重要性减弱了，其实现的复杂性却显得更加突出，新语言大都没有采用这种设计

Algol 60/Pascal/Ada 等语言支持嵌套子程序，程序运行时环境分为三部分

- 静态（全局）环境，局部环境。这两部分我们很熟悉
- 非全局的外围环境，即外围子程序里定义的那些命名对象形成的环境

嵌套子程序

为支持嵌套子程序，实现引用环境时需要在运行栈上维护一个静态链

静态链表示子程序的静态嵌套结构，运行中可以通过静态链找到和使用那些在非局部的外围环境里定义的变量

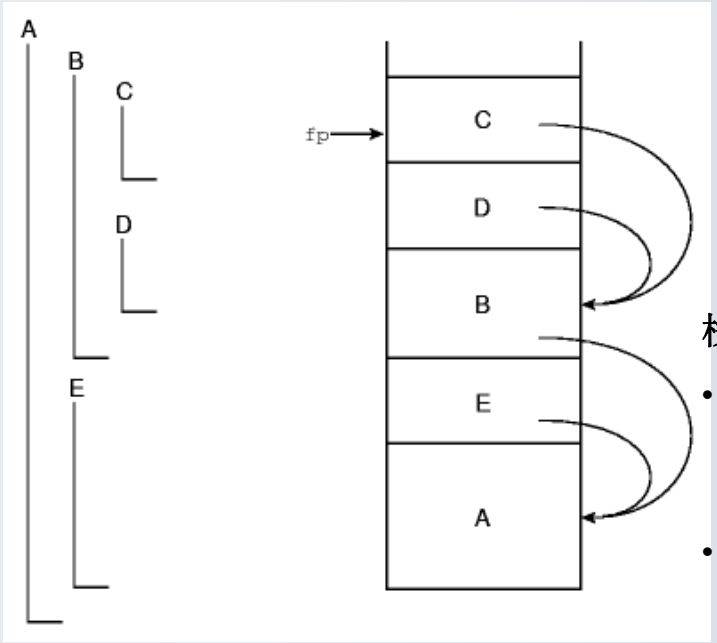
要想支持对非局部环境的访问，必须有这个静态链：

- 在嵌套定义的子程序里使用非局部定义的变量，应使用外围最近的子程序里定义的同名变量（这样找下去，也可能最终确定实际引用的是全局变量）
- 运行中维护静态链，令一个帧的静态链指针指向子程序的直接外围子程序在栈上最近的帧
- 子程序嵌套是静态的，非局部变量引用的定义位置（在当前子程序外面第几层）可静态确定

```
A(): int n, m; real x;
  B(): int m; real y;
    C(): int n;
    {C .. n .. y .. x }C
    D(): int n;
    {D .. n .. m .. x }D
  {B .. m .. x ..  }B
  E(): int k;
    F(): int m;
    {F .. m .. k .. x ..}F
  {E .. k .. }E
{A .. n .. m .. x  }A
```

子程序环境：嵌套子程序

```
A(): int n, m; real x;
  B(): int m; real y;
    C(): int n;
    {C .. n .. y .. x }C
    D(): int n;
    {D .. n .. m .. x }D
  {B .. m .. x ..  }B
  E(): int k;
    F(): int m;
    {F .. m .. k .. x ..}F
  {E .. k .. }E
{A .. n .. m .. x  }A
```



用数组代替链表，可得到 **O(1)** 访问操作

栈上的静态链：

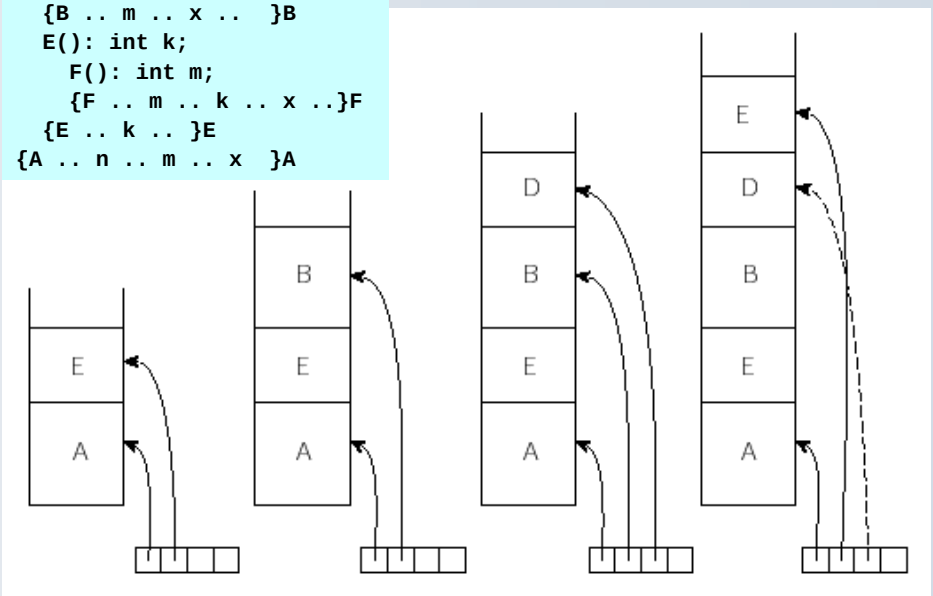
- 访问外围第 **k** 层的变量，先沿静态链前进 **k** 步（找到相应帧），而后通过静态确定的偏移量访问
- 访问开销与嵌套深度有关，不是常量，可能较耗时
- 维护静态链，带来新开销（计算/设置）
- 静态链通常都很短

子程序环境：嵌套子程序

非局部访问的加速技术：区头向量（**display**），通过两次偏移量

```
A(): int n, m; real x;
  B(): int m; real y;
    C(): int n;
    {C .. n .. y .. x }C
    D(): int n;
    {D .. n .. m .. x }D
  {B .. m .. x ..  }B
  E(): int k;
    F(): int m;
    {F .. m .. k .. x ..}F
  {E .. k .. }E
{A .. n .. m .. x  }A
```

维护开销较大，进出子程序时需要保存/恢复的数据多



区头向量数组大小限制了静态嵌套深度

子程序的子程序参数

执行通过参数传递的子程序时，如何为这个子程序建立运行环境？

- 在平坦子程序结构的语言里，这个问题不需要特殊处理，因为每个子程序的运行环境都只有两层：局部环境和全局环境
  - 传递子程序参数时只需要传递一个代码指针（如 C 语言）
- 存在子程序嵌套时，要把子程序作为参数传递，就必须有“闭包”的概念
  - 不仅要传代码指针，还要传一个静态上下文指针 s。以便在实际执行实参子程序而创建栈帧，以这个 s 作为其静态链指针值
  - 闭包(closure)：带着执行环境信息的代码指针
- OO 语言里，实现指向成员函数的指针（C++）也需要“闭包”概念
  - 需要把成员函数的代码地址（一个代码指针）和指向当前调用对象的指针包装成一个闭包（“成员函数指针”不是简单的指针）
  - 最终执行这个方法时，以闭包里的指针作为 this 值（确定调用对象）

子程序：在线展开（inline）

子程序的优点：

- 是一种控制抽象机制，清晰地划分了界面和实现（what 和 how）
- 建立了一个局部计算环境，有利于计算中的信息屏蔽和保护
- 一次定义，可多次调用
- 一份目标代码，多次调用共享（可同时存在多个活动，共享一份代码）

缺点：

- 创建局部环境时的辅助存储开销
- 进入、退出子程序时的时间开销

子程序的在线展开（inline）是为了：

- 保留子程序抽象和局部环境等的原有语义和优点
- 加快执行速度（消除调用时进入退出子程序的时间开销）

子程序：在线展开（inline）

在线展开的想法来自宏汇编（C 语言的宏是汇编语言宏的高级语言形式）

子程序展开的基本想法：

- 把子程序代码直接嵌入每个调用处（在调用处展开，可能带来代码膨胀）
- 严格保持与实际调用相同的语义（保持语义的展开），为此可能需要在展开的代码段前后增加少量代码（类型转换，数据复制、装载和保存等）
- 子程序的局部数据在调用程序的帧里分配（如果需要）

✓ 通常人们只把很小而且调用频繁的子程序描述为 **inline**

✓ **inline** 是一种优化手段，在线展开后编译器可以做进一步优化

✓ 在许多语言里，**inline** 只是对编译器的优化提示，编译器可以根据上下文分析，自由决定是否做、如何做在线展开（例：递归子程序）

✓ **inline** 子程序已经不是程序对象了，因为它们（可能）已经没有运行时的实体了。因此不能取其地址作为子程序抽象来使用

参数机制

子程序的另一重要特征是参数化。

为实现子程序抽象，人们提出了许多参数机制，满足程序设计的不同需要

子程序之间需要共享数据或相互通讯：

- 借助作用域规则，隐式地通过双方都可访问的外层（如全局）变量
- 通过参数，这是私用的通讯通道

- 通过参数通讯的特点是每次调用可以指定不同的变量或者表达式
- 调用语句/表达式描述如何通讯，传入传出什么信息，按什么方式
- 引进参数使子程序更加通用，使之能解决一类问题
- 程序设计语言发展中提出了许多参数模式，主要有值参数、引用参数、结果参数和值结果参数，共享参数等
- 有些参数机制因为语义隐晦且实现复杂而基本被抛弃了，例如换名参数

参数机制：形/实参约束

形式参数是子程序接口定义中的主要部分

- 形参位于子程序的局部环境里，子程序调用时与实际参数建立约束，交换信息（也可能用在子程序返回时向外传递信息）
- 形参的意义、调用时传递的信息由形参的模式确定
- 实参与形参的对应关系通常按位置确定。这种方式很自然，也很方便，但在子程序的参数很多时也容易弄错

有些语言提供了显式指定实参所对应形参名的描述方式。如Ada：

```
prog(a=>10, c=>2.27, f=>100);
```

- 显式表示实参/形参对应关系，有利于阅读
- 在子程序的参数很多时，能够得到更清晰的描述
- 可支持参数的自由缺省（只要有已定义好的默认值）
- 造成形参的作用域不清
- 过程定义与调用间更紧密的耦合（不能随便修改子程序的参数名）

参数模式：值参数

值参数：形参是局部变量，调用时用实参的值初始化（或赋值）。实参可以是任意表达式。值参数通常可以在子程序里作为普通变量使用

- 实践性规则：如果采用值参数没有特别的缺点，就应该用它
- 可能的缺陷是无法改变环境，复制大的数据项会带来显著的开销

C 的参数都是值参数，参数传递就是赋值

C++ 明确把参数传递看作用实参对形参的初始化

- 赋值时变量已有值，可能要处理即将丢掉的值；初始化是创建变量初值
- 对用户定义类型的参数未必能采用按位拷贝，可能需要执行特定动作

Ada 的参数模式概念更高级（**in/out/in out**，更抽象）：

in 参数模式，表示要通过它从子程序外（调用方）向子程序内传信息。标量类型的 **in** 参数通常通过传值（复制）的方式实现。对构造类型，**Ada** 允许编译器选择实现方式，复制或不复制参数是实现细节

参数模式：引用参数

引用参数：实参必须是左值（变量等），调用时形参成为对实参的引用，子程序里对形参的访问被实现为对实参的间接访问：

- **Pascal 的 var 参数，C++ 的引用参数**
- **const (常)引用参数：是引用参数，但不允许子程序修改实参**

引用参数的主要用途

- 使子程序能够直接修改被调用处的环境
- 避免大数据项的复制

```
void swap (int &n, int &m) {
   int k = n;
   n = m;
   m = k;
}

   swap(x, y);
```

能实际交换变量的值，依靠：

- 值语义
- 引用参数

在C语言里只能用指针模拟引用参数

定义中显式做间接，调用时传地址

参数模式：结果参数和值结果参数

- **Ada 结果参数（out）和值结果参数（in out）**
  - 对应的实参必须是左值
  - 子程序结束时，**out** 形参的值被赋给实参；**in out** 参数在函数调用时接受实参的值，在函数结束时把值赋给实参
  - 希望避免引用参数带来的潜在别名问题

in out参数与引用参数不同

```
a : integer = 2;
procedure p1 (n) {
   n := n + a;
   a := n + 3;
   n := n + 1;
}

p1(a);
```

Ada 没有明确引进引用参数，但其设计者也看到 **in**、**out**、**in out** 参数可能产生一次或两次赋值，数组参数有大的复制开销

Ada 标准规定：

对构造类型的 **in out** 参数，实现可选择采用复制方式或引用方式，而且：

“任何程序如果能揭示出这两种实现的差异，它就是错误的程序”

参数模式：共享参数

Java 对基本类型的变量采用值语义，对其他变量采用引用语义

因此，基本类型的参数是值参数，其他类型（类类型和数组）参数按照引用方式实现，实际上是值（值对象）共享，又称共享参数

所有采用引用语义的语言实际上用的都是共享参数机制

```
void swap (T n, T m) {
   T k = n;
   n = m;
   m = k;
}

T x = ..., y = ...;

swap(x, y);
```

能交换x和y的值吗？

不能“交换”被变量引用的对象

要改变两个实参，使其值相互交换，只能通过复制对象的值的方式

```
void swap (T n, T m) {
   T k(n);
   n.copy(m);
   m.copy(k);
}
```


参数和返回值

对于复合数据类型，一个重要问题是调用/返回时是否复制数据

- Pascal 允许数组和记录的值参数，也允许这些类型的引用参数
- C 的数组参数（基于指针的概念实现）相当于引用参数，而对于结构则只有值参数（可以用指针来模拟引用参数）
- C++ 引进了引用参数和 const 引用参数，以方便大型数据对象的信息传递（避免整体复制），以及控制子程序对参数的使用方式

多数语言的返回值就像一个 out 参数，实现单向传输，可以用参数来模拟

有可能对复杂类型的返回值做实现优化（把接受值的对象地址传给子程序，要求子程序最后直接在相应位置构造或者填充）

只有 C++ 提供了返回引用的机制，使对象可以在一连串函数之间传递（注意，返回对象是临时对象，这里有对象生存期问题。请查看 C++ 的规定）

这种机制的一个成功应用是 iostream 库的用户 IO 操作 “<<” 和 “>>”

C++ 的函数对象机制也可以借用这方面的技术

参数的其他问题

默认参数

- 一些语言里允许给参数提供默认值，未提供实参的形参就取其默认值
- 多数语言里，只能是最后的默认参数可缺省（按位置的形/实约束），可以允许最后的几个参数缺省，使用默认值
- **Ada** 语言里提供了显式描述形参和实参对应关系的描述形式，因此可以支持在任意位置的参数缺省（只要其他实参都写清楚对应的形参）

如果可以用任意表达式为参数提供默认值，就有表达式求值时间的问题

- 在函数定义时求值，或者在函数调用时求值
- 如果表达式里有变量，两种求值就可能得到不同结果

C++ 不仅允许函数定义给出参数默认值，也允许在函数原型声明中写默认参数，允许不同原型声明写不同的默认参数（因为只是在调用时使用）

- **C++** 的默认参数在函数调用时求值

参数的其他问题

变长参数表

子程序的参数个数通常是固定的，但有时可能希望允许变动长度的参数表

Lisp 允许变长参数表：

`(define fun (a b.c) ...)` -- `c` 与从第三个实参开始的参数表匹配

Lisp 把与形参（如 **c**）匹配的任意多个参数当作表来处理，通过常规的编程机制去检查和使用，取各个实参就是取表的元素

这种设计实际上还依靠 **Lisp** 语言的动态类型检查机制

函数式语言（脚本语言）一般都是通过动态类型检查提供类型安全的变长参数表，通过类型推理等实现参数的正确使用

常规语言中只有 **C** 语言通过标准库提供变长参数，这种机制的一个重要用途是用于支持紧凑的输入输出函数定义

定义具有变长参数表的函数需要 `#include <stdarg.h>`

这种变长参数表机制牺牲了类型安全性

其他参数问题

C 语言的变长参数表的函数

```
int sum(int n, ...);
```

```
int sum(int n, ...) {
  int i, s = 0;
  va_list vap;
  va_start(vap, n);

  for (i = 0; i < n; i++)
     s += va_arg(vap, int);

  va_end(vap);
  return s;
}
```

```
n = sum(5, 9, 5, 3, 7, 5);

m = sum(3, 101, 234, 511);
```

- 通过函数的命名参数确定无名参数的起始位置
- 程序员描述实参使用方式（形参没有名字，也无类型信息）
- 必须按规定模式定义函数：
  - 初始化va_list变量
  - 程序员确定实参类型和个数
  - 终结处理
- 编译不能检查实参类型或个数
- 实参必须按程序里的顺序排列
- 无类型安全性，但有时有用

泛型子程序

子程序的一个重要发展是泛型(generic)子程序，或称子程序模板(template)

常规语言里的子程序声明要求给出参数和返回值的完全类型描述

但确实有些计算过程并不依赖于参数的具体类型，例如排序、各种数据结构操作（栈、队列和表的操作、二叉树操作等）

一个系统里可能同时有许多类似数据结构，需要排序等通用算法。按常规需要提供一组子程序，它们基本相同，只是操作的数据对象类型不同

要求具体元素类型也限制了库实现：每个库函数只能用于一套具体类型，因此需要做一大批针对各种类型的完成同样功能的库函数（显然不合理）

采用动态类型检查的语言（如一些函数式语言和脚本语言）可以自然地支持泛型。例如 Lisp 语言里的所有表操作都是泛型操作

人们很早就在考虑如何在常规的静态检查的语言里支持泛型功能。存在两种可能性：绕过类型系统（抛弃类型安全性），或者设计新的类型机制

泛型子程序：C 实例

C 标准库的通用排序和检索函数（基于泛型指针 void* 和用法规则）：

```
void qsort(void* base, size_t n, size_t size,
      int (*cmp)(const void* k1, const void* k2));
```

base:数据组起始位置，n:被排序项数，size:元素大小。使用：

```
qsort(a, n, sizeof(ET), compareET)
```

qsort 不知道数组元素类型，因此其实现只能采用 void 指针，要求用户计算出具体的使用细节。这种方式没有类型安全性

```
void* bsearch(const void* key, const void* base,
   size_t n, size_t size,
   int (*cmp)(const void* kv, const void* dt));
```

cmp 在数 kv 与 dt “比较”更大/相等/更小时分别返回正/零/负值

设数组 base[0]..base[n-1]元素按 cmp 序上升排列，bsearch以 *key 为依据查找匹配元素（使 cmp 返回值等于0的元素）

泛型子程序

一个泛型子程序代表一集（潜无穷个）实际子程序，可以认为它有一组类型参数，对每一套适当类型，它相当于一个具体的子程序

泛型子程序应当或者可以直接用于许多不同类型；或者可以通过对一组组的具体类型进行实例化，生成可应用于具体类型的具体子程序

提供泛型机制的语言包括 Ada、C++、Clu、Eiffel、Modula-3、Java、C# 等，还有许多通用的脚本语言

常规语言的泛型机制有多方面诉求：使一个描述可用于多组类型，类型安全，使用方便，有效的静态处理和动态运行，尽可能减少空间开销等

泛型机制最重要的用途是实现各种容器类数据类型（泛型类型）

- 容器是用于保存一集数据元素的数据结构
- 具体例子如表、栈、队列、二叉树等等

泛型的容器数据抽象提供的操作都是泛型子程序

有关情况在讨论数据抽象机制时讨论，下面的一些讨论与之密切相关

泛型子程序：与宏的关系

在 C 语言的实际编程中，人们还常常采用宏机制，通过程序技术实现类似泛型的功能。最简单的情况如：

```
#define min(x,y) ((x)>(y)?(y):(x))
```

这个宏可用于任何数值类型（有泛型的意思）

语言的泛型机制与宏有一些共性，但它们在本质上是不同的：

- 泛型是语言的机制，集成于语言之中。而宏则是由语言之外的机制（预处理器）提供的一套简单的文本替换功能
- 泛型遵守语言的作用域、命名和类型规则。宏什么也不遵守
- 泛型是类型安全的。宏在这方面没有任何保证
- 泛型的类型参数等要通过类型规则的检查（不同语言的安排可能不同）
  - 或者在泛型的定义上下文中解析和检查
  - 或者在泛型实例化的上下文中解析和检查

泛型子程序：实现

两种可能实现方式：

- 为针对每组具体类型的实例化使用生成一份具体化的代码
- 设法使一个泛型子程序的所有（类型具体化）实例共享同一份代码

唯一代码方式是可能的（例如前面 C 实例），可节约代码空间，但是

- 通常会对实例化提出一些限制，例如要求类型实参的元素都具有特定的大小（例如都是引用或者都是指针）
- 另一可能性是增加一层间接，需要为此付出效率方面的代价

例：Java 5 保证每个泛型子程序（方法）只有一份实现代码，因此它只允许针对类类型做泛型实例化。抽象地看所有类类型都是 Object 的子类型，有公共基础。从实现角度看，类类型的变量都是引用，因此大小统一

在此基础上，Java 自动加入必要的（非转换）强制，保证类型正确性。相关的实现技术也称为“类型抹除”（erasure）

泛型子程序：实现

采用为每个（用一组具体类型）实例化生成一份代码的方式，特点：

- 生成多份代码将给编译过程带来（可能很大的）时间开销
- 会造成代码膨胀（至少相当于为需要用的每组类型写一套子程序）
- 使用起来更灵活更少限制（不受具体类型大小的限制）

例：C++ 模板设计希望提供最强的灵活性，允许用任何类型进行实例化，因此选用生成实现方式。一般情况下编译器将为每个实例化生成一份代码

如果编译器足够聪明，也可能发现某些不同的实例化可共享同一份代码

对每个实例化考虑代码生成问题，还受到其他因素的困扰：

- 在常规方式下，由于分别编译的存在，系统在编译一个源文件时，未必知道以前编译其他文件时是否已经为同样的实例化生成过代码
- 系统可能重复生成代码；可能需要判断多份代码是否可合并；也可能因为处理不完全，导致同一份代码的多份拷贝遗留到可执行文件里

泛型子程序：泛型参数

泛型程序单元也是一种抽象，其接口应该为用户提供使用它的信息。一些语言提出了对泛型参数的强制性要求，如 Ada、Java、C#

例如 Java 的泛型排序程序可能写成：

```
public static<T extends Comparable<T>> void sort(T A[]){
    … … if (A[i].compareTo(A[j]) >= 0) … …
}
… …
Integer[] myArray = new Integer[50]; … …
sort(myArray);
```

只允许采用满足限制的类型做泛型实例化，可以保证泛型子程序所要求的操作确实存在。C# 里的规定与 Java 类似

问题：这样定义泛型子程序不能同时用于类类型和非类类型

当然，提供操作只是语法要求，并不能保证操作的语义满足所需（例如，具体类型提供的 compareTo 可能不符合排序所需的比较方向）

泛型子程序：泛型参数

也有些语言（特别是 C++）里的泛型机制没有明确给出的限制描述

```
template<class T> void sort(int size, T A[]) { … }
```

这里有对 T 提出任何进一步要求

采用这样的泛型设计，一个泛型子程序可能同时用类类型或者非类类型进行实例化（C++ 的一个基本设计思想，用户定义类型和基本类型行为相同）

问题：

- 如果实例化类型没提供泛型定义里使用的操作，编译时会报告静态语义错，但很难给出高质量的诊断信息。程序员根据泛型接口无法清晰地预测实例化会不会出错，必须查看泛型定义（不是良好定义的抽象）
- 即使有关类型提供了泛型中使用的操作，也未必保证该操作做的就是这个泛型所需要的事情（只是名字碰巧对了）

后一情况的实例：sort 可能用到 < 运算符。如果用 (char *) 实例化，排序时实际比较的将是字符串地址

泛型子程序：非类型参数

允许什么样的泛型参数：

- 一些语言只允许类型作为泛型的参数，因此一个泛型子程序也就是一个通过类型参数化而得到的一族操作的一个公共表示
- 另一些语言（特别是 C++）允许非类型泛型参数。这就使泛型机制能用于支持更广泛的程序设计技术

一些人们把泛型看作一种一般性程序生成技术（尤其是 C++ 那样通过实例化“生成”程序的泛型机制），研究把泛型发展为一种静态程序设计机制的技术称为元程序设计（其潜力还有待进一步开发和认识）

有人证明了 C++ 的泛型机制具有图灵完全性，也就是说，原则上说，我们可以用它生成出任何可能的程序

如有兴趣，可以读一读有关 generic programming 的书籍和论文

“Modern C++ Programming” 是一本很较广影响的著作

后来有专门讨论 Meta Programming 的著作

泛型子程序：C++

C++ 泛型称为模板，允许定义函数模板（泛型子程序）和类模板，因为 C++ 有独立存在的函数。模板的实例化生成具体的函数或者具体的类

函数模块实例化隐式地自动进行，一旦类型解析发现需要用某函数模板的实例而又没找到，编译器就根据这种需要自动生成针对具体类型的实例

由于函数模块生成中的复杂性，C++ 规定不对生成类型做任何推断，只使用实际参数的具体类型，以防止歧义性

- 由于编译器通常不够聪明，它完全可能生成同样代码的多个拷贝
- 有时根据模板参数无法确定如何进行实例化，需要显式提供类型参数

C++ 允许同样名字的函数模板重载，也允许函数模板与非模板函数的名字重载。语言为重载的函数调用定义了一套解析规则，大致过程：

找出所有实例化后可能用于该调用的模板，去掉其中比其他模板更一般的模板。对剩下的模板进行实例化后与其他可用函数一起进行选择

泛型子程序：Java

Java 没有独立的子程序，所有泛型子程序都是某个类的方法

Java 设计者原本希望通过公共基类 Object 实现各种泛型功能，因此有意不想提供专门的泛型机制。但后来发现这样工作有一些不可避免的缺点：

- 泛型容器的用户必须到处使用强制，这种做法既很讨厌也不美观
- 错误的强制导致运行时 ClassCastException 异常，编译时不能检查
- 运行时强制带来运行时的开销（需要动态检查强制的正确性）

后来在实际应用和用户群强烈要求下不得不考虑显式泛型机制（Java 5.0）

由于是后来的不得已而为之，机制的设计受到许多条件的限制，最后采用的基本设计经过长时间理论研究和比较讨论（有很好的理论基础）

泛型设计的两大困难：

1，不能打破已开发的 Java 程序，因此需要与以前的版本兼容

2，需要继续使用已有的 Java 环境，特别是 Java 虚拟机和现有的库

泛型子程序：Java

Java 泛型的设计基于下面一些想法

- 每个类都是 Object 的子类，以 Object 为参数的方法可应用于任何类类型的对象，重要问题是返回的对象需要正确强制，回到所需要的类型
- 通过泛型机制，可以将使用中所需的强制正确地封装起来
- Java 的泛型设计研究了一套抹除技术，可以把泛型代码翻译为不带泛型机制的普通 Java 代码，其中包含所有必要的强制（保证正确）

这套设计可以保证类型安全性，却无法判断一些合理使用的正确性：

```
public class SList<T> {
   T[] elements; ... ...
   SList (int m) {
      elements = (T[])new Object[m];
   ... ...
```

编译警告：[unchecked] 未经检查的类型使用

这里不允许用泛型参数创建数组，因为可能危害类型安全

泛型子程序

对于泛型的研究历史还不长，认识还不够深入

现在已经有一大批语言里提供了泛型机制，但它们采用的思想和技术差异很大，也说明这一领域尚不成熟

泛型设计的基本需求

- 突破过分僵死的类型系统的束缚（可以看作是突破 Pascal 的过分受限的类型机制工作的继续）
- 提供静态的类型安全性
- 支持更广泛的程序设计的实际需要（特别是库的开发）

程序设计中有一个追求（单点原则）：在一个系统里，每个重要的设计决策应该只有唯一的一个描述

目的是为了支持软件系统的维护和演化

泛型也是向着这一设计理想的一种努力