東南大學 SCHOOL OF INTEGRATED CIRCUITS, SEU

集成电路学院

计算机科学基础I

——函数III、数组定义

东南大学 集成电路学院 朱彬武

E-mail: bwzhu@seu.edu.cn

斐波那契数列循环版

就是斐波那契数列由0和1开始，之后的斐波那契数就是由之前的两数相加而得出。首几个斐波那契数是：1、1、2、3、5、8、……

```
int fibonacci_iter(int n) {

    int prev = 0;
    int curr = 1;
    int nextValue = 1;

    for (int i = 3; i <= n; i++) {
        nextValue = prev + curr;
        prev = curr;
        curr = nextValue;
    }

    return nextValue;
}
```

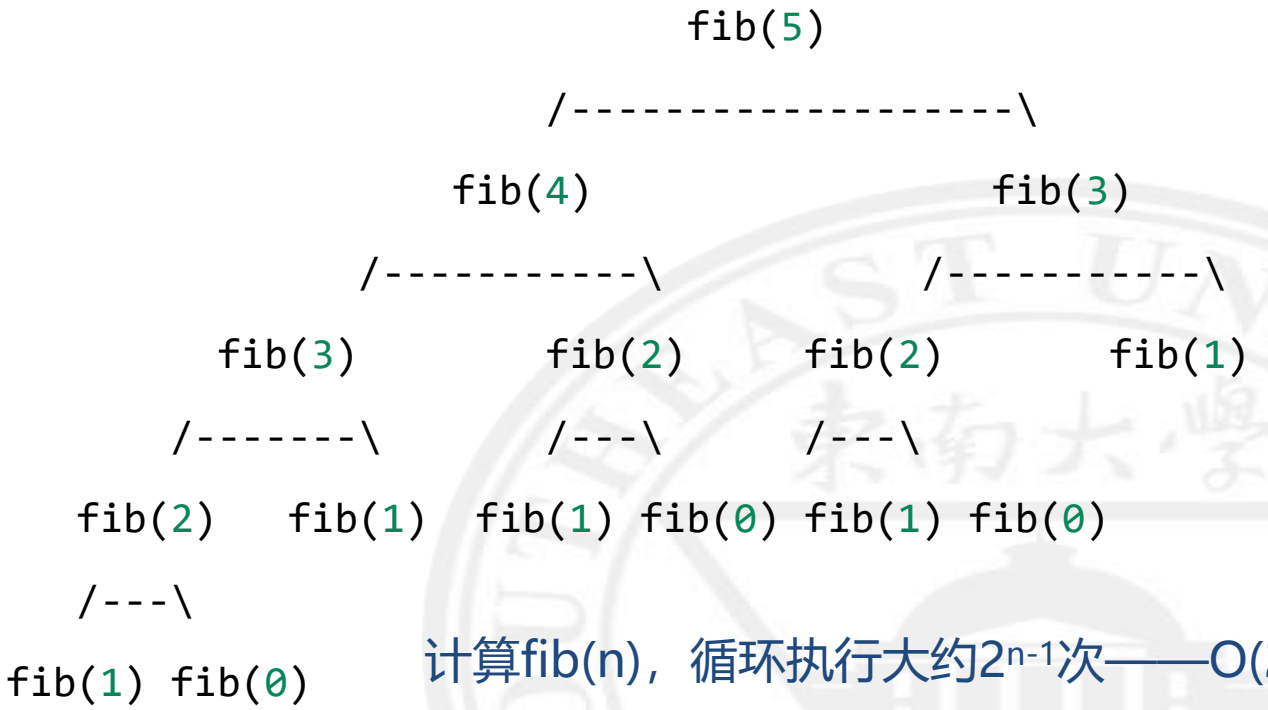
循环轮次	prev	curr	nextValue
1	0	1	1
2	0	1	1
3	1	1	2
4	1	2	3
5	2	3	5

计算fib(n)，循环执行大约n次——O(n)

斐波那契数列的递归实现

递推公式： $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}(n \geq 2)$

```c
int fibonacci_rec(int n) {
    // 终止条件
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fibonacci_rec(n - 1)
             + fibonacci_rec(n - 2);
}
```



计算fib(n)，循环执行大约2^{n-1}次——$O(2^n)$

时间对比

n	循环版本的时间(ms)	递归版本的时间(ms)
10	1.49e-5	2.2e-4
20	3.4e-5	0.029
30	4.04e-5	3.67
40	6.09e-5	441

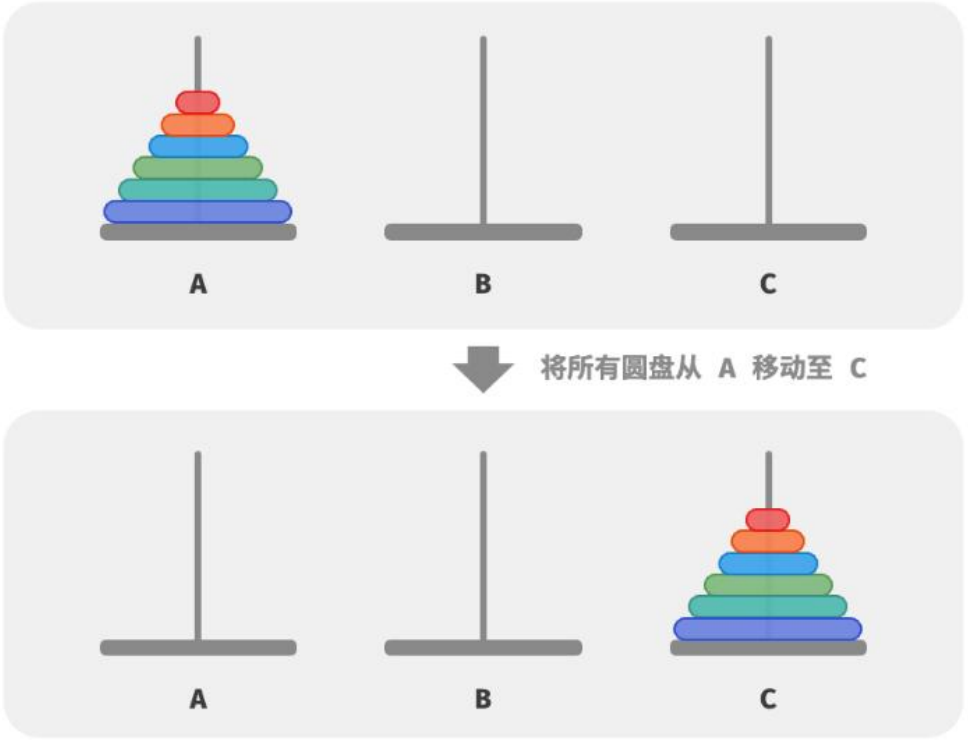
```cpp
int main() {
    // 记录开始时间
    auto start = std::chrono::high_resolution_clock::now();

    // ======= 需要计时的代码 =======

    // ==============================

    // 记录结束时间
    auto end = std::chrono::high_resolution_clock::now();

    // 计算耗时（单位：毫秒）
    std::chrono::duration<double, std::milli> elapsed = end - start;

    std::cout << "耗时: " << elapsed.count() << " ms\n";
    return 0;
}
```

C++11

```cpp
int main() {
    clock_t start = clock();

    // ======= 需要计时的代码 =======

    // ==============================

    clock_t end = clock();

    double elapsed = double(end - start) / CLOCKS_PER_SEC; // 秒

    std::cout << "耗时: " << elapsed << " 秒\n";
}
```

C++03或更早

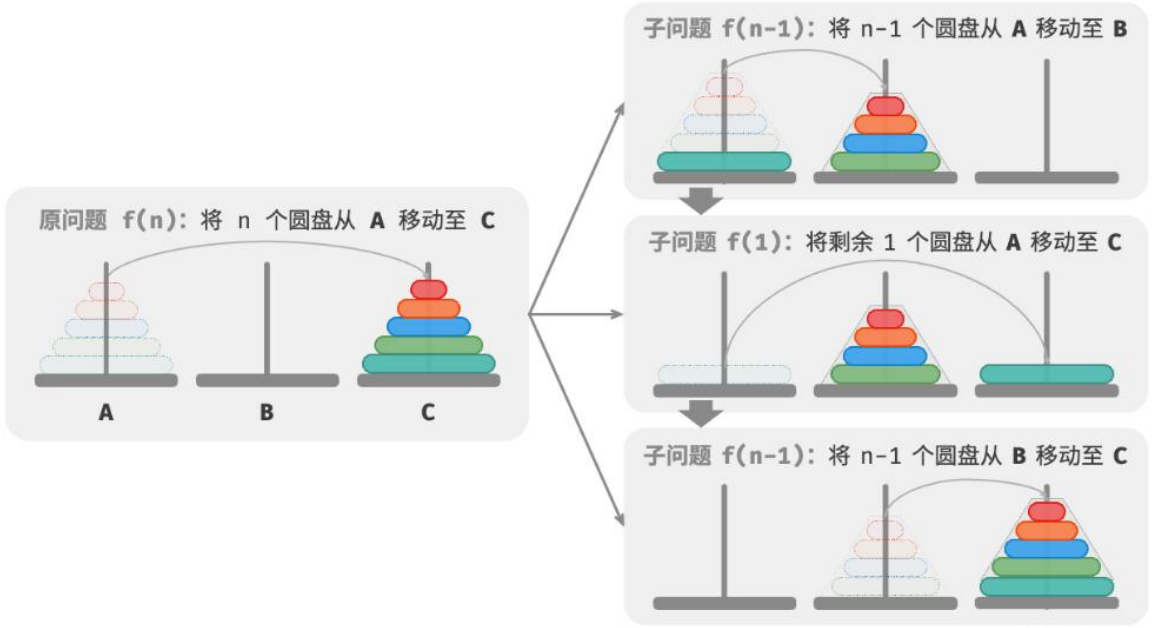
递归经典问题——汉诺塔



给定三根柱子，记为 A、B 和 C 。起始状态下，柱子 A 上套着 n 个圆盘，它们从上到下按照从小到大的顺序排列。我们的任务是要把这 n 个圆盘移到柱子 C 上，并保持它们的原有顺序不变。在移动圆盘的过程中，需要遵守以下规则。

- 圆盘只能从一根柱子顶部拿出，从另一根柱子顶部放入。
- 每次只能移动一个圆盘。
- 小圆盘必须时刻位于大圆盘之上。

递归经典问题——汉诺塔




```
void hanoi(int n, char from, char aux, char to) {
    // 终止条件：只有 1 个盘子时直接移动
    if (n == 1) {
        cout << "Move disk 1 from " << from << " to " << to << endl;
        return;
    }

    // 1. 把上面的 n-1 个盘子从 from 移动到 aux，借助 to
    hanoi(n - 1, from, to, aux);

    // 2. 把最大的盘子从 from 移动到 to
    cout << "Move disk " << n << " from " << from << " to " << to << endl;

    // 3. 最后把 n-1 个盘子从 aux 移动到 to，借助 from
    hanoi(n - 1, aux, from, to);
}
```

移动n个盘子所需的次数$T(n)$:

$$T(n) = 2T(n-1) + 1$$
$$T(1) = 1$$

汉诺塔问题如果用循环来做则会非常复杂，
我们目前学的知识仍然不够

递归和迭代对比

- 递归：函数自己调用自己来解决问题，每次调用都把问题规模缩小一点，直到达到终止条件。代码简洁，易理解，可能性能差

- 迭代：使用循环结构不断重复某些操作，通过变量更新一步一步接近答案。循环实现性能高，但逻辑可能更复杂

默认参数、内联函数和函数重载

默认参数

- 在C++中，定义函数的时候可以让最右边的连续若干个参数有默认值（default），那么调用函数的时候，若相应位置不写参数，参数就是默认值

```
void func(int x, int y = 2, int z = 3){
    cout << x << y << z << endl;
}

int main(){
    func(10); //等效于func(10, 2, 3)
    func(10, 8); //等效于func(10, 8, 3)
    func(10, ,8); //error，只能最右边的连续若干个参数缺省
}
```

```
// 声明：提供缺省参数
void func(int a, int b = 10);

// 定义：不要再写默认值
void func(int a, int b) {
    // ...
}
```

- 若提供了函数声明，则默认参数只能在声明中提供，不能在定义中重复出现

默认参数作用

- 函数参数可以默认的主要目的之一在于提高程序的可扩充性
- 考虑要为某个写好的函数添加新的参数，而原先调用该函数的语句，未必需要使用新增的参数，那么为了避免对原先那些函数调用语句的修改，就可以使用默认参数

```
double calcPrice(int count, double unitPrice) {
    return count * unitPrice;
}

int main() {
    double p1 = calcPrice(3, 9.9);  // 老代码
}
```

```
double calcPrice(int count, double unitPrice, double rate = 1.0) {
    double price = count * unitPrice * rate;

    return price;
}

int main() {
    double p1 = calcPrice(3, 9.9);          // 老代码：不打折
    double p2 = calcPrice(5, 8.0, 0.9);     // 新代码：9折
}
```

内联函数

- 函数调用是有时间开销的，如果函数本身只有几条语句，执行非常快，而且函数被反复执行很多次，相比之下调用该函数所产生的这个开销就会显得比较大

```
int myMin(int a, int b) {
    return (a < b) ? a : b;
}
```

- 为了减少函数调用的开销，引入了内联函数机制。在函数定义前面加“inline”关键字，即可以定义内联函数

```
inline int myMin(int a, int b) {
    return (a < b) ? a : b;
}
```

内联函数机制

- 编译器处理对内联函数的调用语句时，是将整个函数的代码插入到调用语句处，而不会产生调用函数的语句。

```
int main(){
    int x = 2, y = 3;
    int minVal;
    minVal = myMin(x, y);

    //编译时可能会变成
    minVal = (x < y) ? x : y;
}
```

- inline会造成可执行程序体积的增大，通常是给编译器的一个建议，编译器如果觉得不合适也会拒绝inline。

类型不匹配

- 调用函数时给的值与参数的类型不匹配时，会根据情况发生不同的行为，主要有以下三种情况
  - 类型转换
  - 无法转换
  - 函数重载决议失败
- 但是很多情况下，即使类型转换成功了，也可能并不是我们所期望的

```
int pow(int base, int exponent);

pow(1.3, 2);
```

函数重载

- 函数的名字相同，然而参数个数或参数类型不相同，这叫做函数的重载（overload）。

```
int myMax(double f1, double f2){
    //(1)...
}
int myMax(int n1, int n2){
    //(2)...
}
int myMax(int n1, int n2, int n3){
    //(3)...
}
```

- 函数重载使得函数的命名变得简单。

函数重载机制

- 编译器会根据调用语句中的实参的个数和类型判断应该调用哪个函数

```
int myMax(double f1, double f2){
    //(1)...
}
int myMax(int n1, int n2){
    //(2)...
}
int myMax(int n1, int n2, int n3){
    //(3)...
}
```

```
myMax(3.4, 2.5);  //调用(1)
myMax(3, 2);      //调用(2)
myMax(1, 2, 3);   //调用(3)
myMax(3, 2.4);    //error，二义性
```

- 需注意：如果参数个数和参数类型均相同，只有返回类型不同，则不符合重载要求，而是重复定义，编译会发生错误。

pow函数的二义性

- 使用Visual C++ 2010调用的pow函数：使用的标准是C++03

math.h

(Global Scope)

```
497 inline double __CRTDECL pow(_In_ double _X, _In_ int _Y)
498         {return (_Pow_int(_X, _Y)); }

535 inline float __CRTDECL pow(_In_ float _X, _In_ int _Y)
536         {return (_Pow_int(_X, _Y)); }
```

pow(2, 3) //error, 二义性

文件路径：d:\visualC\VC\include\math.h

- Dev C++中调用的pow函数：使用的标准是C++11

```
#if __cplusplus < 201103L
  // _GLIBCXX_RESOLVE_LIB_DEFECTS
  // DR 550. What should the return type of pow(float,int) be?
  inline double
  pow(double __x, int __i)
  { return __builtin_powi(__x, __i); }

  inline float
  pow(float __x, int __n)
  { return __builtin_powif(__x, __n); }

  inline long double
  pow(long double __x, int __n)
  { return __builtin_powil(__x, __n); }
#endif
#endif

  template<typename _Tp, typename _Up>
    inline _GLIBCXX_CONSTEXPR
    typename __gnu_cxx::__promote_2<_Tp, _Up>::__type
    pow(_Tp __x, _Up __y)
    {
      typedef typename __gnu_cxx::__promote_2<_Tp, _Up>::__type __type;
      return pow(__type(__x), __type(__y));
    }
```

数组

逆序输出整数（Lab4）

题目描述　参考答案　反馈

7-3 逆序输出整数　分数 15

查看题目详情　全屏浏览　切换布局　　作者 温彦　单位 山东科技大学

编写程序将整数逆序输出。如输入为9876输出为6789
Main函数中读入n个整数，输出n个整数的逆序数

输入格式:

整数个数n
n个整数

输出格式:

n个整数的逆序数

输入样例:

在这里给出一组输入。例如:

```
3
1234
2323
1112
```

输出样例:

在这里给出相应的输出。例如:

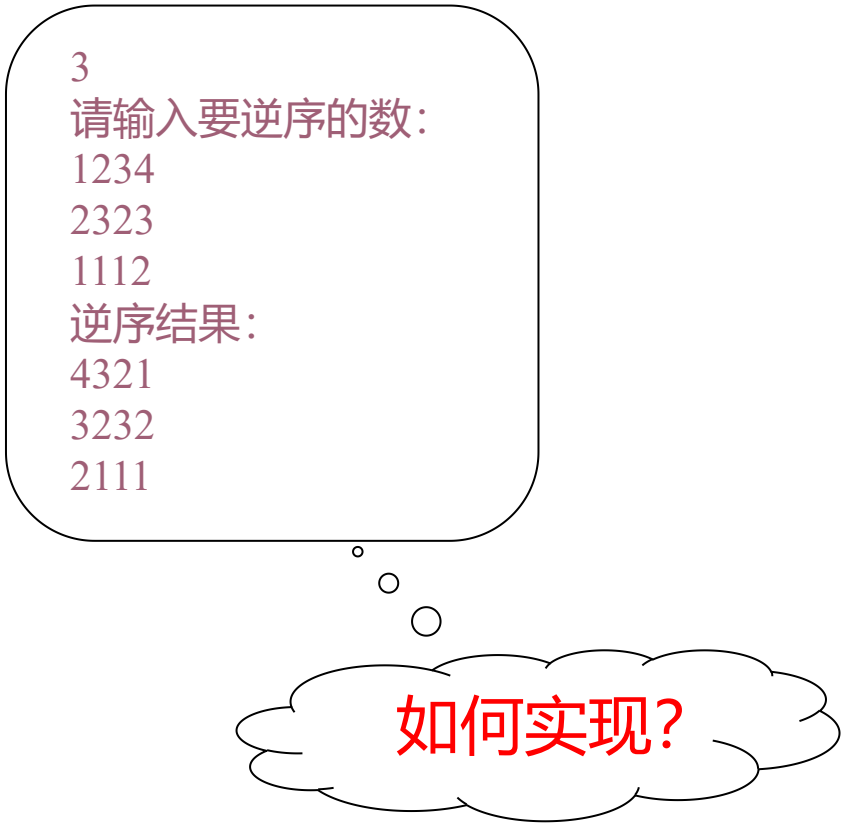
```
4321
3232
2111
```

```cpp
int main() {
    int n;
    cin >> n;

    int x;
    for (int i = 0; i < n; i++) {
        cout << "请输入要逆序的数：";
        cin >> x;
        cout << "逆序结果：" << reverseNumber(x);
        if (i != n - 1)
            cout << "\n";
    }

    return 0;
}
```

```
3
请输入要逆序的数：1234
逆序结果：4321
请输入要逆序的数：2323
逆序结果：3232
请输入要逆序的数： 1112
逆序结果：2111
```

每次输入一个
然后输出一个

如何一次性读入多个数?




```
int num1, num2, ..., numN;
cin >> num1;
cin >> num2;
...
cin >> numN;
for(int i = 0; i <= n; i++){
    //一些操作
}
```

但是无法事先知道需要多少个变量

灵活批量存储数据的场景

- 成绩登记系统：记录每个同学的成绩再排序
- 购物车系统：要支持删改购物车中的物品
- 异常值检测：找出一批数据中的异常值
- 多媒体：保存一张图片、一段声音
- ……

数组引入


```
int main() {
    int n;
    cin >> n;

    int x;
    for (int i = 0; i < n; i++) {
        cout << "请输入要逆序的数: ";
        cin >> x;
        cout << "逆序结果: " << reverseNumber(x);
        if (i != n - 1)
            cout << "\n";
    }

    return 0;
}
```

```
int main() {
    int n;
    cin >> n;

    int arr[100];
    cout << "请输入要逆序的数: \n";

    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    cout << "逆序结果: \n";
    for (int i = 0; i < n; i++) {
        cout << reverseNumber(arr[i]);
        if (i != n - 1)
            cout << "\n";
    }

    return 0;
}
```

数组定义

数组赋值

数组遍历

数组访问

数组（array）是一种容器（container）（存储、管理数据集合），其中所有的元素具有相同的数据类型

数组的定义


```
int array[100];
```

- <类型> 数组名称[元素数量]

```
int grades[100];
double weight[200];
char key[30];
```

- 元素数量必须是整数

```
int arr[100.0]; // error
```

- 在标准C++中，元素数量必须是常量表达式
（如果使用g++编译器是允许的，但这不属于C++标准）

```
int n = 5;
int arr[n];  // error

const int n = 5;
int arr[n];  // OK
```

数组元素

- 数组的每个元素就是数组类型的一个变量，可以出现在赋值号的左边或右边
- 使用数组时放在[]中的数字叫做下标或索引，下标从0开始计数：

```
grades[0]

grades[99]

average[5]
```

有效的下标范围

- 编译器和运行环境都不会检查数组下标是否越界，无论是对数组单元做读还是写
- 一旦程序运行，越界的数组访问可能造成问题，导致程序崩溃
- 写数组的时候应该保证只使用有效的下标值：0～数组的大小-1
- `int a[0]; //可以存在，但是没有用`