step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与封装（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、指针、结构）

归纳与推广（程序的本质/**程序设计的本质**）

程序的本质

- **什么是程序?**
  - 一组连续的相互关联的计算机指令
  - 更通俗地讲，是指示计算机处理某项计算任务的任务书，计算机根据该任务书，执行一系列操作，并产生有效的结果。
  - 程序的本质是对计算机的**计算过程**及其**对象（数据）**进行描述。

程序的本质

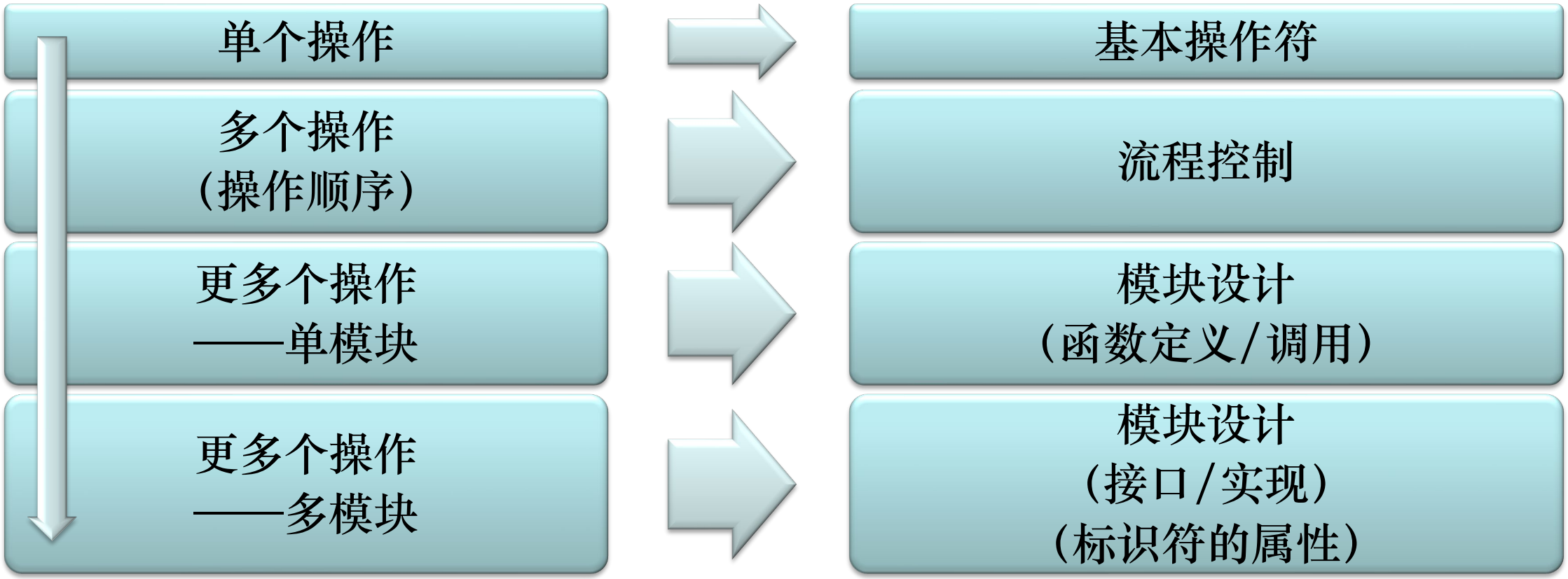
- PASCAL语言之父 Niklaus Emil Wirth（1984年的图灵奖得主）曾给出下列经典公式：

  程序 = **算法** + **数据结构**

算法（algorithm）

- 对数据进行加工的步骤

对应一个程序的总流程



例1

- **问题：求两个正整数的最大公约数。**
  - 输入：两个正整数 m、n
  - 输出：m、n 的最大公约数 gcd

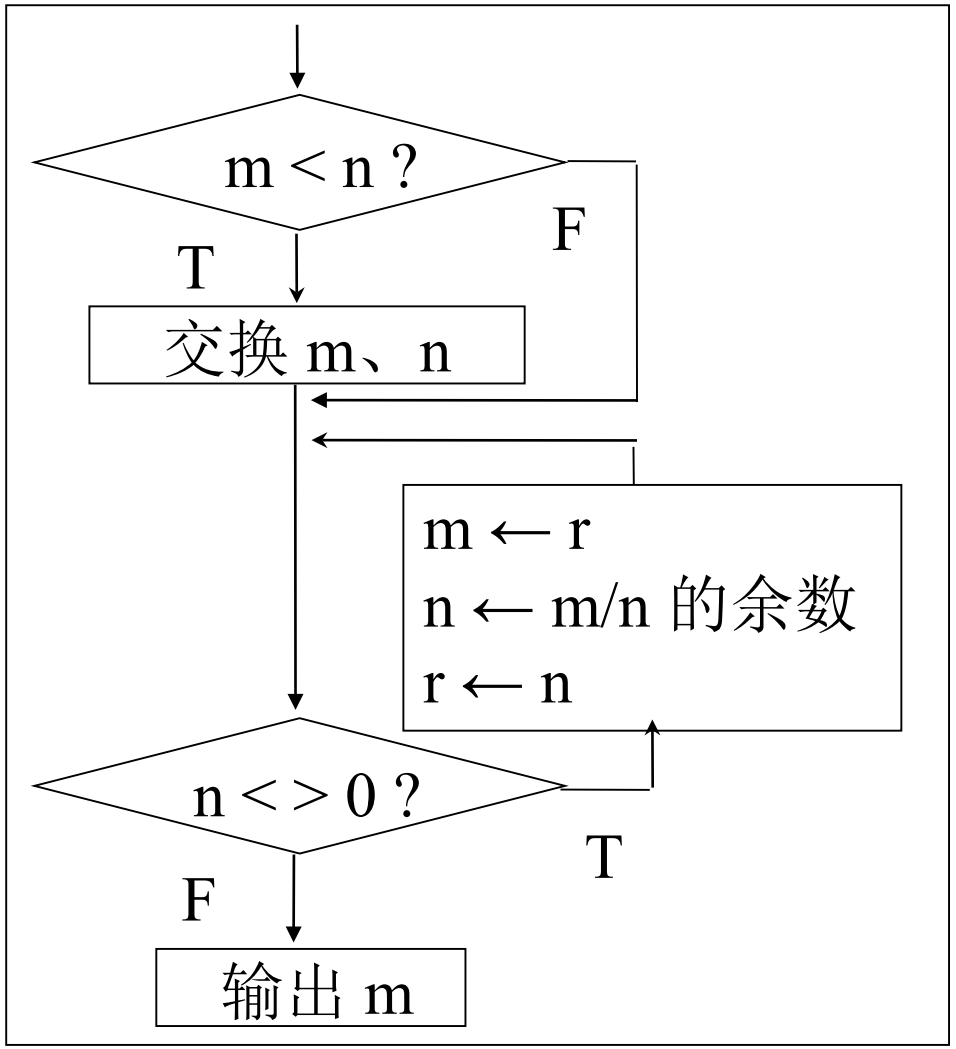
例1

- **问题：求两个正整数的最大公约数。**
  - 输入：两个正整数 m、n
  - 输出：m、n 的最大公约数 gcd
  - 算法：
    - 辗转相除法，又叫欧几里得算法（Euclidean algorithm）
    - （用自然语言描述）

第一步：如果 m < n，则交换 m 和 n，否则转第二步；
第二步：如果 n 为0，则转第五步，否则转第三步；
第三步：将 n 赋值给 r，m 除以 n 的余数赋给 n；
第四步：将 r 赋值给 m，转第二步；
第五步：输出 m，m 为“最大公约数”。

例1

- **问题：求两个正整数的最大公约数。**
  - 输入：两个正整数 m、n
  - 输出：m、n 的最大公约数 gcd
  - 算法：
    - （用流程图描述）



例1

- **问题：求两个正整数的最大公约数。**
  - 输入：两个正整数 m、n
  - 输出：m、n 的最大公约数 gcd
  - 算法：
    - （用伪代码描述）

pseudocode

```
IF m < n THEN SWAP m, n
WHILE n < > 0
    r = n
    n = m MOD n
    m = r
END
OUTPUT m
```

- **算法种类**
  - 数值问题的求解
    - 求面积
    - 解方程
    - …
  - 非数值问题的求解
    - 排序
    - 信息检索
    - 推理
    - …

- **解决一个问题的算法往往有多种**
  - 易于理解（GCD：穷举约数、求交集最大值；　　检索：顺序）
  - 效率更高（ GCD： Stein；　　检索：分治）
  - 便于计算机实现（ GCD： 辗转相除）

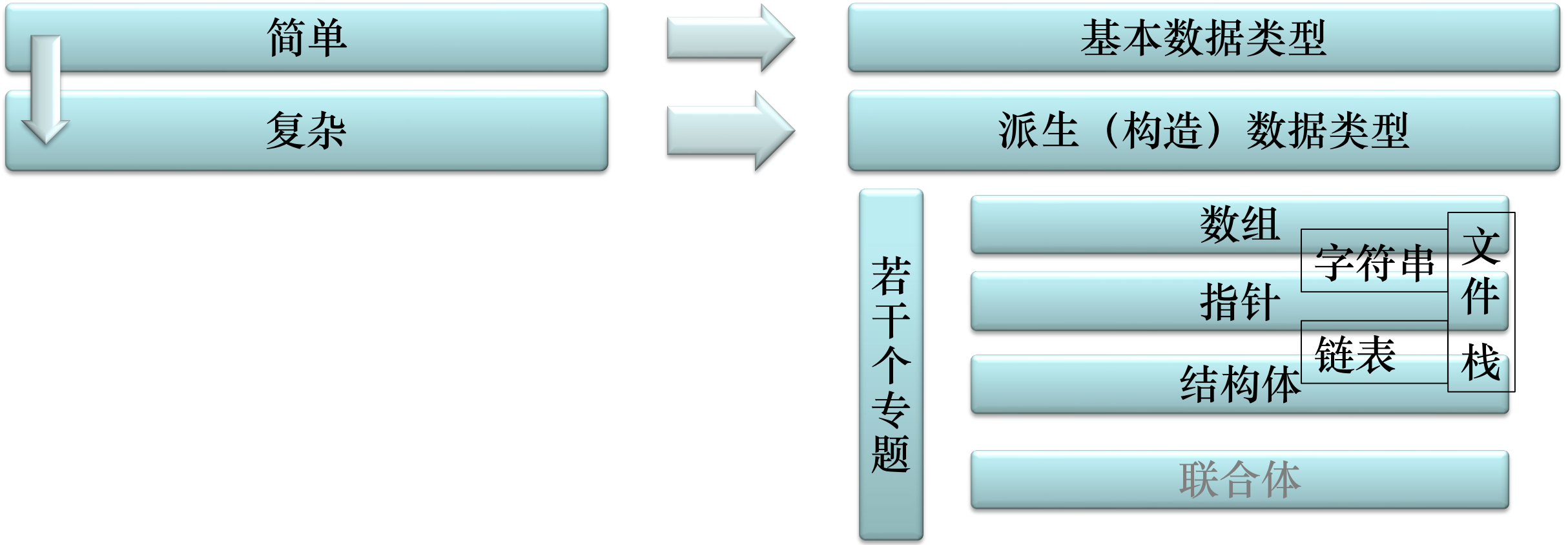
- **目前计算机与人类的计算方式不尽相同，选择适合计算机的算法**
  - 求解高阶线性方程组
    - 克莱姆法则（Cramer‘s Rule）：系数行列式，消去法，计算量大
    - 雅可比迭代法（Jaccobi Iterative Method）

- 随着计算机科学与应用的发展，新的问题不断出现，**需要根据计算机的结构与工作特点设计各种各样的新算法**

- **根据Donald Ervin Knuth（1974年的图灵奖得主）的描述，计算机算法应具有以下特征：**
  - 输入量与**输出**量：一个算法通常有〇个或多个输入量，并有一个或多个输出量，也即，给定初始状态或输入数据，能够得到结果状态或输出数据。（处理轨迹数据，输出是什么？）
  - **确定**性：算法中每一个步骤应当是确定无歧义的，而不能是含糊的、模棱两可的。（排序，从小到大？从大到小？求最值，最大值？最小值？）
  - **有穷**性：一个算法应包含有限的操作步骤，以及有限个输入量与输出量。（输入若干个数，以什么结束？）
  - **有效**性：即可行性，算法中的每一个步骤应能通过已经实现的基本运算有效地执行。（负数的平方根？）

数据结构（data structure）

- 对数据的描述及在计算机中存储和组织的方式



例2

- **问题：洗牌**
  - 输入：一副牌
  - 输出：一副牌

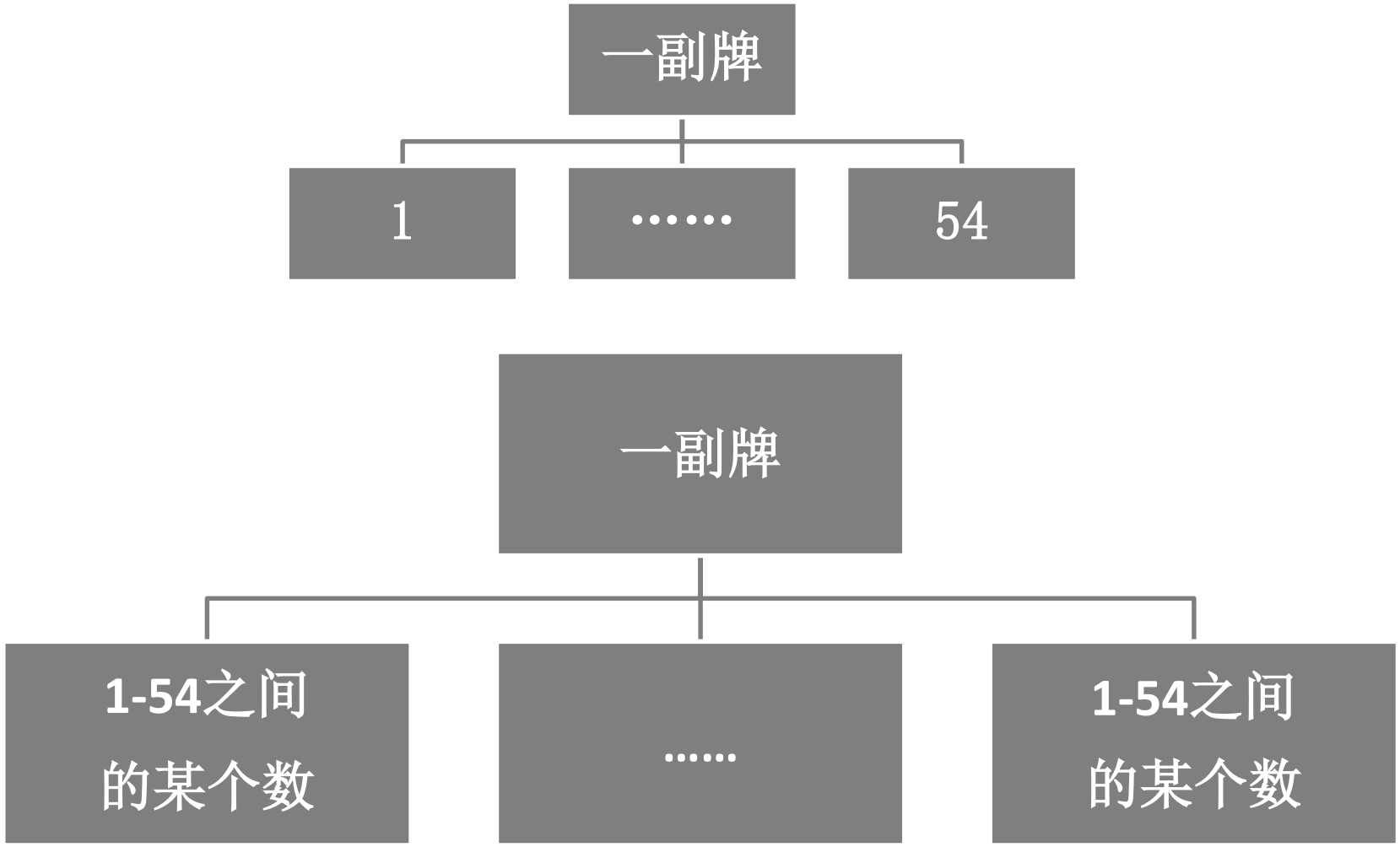
例2

- **问题：洗牌**
  - 输入：一副牌
  - 输出：一副牌
  - 数据结构：
    - （用自然语言描述）

洗牌前：一副牌（用整数 1-54 表示 54 张牌，
用整型数组 acard[54] 表示 54 个位置）
洗牌后：一副牌（1-54 随机存于 acard[54] 的 54 个元素中）

例2

- **问题：洗牌**
  - 输入：一副牌
  - 输出：一副牌
  - 数据结构：
    - （用框图描述）



例2

- **问题：洗牌**
  - 输入：一副牌
  - 输出：一副牌
  - 数据结构：
    - （用伪代码描述）

输入数据结构：

```
INT array_card[1…54]  //位置
INT n  //1…54                //代表牌
```

输出数据结构：

```
FOR i FROM 1 TO 54
        n = random of 1~54
        array_card[i] = n
END
```

算法与数据结构并非泾渭分明

算法：机器学习	• 借助于大量数据训练模型参数
数据结构：链表	• 借助于基本操作、流程控制方法来组织其数据成员

step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与封装（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、指针、结构）

归纳与推广（程序的本质/**程序设计的本质**）

- **例1 设计C程序，实现求解两个整数的最大公约数。**
  - 根据最大公约数的求解算法，该问题的求解程序首先可以用一个分支流程使被除数不小于除数，然后需要一个循环流程辗转求余数，最终得到两个整数的最大公约数。

例1

- **问题：求两个正整数的最大公约数。**
  - 输入：两个正整数 m、n
  - 输出：m、n 的最大公约数 gcd
  - 算法：
    - （用伪代码描述）

pseudocode

```
IF m < n THEN SWAP m, n
WHILE n < > 0
    r = n
    n = m MOD n
    m = r
END
OUTPUT m
```

```
int MyGcd1(int m, int n)
{
        if(m < n)
        {    int temp = m;
             m = n;
             n = temp;
        }
        while(n != 0)
        {    int r = n;
             n = m % n;
             m = r;
        }
        return m;
}
```

```
int main()
{
     ……
     int gcd = MyGcd1(x, y);
     ……
     return 0;
}
```

- 实际实现时，可以将其中两个正整数的交换用一个独立的函数来实现，从而使程序的结构得以优化：

```
int MyGcd2(int m, int n)
{
      if(m < n) MySwap(&m, &n);
      while(n != 0)
      {
            int r = n;
            n = m % n;
            m = r;
      }
      return m;
}
```

```
int main()
{
      ……
      int gcd = MyGcd2(x, y);
      ……
      return 0;
}
```

```
void MySwap(int *pm, int *pn)
{
      int temp = *pm;
      *pm = *pn;
      *pn = temp;
}
```

- 实际实现时，可以将其中两个正整数的交换用一个独立的函数来实现，从而使程序的结构得以优化：

```
int MyGcd2(int m, int n)
{
      //if(m < n) MySwap(&m, &n);
      while(n != 0)
      {
            int r = n;
            n = m % n;
            m = r;
      }
      return m;
}
```

```
int main()
{
      ……
      int gcd = MyGcd2(x, y);
      ……
      return 0;
}
```

- 还可以用递归函数进一步优化程序的结构：

```
int MyGcd3(int m, int n)
{
    if(m % n)
        return MyGcd3(n, m % n);
    return n;
}
```

```
int main()
{
    ……
    int gcd = MyGcd3(x, y);
    ……
    return 0;
}
```

例2

- **问题：洗牌**
  - 输入：一副牌
  - 输出：一副牌
  - 数据结构：
    - （用伪代码描述）

C语言数组的下标从 0 开始

随机数的生成

if (n 未出现过)

输入数据结构：

```
INT array_card[1…54]  //位置
INT n  //1…54                //代表牌
```

输出数据结构：

```
FOR i FROM 1 TO 54
        n = random of 1~54
        array_card[i] = n
END
```

```
#include <cstdlib>
int array_card[54] = {0};                          //初始化所有位置为空
void MyShuffle( )
{   for(int j, i = 0; i <= 53; )
    {     int n = 1+int(54.0*rand()/RAND_MAX);//随机一个数 (1~54)



          if(          )                           //该数未曾出现过
          {     array_card[i] = n;
                ++i;
          }
    }
}
```

```
#include <cstdlib>
int array_card[54] = {0};                          //初始化所有位置为空
void MyShuffle( )
{   for(int j, i = 0; i <= 53; )                   //注意 j 的作用域
    {     int n = 1+int(54.0*rand()/RAND_MAX);//随机一个数 (1~54)
          for(j = 0; j < i; ++j)
                if(n == array_card[j])             //如果该数曾出现过
                     break;
          if(j >= i)                               //该数未曾出现过
          {     array_card[i] = n;
                ++i;
          }
    }
}
```

- 还可以随机一个位置，依次放入 1~54

```
#include <cstdlib>
int array_card[54] = {0};                          //初始化所有位置为空
void MyShuffle( )
{   int n, j;
    for(n = 1; n <= 54; ++n)
    {     int i = 54.0 * rand()/RAND_MAX;          //随机一个下标（0~53）
          if(array_card[i] == 0)                   //如果是空位置
                array_card[i] = n;
          else                                     //如果不是空位置
                ...
```

```
            ...
            for(j = 0; j <= 53; ++j)          //从头找一个空位置
                if(array_card[j] == 0)
                {
                    array_card[j] = n;
                    break;
                }
    }
}
```

程序设计的本质

- **狭义的程序设计：**
  - 由设计好的算法和数据结构产生程序的过程 **coding**
  - 其本质就是选用恰当的程序设计方法（即范型）和程序设计语言元素实现算法和数据结构。

程序设计的本质

- 广义的程序设计：
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- 本课程的主要内容是基于**过程式程序设计范型**和**C语言**，学习选用恰当的程序设计语言元素解决简单的问题，兼顾狭义和广义的程序设计 所涉及的基本方法和概念。
  - 如果有算法和数据结构，根据算法选用…（狭义的程序设计）
  - 如果没有，则要先分析问题，再……（广义的程序设计）
  - 如果有可用的代码、软件，直接使用

流程的控制
模块的设计
操作的描述
数据的描述

- **问题的分析（analysis）**
  - 搞清楚要解决的问题并给出问题的明确定义，即：做什么？
  - 主要任务是通过演绎、归纳等逻辑思维方式找出已知数据和未知数据，以及二者之间的关系
  - 尽量降低子问题之间的耦合程度

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- **算法和数据结构设计（design）**
  - 给出问题的解决方案，即：如何做?
  - 主要任务就是要进行算法的设计，以及已知数据与未知数据的合理组织
  - 不同程序设计范型的程序设计思路不同。
    - 过程式程序设计：设计思路是功能分解（数据结构设计和算法设计往往是分开考虑的）
    - 对象式程序设计：设计思路主要是数据抽象（数据结构和算法结合在对象和类中一并设计）
    - 函数式与逻辑式程序设计：需要明确用哪些函数与规则完成已知数据到未知数据的转换。

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

程序设计范型

- 基于不同的计算模式来描述计算形成不同的程序设计范型（programming paradigms）。

- 典型的程序设计范型有：
  - **过程式（面向功能）**
  - 对象式（面向数据）
  - 函数式
  - 逻辑式
  - …

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

过程式程序设计（procedural programming）

- 一种基于过程调用的程序设计范型。
- 过程：封装了一系列计算步骤的子程序（C 语言中的子程序表现为函数），一个过程可以在程序执行的任一时间点被其他过程或自身所调用。
- 从程序员角度看，采用这种程序设计范型
  - 调用者不必知道数据的具体表示和操作。
  - 这种方式实现了一定程度的软件复用。

- **在过程式程序中，**
  - 一方面，一个过程对应一个子功能，过程内部的计算步骤按顺序、分支或循环流程执行，对计算任务的功能描述比较清晰，
  - 另一方面，对数据的描述和对数据的操作相分离，这种模式与冯·诺依曼体系结构比较吻合。
  - 早期的程序设计大都采用过程式程序设计范型，支持该范型的程序设计语言有FORTRAN、COBOL、BASIC、Pascal、C、C++、Lisp，后期出现的支持对象式程序设计的语言一般也支持过程式程序设计。

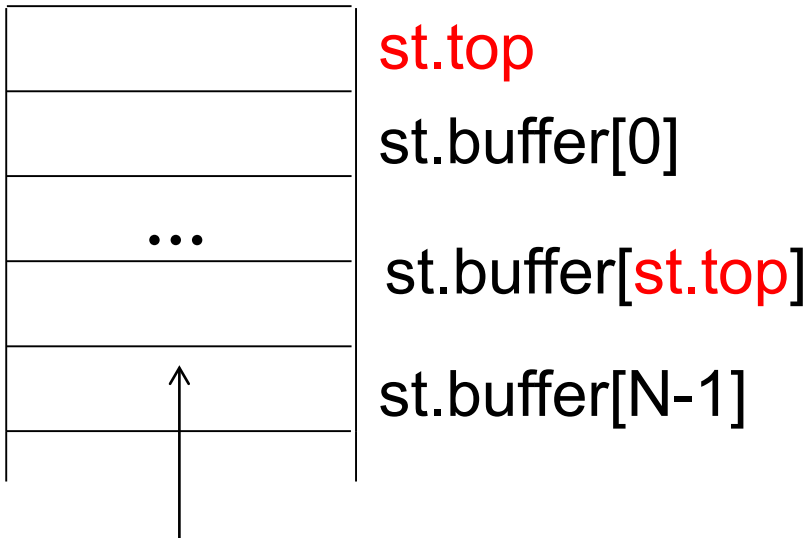
- **过程式程序设计的基本做法，包括流程控制方法、模块设计方法、数据和操作的描述方法等，掌握这些基本做法，可以为学习其他范型的程序设计方法打下坚实的基础。**

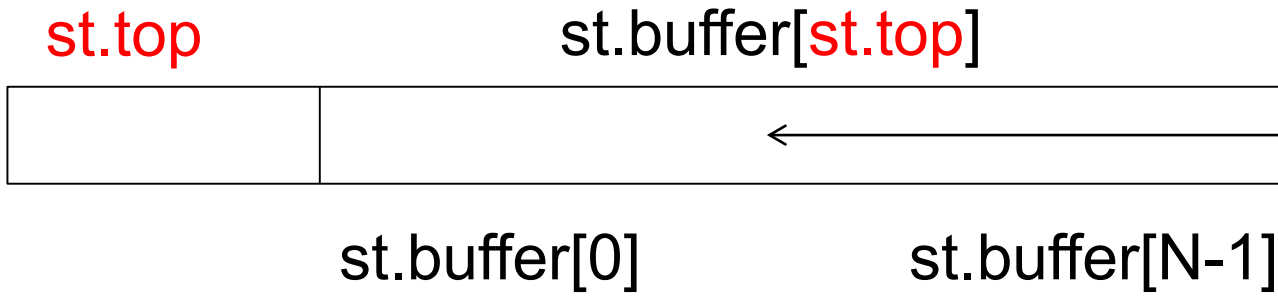
- 过程式程序设计的不足之处
  - 数据类型的定义与操作的定义之间缺乏显式的联系；
  - 对数据缺乏足够的保护，因为数据表示仍然是公开的，可以通过其他函数操作数据或直接操作数据；
  - 程序的功能常常会随着需求的改变而变化，功能的变化往往会导致整个程序结构的变动，从而使程序难以维护。

例：栈

- 一种带有操作约束的结构类型
  - 对栈只能进行两种操作
    - 进栈（增加一个元素）
    - 出栈（删除一个元素）
  - 这两个操作必须在栈的同一端（称为栈顶，top）进行
    - 即后进先出（Last In First Out，简称LIFO）
    - 若 **push(x); pop(y);** 则 **x==y**
  - （函数调用过程中用栈保存函数的局部变量、参数、返回值以及返回地址）

```
typedef struct
{    int top;
     int buffer[N];
}Stack;
Stack st;
```





栈的实现（C/C++，无过程抽象）

方法一

```
const int N = 100;

typedef struct
{
    int top;
    int buffer[N];
}Stack;
```

```
#include <stdio.h>
int main()
{
    Stack st;           //定义栈数据
    st.top = -1;        //初始化栈
    st.top++;
    st.buffer[st.top] = 12;    //存数入栈
    int x = st.buffer[st.top];//取数出栈
    st.top--;
    printf("x: %d\n", x);
    return 0;
}
```

- **方法一：使用者也是实现者**
  - 实现与使用混在一起，使用者（如main函数的编写者）
    - 必须知道数据表示才能完成操作，eg. “top” or “index”；
    - 可以修改数据表示，数据没有得到保护，eg. 将成员buffer的数据类型改为char；
    - 数据表示发生变化将影响操作代码；
    - 会忘记初始化，eg. 误把st.top++写成st.top--；
  - 需求的改变会导致整个程序结构的变动，难以维护
    - eg. 增加一个栈溢出标记，每次操作前先判断，以防操作异常

栈的实现（C/C++，过程抽象）

方法二

```
const int N = 100;

typedef struct
{
    int top;
    int buffer[N];
}Stack;
```

```
#include <stdio.h>
int main()
{
    Stack st;          //定义栈数据
    Init(&st);         //初始化栈
    Push(&st, 12);     //存数入栈

    int x;
    Pop(&st, &x);      //取数出栈
    printf("x: %d\n", x);
    return 0;
}
```

C

- 过程

```
bool Push(Stack *s, int i)
{
    if(s -> top != N-1)
    {
      s -> top++;
      s -> buffer[s -> top] = i;
      return true;
    }
    else
    {
      printf("Stack is overflow\n");
      return false;
}
} //入栈
```

```
bool Pop(Stack *s, int *i)
{
    if(s -> top != -1)
    {
      *i = s -> buffer[s -> top];
      s -> top--;
      return true;
}
    else
    {
      printf("Stack is empty.\n");
      return false;
    }
} //出栈
```

```
void Init(Stack *s)
{
      s -> top = -1;
} //初始化栈
```

栈的实现（C++，过程抽象）

方法三

```
const int N = 100;

typedef struct
{
    int top;
    int buffer[N];
}Stack;
```

```
#include <stdio.h>
int main()
{
    Stack st;           //定义栈数据
    Init(st);           //初始化栈
    Push(st, 12);       //存数入栈

    int x;
    Pop(st, x);         //取数出栈
    printf("x: %d\n", x);
    return 0;
}
```

C++

- 过程

```
bool Push(Stack &s, int i)
{
    if(s.top != N-1)
    {
      s.top++;
      s.buffer[s.top] = i;
      return true;
    }
    else
    {
      printf("Stack is overflow\n");
      return false;
}
} //入栈
```

```
bool Pop(Stack &s, int &i)
{
    if(s.top != -1)
    {
      i = s.buffer[s.top];
      s.top--;
      return true;
}
    else
    {
      printf("Stack is empty.\n");
      return false;
    }
} //出栈
```

```
void Init(Stack &s)
{
      s.top = -1;
} //初始化栈
```

- **方法二：使用者不一定是实现者**
  - 初始化、入栈及出栈过程均抽象为函数来实现
    + 一个函数对应一个功能，可以清晰地描述计算任务，易维护；
    + 能实现一定程度的软件复用，eg. 在pop/push函数里判断导致操作异常的情况
    + 数据表示发生变化不影响使用者的操作代码；
    + 上述函数的声明与结构类型的构造可以放头文件中，以便于调用，调用者不必知道数据的表示；
    - Init、Push、Pop函数在形式上跟其他函数没有区别，其他操作或函数也能操作数据，数据表示仍然是公开的，对数据缺乏足够的保护
    - 仍会忘记调用Init(&st)对栈进行初始化；
  - 数据类型的构造与对数据操作的定义是分开的
    + 与冯诺依曼体系结构吻合
    - 二者之间没有显式的联系
  - 需求的改变仍可能会导致整个程序结构的变动，难以维护
    - eg. 改成在链表上实现栈
- **方法三：与方法二类似；引用可避免指针的弊端**

栈的实现（C/C++，过程抽象的缺陷）

```
const int N = 100;

typedef struct
{
    int top;
    int buffer[N];
}Stack;
```

```
#include <stdio.h>
int main()
{
    Stack st;          //定义栈数据
    Init(&st);         //初始化栈
    Push(&st, 12);     //存数入栈

    int x;
    Pop(&st, &x);      //取数出栈
    printf("x: %d\n", x);
    return 0;
}
```

```
void Fun(Stack *s)
{
    .....跟push不太一样
} //把栈修改成普通数组!
```

```
Fun(st);
```

```
st.top--;
st.buffer[st.top] = 12;
```

栈的实现（C++，数据抽象）

方法四

```
const int N = 100;

class Stack
{
public:
    Stack();
    bool Push(int i);
    bool Pop(int& i);
private:
    int top;
    int buffer[N];
};
```

```
#include <stdio.h>
int main()
{
    Stack st;   //定义栈数据
    Stack st;   //自动调用st.Stack()初始化
                //不允许 st.top = -1;
    st.Push(12);     //不允许 st.top++;或
                //st.buffer[st.top] = 12;
    int x;
    st.Pop(x); //不允许 st.Fun();
    cout << "x: " << x << endl;
    return 0;
}
```

C++

- 对象

```
bool Stack::Push(int i)
{
    if(top != N-1)
    {
        top++;
        buffer[top] = i;
        return true;
    }
    else
    {
        cout << "Stack is overflow\n";
        return false;
    }
} //入栈
```

```
bool Stack::Pop(int& i)
{
    if(top != -1)
    {
        i = buffer[top];
        top--;
        return true;
    }
    else
    {
        cout << "Stack is empty\n";
        return false;
    }
} //出栈
```

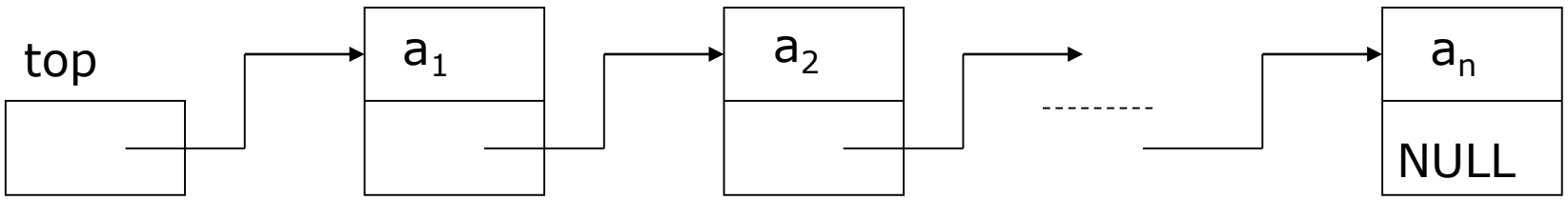
```
Stack::Stack()
{
    top = -1;
} //初始化栈
```

- **方法四：使用者常常不是实现者**
  - 数据类型的构造和对数据操作的定义构成了一个整体，对数据操作的定义是数据类型构造的一部分
  - 只能通过提供的成员函数来操作数据
    - \+ 保护数据
    - \+ 自动进行初始化；
    - \- 冗余
  - 需求的改变通常不会导致整个程序结构的变动，易于维护
    - \- eg．改用链表实现栈，对使用者没有影响
    - \- eg．添加成员函数或其他功能，对已有成员函数或功能代码没有影响

用链表实现栈类Stack

```
class Stack
{
public:
        Stack();
        bool push(int i);
        bool pop(int& i);
private:
        struct Node
        {    int content;
             Node *next;
        } *top;
};
```

```
const int N = 100;
class Stack
{
public:
    Stack();
    bool Push(int i);
    bool Pop(int& i);
private:
    int top;
    int buffer[N];
};
```



```
Stack::Stack() { top = NULL; }

bool Stack::Push(int i)
{    Node *p = new Node;
     if(p == NULL)
     {    cout << "Stack is overflow\n";
          return false;
     }
     else
     {    p -> content = i;
          p -> next = top;
          top = p;
          return true;
     }
}
```

```
bool Stack::Pop(int& i)
{    if(top == NULL)
     {    cout << "Stack is empty\n";
          return false;
     }
     else
     {    Node *p = top;
          top = top -> next;
          i = p -> content;
          delete p;
          return true;
     }
}
```

```
int main()
{    Stack st;
     st.Push(12);
     int x;
     st.Pop(x);
     …
     return 0;
} //修改栈的实现之后，使用者的代码不受影响
```

- **栈**是一种数据结构。其特点是LIFO。其实现容易看出过程抽象与数据抽象两种方法的特点。

用C实现数据抽象（不方便，不完善）

```
//一个C++程序
class A
{
      int x, y;
public:
      void f();
      void g(int i)
      {
            x = i;
            f();
      }
};
......
```

```
A a, b;
a.f();
a.g(1);
b.f();
b.g(2);
```

```
//功能上等价的C程序
typedef struct
{
      int x, y;
}A;

void f_A(A *this);
void g_A(A *this, int i)
{
      this -> x = i;
      f_A(this);
}
......
```

```
A a, b;
f_A(&a);
g_A(&a, 1);
f_A(&b);
g_A(&b, 2);
```

- **基于不同的计算模式来描述计算形成不同的程序设计范型（programming paradigms）。**

- 典型的程序设计范型有：
  - 过程式（面向功能）
  - **对象式（面向数据）**
  - 函数式
  - 逻辑式
  - …

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

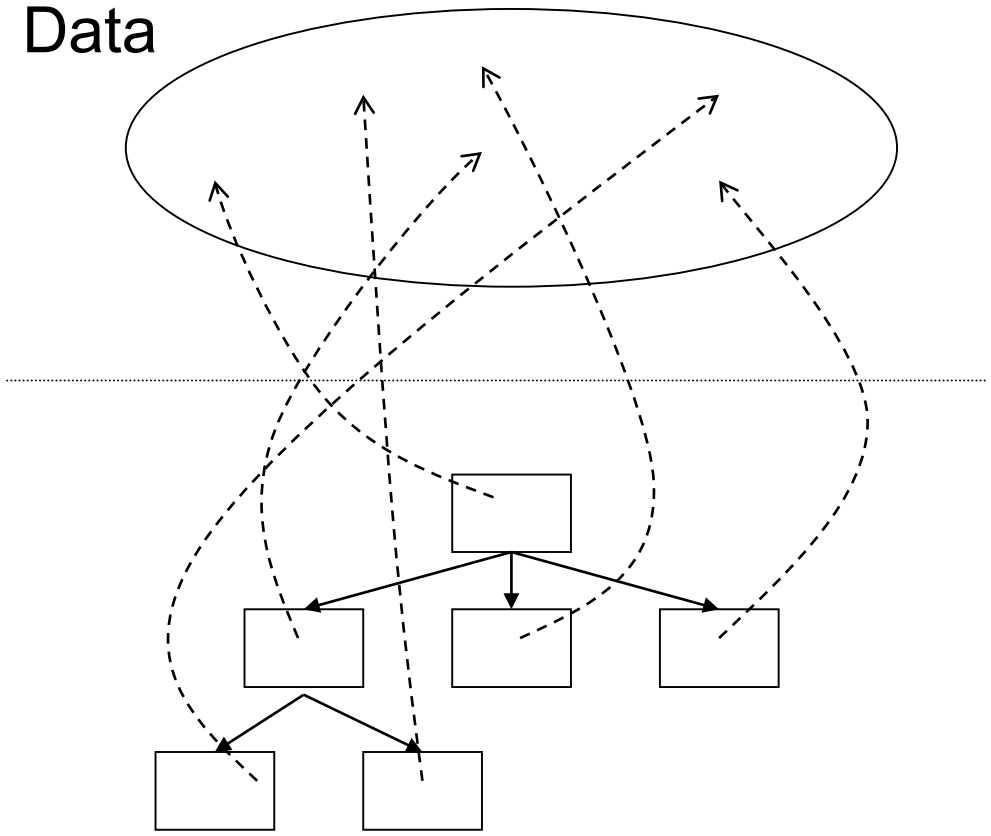
对象式程序设计（object-oriented programming）

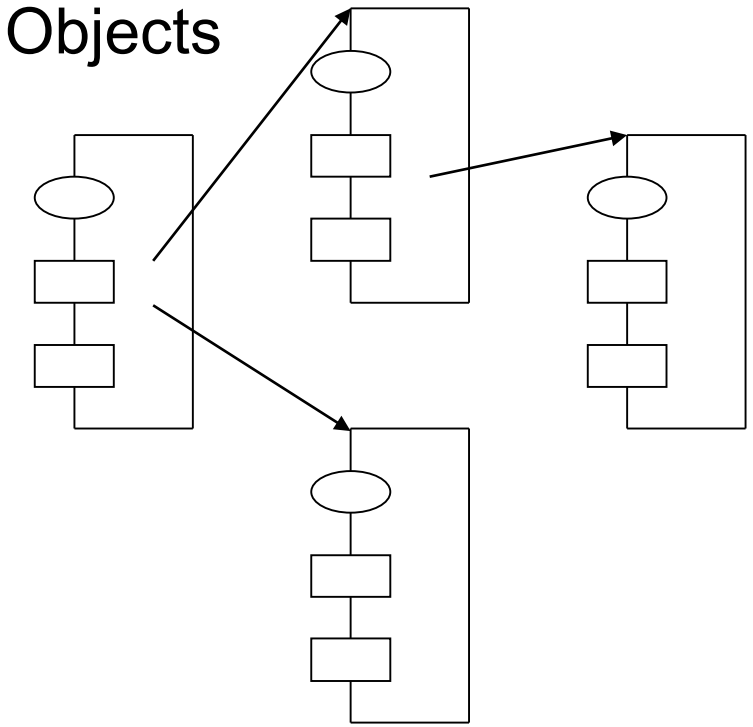
- 一种基于对象及其操作的程序设计范型。
- 这里的对象指的是一组数据（若干描述实体属性的变量值）的实体。
- 对数据的操作通过向对象发送消息（调用对象的操作）来实现。
- 对象的数据特征（数据成员）及其操作特征（成员函数）在相应的类（相当于过程式程序中构造的类型）中描述。
- 一个类所描述的对象特征可以从其他的类继承（派生类，具有相同数据特征和操作特征，并添加新的数据成员和成员函数）。

- 对象式程序设计是目前大型程序的主流设计范型，能比较好地支持这种范型的程序设计语言有Simula、Ada、Smalltalk、Java、C++和Python等。

- 在对象式程序设计中，
  - 对数据的操作必须通过相应的对象来调用，这种模式加强了对数据的保护
  - 一个对象对应一个待处理的数据实体，对数据的描述较清晰
  - 数据的描述通常是实际应用中相对稳定的实体，功能上的变化通常只涉及程序局部范围内的操作调整，对象式程序设计使得程序的开发与维护不至于牵一发而动全身，从而能够更好地适应需求的改变

  - 例如，对于一个企业信息管理系统，原有的产品信息发布功能需要增加某特殊产品的发布前审核功能，那么，在对象式程序中，只要派生一个特殊产品类，在其中增加审核操作函数，然后在产品发布程序前增加该特殊产品对象的审核操作调用即可，而若是在过程式程序中，则需要增加一个产品属性变量，然后可能需要修改所有产品相关的操作子程序


Data
Sub-programs


Objects

- 对象式程序设计的不足之处在于对程序的整体功能描述不明显，程序会包含较多的冗余信息，程序的效率往往不高，不太适合小型应用软件的开发。

- **过程式程序是对象式程序的基础，对象式程序的局部及类定义中的数据操作一般都是过程式程序。**

- 基于不同的计算模式来描述计算形成不同的程序设计范型（programming paradigms）。

- 典型的程序设计范型有：
  - 过程式（面向功能）
  - 对象式（面向数据）
  - **函数式**
  - 逻辑式
  - …

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- **函数式程序设计（functional programming）**
  - 一种基于函数计算的程序设计范型
  - 这里的函数用来描述变量之间的关系，也可以看作数值，函数的参数也可以是函数，从一连串的函数计算得出最终计算结果，递归函数理论和lambda演算是其理论基础。
  - (* 5 2)
  - (if (> 10 20) (print "Hello") (print "World"))

```
int n = [](int x, int y)->int { return x*y; }(3, 4);
double x = integrate([](double x)->double { return x*x; }, 0, 1);
```

- 通常用于人工智能领域程序的开发，能够比较好地支持这种范型的程序设计语言有Haskell、Lisp和Scheme等

- 代码简洁，开发快速，Lisp代码的长度可能是C代码的二十分之一；接近自然语言，易于理解；更方便的代码管理，易于"并发编程"

- **函数式程序设计秉承了过程式程序设计的模块化与代码重用思想，二者有类似的函数调用、参数、返回值及变量的作用域等概念，所不同的是，函数式程序设计语言几乎没有命令元素，很少控制程序的执行次序。**

- 基于不同的计算模式来描述计算形成不同的程序设计范型（programming paradigms）。

- 典型的程序设计范型有：
  - 过程式（面向功能）
  - 对象式（面向数据）
  - 函数式
  - **逻辑式**
  - …

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

逻辑式程序设计（logical programming）

- 是一种基于事实和规则及推理的程序设计范型
  - 规则用来描述变量之间的关系，从一连串基于事实和规则的推理得出最终计算结果，谓词演算（Predicate Calculus）是其理论基础。

```
gcd (A, 0, A).
gcd (A, B, D):-(A>B), (B>0), R is A mod B,  gcd(B, R, D).
gcd (A, B, D):-(A<B),  gcd (B, A, D).
```

  - 逻辑式程序设计通常用于专家系统和自动化定理证明程序的开发，支持这种范型的程序设计语言有Mercury和Prolog等。

what ~~how~~

- 逻辑式程序设计往往关注于问题是什么，而不是如何解决一个问题。数学家和哲学家认为逻辑是有效的理论分析工具，是解答问题的可靠方法，逻辑式程序设计实现了这一过程的自动化。随着大数据时代的到来，面对浩瀚的数据，人们在迷惑“这些数据能解决什么问题” 之前往往先被“这些数据是什么？”所困惑，逻辑式程序设计正在发挥日益重要的作用。

- **不过，基于冯·诺依曼体系结构的计算机运行逻辑式程序效率不高，有经验的程序员往往会在能保证正确性的前提下，在逻辑式程序中嵌入部分过程式程序，以提高程序的效率**

- 对象式程序的局部及类定义中的数据操作一般都是过程式程序。

- 函数式程序设计秉承了过程式程序设计的模块化与代码重用思想，二者有类似的函数调用、参数、返回值及变量的作用域等概念。
  - `(* 5 2)`
  - `(if (> 10 20) (print "Hello") (print "World"))`

- 基于冯·诺依曼体系结构的计算机运行逻辑式程序效率不高，有经验的程序员往往会在能保证正确性的前提下，在逻辑式程序中嵌入部分过程式程序，以提高程序的效率。

```
gcd (A, 0, A).
gcd (A, B, D):-(A>B), (B>0), R is A mod B,  gcd(B, R, D).
gcd (A, B, D):-(A<B),  gcd (B, A, D).
```

程序设计语言

- **即编程语言，是一套用于编写计算机程序的符号与规则。**
  - 编程语言可以让程序员准确地在计算机世界为现实世界建立对应的模型，即定义现实世界实体的属性（数据结构）和不同情况下所实施的操作（算法）。
  - 根据与计算机指令和人类自然语言的接近程度，通常把编程语言分为：
    - 低级语言
    - 高级语言

    (通常所讲的程序设计语言往往指的是高级语言)

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- **低级语言（low-level programming language）**是一类与计算机指令直接对应，与人类自然语言相距甚远的编程语言

  - 机器语言（machine language）采用二进制形式的指令码和数据的存储位置来表示操作与操作数。硬件构成的裸机只能识别用机器语言表示的指令。（穿上C语言编译器这层外衣的虚拟机可以执行由C语言所表示的基本操作或语句）

  - 汇编语言（assembly language）采用助记符来表示操作和数据的存储位置，以帮助程序员记忆和运用，提高程序的可读性，汇编语言程序经汇编器（assembler，一种软件）翻译成机器语言程序即可被CPU执行。

- 对于表达式 `r = a + b * c - d`的求解，可以用机器语言描述为：

```
...
0000 0000 0000 0010 1100 1000 1011 0100 0101 1111 1100
…
```

- 可用汇编语言描述为（其中的ax为寄存器，即CPU中暂存数据的器件）：

```
……
mov ax, b
mul ax, c
add ax, a
sub ax, d
mov r, ax
```

- 低级语言与特定计算机指令系统密切相关，低级语言程序一般占用空间少，不需要进行大量的编译就可以被CPU执行，所以运行程序的效率比较高。

- 不过，由于不同型号的计算机指令系统不尽相同，低级语言程序可移植性不好，而且，对于程序员而言，低级语言程序难以编写和维护，不容易实现对数据和操作的抽象，因此，随着计算机硬件速度的提高、存储空间的增大，加上程序规模大和复杂度高的影响，低级语言逐渐被现代编程所淘汰。

- **高级语言（high-level programming language）是一类以人类自然语言为基础的编程语言。它通常采用人们熟悉的数学符号和精炼的英语单词来描述操作与操作数。**
  - `r = a + b * c - d`
- 高级语言程序（即源程序）必须被**编译器（compiler，一种软件）**或**解释器（interpreter，一种软件）**翻译成机器语言程序（即目标程序）才能被CPU执行。

高级语言的翻译

- 编译执行方式下（例如，FORTRAN、COBOL、Pascal、C、Simula、Ada、C++等语言），编译器将适于某种执行环境的源程序翻译成等价的目标程序，以便形成该环境下的可执行程序，**执行时不再需要源程序**。
  - 或先翻译成汇编语言程序，再通过汇编器翻译成目标程序。
  - 编译执行效率较高

- 解释执行方式下（例如，BASIC、Smalltalk、Java、Lisp、Prolog、Python等语言）的源程序，一边由解释器根据当前执行环境逐条翻译，一边被逐条执行，**翻译过程中不产生完整的目标程序**。
  - 解释执行可移植性较好
  - 有的高级语言，例如Java，其源程序可以被一次性翻译成平台无关的接近机器语言的中间码，执行时，只需要根据当前执行环境逐条翻译中间码并执行即可，这样既可以保证可移植性，又可以提高执行效率。

- 高级语言避免了低级语言的缺陷，程序员不必过分关注计算机的硬件，而是可以基于一种虚拟机来进行编程，从而降低了编程的复杂度。虽然高级语言程序一般比低级语言程序占用的空间大，翻译为目标程序的工作量大，运行的效率低，但是随着计算机硬件技术的发展，这些已经不是程序员需要担心的主要问题。

- 现代编程一般都使用高级语言，例如用于科学计算的FORTRAN语言，用于商务处理的COBOL语言，用于教学的Pascal语言，用于嵌入式或实时系统的Ada语言，用于系统软件开发的C/C++语言，用于网络应用的Java语言等等

C语言的特点

- 是一种编译执行的语言
- 是一种“中级”语言：
  - 高级语言：提供了丰富的类型和结构化流程控制机制。
  - 低级语言：提供了内存地址操作和位操作机制，部分硬件访问能力。
- 通用
- 程序运行效率比较高
- 对程序员的素质要求较高，初学者容易感到困惑
- 存在安全隐患，对可能导致错误的用法不加限制
- 对某些应用的支持不够好，如基于Internet的应用

- 通用
  - 支持基本的程序设计思想、概念和技术，支持多种程序设计范型，适用范围广泛
- 程序运行效率比较高
  - 是一种静态类型语言，即要求在静态的程序（运行前的程序）中指定每个数据的类型，这样在编译时就能知道程序中每个数据的类型，可以由编译器检查一些与类型相关的程序错误，并且为数据测算他们所需要的存储空间，从而可以产生高效的目标程序代码
  - 是一种弱类型语言，即对类型检查不完全，程序一些类型方面的错误只有在得出错误结果或异常终止时才会被发现，很少做运行时的其他合法性检查，例如数组下标越界等，从而减少了运行程序时的开销

- C++也具有C语言以上特点
- C++与C的常见不同之处
  - 对面向对象的支持
    - C：**很难支持**（结构、联合）
    - C++：**比较好地支持**（对象与类、继承、模板与标准模板库（STL））
  - I/O方式
  - 创建/删除动态变量
  - 引用类型
  - 类型机制
  - 名空间
  - 函数名重载
  - 头文件
  - 异常处理

语言的设计、实现以及使用

- **程序设计语言经过设计和实现，才能被程序员使用。**
- **语言的设计是指语言的定义，需考虑语言的语法、语义和语用**
  - **语法**（syntax）：一系列语言成分之间的组合规则，规定了程序的结构和形式；
  - **语义**（semantics）：各个语言成分的含义，代表程序的内容和功能；
    （My deskmate is Susan./ `if(x == 0) printf(" zero ");`）
  - **语用**（pragmatics）：语言成分的使用场合及所产生的实际效果。
    （下星期六-next Saturday/ `int i=1; (++i)+(++i)+(++i);` ）
  - 在设计阶段不易预料这样的问题，程序员在使用语言的时候应尽量避免出现这种歧义
- **语言的实现是指在某种计算机平台上开发出语言的翻译软件，针对某种语言可以有多种实现。**
- **语言的使用是指用语言来编写（设计）解决各种问题的程序。**

- **算法和数据结构的实现（coding）**
  - **选择程序设计语言**：根据应用系统的需求特点、程序设计范型、软硬件平台等因素综合考虑。实际上，现有各种语言大多基于冯·诺依曼体系结构，在表达能力上几乎等价，所以，一些非技术因素，例如人员、时间和资金等，往往会起决定作用。
  - **选择适当的语言元素**来表达算法与数据结构。

对于同一个设计，不同的人会写出不同风格的程序。风格的优劣会影响程序的质量。高质量的程序缺陷少、效率高、易维护、能容错。程序设计风格取决于编程人员对程序设计基本思想方法、语言和开发环境掌握的程度，以及工作作风和习惯的优劣。初学者可以通过刻意学习和训练来培养良好的程序设计风格。

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

程序的执行

- 对于实现算法和数据结构的程序，最终要在计算机系统中执行才能产生有效的结果。
- 一般情况下，还需要操作系统、编译系统等软件的支撑。

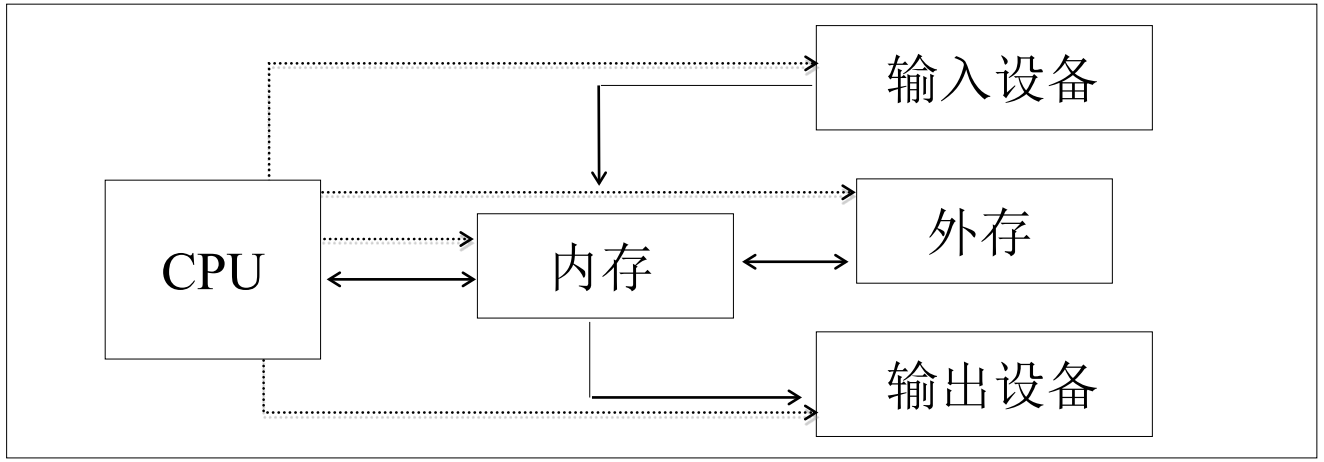
广义的程序设计：

- 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
  - **逻辑学、数学**
  - **算法**
  - **数据结构**
  - **程序设计范型**
  - **程序设计语言**
  - **程序开发环境（计算机系统、操作系统、编译系统）**
  - **软件工程**
  - **……**

计算机的工作模型

《编码的奥秘》、极简版《计算机组成原理》

- 早期的计算机执行程序时，数据位于存储单元中，程序以一种外插型的电路手工接入系统。
- 上世纪四十年代，匈牙利人John von Neumann提出更为实用的冯·诺依曼体系结构，一直沿用至今。
  - 采用该体系结构的计算机含存储器、运算器、控制器、输入及输出设备五个单元
  - 程序执行时，数据和程序均位于存储器中



虚线箭头表示控制流

实线箭头表示数据流

- 冯•诺依曼计算机的工作过程
  - 待执行的程序装入内存后，⇓
  - 带有运算器和控制器的 CPU 从内存逐条地取程序中的指令加以执行，⇓
  - 并从内存（例如变量、字符数组）或输入设备（例如键盘、磁盘文件）中获得所需要的数据，运算器进行运算，控制器发出控制信号控制整个系统，⇓
  - 程序执行产生的临时结果保存在内存（例如变量、字符数组）中，⇓
  - 程序执行的最终结果通过输出设备（例如显示器、磁盘文件）输出。

- 采用冯·诺依曼体系结构的计算机，通过不断改变程序的状态实现计算任务，
- 不同时刻存储器中的数据（一般为变量的值）反映了程序在不同时刻的状态，
- 变量值的改变通常是通过赋值操作来实现的，
- 所以，赋值操作构成了采用冯·诺依曼体系结构的计算机的一个重要特征。

- 由于计算机各个部件存在访问速度上的差别，快速部件往往要花费大量的时间等待慢速部件的访问，因此，涉及CPU、内存以及输入/输出设备之间数据传输的赋值操作形成一个性能瓶颈。

- 不过，程序的执行及其对数据的访问通常具有局部性特征，所以现代计算机往往利用这一特征，在CPU中为内存提供内存高速缓存（cache），暂存一批局部数据，以减少CPU访问内存的次数，在内存中为外存提供磁盘高速缓存（disk cache），以减少CPU访问外存的次数，从而可以从一定程度上解决部件之间速度不匹配问题。

- **程序的测试（test）与调试（debug）**
  - 编写的程序中可能含有语法、逻辑（或语义）或执行异常错误
    - **语法错误**可以由编译器检查发现
    - **逻辑错误**是指程序不符合所设计的算法或数据结构，或者算法或数据结构本身就不符合问题的求解
    - **执行异常错误**是指程序对执行环境的缺陷或用户操作的失误考虑不足而引起的程序异常终止(内存空间不足、打开不存在的文件进行读操作、数组下标越界、程序执行了除以0的指令等等)。

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- **程序的测试（test）与调试（debug）**
  - 部分逻辑错误和执行异常错误可以通过少量模拟数据进行测试（test）来发现，并通过调试（debug）来对错误进行定位和排除
    - 测试工作往往可以先分单元分模块测试，后进行集成化的整体测试
    - 调试工作也可以分段进行
    - 通过测试能够执行的程序并不一定是没有错误的程序
    - 何时结束调试阶段一般由程序员的经验和实际需求等因素决定

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

- **输入/输出（简称I/O）是程序的一个重要组成部分**
  - 程序运行所需要的数据往往要从内存或外设得到；程序的运行结果通常也要存入内存或外设中去 。
- **分类：**
  - 基于控制台的I/O
    - 从标准输入设备（eg. 键盘）获得数据；把程序结果从标准输出设备（eg. 显示器）输出
  - 基于字符数组的I/O
    - 从程序中的字符数组中获得数据；把程序结果保存到字符数组中
  - 基于文件的I/O
    - 从外存文件获得数据；把程序结果保存到外存文件中
- **在C/C++中，I/O不是语言定义的成分，而是作为标准库的功能由具体的实现（编译器）来提供，有两种途径：**
  - 过程式——I/O库函数（C/C++）
  - 对象式——I/O类库(C++)

I/O流

- **在C/C++中，I/O操作是一种基于字节流的操作：**
  - 在进行输入操作时，可把输入的数据看成逐个字节地流入到计算机内部（内存）；
  - 在进行输出操作时，则把输出的数据看成逐个字节地从内存流出。

- **在C/C++的标准库中，除了提供基于字节的I/O操作外，为了方便使用，还提供了基于C/C++基本数据类型数据的I/O操作。**

- **在C++程序中也可以对类库中I/O类的一些操作进行重载，使其能对自定义类的对象进行I/O操作。**

- **程序的运行（run）与维护 （maintenance）**
  - **所有的测试手段只能发现程序中有无错误，不能证明程序是否完全正确。**
  - 在运行使用过程中发现错误并改错称为维护
    - 正确性维护
    - 完善性维护：指根据用户的要求使得程序功能更加完善
    - 适应性维护：把程序移植到不同的计算平台或环境中，使之能够运行
  - 程序维护所花费的人力和物力往往是很大的，因此，在设计与实现阶段要设法提高程序的易维护性

- **广义的程序设计：**
  - 包括问题分析、算法和数据结构的设计、算法和数据结构的实现、测试和运行 等一系列过程，其本质涉及多方面内容
    - **逻辑学、数学**
    - **算法**
    - **数据结构**
    - **程序设计范型**
    - **程序设计语言**
    - **程序开发环境（计算机系统、操作系统、编译系统）**
    - **软件工程**
    - **……**

程序设计过程

- 问题分析
- 算法和数据结构的设计
- 算法和数据结构的实现
- 程序的执行、测试和调试
- 程序的运行和维护

- 1971年，Niklaus Emil Wirth首次提出“结构化程序设计”（structured programming）理念
  - **问题分析阶段：“自顶向下”、“逐步求精”** ??
  - **设计阶段：“模块化”** ?
  - **实现阶段：“结构化”** √

例3* N 阶线性方程组的 Jaccobi 迭代法求解简化程序。

- [分析]
  - 有没有现成的软件包或库?
  - 对于 N 阶线性方程组 Ax = b，A 是方程组的**系数矩阵**，x 是**未知数的列向量矩阵**，b 是**常数项的列向量矩阵**。线性方程组可以写成：Dx = b - Rx，其中 D 为 A 中的正对角部分，R 为 A 中的剩余部分。求解未知数的 Jaccobi **迭代**公式为 $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$，计算每个元素的迭代公式为：

$$x_i^{(k+1)} = a_{ii}^{-1}\left(b_i - \sum_{j\neq i} a_{ij}x_j^{(k)}\right)\ ,\ i, j = 1,\ 2,\ \ldots,\ N$$

- [设计]
  - 向量
  - 迭代法
  - 可以采用过程式程序设计范型
- [实现]
  - 选用 C 语言实现该公式的计算
  - 选择开发环境
  - 可以用一维数组 a[] 表示系数矩阵 A， 用一维数组 x[] 表示未知数的列向量矩阵，常数项的列向量可以用一维数组 b[] 表示，用 double 型变量 dErr 表示精度。
  - 用独立的函数 Solve 实现 Jaccobi 迭代算法，用嵌套的 for 循环计算公式中的求和项，迭代过程对应外层大循环，每大循环一次，用新的迭代结果替换旧值，当迭代终止条件满足时，用 break 语句折断循环流程。
- 测试、调试（设计样例）
- 运行、维护（实际应用）

程序与程序设计的本质

- 程序的本质
  - 算法
  - 数据结构
- 程序设计的本质
  - **狭义**
  - 广义
- **程序设计范型**
  - **过程式（面向功能）**
  - 对象式（面向数据）
  - 函数式
  - 逻辑式
- **程序设计语言**
  - 低级
  - 高级
  - 语言的设计、实现及使用
- 程序设计过程
  - 问题分析；算法和数据结构的设计；编码实现；测试与调试；运行与维护

- **思考：对于一个具体的问题，我们怎么设计程序来解决它?**
  - 比如，做一个电子日历。
- **你现在的思路跟 学期初看到这个问题时的 思路 有什么不同?**

祝大家期末考试取得好成绩！

Thanks!