第三章 分支与函数初步

C++简明双链教程（李昕著，清华大学出版社）

作者：李昕

PPT制作者：刘镇毅

目录

01 分支结构

1.1 单分支语法规则

如果条件成立，则执行语句块1，否则直接执行后继语句，语句块1被忽略，执行后继语句。

```
if(条件)
{       语句块1;    }
```

例题3.1

输入两个数m和n，对这两个数进行降序排序。

样例输入	样例输出
3 5	sorted: 5 3

两变量排序：

```cpp
#include<iostream>
using namespace std;
int main()
{
    int m,n;
    cin>>m>>n;
    if(m<n)      //此处末尾不能有分号，千万不要写成if(m<n);
    {
        int t=m;
        m=n;
        n=t;
    }
    cout<<"sorted: "<<m<<' '<<n<<endl;
}
```

例题3.1

输入两个数m和n，对这两个数进行降序排序。

样例输入	样例输入
3 5	sorted: 5 3

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int m,n;
6       cin>>m>>n;
7       if(m<n)      //此处末尾不能有分号，千万不要写成if(m<n);
8       {
9           int t=m;
10          m=n;
11          n=t;
12      }
13      cout<<"sorted: "<<m<<' '<<n<<endl;
14  }
```

- 第 7 行为条件判断，**如果 m<n**，则执行第9-11行，交换 m 和 n 的值。**如果条件不成立**，则跳过第 8-12 行，直接执行第 13 行。
- 第 7 行的右小括号右侧不能加分号，因为语句未结束。如果添加了分号，表示条件成立时执行空语句（单个分号表示空语句），第 8-12 行与第 7 行的 if 语句就没有任何关联性了。
- if 语句后面只能接一条语句或一个语句块，如果需要执行多条语句，就用大括号将多条语句封装成语句块。
- 真正的程序员必须保证代码具有**良好的缩进**。

例题3.2

对三个数a,b,c进行降序排序

样例输入	样例输出
3 2 5	sorted: 5 3 2

三变量排序：

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    cin>>a>>b>>c;
    if(a<b)
    {   swap(a,b);  }
    if(a<c)
    {   swap(a,c);  }
    if(b<c)
    {   swap(b,c);  }
    cout<<"Sorted: "<<a<<' '<<b<<' '<<c<<endl;
}
```

例题3.2

对三个数a,b,c进行降序排序

样例输入	样例输出
3 2 5	sorted: 5 3 2

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    cin>>a>>b>>c;
    if(a<b)
    {   swap(a,b);  }
    if(a<c)
    {   swap(a,c);  }
    if(b<c)
    {   swap(b,c);  }
    cout<<"Sorted: "<<a<<' '<<b<<' '<<c<<endl;
}
```

- 通过组合使用代码中的**“比较+交换”**操作，可以达成对多个数据排序的目的。当数据量比较大的时候，需要用到循环和数组，将在后继章节中介绍。
- 第 7-10 行保证 a 中存储了三个数的最大值，然后第 11-12 行对 b 和 c 进行了“比较+交换”，从而保证了三个数的降序。同样道理，如果第7行处理 a<c ，第 9 行处理 b<c，可以让 c 中先存储最小值，然后再对 a 和 b 进行“比较+交换”。

随堂练习

填充右侧代码下划线部分，

达成对三个数 a,b,c 进行降序排序。

```
#include<iostream>
using namespace std;
int main()
{
    int a,b,c;
    cin>>a>>b>>c;
    if(a<b)
    {   swap(a,b);  }
    if(b<c)
    {   swap(b,c);  }
    if(____<____)
    {   swap(____, ____);   }
    cout<<"Sorted: "<<a<<' '<<b<<' '<<c<<endl;
}
```

知识点

索引	要点	正链	反链
T311	掌握单分支条件语句的基本写法，千万注意不要添加额外的分号		
T312	掌握少量数据的排序方法	T26B	
T313	良好的代码缩进是编程基本素养，遇到左大括号缩进，遇到右大括号缩进完成		

1.2 双分支语法规则

如果条件成立，则执行语句块1，否则执行语句块2，二者必然有一项要被执行。

```
if(条件)
{     语句块1;    }
else
{     语句块2;    }
```

PS：else后面不写条件，但是具有隐含条件，即 if 的条件不成立。

例题3.3

输入一个整数m，判断m的奇偶性。

样例输入	样例输出
3	奇数

判断奇偶性：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int m;
6       cin>>m;
7       if(m%2==0)
8           cout<<"偶数"<<endl;
9       else
10      {
11          cout<<"奇数"<<endl;
12      }
13  }
```

这里 if 和 else 接的语句都是单条语句，可以像第 8 行那样不加大括号，也可以像第 11 行那样增加大括号。

例题3.3

输入一个整数m，判断m的奇偶性。

样例输入	样例输出
3	奇数

根据整型和布尔型之间的对应关系，以上程序也可以改写为：

```
#include<iostream>
using namespace std;
int main()
{
    int m;
    cin>>m;
    if(m%2)                    //或if(m&1)
        cout<<"奇数"<<endl;
    else
        cout<<"偶数"<<endl;
}
```

例题3.3

输入一个整数m，判断m的奇偶性。

样例输入	样例输出
3	奇数

进一步可以改写为问号表达式：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int m;
6       cin>>m;
7       cout<<(m%2?"奇数":"偶数")<<endl;
8   }
```

问号表达式：条件?条件成立语句:条件不成立语句

首先进行条件判断，条件成立时执行冒号前的语句，否则执行冒号后的语句。这是C/C++中的**三目运算符**，极大简化了双分支结构的书写。

知识点

索引	要点	正链	反链
T314	掌握双分支条件语句的基本写法，else部分没有显式条件，但有隐含条件。		T315, T317
T315	掌握问号表达式的用法	T314	
T316	在进行0或非0判断时，不需要写关系表达式	T244	

1.3 多分支语法规则

如果条件1成立，则执行语句块1，否则执行条件2的判断，如果条件2成立，执行语句块2，否则执行语句块3。必然有一个分支要被执行。

```
if(条件1)
{     语句块1;    }
else if(条件2)
{     语句块2;    }
else
{     语句块3;    }
```

1.3 多分支语法规则

PS：① **else**具有隐含条件，第3行的else表示条件1不成立，第5行的else表示条件1和条件2都不成立。

②C/C++提供**switch**结构进行多分支，但是其执行效率与else if的多分支相同，不建议使用switch。

```
if(条件1)
{     语句块1;    }
else if(条件2)
{     语句块2;    }
else
{     语句块3;    }
```

例题3.4

编写一个程序，根据用户输入的期末考试成绩，输出相应的成绩评定信息。成绩大于等于90分输出“优”；成绩大于等于80分小于90分输出“良”；成绩大于等于60分小于80分输出“中”；成绩小于60分输出“差”。

样例输入	样例输出
83	良

成绩评定：

```cpp
#include<iostream>
using namespace std;
int main()
{
    int score;
    cin>>score;
    int grade=score/10;
    if(grade>=9)
        cout<<"优"<<endl;
    else if(grade==8)           //建议写成if(8==grade)
        cout<<"良"<<endl;
    else if(grade>=6)           //不需要写成if(grade<8 && grade>=6)
        cout<<"中"<<endl;
    else
        cout<<"差"<<endl;
}
```

例题3.4

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      int score;
6      cin>>score;
7      int grade=score/10;
8      if(grade>=9)
9          cout<<"优"<<endl;
10     else if(grade==8)          //建议写成if(8==grade)
11         cout<<"良"<<endl;
12     else if(grade>=6)          //不需要写成if(grade<8 && grade>=6)
13         cout<<"中"<<endl;
14     else
15         cout<<"差"<<endl;
16 }
```

- 这里用到了整除特性，将百分制转换为10个区间，然后用多分支划分为5个区域。
- 第10行的 == 如果误写为 =，则程序流程为先将grade赋值为8，然后判断grade的布尔值。因为8为非零数，总是代表true，所以全部小于90的分数都会被输出良好。因此要特别注意 == 和 = 的区别。
- 当判断变量和常量是否相等时，建议把常量写在左侧，如果 == 被误写为 = ，程序编译时会报错，因为常量不能被赋值。

先特殊后一般原则

当进行多分支判断时，要注意条件的判断顺序，遵循先特殊后一般的原则。

例如判断一个三角形的类型，划分为5个类别：1）锐角三角形；2）直角三角形；3）钝角三角形；4）等腰三角形；5）等边三角形。

如果按照题目陈述顺序判断，最后两个条件永远也用不到。按照先特殊后一般的准则，正确判断顺序应该为 **5）4）3）2）1）** 。这样才能得到正确的答案。

知识点

索引	要点	正链	反链
T317	掌握多分支条件语句的基本写法，特别注意else的隐含条件，不要重复书写	T261, T314	
T318	在进行多分支判断时，要遵循先特殊后一般的顺序		

1.4 分支嵌套

- 每个 else 部分总是与它前面最近的那个缺少对应的 else 部分的 if 语句配对。
- 建议使用大括号避免二义性。

```
if(条件1)
{
    if(条件2)
    {    语句块2;    }
    else
    {    语句块3;    }
}
else
{   语句块4;    }
```

例题3.5

将例题3.4改写为分支嵌套结构

分支嵌套结构改写例题3.4：

```
#include<iostream>
using namespace std;
int main()
{
    int score;
    cin>>score;
    if(score>=60)
    {
        if(score<70)
            cout<<"及格"<<endl;
        else if(score<80)
        {   cout<<"一般"<<endl;    }
        else if(score<90)
        {   cout<<"良好"<<endl;    }
        else
        {   cout<<"优秀"<<endl;    }
    }
    else
    {   cout<<"不及格"<<endl;  }
}
```

例题3.5

将例题3.4改写为分支嵌套结构

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int score;
6       cin>>score;
7       if(score>=60)
8       {
9           if(score<70)
10              cout<<"及格"<<endl;
11          else if(score<80)
12          {    cout<<"一般"<<endl;     }
13          else if(score<90)
14          {    cout<<"良好"<<endl;     }
15          else
16          {    cout<<"优秀"<<endl;     }
17      }
18      else
19      {    cout<<"不及格"<<endl;   }
20  }
```

- 以第11行为例，不要将其写成else if(score>=70 && score<80)，这里的else代表的含义就是 score>=70，不需要重复表达。
- 每个if或else可以接一条简单语句（第10行）或一条复合语句（第12,14,16,19行），复合语句即将多条语句用大括号包含到一起。对于初学者，建议采用**复合语句**的形式编写代码，减少犯错概率。

知识点

索引	要点	正链	反链
T319	掌握嵌套结构的写法，尽量全部使用复合语句		

02 分支程序优化

例题3.6

中国有句俗语叫“三天打鱼两天晒网”。假设某人从某天起，开始三天打鱼两天晒网”，问这个人在以后的第N天中是"打鱼“还是晒网”？

【输入】
输入在一行中给出1个不超过1000的正整数N
【输出】
在一行中输出此人在第N天中是'Fishing"(打鱼")还是“Drying"(晒网")，并且输出"on day N"。

样例输入	样例输出
103	Fishing on day 103
34	Drying on day 34

三天打鱼两天晒网：

```cpp
#include<iostream>
using namespace std;
int main()
{
    int day;
    cin>>day;
    if (day%5==1||day%5==2||day%5==3)
        cout<<"Fishing on day "<<day<<endl;
    else
        cout<<"Drying on day "<<day<<endl;
    return 0;
}
```

例题3.6

中国有句俗语叫“三天打鱼两天晒网”。假设某人从某天起，开始三天打鱼两天晒网”，问这个人在以后的第N天中是"打鱼“还是晒网”？

【输入】
输入在一行中给出1个不超过1000的正整数N
【输出】
在一行中输出此人在第N天中是'Fishing"(打鱼")还是“Drying"(晒网"),并且输出"on day N"。

样例输入	样例输出
103	Fishing on day 103
34	Drying on day 34

考虑晒网的天数少，可以书写上减少一个条件，因此第7-10行可以改写为：

```
if (day%5==4||day%5==0)
    cout<<"Drying on day "<<day<<endl;
else
    cout<<"Fishing on day "<<day<<endl;
```

连续数字可以用范围表示，但是0和4在这个范围的两侧，不好归结。这时就可以考虑数学变换。第7-10行可以改写为：

```
if ((day-1)%5<3)
    cout<<"Fishing on day "<<day<<endl;
else
    cout<<"Drying on day "<<day<<endl;
```

最后，还可以用问号表达式进行书写优化：

```
cout<<((day-1)%5<3?"Fishing":"Drying")<<" on day "<<day<<endl;
```

例题3.7

你买了一箱n个苹果，很不幸的是买完时箱子里混进了一条虫子。虫子每x小时能吃掉一个苹果假设虫子在吃完一个苹果之前不会吃另 一个,那么经过y小时你还有多少个完整的苹果?

【输入】输入仅一行，包括n，x和y (均为整数)
【输出】输出仅一行，剩下的苹果个数

样例输入	样例输出
10 4 9	7

虫子吃苹果：

```cpp
#include<iostream>
using namespace std;
int main()
{
    int n,x,y;
    cin>>n>>x>>y;
    if (y%x==0)
        cout<<(n-y/x)<<endl;
    else
        cout<<(n-y/x-1)<<endl;
    return 0;
}
```

程序思想非常简单，如果y是x的倍数，则直接得到结果，否则需要多减一天。

例题3.7

你买了一箱n个苹果，很不幸的是买完时箱子里混进了一条虫子。虫子每x小时能吃掉一个苹果假设虫子在吃完一个苹果之前不会吃另 一个,那么经过y小时你还有多少个完整的苹果?

【输入】输入仅一行，包括n，x和y (均为整数)
【输出】输出仅一行，剩下的苹果个数

样例输入	样例输出
10 4 9	7

这里实际上是一种向上取整的方法，可以通过数学进行优化。因为x和y都是整数，所以最小的差距是1。如果在y的基础上增加x-1，那么只要有余数，被吃掉苹果的数量就会加1，如果正好整除，x-1会被整除操作自动舍弃。以下代码优化后去掉了分支操作：

```
1   #include<iostream>
2   using namespace std;
3   int main()
4   {
5       int n,x,y;
6       cin>>n>>x>>y;
7       cout<<(n-(y+x-1)/x)<<endl;
8       return 0;
9   }
```

例题3.7

你买了一箱n个苹果，很不幸的是买完时箱子里混进了一条虫子。虫子每x小时能吃掉一个苹果假设虫子在吃完一个苹果之前不会吃另 一个,那么经过y小时你还有多少个完整的苹果?

【输入】输入仅一行，包括n，x和y (均为整数)
【输出】输出仅一行，剩下的苹果个数

样例输入	样例输出
10 4 9	7

C/C++中提供了ceil函数进行向上取整。书写上比数学方法麻烦，但是更容易理解。

```cpp
#include<iostream>
#include<cmath>                                  //支持ceil函数的定义
using namespace std;
int main()
{
    int n,x,y;
    cin>>n>>x>>y;
    cout<<(n-ceil(y/(float)x))<<endl;    //通过强制类型转换，把整除改为浮点数除法
    return 0;
}
```

总而言之，可以通过语法、数学方法或库函数对程序进行优化。

知识点

索引	要点	正链	反链
T321	优化是程序员不断追求的目标，既可以锻炼思维逻辑，又可以减小出错概率。简单的书写优化可以通过语法、数学方法或库函数进行	T291	
T322	掌握对数值进行向上取整的方法	T256	

03 自定义函数

3.1 函数定义

- 从之前使用的库函数可以知道，一个功能独立的函数，可以被**一次定义，多次执行**。
- 函数每次执行时可以输入不同的参数，从而得到不同的结果。
- 函数包括多条语句，完成了一个独立的功能，是对**一个行为的封装**。
- 除了C/C++中提供的库函数外，也可以自行定义函数。

```
函数类型   函数名(参数声明列表)
{
   语句块
   return 返回值;
}
```

3.1 函数定义

- **函数类型：**函数返回值的数据类型，如int，char；若没有返回值，类型应为void。函数类型与第4行return返回值数据类型相对应。
- **函数名**是调用这个函数的唯一标识。
- **参数声明列表**用逗号分隔的一组变量说明，形式：类型参数1，类型参数2，…，其中若有多个参数，每个参数前的数据类型都要写。如果没有参数列表，成为无参函数。
- 第1行称为**函数头**，第2-5行称为**函数体**。

```
函数类型   函数名(参数声明列表)
{
   语句块
   return 返回值;
}
```

3.1 函数定义示例

- 第4-9行称为函数定义，第15行称为函数调用。
- 定义时使用的参数称为**形参**，例如x, y。调用时使用的参数称为**实参**，例如a, b。
- 形参的每个参数的数据类型都要写，实参不能写数据类型。
- 执行第15行时，对应实参对形参进行赋值。即a对x赋值，b对y赋值。
- 实参和形参的数量、顺序和对应参数的数据类型必须相同。
- 第15行函数调用结束后，将函数的返回值按照函数类型返回给main函数并进行输出。

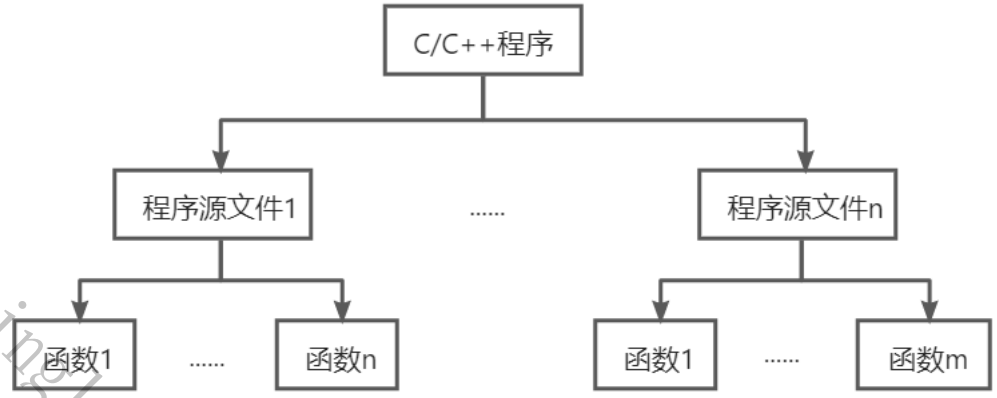
```
1   #include<iostream>
2   using namespace std;
3
4   int my_max(int x,int y)
5   {
6       int z;
7       z = x > y ? x : y;
8       return z;
9   }
10
11  int main()
12  {
13      int a,b;
14      cin>>a>>b;
15      cout<<my_max(a,b)<<endl;
16      return 0;
17  }
```

知识点

索引	要点	正链	反链
T331	掌握函数定义的基本写法		

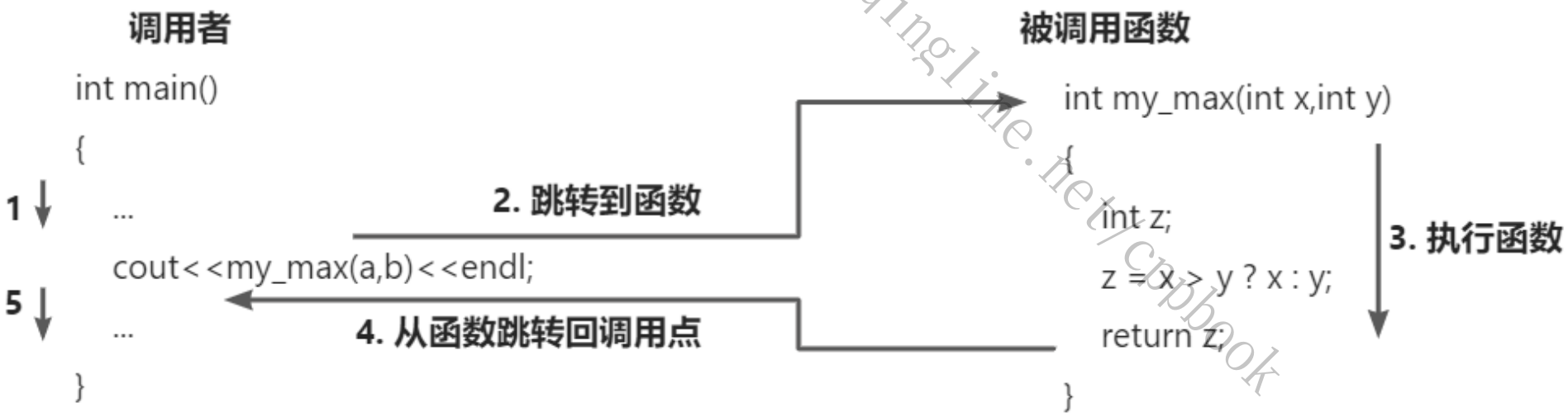
3.2 函数执行顺序

一个C/C++程序可以由一个或多个源程序文件组成，一个C/C++源程序文件可以由一个或多个函数组成。但一个C/C++程序中，有且仅有一个**主函数main**，它是一个程序的起点。程序从main函数的第一条语句开始执行，到main函数的最后一条语句执行完结束。



3.2 函数执行顺序

函数定义用于向编译器解释函数的接口和实现，并不会导致函数被执行。一个函数是否被执行，由其是否被main函数直接或间接调用决定。当调用一个函数时，所有当前信息会被保存，包括当前执行的语句的位置。然后用实参对形参进行赋值，开始执行被调用的函数。函数执行结束后，返回到之前的调用点，返回值被传递给调用者。



知识点

索引	要点	正链	反链
T332	掌握函数的执行顺序		

3.3 函数声明

在程序中调用函数应遵循**“先定义后使用”**的原则，即被调函数的定义要出现在主调函数的定义之前，与**变量必须先定义后使用**一样。也可以采用声明的方式改变顺序，如右侧代码所示：

```cpp
#include<iostream>
using namespace std;

int my_max(int x,int y);

int main()
{
    int a,b;
    cin>>a>>b;
    int ret = my_max(a,b);
    cout<<ret<<endl;
    return 0;
}

int my_max(int x,int y)
{
    int z;
    z = x > y ? x : y;
    return z;
}
```

3.3 函数声明

第4行称为函数声明，可以简单地将第15行的函数头复制一遍，并以；结束，如果有函数声明，那么函数定义可以出现在程序的任何地方。

实际上，每个库函数在使用前，必须包括对应的头文件，包含了函数的定义或声明。

PS：对于库函数，官方文档https://cplusplus.com/reference/中给出的全部是函数声明，需要掌握通过官方文档给出的函数声明，了解一个库函数的使用方法。

```
1  #include<iostream>
2  using namespace std;
3
4  int my_max(int x,int y);
5
6  int main()
7  {
8      int a,b;
9      cin>>a>>b;
10     int ret = my_max(a,b);
11     cout<<ret<<endl;
12     return 0;
13 }
14
15 int my_max(int x,int y)
16 {
17     int z;
18     z = x > y ? x : y;
19     return z;
20 }
```

例题3.5

对下面问题，分析已知条件和结果，写出函数声明

1 求两个实数的和。

2 求两个整数的最大公约数。

3 判断任一个正整数n是否为素数。

4 在一行中输出指定个数的星号（*）

分支嵌套结构改写例题3.4：

```
1   float sum(float x, float y);
2   int   common_divisor(int m, int n);
3   int   prime(int n);
4   void  print_star(int n);
```

1. 第一个问题要求输入两个实数，因此设定两个浮点数类型的形参；要求得到两个实数的和，该和也应该是实数类型，因此函数的返回类型也为float。
2. 第四个问题要求打印，不需要任何返回，因此函数类型设置为void，表示不需要返回值。

知识点

索引	要点	正链	反链
T333	掌握函数声明的基本写法，掌握通过查阅官方文档掌握一个库函数的使用		

3.4 函数返回值

函数类型决定了函数返回值的类型，函数的返回值通过return语句返回。return的作用是**终止函数运行，返回调用者**，若有返回值，将返回值传递给调用者。return可以采用右侧形式：

return (表达式);	表达式结果类型与函数类型相同
return 表达式;	表达式结果类型与函数类型相同
return ;	对应void函数类型

3.4 函数返回值

```
1   float max(float x, float y)
2   {
3       if(x > y)  return x ;
4       else  return y ;
5   }
```

简化

```
1   float max(float x, float y)
2   {
3       return (x > y) ? x:y;
4   }
```

一个函数**只能有一个返回值**，但可以有多个return。因为return会终止当前函数，所以即使一个函数有多个return，也只是有一个返回值。

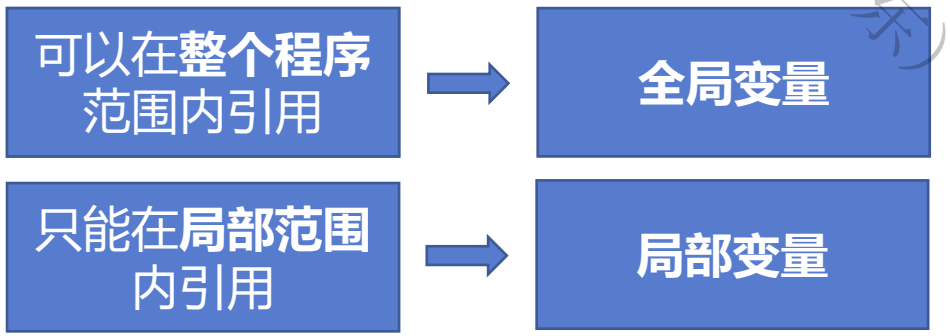
PS: 当return语句中表达式的类型和函数返回值的类型不匹配时，会发生**隐式类型转换**。函数的返回值转换为函数类型指定的数据类型。

知识点

索引	要点	正链	反链
T334	掌握函数类型和函数返回值，return可以有多个，但是只有一个被执行。当返回值与函数类型不一致时，会发生隐式类型转换。		

3.5 变量的作用域

变量的作用域指**变量有效的范围**。



C/C++中用大括号定义一个代码块，代码块可以层级嵌套。每个局部变量只在定义它的那个大括号内生效，如果不在任何大括号里，即为全局变量。

```
1   #include<iostream>
2   using namespace std;
3
4   int a=9;
5   void f(int a)
6   {
7       a=5;
8       {
9           a++;
10          cout<<a<<endl;
11          int a=1;
12          cout<<a<<endl;
13      }
14      cout<<a<<endl;
15  }
16  int main()
17  {
18      cout<<a<<endl;
19      int a=7;
20      cout<<a<<endl;
21      f(a);
22      cout<<a<<endl;
23  }
```

输出

```
9
7
6
1
6
7
```

3.5 变量的作用域

- 在同一个层级里，一个变量名称只能使用一次，但是在不同层级里，变量名称可以相同。
- 当不同作用范围的同名变量发生冲突时，以**作用范围最小的变量为准**。函数的参数是属于该函数的局部变量。
- 一个局部变量脱离了它的作用域，就会被自动释放内存，不能再被使用。

```
1   #include<iostream>
2   using namespace std;
3
4   int a=9;
5   void f(int a)
6   {
7       a=5;
8       {
9           a++;
10          cout<<a<<endl;
11          int a=1;
12          cout<<a<<endl;
13      }
14      cout<<a<<endl;
15  }
16  int main()
17  {
18      cout<<a<<endl;
19      int a=7;
20      cout<<a<<endl;
21      f(a);
22      cout<<a<<endl;
23  }
```

输出

```
9
7
6
1
6
7
```

3.5 变量的作用域

- 程序从第18行开始执行，此时只有一个全局变量a生效，因此输出9。
- 第20行存在两个a，全局变量和第19行定义的main函数中的局部变量a，局部变量的作用域更小，因此局部变量生效，输出结果为7。
- 第21行开始转入函数f执行，用第19行定义的a，对第5行定义的形参a进行赋值。

```
1   #include<iostream>
2   using namespace std;
3
4   int a=9;
5   void f(int a)
6   {
7       a=5;
8       {
9           a++;
10          cout<<a<<endl;
11          int a=1;
12          cout<<a<<endl;
13      }
14      cout<<a<<endl;
15  }
16  int main()
17  {
18      cout<<a<<endl;
19      int a=7;
20      cout<<a<<endl;
21      f(a);
22      cout<<a<<endl;
23  }
```

输出

```
9
7
6
1
6
7
```

3.5 变量的作用域

- 第19行的a是main函数的局部变量，它在函数f的执行过程中是完全不可见的。
- 第9-10行存在两个a，全局变量和第5行定义的f函数中的局部变量a，局部变量的作用域更小，因此局部变量生效，输出结果为6。

```
1   #include<iostream>
2   using namespace std;
3
4   int a=9;
5   void f(int a)
6   {
7       a=5;
8       {
9           a++;
10          cout<<a<<endl;
11          int a=1;
12          cout<<a<<endl;
13      }
14      cout<<a<<endl;
15  }
16  int main()
17  {
18      cout<<a<<endl;
19      int a=7;
20      cout<<a<<endl;
21      f(a);
22      cout<<a<<endl;
23  }
```

输出

```
9
7
6
1
6
7
```

3.5 变量的作用域

- 第12行存在三个a，全局变量、第5行定义的f函数中的局部变量a、第11行定义的代码块中的a，第11行定义的a作用域最小，从第11行开始生效，输出结果为1。
- 执行到14行时，第11行定义的a已经失效，不再其作用，因此这里的a是第5行定义的a，经过第7和9行的处理后，值为6。
- 函数f执行结束，返回到main函数的第22行，此时f函数中定义的所有a都失效了，因此使用的是19行定义的a，其值为7。

```
1   #include<iostream>
2   using namespace std;
3
4   int a=9;
5   void f(int a)
6   {
7       a=5;
8       {
9           a++;
10          cout<<a<<endl;
11          int a=1;
12          cout<<a<<endl;
13      }
14      cout<<a<<endl;
15  }
16  int main()
17  {
18      cout<<a<<endl;
19      int a=7;
20      cout<<a<<endl;
21      f(a);
22      cout<<a<<endl;
23  }
```

输出

```
9
7
6
1
6
7
```

知识点

索引	要点	正链	反链
T335	掌握变量的作用域		T341, T781

3.6 参数传递和引用

- **实参**是定义在调用函数范围内的变量，**形参**是定义在被调用函数中的局部变量。两者必须在数量、顺序、类型上完全匹配。
- 当实参与形参的类型不匹配时，会发生**隐式类型转换**，函数中按照形参的类型进行计算，否则将报错。
- 除了在调用时，实参对形参有一次赋值操作外，二者在定义或存储空间等其他方面没有任何交集。因此形参的改变不会影响实参。这种方式被称为**传值方式**。

```cpp
#include<iostream>
using namespace std;

void my_swap ( int  x, int y )
{
    int  temp = x;
    x = y;
    y = temp;
    cout<<"x="<<x<<" y="<<y<<endl;
}
int main (  )
{
    int a, b;
    cin>>a>>b;
    my_swap(a, b);
    cout<<"a="<<a<<" b="<<b<<endl;
}
```

样例输入	样例输出
3 5	x=5 y=3 a=3 b=5

3.6 参数传递和引用

上页代码第15行函数调用时，用a对x赋值，用b对y赋值，此后a,b和x,y就没有任何关系了。a,b和x,y在存储空间上是完全不相关的。

如果希望通过函数交换实参，就需要使用新的概念——**引用**。

语法上在定义变量时，在其前面加一个**&**符号。它表示新的变量与原有变量共享存储空间，是对原有变量存储空间增加了一个新的变量名称。（一个存储空间同时有两个名字）

```
1  #include<iostream>
2  using namespace std;
3
4  void ref_swap ( int  &x, int &y)
5  {
6      int  temp = x;
7      x = y;
8      y = temp;
9      cout<<"x="<<x<<" y="<<y<<endl;
10 }
11 int main (  )
12 {
13     int a, b;
14     cin>>a>>b;
15     ref_swap(a, b);
16     cout<<"a="<<a<<" b="<<b<<endl;
17 }
```

样例输入	样例输出
3 5	x=5 y=3 a=3 b=5

3.6 参数传递和引用

- 第4行的引用表示x和a指向了同一个存储空间，y和b指向了同一个存储空间，对x和y的修改，其实就是对a和b的修改。
- 第6-8行交换了形参x和y的值，因此实参a和b也被交换了。
- 引用是C++中提出的概念，C语言中没有引用的概念。引用可以通俗地理解为一个存储空间的多个变量名。也可以理解为把实参的地址传递给了形参，形参用这个地址来访问变量。因此这种方式也称为**传地址方式**。

```
1  #include<iostream>
2  using namespace std;
3
4  void ref_swap ( int  &x, int &y)
5  {
6      int  temp = x;
7      x = y;
8      y = temp;
9      cout<<"x="<<x<<" y="<<y<<endl;
10 }
11 int main (  )
12 {
13     int a, b;
14     cin>>a>>b;
15     ref_swap(a, b);
16     cout<<"a="<<a<<" b="<<b<<endl;
17 }
```

样例输入	样例输出
3 5	x=5 y=3 a=3 b=5

3.6 参数传递和引用

C++中库函数swap和ref_swap基本相同，实现了变量交换。但C++中swap是使用模板完成的，更加复杂一些。

如果两个或多个函数同名，但是参数的类型、顺序或数量不同，会被认为是不同的函数，这种现象被称为**函数重载**。但是如果两个函数的函数名和参数完全一致，只有返回类型不同，会被认为是重复定义，报语法错误。

```cpp
#include<iostream>
using namespace std;

void ref_swap ( int  &x, int &y)
{
     int  temp = x;
     x = y;
     y = temp;
     cout<<"x="<<x<<" y="<<y<<endl;
}
int main (  )
{
     int a, b;
     cin>>a>>b;
     ref_swap(a, b);
     cout<<"a="<<a<<" b="<<b<<endl;
}
```

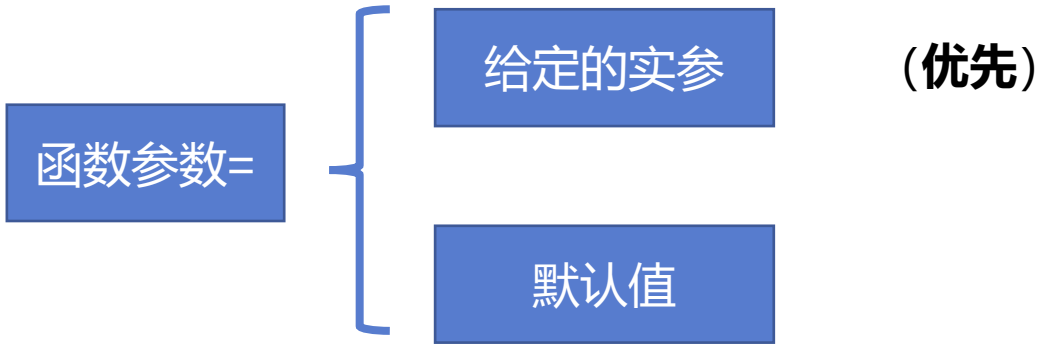
样例输入	样例输出
3 5	x=5 y=3 a=3 b=5

知识点

索引	要点	正链	反链
T336	掌握实参与形参的关系，实参与形参必须在数量、顺序、类型上完全匹配 掌握参数传递的传值方式和引用方式		T338, T613
T337	掌握函数重载		

3.7 参数的默认值*

形参可以在定义时以赋值的形式给出默认值。在调用时，如果形参有对应的实参，则用实参赋值，如果没有对应的实参，用默认值。参数的默认值必须**从右向左**提供，即无默认值的参数不能出现在有默认值参数的右边。



（优先）

```cpp
#include<iostream>
using namespace std;

int func(int a,int b=4,int c=5)
{
    return  a+b+c;
}
int main (  )
{
    int a,b,c;
    cin>>a>>b>>c;
    cout<<func(a)<<endl;
    cout<<func(a,b)<<endl;
    cout<<func(a,b,c)<<endl;
}
```

样例输入	样例输出
6 7 8	15 18 21

3.7 参数的默认值*

- 第12行只给定了一个实参，因此b和c用默认值，得到结果为6+4+5=15
- 第13行形参b用实参值7，得到结果为6+7+5=18
- 第14行都采用传入的实参，得到结果为6+7+8=21
- 由此可知，当参数个数为1,2,3时，都可以调用func函数，这样可以避免重复写三个重载函数。形参a没有默认值，调用时不可以省略实参。
- 参数的默认值必须**从右向左**提供，因此int func(int a=3,int b,int c=5)是不允许的，但是可以全部带有默认值，例如int func(int a=3,int b=4,int c=5)是可以的。

```
1   #include<iostream>
2   using namespace std;
3
4   int func(int a,int b=4,int c=5)
5   {
6       return  a+b+c;
7   }
8   int main (  )
9   {
10      int a,b,c;
11      cin>>a>>b>>c;
12      cout<<func(a)<<endl;
13      cout<<func(a,b)<<endl;
14      cout<<func(a,b,c)<<endl;
15  }
```

样例输入	样例输出
6 7 8	15 18 21

知识点

索引	要点	正链	反链
T338	掌握参数默认值的语法规则	T336	

04 局部变量和函数的内存模型

4.1 局部变量和函数的内存模型

程序和数据平常存储在硬盘等存储器上，运行时，这些内容都会被加载到内存中。为了更好地管理内存，内存被以字节为单位进行编号，这些编号称为内存地址。也就是说一个内存地址代表一个字节（8bit）的存储空间。一个32位的电脑，最多支持2^{32}=4GB的内存空间，而64位电脑就可以支持2^{64}的内存空间。当代计算机的物理内存通常大于4GB，这也是这些计算机必须安装64位操作系统的原因。

4.1 局部变量和函数的内存模型

编程中的每一行代码，代码中用到的每个数据，都需要在内存上有其映射地址。当定义一个变量时，根据数据类型分配一个相应的内存空间，比如int类型分配 **sizeof(int)=4** 个字节的内存空间，这4个字节必须连续，4个字节对应4个地址，其中最小的一个地址作为这个变量的地址。当使用一个变量时，编译器实际上是通过变量名获知变量的地址，然后根据数据类型告诉编译器要从该地址开始的多少字节用来解释，按照什么方式进行解释。

```
#include<iostream>
int main (  )
{
    char c='A';
    int a = 65;
    printf("%d %c %d %c",c,c,a,a);
}
```

样例输入	样例输出
无	65 A 65 A

4.1 局部变量和函数的内存模型

- 字符实际上先被映射成ASCII码，然后才能在计算机中存储。因此字符'A'和整数65实际上在内存中的二进制表示是完全相同的。
- 占位符%d要求将变量按照十进制整数进行解释，%c要求将变量按照字符类型进行解释，因此同一个变量，因为要求的解释方式不同，就会输出不同的结果。本质上是因为它们在内存中的二进制表示形式是完全相同的。

```
1  #include<iostream>
2  int main (  )
3  {
4      char c='A';
5      int a = 65;
6      printf("%d %c %d %c",c,c,a,a);
7  }
```

样例输入	样例输出
无	65 A 65 A

4.1 局部变量和函数的内存模型

当一个程序开始运行时，会分配一个称为堆栈（stack，简称为栈）的空间。栈是一个一端封闭，一端开放的数据结构。当有新的数据分配需求时，从栈顶开始依次分配。当需要释放空间时，也只能先释放栈顶的空间。栈的这种特性被称为**“后进先出”**。栈底在高地址，栈顶在低地址。因此先定义的变量会相对靠下，后定义的变量会相对靠上，总体来说是从高到低分配，这就解释了下面代码中 abcde 变量为什么地址会从低到高。理论上这些变量的内存地址应该是相邻的，但是因为操作系统一些机制（例如编译器数据对齐，64位操作系统会尽量以8个字节为“单位”）的存在，连续定义的多个局部变量在内存中并不一定连续。

PS: 局部变量内存空间的分配与回收是由编译器自动管理的，不需要用户人工干预。

4.1 局部变量和函数的内存模型

用操作符 sizeof 获得每个变量占据的内存空间大小，在变量前添加&获取该变量的内存地址。

C/C++中，将char类型地址理解为字符数组的起点，如果该数组中存在\0字符，则将这个字符数组理解为从起点到结束标记**\0**的字符串。当cout遇到char类型的地址时，将会从该地址开始，将遇到所有内容按照字符串来进行解释并输出，直到遇到\0停止。

```cpp
#include<iostream>
using namespace std;
void dummy(){
    int x;
    cout<<"x("<<x<<")\t的地址为"<<(&x)<<"，占据"<<sizeof(x)<<"字节"<<endl;
}
int main (  )
{
    int a=3;
    double b=1.2;
    char c='A';
    long long d=4;
    float e=8.7;
    dummy();
    cout<<"a("<<a<<")\t的地址为"<<(&a)<<"，占据"<<sizeof(a)<<"字节"<<endl;
    cout<<"b("<<b<<")\t的地址为"<<(&b)<<"，占据"<<sizeof(b)<<"字节"<<endl;
    cout<<"c("<<c<<")\t的地址为"<<(void*)(&c)<<"，占据"<<sizeof(c)<<"字节"<<endl;
    cout<<"d("<<d<<")\t的地址为"<<(&d)<<"，占据"<<sizeof(d)<<"字节"<<endl;
    cout<<"e("<<e<<")\t的地址为"<<(&e)<<"，占据"<<sizeof(e)<<"字节"<<endl;
}
```

样例输入	样例输出
无	x(32762) 的地址为0x61fdbc，占据4字节 a(3) 的地址为0x61fe1c，占据4字节 b(1.2) 的地址为0x61fe10，占据8字节 c(A) 的地址为0x61fe0f，占据1字节 d(4) 的地址为0x61fe00，占据8字节 e(8.7) 的地址为0x61fdfc，占据4字节

4.1 局部变量和函数的内存模型

因此如果不做强制类型转换，第17行将会输出一个字符串。而第17行期望输出地址。因此将该地址强制转换为无类型进行输出，这样才能正确输出一个地址。实际上可以将其转换为任意非字符地址类型，都可以正确输出。char类型地址只是一个特例。其中的符号*表示指针，即存储地址的变量，将在后继章节中进行详细展开。

```
1  #include<iostream>
2  using namespace std;
3  void dummy(){
4      int x;
5      cout<<"x("<<x<<")\t的地址为"<<(&x)<<"，占据"<<sizeof(x)<<"字节"<<endl;
6  }
7  int main (  )
8  {
9      int a=3;
10     double b=1.2;
11     char c='A';
12     long long d=4;
13     float e=8.7;
14     dummy();
15     cout<<"a("<<a<<")\t的地址为"<<(&a)<<"，占据"<<sizeof(a)<<"字节"<<endl;
16     cout<<"b("<<b<<")\t的地址为"<<(&b)<<"，占据"<<sizeof(b)<<"字节"<<endl;
17     cout<<"c("<<c<<")\t的地址为"<<(void*)(&c)<<"，占据"<<sizeof(c)<<"字节"<<endl;
18     cout<<"d("<<d<<")\t的地址为"<<(&d)<<"，占据"<<sizeof(d)<<"字节"<<endl;
19     cout<<"e("<<e<<")\t的地址为"<<(&e)<<"，占据"<<sizeof(e)<<"字节"<<endl;
20 }
```

样例输入	样例输出
无	x(32762) 的地址为0x61fdbc，占据4字节 a(3) 的地址为0x61fe1c，占据4字节 b(1.2) 的地址为0x61fe10，占据8字节 c(A) 的地址为0x61fe0f，占据1字节 d(4) 的地址为0x61fe00，占据8字节 e(8.7) 的地址为0x61fdfc，占据4字节

4.1 局部变量和函数的内存模型

内存地址通常用16进制表示，代码3.25中main函数的5个连续变量的内存地址在整体上从高到低分配。

连续定义的变量内存上有时连续，例如：

0x61fe10-0x61fe0f=1（c占据的字节数），

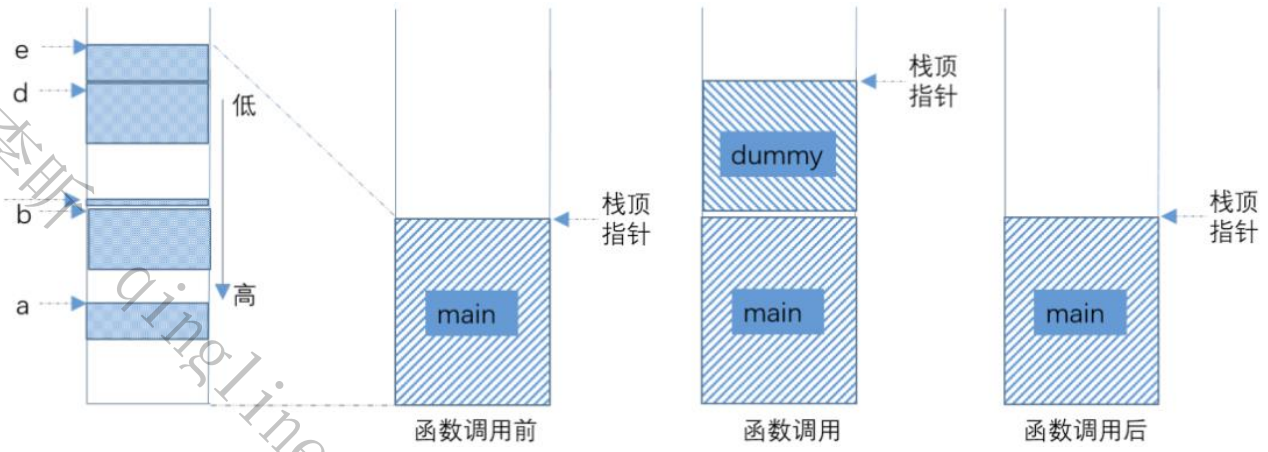
0x61fe00-0x61fdfc=4（e占据的字节数）；

有时不连续，例如：

0x61fe1c-0x61fe10=12（不等于b占据的字节数8），

0x61fe0f-0x61fe00=15（不等于d占据的字节数8），

如右图所示。



4.1 局部变量和函数的内存模型

内存的最终分配结果与编译器、计算机和操作系统都紧密相关，当这些条件不一致时，都可能会导致结果不相同。同一个程序在不同时间运行，内存地址也会不同。但其基本原理不会发生变化。

当一个函数被调用前，它是不占据内存空间的。只有调用时才从栈顶分配空间。函数调用结束后，该空间被自动释放。因为函数调用还要保存调用点等信息，因此x的内存地址和e并不连续。但是要注意栈空间是有大小限制的，如果分配的变量空间过多，或调用的函数过多，会形成栈溢出。因此不要分配过多的变量（主要指数组的元素数量不能过大），不要形成无限递归调用。

4.1 局部变量和函数的内存模型

本小节内容的理解对于初学者过于复杂，因此只需要谨记三个原则：

a) 每个变量都会根据其数据类型分配内存，因此一定存在对应的内存地址，即在内存中的位置编号。

b) 局部变量会自动分配和释放内存。对于连续定义的变量，其内存地址不一定连续。

c) 数组的元素数量不能过大。可以自行尝试大小，一般情况下足够用，一旦超出运行时会报错。

知识点

索引	要点	正链	反链
T341	掌握局部变量和函数调用的内存模型，这是掌握指针的基础	T335	T513, T611, T851

05 变量的深度理解*

5.1 变量的深度理解*

变量名是一个**标识符**(identifier)，用来指代一块内存区域，即变量。这块区域的值一般是可以更改的，这就是它“变”的由来。如果使用如const等一些修饰符号来限定这一内存区域的操作特性，例如const int b;，const修饰使变量不能更改，这样的变量称为常变量。变量使程序代码操作内存更加方便，因此C/C++被称为高级语言。定义int a;时,编译器分配4个字节内存，并将该4个字节的空间名字为a(即变量名)，当用到变量名a时,就是在使用对应4个字节的内存空间。

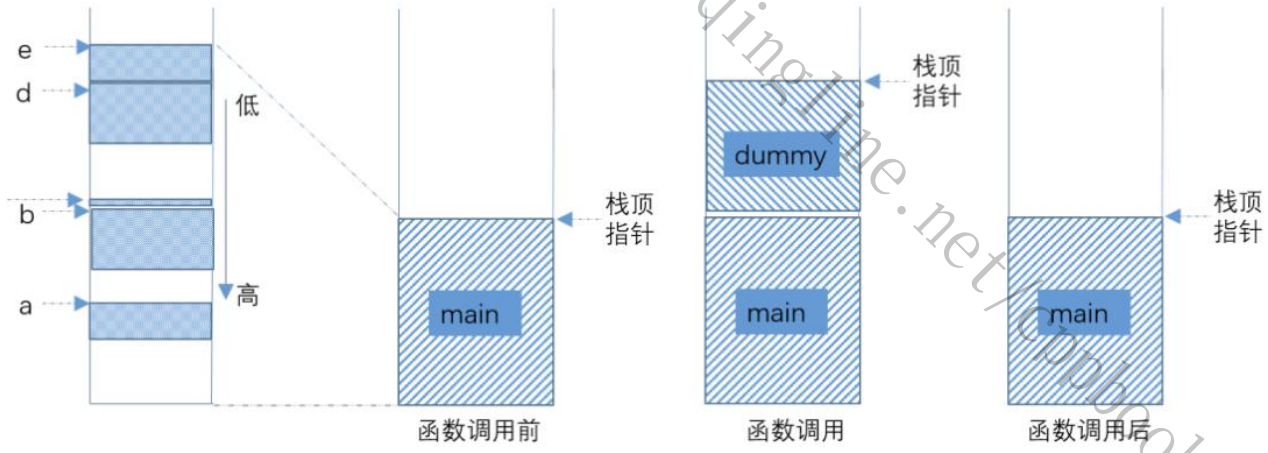
5.1 变量的深度理解*

一个变量与一块内存空间绑定，那么变量怎样存储地址呢？

在C/C++等编译型程序中，变量实际上不存储地址。变量名是给程序员看的，让程序员可以方便直观的操作内存地址。一段代码经过编译、链接之后形成二进制的机器代码，然后才能够执行，机器代码是给计算机使用的。在经过编译链接后，所有的变量在机器代码中都被直接替换为对应的地址，也就是说，机器代码中不存在变量名。当程序中操作一个变量时，最终是通过地址访问对应的内存区域，完成相应的操作。例如a=5;就是把a指向的内存区域的值修改为5；b=a;就是将a指向的内存区域的值复制到b指向的内存区域中。可用取地址符号&来获得它所代表的变量的存放地址。

5.1 变量的深度理解*

在一个局部区域定义两个变量如果同名，都会在机器代码中被编译为相同的地址，不能被区分为两个变量，因此相同局部区域的变量不能同名。如果在不同局部区域，例如下图中的main和dummy，这两个区域不在一起，因此在两个区域里即使存在同名变量，也会被编译成两个不同的地址，因此不同区域的变量可以同名。



5.1 变量的深度理解*

实参和形参被分配到两个不同的内存块中，地址完全不同，除了调用时有一次赋值操作外，实参与形参完全不相关，因此形参的修改不会影响实参，这也是**传值方式**的特性。

当采用引用定义变量时，新变量和被引用变量在机器代码中被编译成相同的地址，并没有新的存储空间被分配。而变量是通过地址操作相应的存储空间。因此采用**传地址方式**时，修改形参实际上就是在修改实参。

知识点

索引	要点	正链	反链
T351	掌握变量在编译后的表示方法		

THANKS