06 函数

定义处理、调用功能、联系数据

本讲学习目标

- 说明怎样用函数安排不同处理，在需要时调用，并再次使用同一种处理。
- 辨认函数的定义、声明与调用，追踪一次调用中的数据和执行。
- 编写职责明确的简单函数，使用结果，并用正常值与边界值检查程序。
- 解释调用双方的变量变化，定位和修正常见函数错误。
- 初步认识递归的结束条件与调用关系，不要求独立设计递归算法。

函数：从具体操作到功能组合

程序 = 数据对象 + 操作 + 控制 + 组合

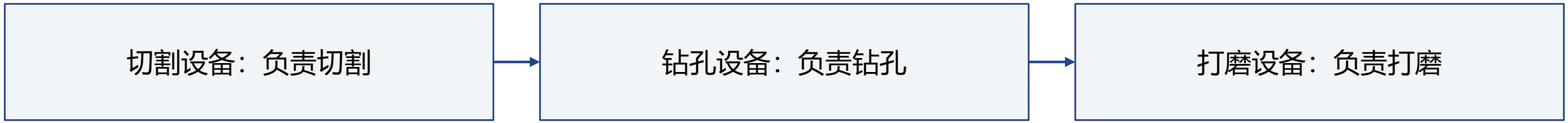
- 函数将相关操作组织为职责明确的功能单元，使程序按功能组织处理过程。
- 函数通过**定义与调用**实现功能封装和复用，不同功能可以组合完成较复杂的任务。

1 函数的用途

认识函数的功能，区分定义与调用

专用功能与任务分工

加工完成一件工件，需要按顺序完成多项处理。不同处理可以交给不同的专用设备。



每台设备承担明确的加工任务；生产安排把这些任务组织为完整过程。

程序也可以把不同处理组织为职责明确的**函数**：由定义规定处理、可以被调用的程序组成部分。

同一台切割设备可以按不同长度要求加工，后面再讨论本次任务的数据。

功能的定义与调用

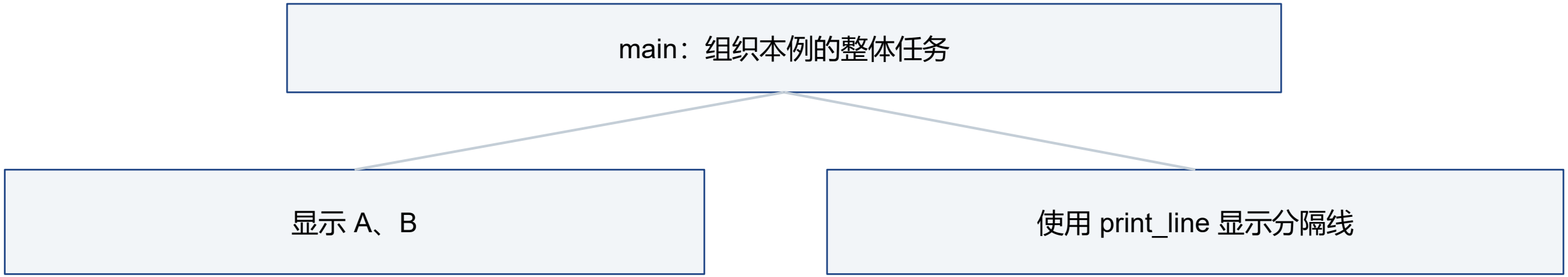
工厂设备	程序函数
预先规定设备的工作程序	函数定义规定处理
按任务要求启动设备	调用函数
启动后执行一次加工	调用后执行一次处理
再次启动同一设备	再次调用同一函数

定义规定处理内容，调用才使处理执行；调用正常完成后，调用方继续后续任务。

例如，print_line 负责显示一行分隔线。需要时使用 print_line();，也可以再次调用它。

整体任务与功能组合

生产安排组织不同设备。本例由 main 安排整体任务，print_line 负责显示分隔线。



- main 是程序入口，本身也是函数；本例由它组织显示与调用。
- print_line 完成专门处理，同一函数可以在两个位置调用。

图中连线表示任务组织关系，不表示执行顺序；具体调用与返回由后面的代码观察。

接下来看实际代码：调用怎样使 print_line 执行，完成后从哪里继续?

2 函数的定义与调用

定义规定处理，调用后执行并返回

函数调用时，程序怎样执行?

```c
#include <stdio.h>

void print_line(void)
{
    puts("---");
}

int main(void)
{
    puts("A");
    print_line();
    puts("B");
    print_line();
    return 0;
}
```

已有处理

print_line 中的处理是显示一行 ---

print_line(); 要求执行这项处理

puts("A") 显示 A 并换行；显示分隔线的 puts 用法相同

分步预测

程序开始时，从哪个函数进入执行?

main 第一次遇到 print_line(); 时，需要执行哪段处理?

这段处理完成后，你预测 main 接着执行哪一句？说明依据

再次遇到 print_line(); 时会发生什么？据此预测输出顺序

函数调用与返回的过程

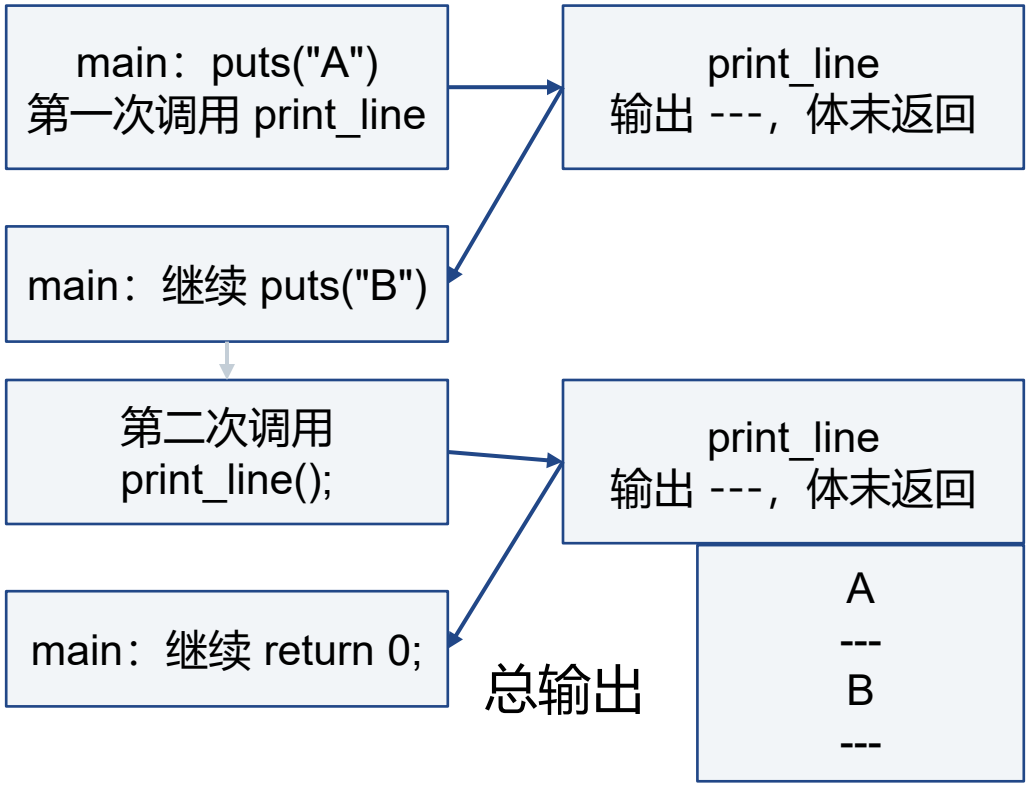
```c
#include <stdio.h>

void print_line(void)
{
    puts("---");
}

int main(void)
{
    puts("A");
    print_line();
    puts("B");
    print_line();
    return 0;
}
```

函数调用使程序执行指定函数。**调用方**发起调用，**被调用方**执行处理。

main 等待 print_line 完成后才继续。



函数定义的组成

```
1 void print_line(void)
2 {
3     puts("---");
4 }
```

定义的组成

函数定义规定名称、返回类型、参数列表及函数体

print_line 是名称；前面的 void 表示无返回值

括号内的 void 表示无参数

函数体

函数体是定义中用花括号括起的复合语句，规定函数执行的处理；本例显示一行分隔线

定义规定处理内容；处理在调用时执行

调用前的函数声明

```c
void print_line(void);   /* 原型 */

void print_line(void)    /* 定义 */
{
    puts("---");
}
```

原型与定义

函数声明提供函数的名称和类型信息；**函数原型**明确返回类型和参数类型

本例原型以分号结束；定义带函数体，也具有声明作用

调用前的声明

若把定义移到 main 后，需要在调用前写原型

定义已在调用前：可以直接使用该定义提供的声明

声明、定义和调用的类型信息须一致

原型提供类型信息，不要求执行函数体

函数接口：类型信息与任务约定

```
void print_line(void);
```

- **函数接口**说明怎样使用函数，包括类型信息与任务约定。
  - 类型信息：不接收参数，不提供返回值。
  - 任务约定：每次调用显示一行分隔线；这一效果无法仅从类型得知。
- 有参数、有返回值的函数，还需说明接收什么数据、返回值表示什么。

3 参数、返回值与局部状态

传入数据、使用结果，区分局部变量

功能的输入与任务数据

前面的同一台切割设备，可以处理不同长度的任务：设备功能不变，本次数据可以不同。

观察层次	工厂任务	函数中的对应
事先规定	操作说明要求提供长度	函数定义规定需要接收什么数据
本次提供	任务单填写长度 30	调用时提供本次实际数据
执行处理	按本次长度完成切割	函数使用收到的数据执行处理

下面用代码区分函数定义中的参数与调用时提供的表达式。

函数可以提供返回值，也可以完成输出等处理而不提供返回值。

用函数计算整数的平方

```c
#include <stdio.h>

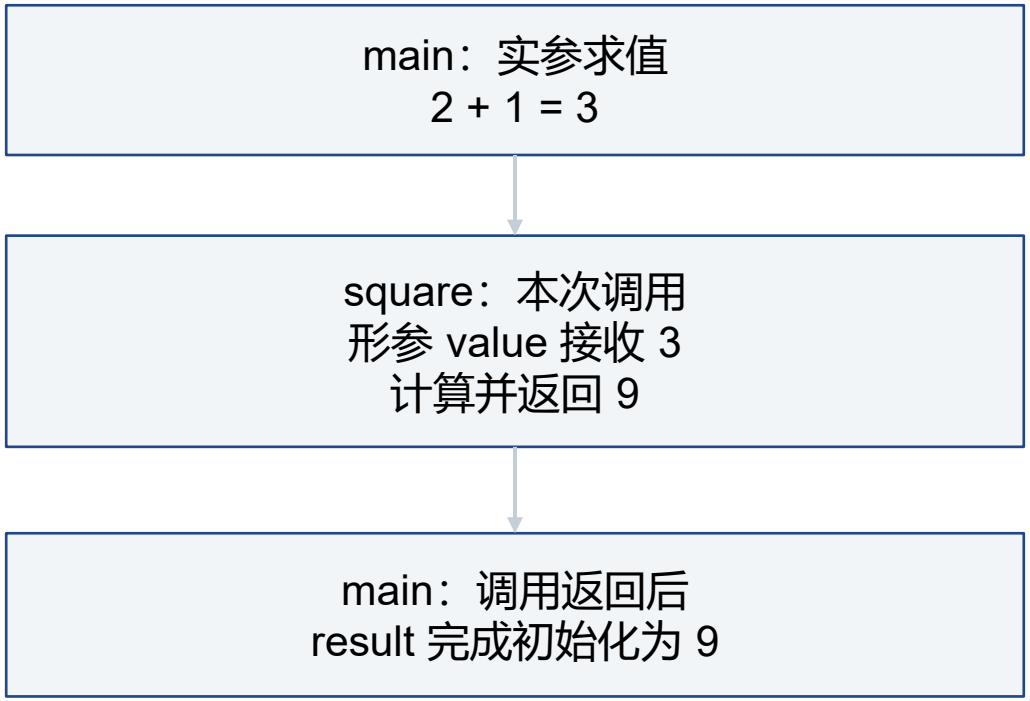
int square(int value);

int main(void)
{
    int result = square(2 + 1);
    printf("%d\n", result);
    return 0;
}

int square(int value)
{
    return value * value;
}
```

实参：调用时提供的表达式。

形参：本次调用接收值的参数对象。



比较函数返回值与表达式结果

```
1 int square(int value)
2 {
3     return value * value;
4 }
```

计算三个表达式

square(3)、square(2 + 1)、1 + square(3)

分别求出整个表达式的值，并说明其中的 square 调用传入什么值、返回什么值

返回后的运算

在 1 + square(3) 中，square(3) 返回后，还需要完成哪一步计算?

返回值参与表达式求值

核对刚才的三个表达式：square 调用都传入 3、返回 9，但整个表达式的值不全相同。

表达式	square 的传入值	square 的返回值	整个表达式的值
square(3)	3	9	9
square(2 + 1)	3	9	9
1 + square(3)	3	9	10

- **返回值**是函数正常返回时，按返回类型转换并交给调用方的值。
- 1 + square(3)：square 返回 9 后，还需计算 1 + 9，所以整个表达式的值是 10。
- **return 语句**结束当前函数调用；带表达式的形式还提供结果。
  - 调用方继续完成尚未完成的运算或处理。
  - 返回位置不一律是“下一条语句”。

实参与形参的匹配

```
1 int difference(int left, int right)
2 {
3     return left - right;
4 }
```

两种参数顺序

difference(8, 3)：left=8，right=3，结果为 5

difference(3, 8)：left=3，right=8，得到 -5

位置与类型要求

实参与形参按对应位置匹配，并遵守参数类型转换规则

调用要检查参数数量、顺序和类型；原型可以写成 int difference(int, int);

求值顺序的边界

参数位置不规定实参的求值顺序；本讲不使用相互影响的实参

没有返回值的函数

```c
int square(int value)
{
    return value * value;
}

void show_square(int value)
{
    printf("%d\n", value * value);
}
```

无返回值的函数

show_square 前的 void 表示函数没有返回值

show_square(3); 传入 3、显示 9，但不向调用方提供可用于赋值或计算的返回值

与有返回值的函数对照

int r = square(3);：r 得到 9，不自动显示

square(3);：返回的 9 被忽略，不自动显示

返回值可用于赋值、计算或判断，也可能被忽略

参数与处理职责

void 函数仍可接收参数、执行输出

接口应说明函数负责计算、判断还是显示

用 return 提前结束函数调用

```c
void show_positive(int value)
{
    if (value <= 0) {
        return;
    }
    printf("%d\n", value);
}
```

两条执行路径

return; 不提供值，适用于 void 函数

value<=0：本次调用提前结束，不执行 printf

value>0：显示该值，到达函数体末尾返回

返回后的处理

返回之后，调用方仍继续它尚未完成的处理

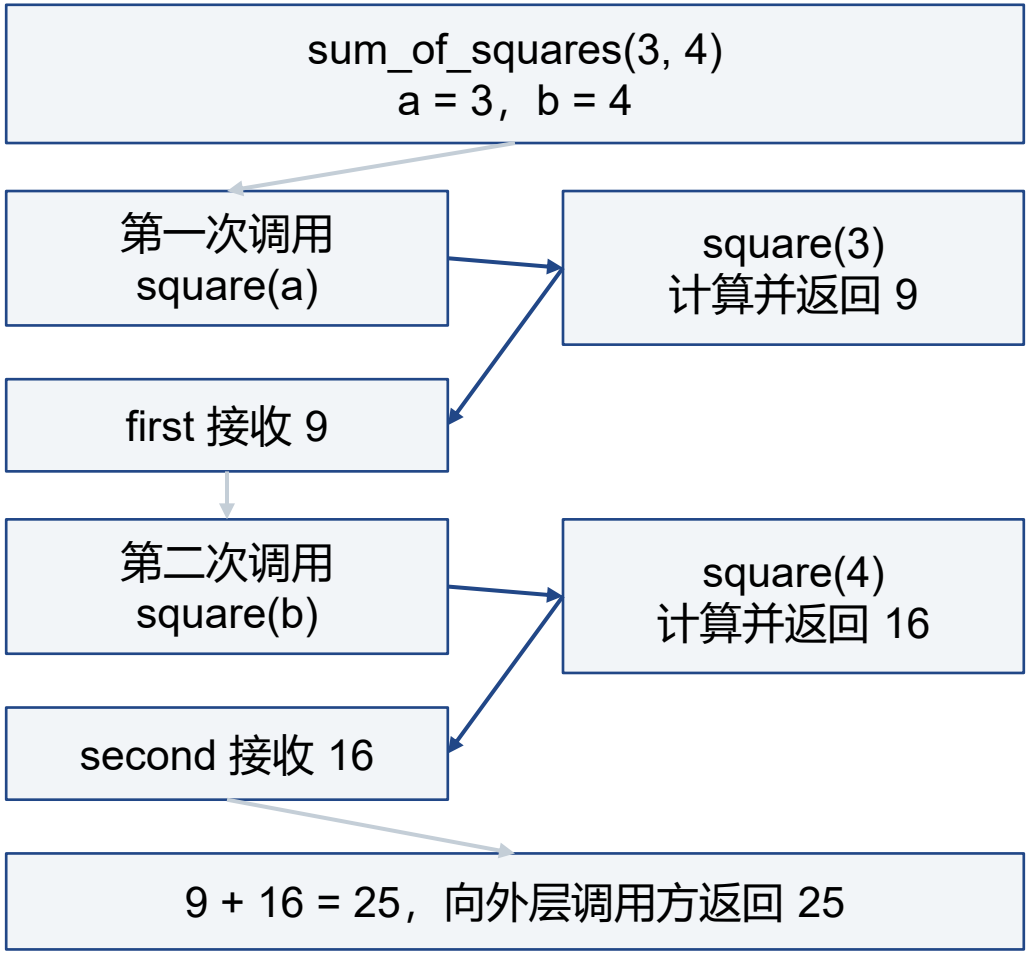
数值结果的要求

需要数值结果的函数，应在相关执行路径提供约定的返回值

一个函数可以调用另一个函数

```
1  int square(int value)
2  {
3      return value * value;
4  }
5
6  int sum_of_squares(int a, int b)
7  {
8      int first = square(a);
9      int second = square(b);
10     return first + second;
11 }
```

sum_of_squares 先后调用两次 square，再合并结果。辅助函数也能组织调用。



函数内的修改会改变 main 的变量吗?

```c
#include <stdio.h>
int increase(int n);
int main(void)
{
    int n = 5;
    int result = increase(n);
    printf("%d %d\n", n, result);
    n = increase(n);
    printf("%d\n", n);
    return 0;
}
int increase(int n)
{
    n = n + 1;
    return n;
}
```

观察两个 n

main 声明变量 n；increase 的形参也叫 n，函数内对它加一

分析两次输出

执行 int result = increase(n); 后，写出 main 中 n、result 的值及第一次 printf 的输出

再执行 n = increase(n); 后，写出 main 中 n 的值及第二次 printf 的输出

解释变化

increase 中的形参 n 与 main 的变量 n 是同一个对象吗?

哪一处赋值改变了 main 的 n?

数值形参与调用方变量是不同对象

- **传值**：实参表达式求得的值按对应参数类型转换后，用于建立本次形参的值。



位置	main 的 n	increase 的形参 n	main 的 result
第一次调用前	5	尚未建立	尚未完成初始化
传入后	5	5	尚未完成初始化
函数内加一后	5	6	尚未完成初始化
返回并完成初始化后	5	本次形参不再存在	6

- 第一行输出是 5 6。
- n = increase(n); 中，由 main 的赋值把返回的 6 保存到自己的 n，第二行输出 6。
- 相同名称或相同值，不表示同一个变量对象；返回本身不自动写回原变量。

局部变量的作用域与存续

```c
#include <stdio.h>
int increase(int n);
int main(void)
{
    int n = 5;
    int result = increase(n);
    printf("%d %d\n", n, result);
    n = increase(n);
    printf("%d\n", n);
    return 0;
}
int increase(int n)
{
    n = n + 1;
    return n;
}
```

局部变量

局部变量在相应局部作用域中声明；形参也属于本次调用的局部对象

名称的作用域

作用域是名称在源代码中能够指代相应实体的范围

- main 中的 n 指 main 的变量
- increase 函数体中的 n 指本次形参，不能靠相同名字访问 main 的 n

对象的存续

存续是对象在运行中存在的期间

- 本例形参与普通自动局部对象随相应执行建立，离开时结束存续
- 返回交回值，不把局部变量对象搬给调用方

名称可用范围与对象存在期间是两个问题

重复调用会保留上次的局部变量值吗?

```c
#include <stdio.h>

int local_count(void)
{
    int count = 0;
    count = count + 1;
    return count;
}

int main(void)
{
    int first = local_count();
    int second = local_count();
    printf("%d %d\n", first, second);
    return 0;
}
```

分析两次调用

写出两次 local_count 调用返回的值，以及 first、second 的值

说明依据

第二次调用时，count 从什么值开始？依据是哪条语句？

每次调用建立自己的普通局部状态

调用	初始化后的 count	加一后	返回值
第一次 local_count	0	1	1
第二次 local_count	0	1	1

- 输出：1 1。
- 两次调用执行同一份定义，但各自建立局部对象并执行初始化。
- 类似每次工作使用一张新的任务单：本例在每次调用中执行 int count = 0; 中的初始化，从 0 开始记录；依据是代码中的初始化，并非所有局部变量都会自动清零。
- 若删除 = 0，新建立的 count 也不会自动保留上次结果；此时变量未初始化，不能读取它来分析输出。
- 本讲只讨论普通自动局部状态；函数调用不自动保存可供下次继续使用的计数。

4 用函数实现与组合任务

编写判断函数，组合任务，检查边界

按质数规则完成判断

- **质数**是大于 1、正因数只有 1 和自身的整数。
  - 1 不是质数；2 是质数；9 还有因数 3，因此不是质数。
- 在 main 中判断 n 是否为质数，显示 1 或 0；本例先使用 n=17。
- 试除规则：
  - n<2：非质数。
  - 检查 2～n-1 的整数 d；若 n % d == 0，n 为非质数，可以停止试除。
  - 全部检查完仍未找到整除数：质数。
- 先按规则写出 n=1、2、9、17 应显示的结果，再编写代码。

在 main 中直接判断质数

```c
#include <stdio.h>

int main(void)
{
    int n = 17;
    int result = n >= 2;
    for (int d = 2; d < n; d++) {
        if (n % d == 0) {
            result = 0;
            break;
        }
    }
    printf("%d\n", result);
    return 0;
}
```

判断与显示

main 中完成试除，用 result 保存 1 / 0，再显示结果

n >= 2 先排除小于 2 的整数；找到整除数时置 0，用 break 退出循环

将判断组织成函数

使用接口 int is_prime(int n);，质数返回 1，否则返回 0

把判断写进函数，再由 main 调用并显示；下一页核对实现

质数判断函数的返回路径

```c
int is_prime(int n)
{
    if (n < 2) {
        return 0;
    }
    for (int d = 2; d < n; d++) {
        if (n % d == 0) {
            return 0;
        }
    }
    return 1;
}
```

返回 0 的路径

小于 2：直接返回 0

找到一个整除数：返回 0，结束本次调用

直接版用 break 退出循环；此处 return 结束判断函数

返回 1 的路径

循环结束仍未找到：返回 1

n=2 时，d < n 一开始为假；没有进入循环，仍会执行最后的 return 1;

返回位置的要求

返回 1 必须放在全部试除结束之后，不能只因第一个 d 未整除就认定质数

调用质数判断函数并显示结果

```c
#include <stdio.h>

int is_prime(int n);

int main(void)
{
    int n = 17;
    int result = is_prime(n);
    printf("%d\n", result);
    return 0;
}
```

补齐函数定义

将前一页的 is_prime 定义加入同一源文件，构成完整程序

传入与接收

main 传入 n 的值，result 接收返回结果

显示与使用结果

本例显示 1，表示 17 是质数；is_prime 内部不执行 printf

结果也能用于条件判断，而不必立即显示

用已有函数查找区间内的孪生质数

- **孪生质数**是相差 2 的两个质数，例如 3 和 5。
- 给定整数 low、high，满足 0 <= low <= high <= 100。
- 输出闭区间 [low, high] 内的所有孪生质数对；两个数都须在区间内，每对按较小的数在前，以 (n,n+2) 格式显示，各对之间用空格分隔，最后换行。
- 可用接口：int is_prime(int n);
  - 质数返回 1，非质数返回 0；沿用前面编写的 is_prime 定义。
- 独立编写 main 中的遍历、调用与显示；复用判断函数，不重复试除算法。
- 写出 [1,20]、[3,5]、[3,4] 的预期输出，并说明 main 与 is_prime 的职责。

组合两个质数判断，并控制区间边界

```c
#include <stdio.h>
int is_prime(int n);
int main(void)
{
    int low = 1;
    int high = 20;
    int first_pair = 1;
    for (int n = low; n + 2 <= high; n++) {
        if (is_prime(n) && is_prime(n + 2)) {
            if (!first_pair) putchar(' ');
            printf("(%d,%d)", n, n + 2);
            first_pair = 0;
        }
    }
    putchar('\n');
    return 0;
}
```

遍历与区间边界

n 为较小数；n + 2 <= high 使两个数都在区间内

[3,5] 输出 (3,5)；[3,4] 只有换行，5 已超出区间

组合两次判断

is_prime(n) && is_prime(n + 2) 要求两数都为质数

第一次为假时，&& 不再执行右侧调用

显示与复用

first_pair 记录是否尚未输出；只在后续各对前显示空格，最后换行

输出 (3,5) (5,7) (11,13) (17,19)，行末没有空格

main 遍历并显示，is_prime 只判断一个整数

5 可以参与 (3,5) 和 (5,7)，因此 n 每次加 1

加入前面编写的 is_prime 定义，才构成完整程序

按辗转相除法设计最大公约数函数

- **最大公约数**是能同时整除两个给定整数的最大正整数。
- 输入 a、b 非负且不同时为 0；一个为 0，结果为另一个。
- b!=0：求 r=a%b，再更新 a=b、b=r，重复；b==0：返回 a。

a	b	本轮余数
48	18	12
18	12	6
12	6	0
6	0	不再求余数，结果为 6

- 独立编写函数及调用：自定名称、参数、返回类型；函数计算，main 保存并显示结果。
- 检查声明、定义、调用是否一致，返回路径是否完整；可查规则与语法。
- 先写出 (48,18)、(18,48)、(7,3)、(12,0)、(0,12) 的预期。

最大公约数函数的参数与计算

```
1 int gcd(int a, int b)
2 {
3     while (b != 0) {
4         int r = a % b;
5         a = b;
6         b = r;
7     }
8     return a;
9 }
```

输入与结果约定

接口：int gcd(int a, int b);

两个参数提供计算输入，返回值提供结果

任务约定要求非负且不同时为 0；int 类型本身不保证这一条件

局部计算与结束条件

r 保存旧 a、b 的余数，再更新两个形参值

b==0 时直接返回 a；没有以 0 为除数的求余操作

(12,0) 不进入循环，返回 12；(0,12) 执行一轮后返回 12

形参修改的范围

修改 a、b 只改变本次形参，不改变调用方的普通数值变量

调用最大公约数函数，原变量保持不变

```c
#include <stdio.h>
int gcd(int a, int b)
{
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
int main(void)
{
    int a = 48;
    int b = 18;
    int result = gcd(a, b);
    printf("%d %d %d\n", a, b, result);
    return 0;
}
```

调用与返回

main 将 a、b 的值传给 gcd，result 接收返回的最大公约数

输出与状态对应

本例输出 48 18 6

函数内部更新的是本次形参 a、b

main 的 a、b 不变；result 保存返回的 6

同名对象的区别

名称相同不使两个函数共享变量对象

5 检查函数相关错误

检查参数、返回值和局部变量的使用

检查函数返回值的使用

```c
#include <stdio.h>

int increase(int n)
{
    return n + 1;
}

int main(void)
{
    int n = 5;
    increase(n);
    printf("%d\n", n);
    return 0;
}
```

任务

把 main 中的 n 增加 1，并显示更新后的值

分析与修改

根据代码写出输出，比较它是否符合任务要求

找出哪一步没有完成任务，只修改必要的一处

修改后再检查

初始 n 分别为 0、5 时，写出修改后应有的输出，并说明哪条语句改变了 main 的 n

通过赋值使用函数返回值

```
1 n = increase(n);
2 printf("%d\n", n);
```

原因与修改

原版 increase(n); 返回 6，但 main 没有保存返回值，所以显示的 n 仍是 5

将返回值赋给 main 的 n，才能更新这个变量

两组输入核对

初始 n=0：原版显示 0，修改后显示 1

初始 n=5：原版显示 5，修改后显示 6

返回与赋值

函数先返回值，调用方再赋值更新变量；这是两项操作

检查函数的返回语句

```
1 int abs_value(int n)
2 {
3     if (n < 0) {
4         return -n;
5     }
6 }
```

任务与输入范围

输入整数范围为 -100 ~ 100，函数应返回该整数的绝对值

分析不同输入

n 为 -3、0、3 时，分别会执行哪些语句？函数是否返回了结果?

找出漏写的处理并修改，写出修改后这三个输入应得到的结果

检查原代码

阅读代码并查看编译器的错误或警告信息；不要运行没有返回结果的情况来猜输出

补全函数的返回语句

```
1 int abs_value(int n)
2 {
3     if (n < 0) {
4         return -n;
5     }
6     return n;
7 }
```

修改后的结果

n=-3：执行 return -n;，结果为 3

n=0：执行 return n;，结果为 0

n=3：执行 return n;，结果为 3

为什么要补 return

原版在 n>=0 时没有返回结果；补上 return n; 后，这些输入也能得到正确的绝对值

编译通过后，仍要核对不同输入的结果是否符合任务要求

函数常见错误与检查方法

检查函数代码时，可按下面五项逐一核对。

检查什么	常见错误	怎样检查
原型与定义	返回类型或形参类型不一致	对照两处的类型
实参与形参	数量、顺序或类型不符	按位置核对实参、形参及用途
返回值的使用	忽略所需结果；误用 void 返回值	核对返回类型及结果的使用
return 语句	某些路径漏写 return	逐条检查需要返回结果的路径
局部变量	混淆同名变量；使用未初始化的值	核对所属函数、作用域与初始化

- 用编译错误或警告定位名称、类型问题；编译通过后仍需核对变量值和输出。
- 修改后用普通输入和易暴露错误的输入复查；绝对值函数可测负数、零、正数。

6 递归的初步认识

沿调用过程理解递归的结束与返回

什么叫递归函数?

```
1 int sum_to(int n)
2 {
3     if (n == 0) {
4         return 0;
5     }
6     return n + sum_to(n - 1);
7 }
```

任务与递归调用

sum_to 计算从 1 到 n 的整数之和；输入为 0 ~ 10，n=0 时返回 0

本例 sum_to 在函数体内调用自身，是一个**递归函数**

函数执行中再次调用自身，称为**递归调用**；本例调用 sum_to(n - 1)

分析调用过程

从 sum_to(3) 开始，依次写出每次调用接收的 n 值

哪次调用不再调用自身？哪些加法要等下一次调用返回后才能完成?

区分每次调用

传入 n-1 时，原调用中的 n 会改变吗？联系形参的传值说明理由

递归的调用过程

从 sum_to(3) 开始，每次调用都建立自己的形参 n。

本次调用	本次形参 n	返回前还需完成的计算
sum_to(3)	3	3 + sum_to(2)
sum_to(2)	2	2 + sum_to(1)
sum_to(1)	1	1 + sum_to(0)
sum_to(0)	0	不再调用，直接返回 0

- 每次调用的形参 n 分别保存传入的值。
- sum_to(3) 调用 sum_to(2) 时，原调用的 n 仍为 3，新调用的 n 为 2。
- 原调用等待新调用的结果，随后继续完成自己的加法。

递归的结束条件

```
1 int sum_to(int n)
2 {
3     if (n == 0) {
4         return 0;
5     }
6     return n + sum_to(n - 1);
7 }
```

结束条件

递归结束条件是使本次调用不再调用自身的条件；本例为 n==0，此时返回 0

输入怎样接近结束条件

推进指下一次调用的输入逐渐接近结束条件；本例传入 n-1

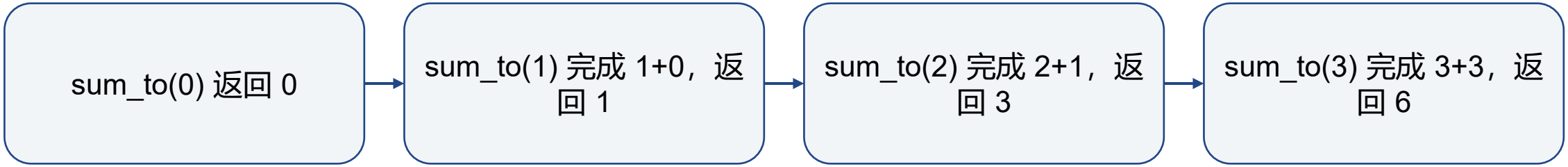
每次调用的输入减少 1，从允许的非负输入能够到达 0

若改为 n+1，从正整数开始便会离 0 越来越远

两者缺一不可

既要有结束条件，也要保证允许的输入经过后续调用能够达到该条件

递归的返回过程



- 每次调用返回后，调用方把返回值代入尚未完成的加法，再返回自己的结果。
- 例如 sum_to(2) 得到 sum_to(1) 返回的 1 后，完成 2+1，返回 3。

传入的 n	sum_to(n) 的结果
0	0
1	1
3	6
10	55

最大公约数函数的递归写法

```
1 int gcd(int a, int b)
2 {
3     if (b == 0) {
4         return a;
5     }
6     return gcd(b, a % b);
7 }
```

沿用计算任务

输入为非负整数 a、b，不同时为 0；返回最大公约数

结束与参数变化

b==0 时返回 a，不再调用自身，也不执行求余

否则传入 b 和 a%b，对应循环版中两次变量更新

下一次调用的第二个参数是余数，继续调用直到它为 0

调用与返回

gcd(48,18) → gcd(18,12) → gcd(12,6) → gcd(6,0)

最后一次返回 6，各次调用再把这个结果返回给调用方

函数的组织作用

已经观察到的做法	组织方法
查找孪生质数需要遍历、判断和显示；单数判断内部再安排试除步骤	**层次化分解**：把整体任务逐层划分为职责明确、相互联系的子任务
使用 is_prime 时关注输入整数与判断结果，暂不展开试除细节	**抽象**：保留当前使用所需的含义与约定，暂不展开内部实现
对不同整数反复调用同一个 is_prime	**复用**：同一种处理定义一次，在需要的位置再次使用
main 用判断结果安排配对与显示，完成查找	组合：让不同处理通过调用和结果使用共同完成任务

定义规定处理，调用执行处理，参数与返回值联系各次任务的数据。

函数封装同一种处理，减少重复实现；循环按规则安排重复执行。is_prime 的复用与 main 的循环遍历承担不同职责。

解释函数组织与调用的关键关系

可查看已学代码，用自己的话回答：

1. 一个函数只调用一次，为什么仍可能有价值？
2. 修改形参为什么不改变调用方的普通数值变量？返回值怎样用于更新它？
3. 为什么 is_prime 只判断一个整数，main 却能用它查找孪生质数对？
4. 递归为什么需要结束条件与朝它推进？新调用与原调用的变量有什么关系？

回答要说明职责、对象或调用关系，不能只列术语。

核对关键认识，并衔接数组学习

问题	核对依据
单次调用的价值	按职责组织并抽象处理，使整体关系更清楚
形参修改与返回	数值形参是不同对象；调用方通过赋值使用返回结果
判断与显示分工	判断通过接口提供结果，整体程序使用它组织筛选与显示
递归的结束与状态	允许输入须向结束条件推进；新调用有独立局部状态，返回后原调用继续

- 教材阅读：《C Primer Plus》第 9 章。
  - 9.1、9.2：定义、调用、参数、返回与原型。
  - 9.3.1 ~ 9.3.2：递归的基本调用关系。
- 后续学习：先认识数组及其遍历，再讨论数组作为函数参数。