软件构造
SOFTWARE CONSTRUCTION

主讲教师：李翔　冯桂焕

2014年秋季

Sub-Part III:

- 子程序的设计
  - 何时需要子程序
  - 在子程序层上设计
  - 子程序命名
  - 子程序长度
  - 子程序参数
  - 宏子程序和内联子程序
  - 其它问题

什么是子程序(routine)

- 子程序
  - 为实现一个特定目的而编写的一个可被调用的方法(method)或过程(procedure)
  - 是节约空间和提高性能的最重要手段

- 什么是高质量的子程序？

子程序举例

C++示例：低质量的子程序

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr, EMP_DATA empRec,
   double & estimRevenue, double ytdRevenue, int screenX, int screenY,
   COLOR_TYPE & newColor, COLOR_TYPE & prevColor, StatusType & status,
   int expenseType )
{
int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;  =0?
newColor = prevColor;
status = SUCCESS;
if ( expenseType == 1 ) {
    for ( i = 0; i < 12; i++ )
        profit[i] = revenue[i] - expense.type1[i];
    }
else if ( expenseType == 2 ) {
        profit[i] = revenue[i] - expense.type2[i];
        }
else if ( expenseType == 3 )
        profit[i] = revenue[i] - expense.type3[i];
        }
```

差劲的子程序

为什么创建子程序？

- 降低复杂度
  - 当内部循环或条件判断的嵌套层次太深时，需提取为子程序

- 引用中间、易懂的抽象

```
if ( node <> NULL ) then
   while ( node.next <> NULL ) do
      node = node.next
      leafName = node.name
   end while
else
   leafName = ""
end if
```

leafName = GetLeafName(node)

为什么创建子程序？-续

- 避免代码重复
- 支持子类化
- 隐藏顺序
- 隐藏指针操作
- 提高可移植性
- 简化复杂的布尔判断
- 改善性能
- 确保所有子程序都很小？No!

应该为简单目的编写简单的子程序吗？

- 使用子程序前

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

伪代码示例：用函数来完成计算

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
     ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

- 使用子程序后

```
points = DeviceUnitsToPoints( deviceUnits )
```

- 优点：更易读，更易修改
  - DeviceUnitPerInch()==0?

在子程序层设计

- 让每个子程序只把一件事做好
  - 不再做任何其他事情，如cosine()
  - 强内聚性Cohesion
    - 内聚性描述了子程序中各种操作及数据之间互相联系的紧密程度

Carch 和 Agresti在1986调查450 个 Fortran 子程序：

50%的强内聚性子程序是没有错误的，而只有18%的弱内聚性子程序才是无错的。

Selby 和 Basili 1991调查450 个 Fortran 子程序：

弱内聚性子程序的出错机会是强内聚性出错机会的7倍，而修正成本是后者的20倍。

可取的内聚性

- 功能内聚性（最高）
- 顺序内聚性（较高）
- 通信内聚性（中）
- 临时内聚性（低）

功能内聚性

- 功能内聚性是最强也是最好的一种内聚
  - 指一个子程序仅执行一项操作

功能性内聚

Sin()：计算三角函数sin的值

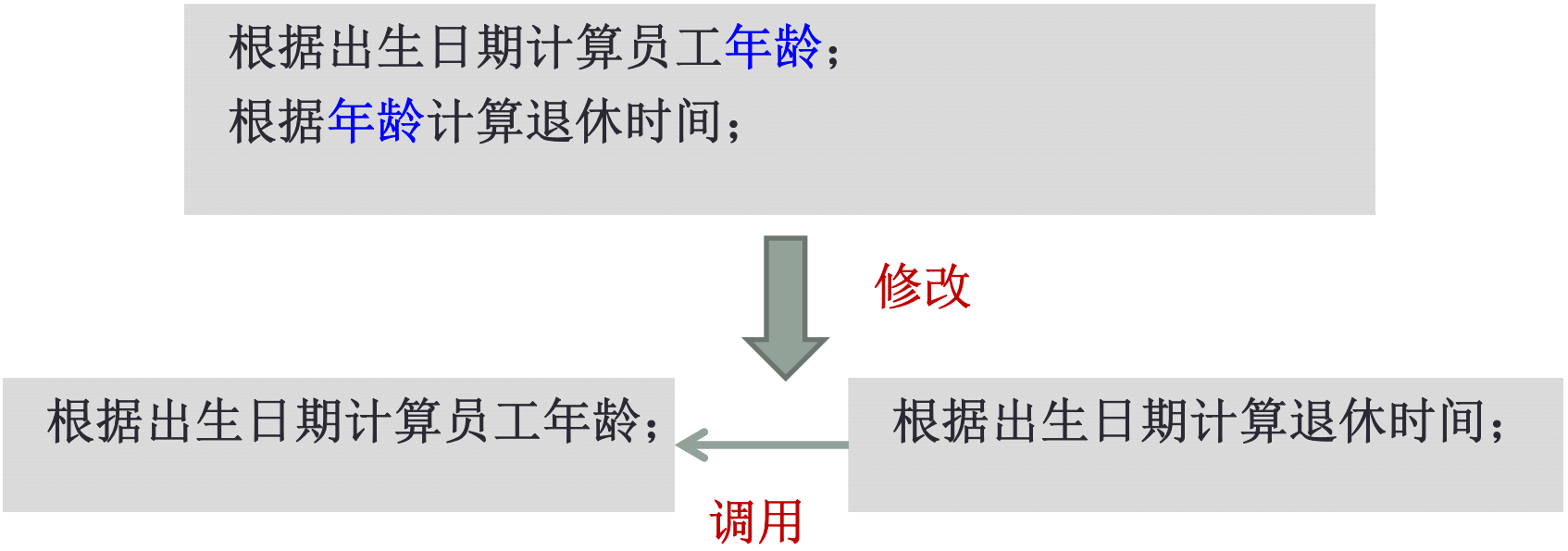
GetCustomerName()：获得客户姓名

EraseFile()：从硬盘删除文件

前提：子程序的名称与其内容相符

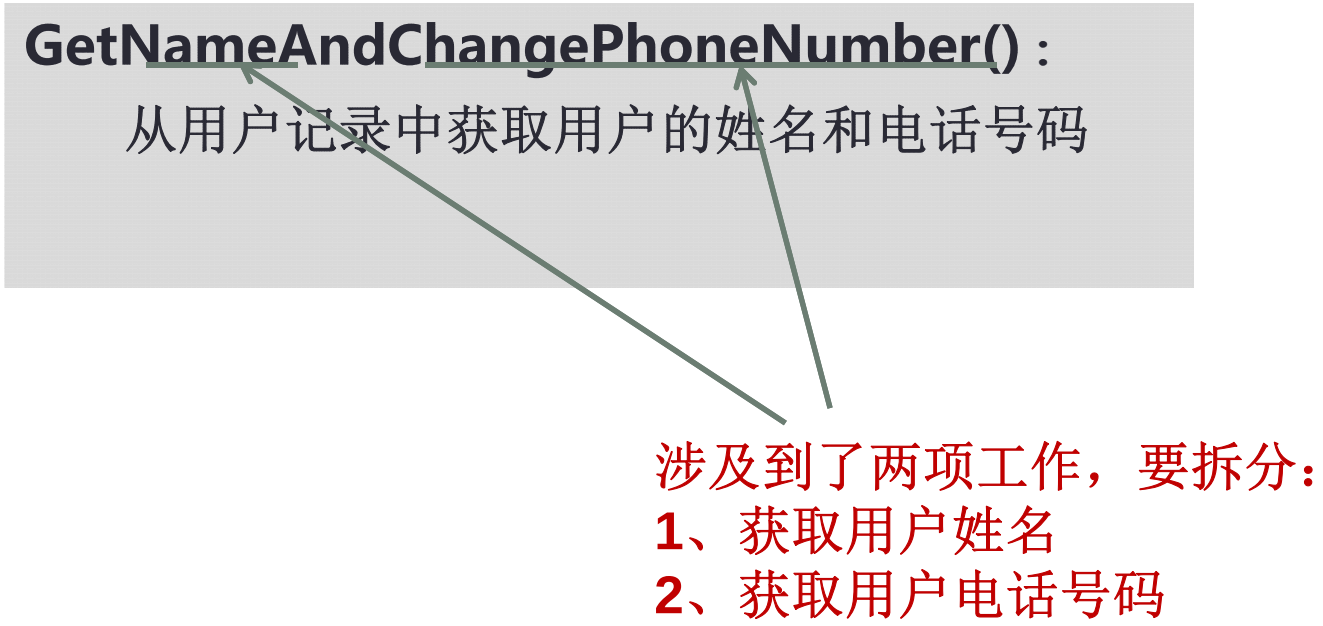
顺序内聚性

- 子程序内包含需要按特定顺序执行的操作，这些步骤共享数据，只有全部执行完毕后才完成一项完整的功能



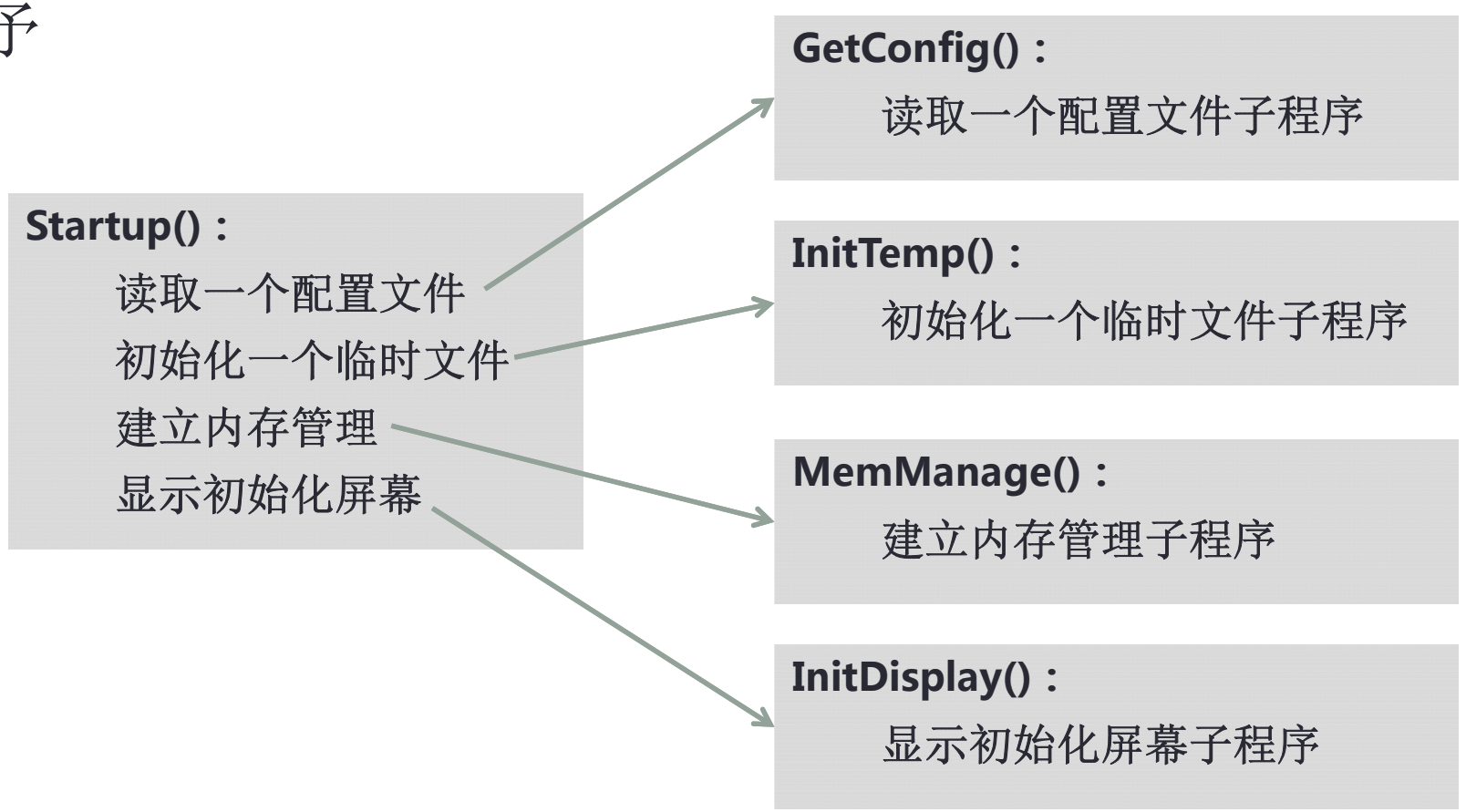
通信内聚性

- 在一个子程序中，两个操作只是使用相同数据，而不存在其它任何联系



临时内聚性

- 因为需要同时执行的原因才被放入同一个子程序



不可取的内聚性

- 不可取的内聚性往往导致一些组织混乱而又难以调试和改进的代码

- 应尽量避免不可取的内聚性的出现
  - 过程内聚性
  - 逻辑内聚性
  - 偶然内聚性

过程内聚性

- 当子程序中的操作是按某一特定顺序进行的，就是过程内聚性
- 过程内聚性vs.顺序内聚性
  - 顺序内聚性中的顺序操作使用的是相同数据
  - 过程内聚性中的操作使用的并不是相同数据

PrintReport()：按一定顺序打印报表

1、打印销售收入报表操作
2、打印支出报表操作
3、打印雇员名单报表操作
4、打印客户名单报表操作

FileOperation()：文件读写

1、打开文件操作
2、读文件操作
3、处理文件内容操作
4、输入结果操作
5、关闭文件操作

逻辑内聚性

- 若干操作被放入一个子程序，通过传入的控制标志选择执行其中的一项操作
  - 其内部操作仅仅是因为控制流，或者说“逻辑”的原因才联系到一起，而不是因为操作之间有何逻辑关联
  - 如InputAll()、ComputAll()、SaveAll()……
  - 如特定条件调用其他子程序的语句组成，通常也是可以的

InputData()：输入用户名、雇员时间卡信息或者库存数据

```
If 输入用户名 then ……
If 输入雇员时间卡 then ……
If 输入库存数据 then ……
Else ……
```

如果代码仅由一系列if或case语句，以及调用其他子程序的语句组成，那么通常也是可以的，如果只发布命令，不做任何处理，称为“事件处理器”

巧合的内聚性

- 各个操作之间没有任何可以看到的关联
  - 如子程序部分最初的例子
  - 难以改进，通常需要重新设计和实现

- 理解各内聚性的内涵
- 把注意力集中在功能上的内聚性
  - 编写功能上内聚性的子程序几乎总是可能的

子程序命名：指导原则

- 描述子程序所作的所有事情
  - 输出结果+副作用
  - 当很难用简单的名称描述时，重新设计
    - 如ComputeReportTotalsAndOpenOutputFile()

- 避免使用无意义的、模糊或表述不清的动词
  - 如HandleCalculation(), PerformServices()
  - 分辨导致含糊的原因是命名还是子程序设计
    - 对目的不明确的子程序重新组织

- 不要仅通过数字来形成不同的子程序名字
  - 如Part1, Part2, OutputUser1, OutputUser2
- 给函数命名时要对返回值有所描述
  - 如cos(), customerId.Next(), printer.IsReady()
- 给过程起名时使用语气强烈的动词加宾语
  - 如PrintDocument(), CheckOrderInfo()
  - 在OO语言中，对象本身已包含在调用语句中
    - Document.Print()而非document.PrintDocument()
    - 更便于继承，如check.PrintDocument()不恰当

- 准确使用对仗词

add/	increment/	open/
begin/	insert/	show/
create/	lock/	source/
first/	min	start
get/	next/	up/
get/	old/	

- 为常用操作确立命名规则

employee.id.Get()
dependent.GetId()
supervisor()
candidate.id()

没人能记住哪个对象应用哪些子程序

子程序的长度

- 理论上，最大长度为一屏代码或打印出来1-2页，约50～150行
  - 对复杂算法，可增长到100～200行（不包括注释行和空行）
  - 对超过200行的代码要小心
- 让内聚性、嵌套层次、变量数量、解释子程序所需的注释数量等来决定子程序的长度

如何使用子程序参数

- 按照 输入—修改—输出 的顺序排列参数
  - 暗含了内部操作发生的顺序
- 考虑自己创建in和out关键字
  - 只起说明性作用
  - 编译器不会强制其使用
    - 可使用const关键字
- 使用所有的参数

C++示例：定义你自己的IN和OUT关键字

```
#define IN
#define OUT
void InvertMatrix(
    IN Matrix originalMatrix,
    OUT Matrix *resultMatrix
);
...

void ChangeSentenceCase(
    IN StringCase desiredCase,
    IN OUT Sentence *sentenceToEdit
);
...

void PrintPageNumber(
    IN int pageNumber,
    OUT StatusType &status
);
```

- 把状态或出错变量放在最后
  - 附属于程序的主要功能，是仅用于输出的参数
- 不要把参数用做工作变量
  - 强调了数据来自何方，C++中可使用const关键字

```
int Sample( int inputVal ) {
   int workingVal = inputVal;
   workingVal = workingVal * CurrentMultiplier( workingVal );
   workingVal = workingVal + CurrentAdder( workingVal );
   ...
   ...
   return workingVal;
}
```

这里，inputVal还维持最初的值

- 若几个子程序都是用了类似参数，应该让这些参数的排列顺序保持一致
  - C语言中strncpy()和memcpy()的参数为目标字符串、源字符串、最大字节数

- 在接口中对参数的假定加以说明
  - 同时在子程序内部和调用子程序的地方说明
  - 断言assertion比注释要好
    - 参数是仅用于输入的、要被修改的、还是仅用于输出的；
    - 表示数量的参数的单位（英寸、英尺、米等）；
    - 如果没有用枚举类型的话，应说明状态代码和错误值的含义；
    - 所能接受的数值的范围；
    - 不该出现的特定数值。

- 把子程序的参数个数限制在大约7个以内
  - 若总需要传递很多参数，说明子程序之间的耦合太过紧密
  - 若向很多不同的子程序传递相同数据，可把这些子程序组成一个类，并将常用数据用作类的内部数据

- 对参数采用某种表示输入、修改、输出的命名规则
  - 如加上i_、m_、o_前缀

- 为子程序传递用以维持其接口抽象的变量或对象
  - E.g. 10个访问器子程序暴露数据成员，当前只需要3个数据
  - 传递数据：数据碰巧由一个对象提供
  - 传递对象：子程序要对这一对象执行操作or经常修改参数表，且每次修改参数都来自同一对象
- 使用具名参数
  - 让形参与实参对应起来
  - 避免将参数放错位置，而编译器却检测不到
- 确保实参和形参相匹配
  - 留意编译器给出的关于参数类型不匹配的警告

有关函数Function

- 函数：有返回值的子程序
  - 最好只有一个返回值，即只能接受仅用于输入的参数
  - 通过返回的值来命名，如sin()、CustomerID()
- 过程Procedure：没有返回值的子程序
  - 可接受任意数量的输入、修改和输出参数

```
if ( report.FormatOutput( formattedReport ) = Success ) then ...
```

函数 or 过程？

改进

- 一：使用过程—用状态变量作为显式参数

```
report.FormatOutput( formattedReport, outputStatus )

if ( outputStatus = Success ) then ...
```

当子程序的主要用途是返回由其名字所指明的返回值时，使用函数，否则使用过程！

- 二：使用函数

```
outputStatus = report.FormatOutput( formattedReport )

if ( outputStatus = Success ) then ...
```

如何设置函数的返回值

- 检查所有可能的返回路径
  - 可在函数开头用一个默认值来初始化返回值
- 不要返回指向局部数据的引用或指针
  - 当一个对象需要返回有关其内部数据信息时，把这些信息保存为类的数据成员
  - 并提供可以返回这些数据成员的访问器子程序

宏子程序和内联子程序（适合C++语言）

- 把宏表达式整个包含在括号内
  - 确保代码按照所预期的方式被展开

- 举例

```
#define Cube( a ) a*a*a
```

  - 存在的问题
    - Cube(x+1)展开为；x+1*x+1*x+1
    - 当a可分割时出现错误

- 修改

```
#define Cube( a ) (a)*(a)*(a)
```

  - 存在的问题
    - 3/Cube(x) = 3/x*x*x ≠ 3/(x*x*x)

- 再修改

```
#define Cube( a ) ( (a)*(a)*(a) )
```

- 把含有多条语句的宏用大括号括起来
- 举例

```
#define LookupEntry( key, index ) {     \
   index = (key - 10) / 5;                \
   index = min( index, MAX_INDEX );       \
   index = max( index, MIN_INDEX );
...
  }

for ( entryCount = 0; entryCount < numEntries; entryCount++ )
   LookupEntry( entryCount, tableIndex[ entryCount ] );
```

只有该条语句被循环执行

- 用给子程序命名的方法来给展开后代码形同子程序的宏命名
  - 当想用子程序来替换宏的时候，只需要修改相关的子程序

宏的替换方案

- 除非万不得已，否则不要用宏来代替子程序
  - const用于定义常量
  - inline定义可被编译为内联代码的函数
  - template用于以类型安全的方式定义各种标准操作，如min,max
  - enum用于定义枚举类型
  - typedef用于定义简单的类型替换

内联子程序

- 允许程序员在编写代码时把代码当作子程序
  - 编译器在编译期间将每一处调用inline子程序的地方都转换为内嵌的代码，从而避免子程序调用的开销
  - 可产生非常高效的代码
- 注意：节制使用inline子程序
  - C++中需将inline子程序的实现代码写在头文件里，违反了封装原则
  - 会增加整体代码的长度

CHECKLIST: High-Quality Routines
核对表：高质量的子程序

大局事项

- ❑ 创建子程序的理由充分吗？
- ❑ 一个子程序中所有适于单独提出的部分是不是已经被提出到单独的子程序中了？
- ❑ 过程的名字中是否用了强烈、清晰的“动词＋宾语”词组？函数的名字是否描述了其返回值？
- ❑ 子程序的名字是否描述了它所做的全部事情？
- ❑ 是否给常用的操作建立了命名规则？
- ❑ 子程序是否具有强烈的功能上的内聚性？即它是否做且只做一件事，并且把它做得很好？
- ❑ 子程序之间是否有较松的耦合？子程序与其他子程序之间的连接是否是小的（small）、明确的（intimate）、可见的（viaible）和灵活的（flexible）？
- ❑ 子程序的长度是否是由其功能和逻辑自然确定，而非遵循任何人为的编码标准？

参数传递事宜

- 整体来看，子程序的参数表是否表现出一种具有整体性且一致的接口抽象？
- 子程序参数的排列顺序是否合理？是否与类似的子程序的参数排列顺序相符？
- 接口假定是否已在文档中说明？
- 子程序的参数个数是否没超过 7 个？
- 是否用到了每一个输入参数？
- 是否用到了每一个输出参数？
- 子程序是否避免了把输入参数用做工作变量？
- 如果子程序是一个函数，那么它是否在所有可能的情况下都能返回一个合法的值？