第六章 程序设计函数探幽

主讲教师：同济大学电子与信息工程学院 陈宇飞
同济大学电子与信息工程学院 龚晓亮

目录

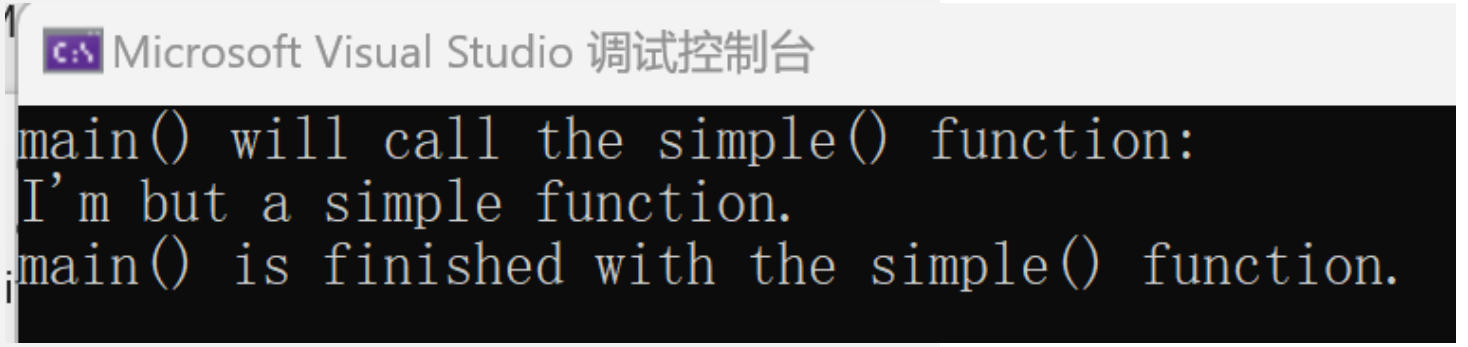
- 复习函数的基本知识
- 函数参数和按值传递
- 递归
- 内联函数
- 默认参数
- 函数重载
- 函数模板

目录

- 复习函数的基本知识
  - 定义函数
  - 函数原型和函数调用

1.1 定义函数

```
// calling.cpp -- defining, prototyping, and calling a function
#include <iostream>
void simple();    // function prototype函数原型
int main()
{
    using namespace std;
    cout << "main() will call the simple() function:\n";
    simple();     // function call函数调用
    cout << "main() is finished with the simple() function.\n";
    return 0;
}
// function definition函数定义
void simple()
{
    using namespace std;
    cout << "I'm but a simple function.\n";
}
```



1.1 定义函数

✓函数分为两类：没有返回值的函数和有返回值得函数

```
void functionName(parameterList)
{
    statement(s)
    return;  // optional
}
```

```
void cheers(int n) // no return value
{
    for (int i = 0; i < 0; i++)
          std::cout << "Cheers!";
    std::cout << std::endl;
}
```

```
typeName  functionName(parameterList)
{
    statement(s)
    return value;  // value is type cast to type typeName
}
```

```
int bigger(int a, int b)
{
    if(a>b)
       return a;
    else
       return b;
}
```

1.1 定义函数

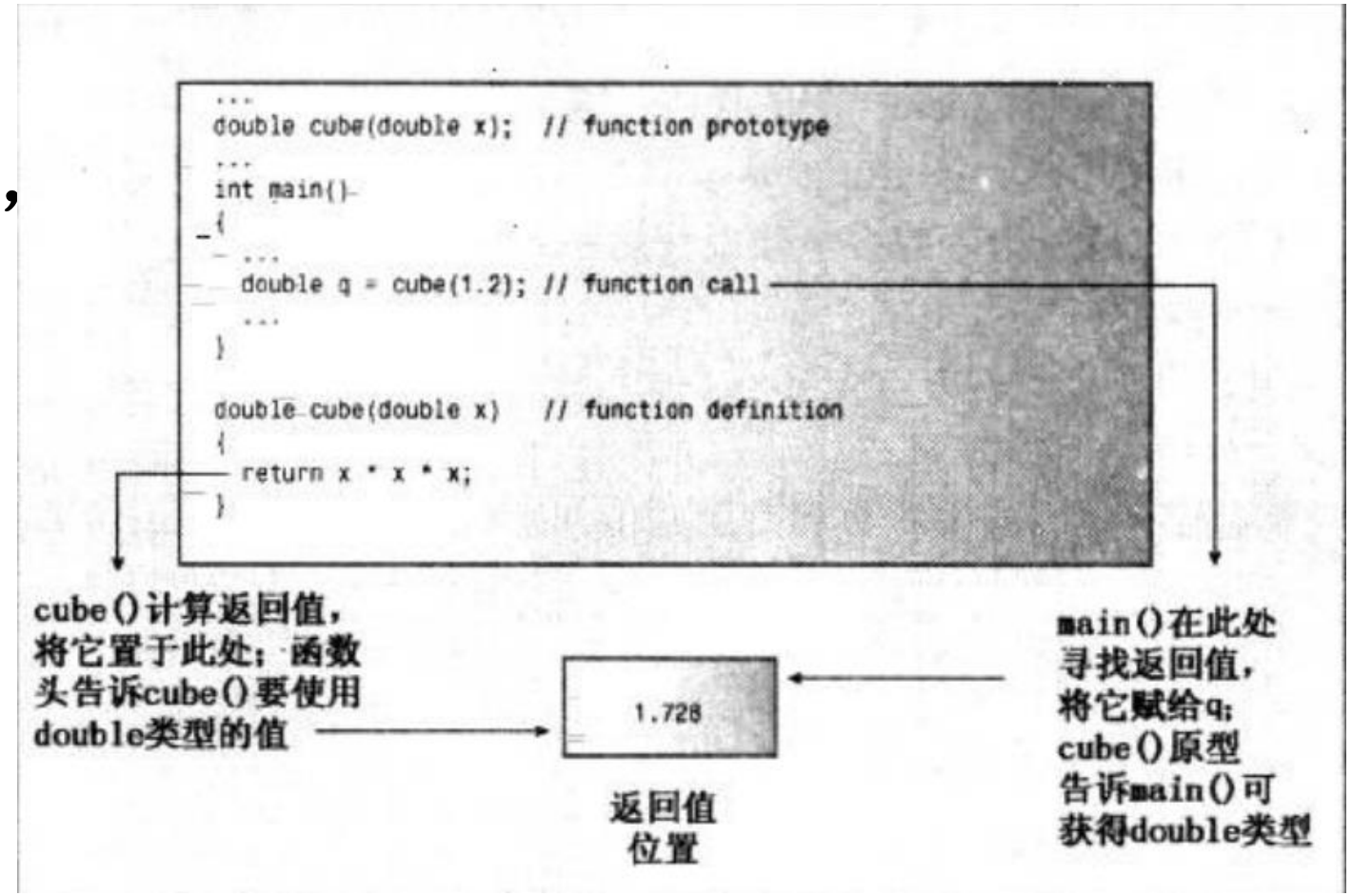
```
typeName  functionName(parameterList)
{
   statement(s)
   return value;  // value is type cast to type typeName
}
```

- ✓函数分为两类：没有返回值的函数和有返回值得函数
- ✓对于有返回值得函数，必须使用返回语句，以便将值返回给调用函数
- ✓返回值本身可以是常量、变量，也可以是表达式，只是其结果的类型必须为typeName类型，或可以被（强制）转换为typeName，然后函数将最终的值返回给调用函数

1.1 定义函数

```
typeName  functionName(parameterList)
{
   statement(s)
   return value;  // value is type cast to type typeName
}
```

- ✓ C++对返回值得类型有一定的限制：不能是数组，但可以是其他任何类型-整数、浮点数、指针，甚至可以是结构和对象。可以将数组作为结构或对象组成部分来返回



典型的返回值机制

```
// protos.cpp -- using prototypes and function calls
#include <iostream>
void cheers(int);        // prototype: no return value
double cube(double x);   // prototype: returns a double
int main()
{
    using namespace std;
    cheers(5);      // function call
    cout << "Give me a number: ";
    double side;
    cin >> side;
    double volume = cube(side);     // function call
    cout << "A " << side << "-foot cube has a volume of ";
    cout << volume << " cubic feet.\n";
    cheers(cube(2));    // prototype protection at work
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

```
void cheers(int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << endl;
}
```

```
double cube(double x)
{
    return x * x * x;
}
```

1.2 函数原型

✓原型描述了函数到编译器的接口，它将函数返回值得类型（如果有的话）以及参数的类型和数量告诉编译器

```
void cheers(int);        // prototype: no return value
double cube(double x);   // prototype: returns a double
```

✓C++允许将一个程序放在多个文件中，单独编译这些文件，然后再将它们组合起来。在这种情况下，编译器在编译main()时，可能无权访问函数代码

✓C++的编程风格是将main()放在最前面，提供程序的整体结构

1.2 函数原型

✓函数原型是一条语句，必须以分号结束。获得原型最简单的方法是，复制函数定义中的函数头，并添加分号

```
void cheers(int);        // prototype: no return value
double cube(double x);   // prototype: returns a double
```

✓通常，在原型的参数列表中，可以包括变量名，也可以不包括

✓原型中的变量名相当于占位符，不必与函数定义中的变量名相同

```
void say_hi();        // prototype: 函数没有参数
void say_hello(void); // prototype: 函数没有参数
```

1.2 函数原型

✓原型的功能

- ❖编译器正确处理函数返回值
- ❖编译器检查使用的参数数目是否正确
- ❖编译器检查使用的参数类型是否正确；如果不正确，则转换为正确的类型（如果可能）

✓通常，原型自动将被传递的参数强制转换为期望的类型

1.2 函数原型

- ✓自动类型转换并不能避免所有可能的错误。当较大的类型被自动转换为较小类型时，有些编译器将发出警告，指出这可能会丢失数据
- ✓原型不会将整数转换为结构或指针
- ✓在编译阶段进行的原型化被称为静态类型检查（static type checking）。静态类型检查可以捕获许多在运行阶段非常难以捕获的错误

目录

- 函数参数和按值传递
  - 参数传递
  - 多个参数
  - 数组名做参数

2.1 参数传递

✓C++通常按值传递参数，将数值参数传递给函数，后者将其赋给一个新的变量

✓用于接收传递值的变量被称为形参（parameter）

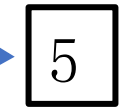
✓传递给函数的值被称为实参（argument）

```
void cheers(int);
cheers(5);
```

2.1 参数传递

```
...
double cube(double x);
int main()
{
      ...
      double side = 5;
      double volume = cube(side);
      ...
}
double cube(double x)
{
      return x * x * x;
}
```

创建side变量，将5赋给它



原始值

side

将值5传递给cube()函数

创建变量x，将传递的值5赋给它

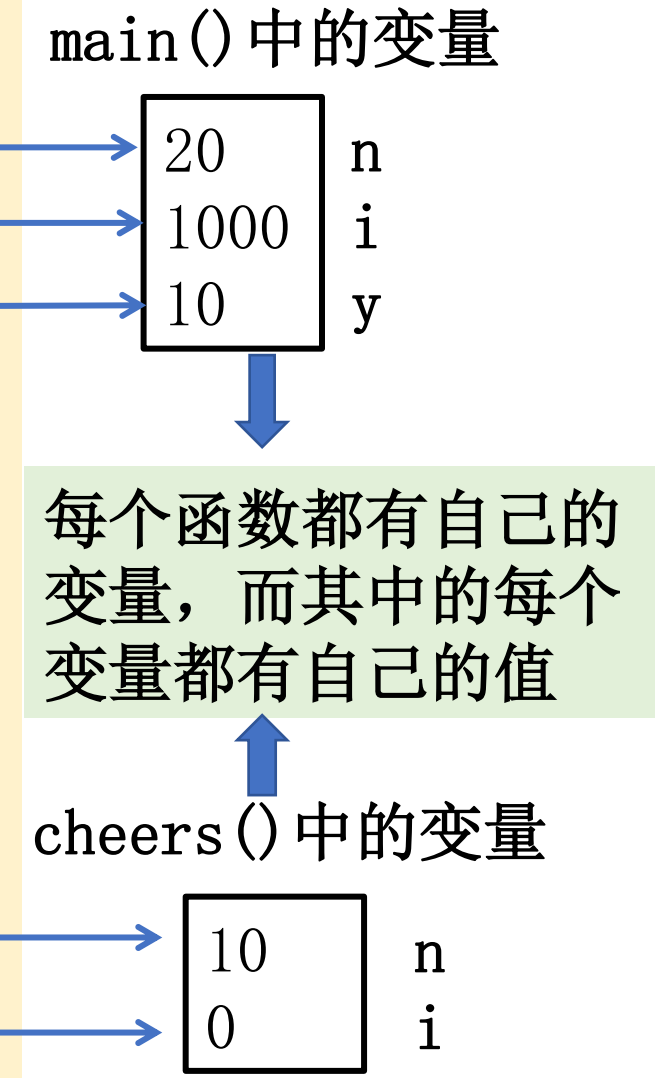
复制的值

x

按值传递

```
...
void cheers(int n);
int main()
{
        int n = 20;
        int i = 1000;
        int y = 10;
        ...
        cheers(y);  // function call
        ...
}
void cheers(int n)
{
        using namespace std;
        for (int i = 0; i < n; i++)
                cout << "Cheers! ";
        cout << endl;
}
```

局部变量



在函数被调用时，计算机将为这些变量分配内存；在函数结束时，计算机释放这些变量使用的内存。这样的变量被称为局部变量，它们被限制在函数中，也被称为自动变量，在程序执行过程中自动被分配和释放

2.2 多个参数

✓函数可以有多个参数，在调用时，只需使用逗号，将这些参数分开：

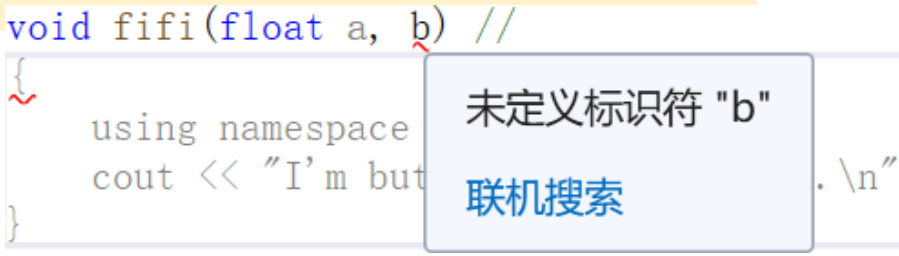
```
n_chars('R', 25);
```

✓在函数定义时，也在函数头中使用逗号分隔参数声明列表：

```
void n_chars(char c,int n) {}
```

✓如果函数的两个参数的类型相同，则必须分别指定参数的类型

```
void fifi(float a, float b) // declare each variable separately
void fifi(float a, b) // ×not acceptable
```



2.2 多个参数

✓和一个参数的情况一样，原型中的变量名不必与定义中的变量名相同，而且可以省略：

```
void n_chars(char c, int n); // prototype, style 1
void n_chars(char, int);     // prototype, style 2
```

```
// twoarg.cpp -- a function with 2 arguments
#include <iostream>
using namespace std;
void n_chars(char, int);
int main()
{
    int times;
    char ch;
    cout << "Enter a character: ";
    cin >> ch;
    while (ch != 'q') // q to quit
    {
        cout << "Enter an integer: ";
        cin >> times;
        n_chars(ch, times);  // function with two arguments
        cout << "\nEnter another character or press the"
             " q-key to quit: ";
        cin >> ch;
    }
    cout << "The value of times is " << times << ".\n";
    cout << "Bye\n"; return 0;
}
```

Microsoft Visual Studio 调试控制台

```
Enter a character: W
Enter an integer: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Enter another character or press the q-key to quit: a
Enter an integer: 20
aaaaaaaaaaaaaaaaaaaa
Enter another character or press the q-key to quit: q
The value of times is 20.
Bye
```

```
void n_chars(char c, int n) // displays c n times
{
    while (n-- > 0)  // continue until n reaches 0
        cout << c;
}
```

- 当用户对程序提示做出响应时，必须在每行的最后按Enter键，以生成换行符
- cin.get(ch)或ch=cin.get()读取所有的输入字符，包括空格和换行符
- cin>>ch 方法可以跳过空格和换行符

2.2 多个参数

需求：采用某种纸牌游戏的形式来发行彩票，让参与者从卡片中选择一定数目的选项。例如，从51个数字中选取6个数。

如果参与者选择的数字与这6个完全相同，将赢得大量奖金。我们用函数计算中奖的几率。

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}$$

Part1

```
// lotto.cpp -- probability of winning
#include <iostream>
// Note: some implementations require double instead of long double
long double probability(unsigned numbers, unsigned picks);
int main()
{
    using namespace std;
    double total, choices;
    cout << "Enter the total number of choices on the game card and\n"
            "the number of picks allowed:\n";
    while ((cin >> total >> choices) && choices <= total)
    {
        cout << "You have one chance in ";
        cout << probability(total, choices);      // compute the odds
        cout << " of winning.\n";
        cout << "Next two numbers (q to quit): ";
    }
    cout << "bye\n";
    return 0;
}
```

Part2

```
// the following function calculates the probability of picking picks
// numbers correctly from numbers choices
long double probability(unsigned numbers, unsigned picks)
{
    long double result = 1.0;  // here come some local variables
    long double n;
    unsigned p;

    for (n = numbers, p = picks; p > 0; n--, p--)
        result = result * n / p;
    return result;
}
```

```
Enter the total number of choices on the game card and
the number of picks allowed:
49 6
You have one chance in 1.39838e+07 of winning.
Next two numbers (q to quit): 51 6
You have one chance in 1.80095e+07 of winning.
Next two numbers (q to quit): 38 6
You have one chance in 2.76068e+06 of winning.
Next two numbers (q to quit): q
bye
```

- 由于选择的数目和总数目都为正，因此该程序这些变量声明为unsigned int类型（简称unsigned）
- 将若干整数相乘可以得到相当大的结果，因此将该函数的返回值声明为long double类型
- 函数中使用了两种局部变量：形参从调用probability()函数获得值，而其他局部变量从函数中获得自己的值

2.3 数组名做参数

```
// arrfun1.cpp -- functions with an array argument
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);          // prototype
int main()
{
    using namespace std;
    int cookies[ArSize] = { 1,2,4,8,16,32,64,128 };
    // some systems require preceding int with static to
    // enable array initialization

    int sum = sum_arr(cookies, ArSize);
    cout << "Total cookies eaten: " << sum << "\n";
    return 0;
}
```

```
Total cookies eaten: 255
```

```
//return the sum of an integer array
int sum_arr(int arr[], int n)
{
    int total = 0;

    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

- int arr[]，方括号指出arr是一个数组，而方括号为空则表明，可以将任何长度的数组传递给该函数
- arr实际上并不是数组，而是一个指针（后续深入讲解）

目录

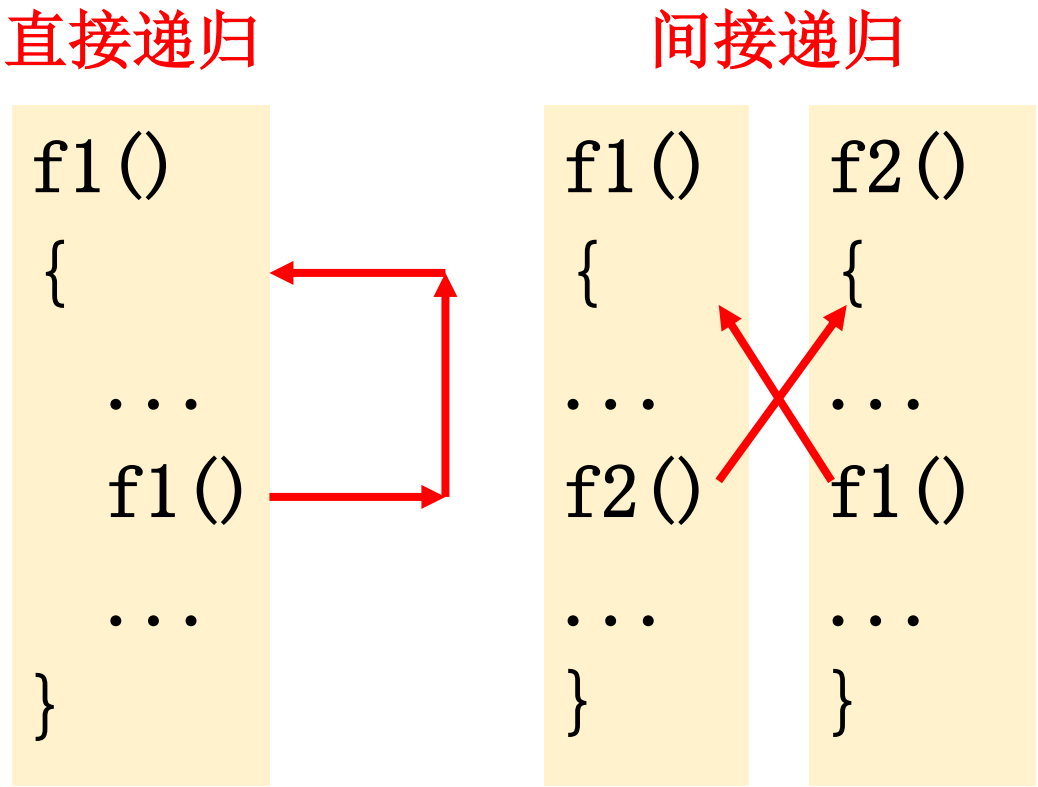
- 递归
  - 包含一个递归调用的递归
  - 包含多个递归调用的递归

3.1 包含一个递归调用的递归

✓C++函数可以调用自己，这种功能被称为递归（与C语言不同，C++不允许main()调用自己）

✓通常将递归调用放在if语句中

```
void recurs(argumentlist)
{
  statements1
  if(test)
     recurs(arguments)
  statements2
}
```



借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k'); }
```

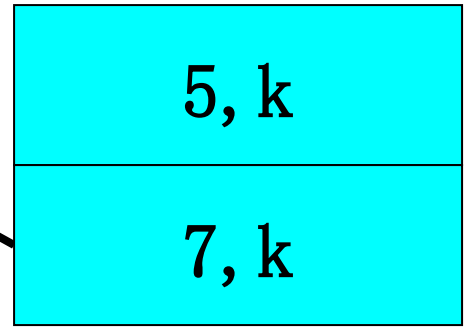
借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

7,k

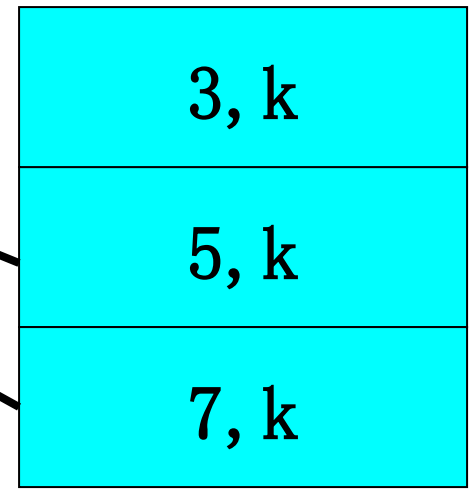
借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

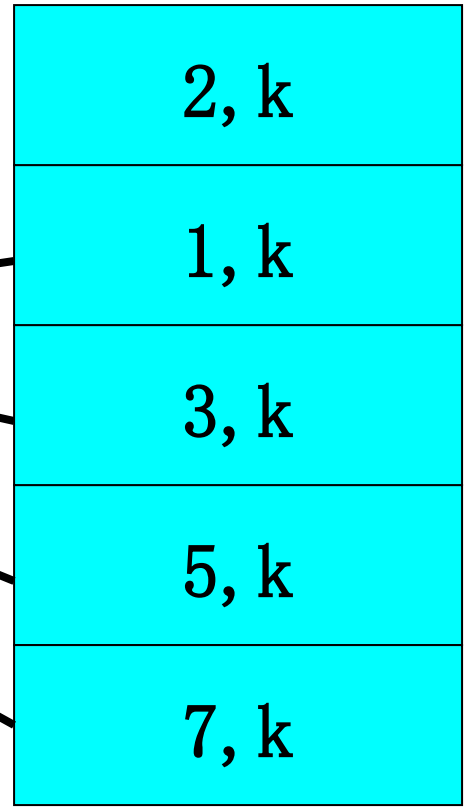


借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

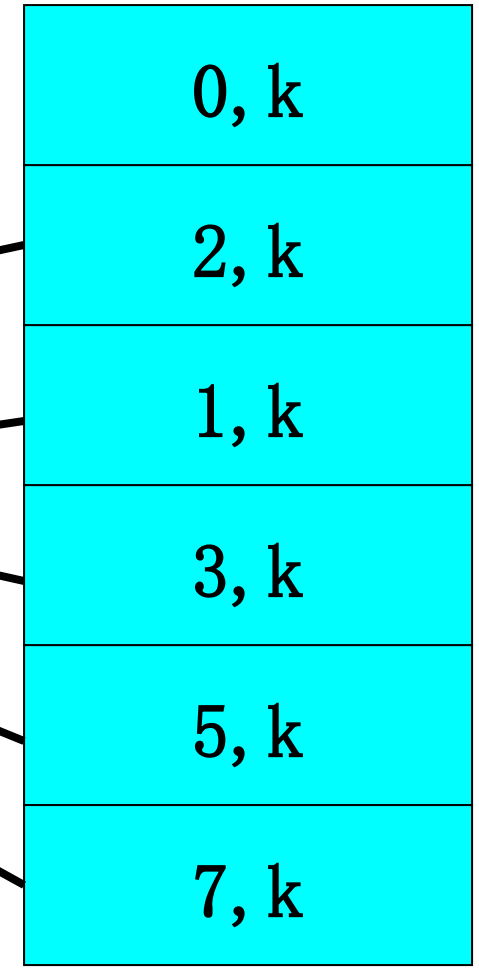
借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



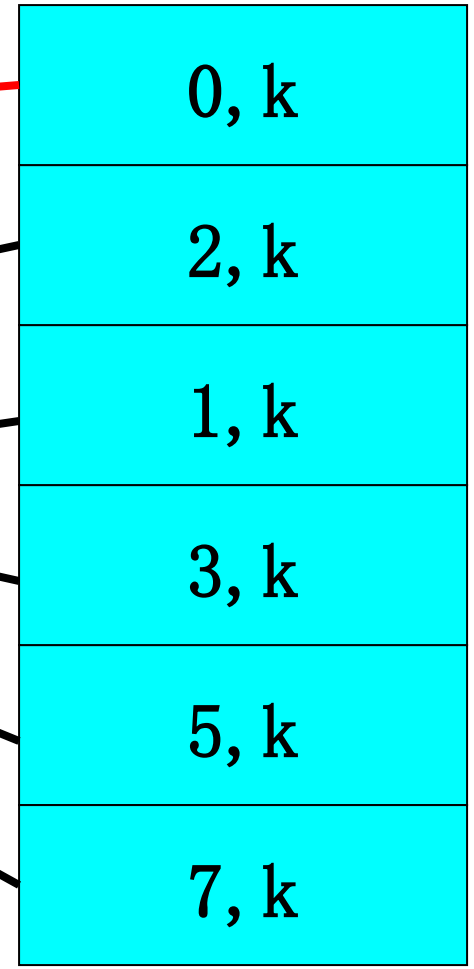
借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)                    ①
        return;
    if (n>1)
        f(n-2,ch);               ②
    else
        f(n+1, ch);
    cout << char(ch+n);          ③
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

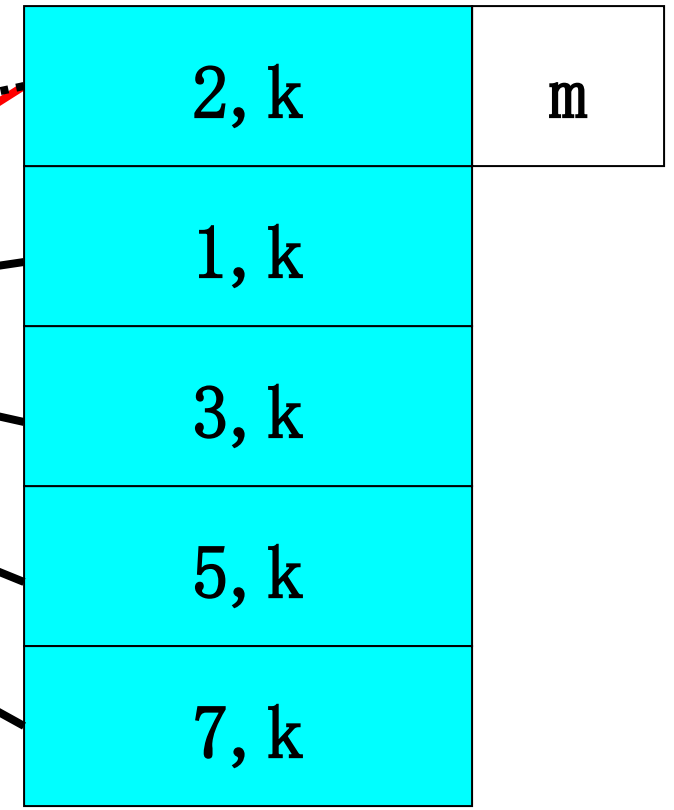
```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

黑虚:上次保存现场位置
红实:本次恢复现场位置

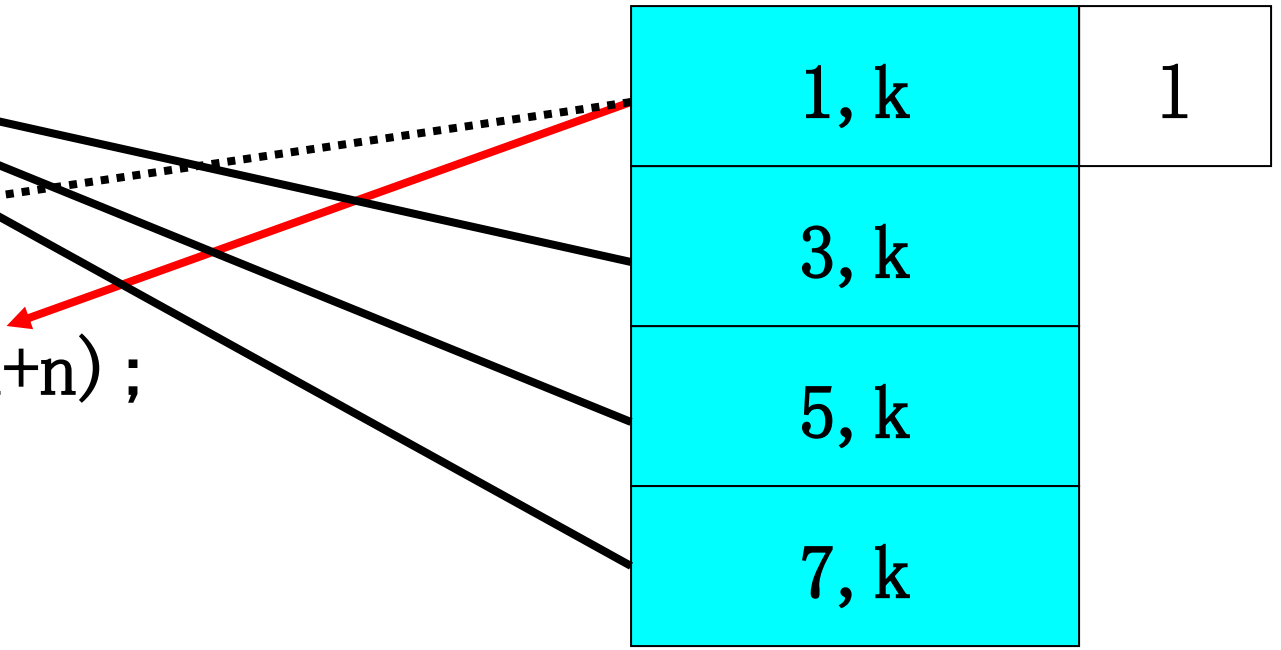
```
void f(int n, char ch)
{   if (n==0)                    ①
        return;
    if (n>1)
        f(n-2,ch);               ②
    else
        f(n+1, ch);
    cout << char(ch+n);          ③
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

黑虚:上次保存现场位置
红实:本次恢复现场位置

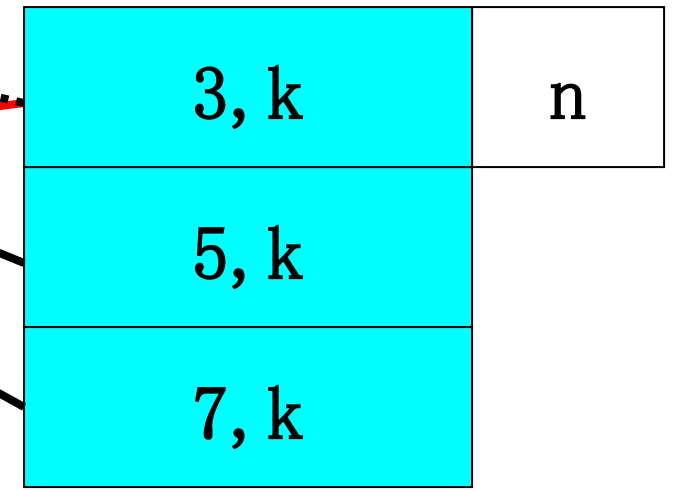
```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

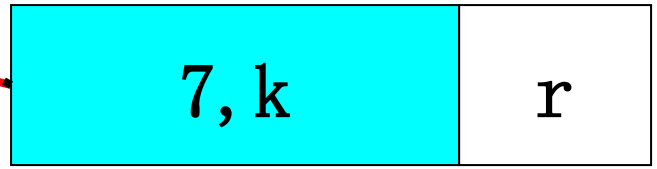
黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```

借助栈来记录调用的层次

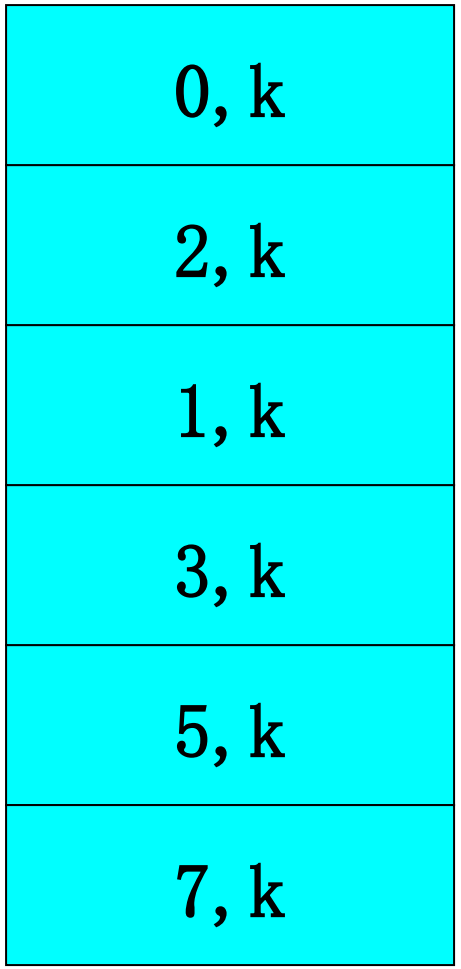
黑虚:上次保存现场位置
红实:本次恢复现场位置

```
void f(int n, char ch)
{   if (n==0)
①       return;
    if (n>1)
        f(n-2,ch);
②   else
        f(n+1, ch);
③   cout << char(ch+n);
}
int main()
{   f(7, 'k');
}
```



借助栈来记录调用的层次

```
void f(int n, char ch)
{   if (n==0)                    ①
        return;
    if (n>1)
        f(n-2,ch);               ②
    else
        f(n+1, ch);
    cout << char(ch+n);          ③
}
int main()                       mlnpr
{   f(7, 'k');
}
```



3.1 包含一个递归调用的递归

```
// recur.cpp -- using recursion
#include <iostream>
void countdown(int n);
int main()
{
    countdown(4);   // call the recursive function
    return 0;
}
void countdown(int n)
{
    using namespace std;
    cout << "Counting down ... " << n << endl;
    if (n > 0)
        countdown(n - 1);     // function calls itself
    cout << n << ": Kaboom!\n";
}
```

```
Counting down ... 4
Counting down ... 3
Counting down ... 2
Counting down ... 1
Counting down ... 0
0: Kaboom!
1: Kaboom!
2: Kaboom!
3: Kaboom!
4: Kaboom!
```

3.1 包含一个递归调用的递归

```
// recur.cpp -- using recursion
#include <iostream>
void countdown(int n);

int main()
{
    countdown(4);   // call the recursive function
    return 0;
}
void countdown(int n)
{
    using namespace std;
    cout << "Counting down ... " << n << "(n at" <<&n<<")"<<endl;
    if (n > 0)
        countdown(n - 1);     // function calls itself
    cout << n << ": Kaboom!" << "(n at" << &n << ")" << endl;;
}
```

```
Counting down ... 4(n at000000E0324FF520)
Counting down ... 3(n at000000E0324FF420)
Counting down ... 2(n at000000E0324FF320)
Counting down ... 1(n at000000E0324FF220)
Counting down ... 0(n at000000E0324FF120)
0: Kaboom!(n at000000E0324FF120)
1: Kaboom!(n at000000E0324FF220)
2: Kaboom!(n at000000E0324FF320)
3: Kaboom!(n at000000E0324FF420)
4: Kaboom!(n at000000E0324FF520)
```

- 每个递归调用都创建自己的一套变量，当程序到达第5次调用时，有5个独立的n变量，每个变量的值都不同，存储的内存单元均不相同
- Counting down阶段和Kaboom阶段的相同层级，n的地址相同

3.1 包含一个递归调用的递归

- ✓ 递归的求解过程
  - ❖ 回推：到一个确定值为止(递归不再调用)
  - ❖ 递推：根据回推得到的确定值求出要求的解

3.1 包含一个递归调用的递归

✓ 递归的求解过程

例：求解5个学生的年龄分别是多少

题目描述：共5个学生

问第5个学生几岁，答：我比第4个大2岁；

问第4个学生几岁，答：我比第3个大2岁；

问第3个学生几岁，答：我比第2个大2岁；

问第2个学生几岁，答：我比第1个大2岁；

问第1个学生几岁，答：我10岁；

3.1 包含一个递归调用的递归

✓ 递归的求解过程

例：求解5个学生的年龄分别是多少

回推

```
age(5) = age(4) + 2;
age(4) = age(3) + 2;
age(3) = age(2) + 2;
age(2) = age(1) + 2;
age(1) = 10;
```

递推

3.1 包含一个递归调用的递归

- ✓ 如何写递归函数
  - ❖确定递归何时终止
  - ❖假设第n-1次调用已求得确定值，确定第n次调用和第n-1次调用之间存在的逻辑关系

```
//求解5个学生的年龄分别是多少
#include <iostream>
using namespace std;
int age(int n);
int main()
{
	int n;
	cout << "Enter the number of the person: ";
	cin >> n;
	for (int i = 1; i <= n; i++)   //输出每个人的年龄
	{
		cout << "The age of the " << i << "th person is " << age(i) << endl;
	}
	return 0;
}
int age(int n)
{
	if (n == 1)
		return 10;
	else
		return age(n - 1) + 2;
}
```

```
Enter the number of the person: 5
The age of the 1th person is 10
The age of the 2th person is 12
The age of the 3th person is 14
The age of the 4th person is 16
The age of the 5th person is 18
```

3.2 包含多个递归调用的递归

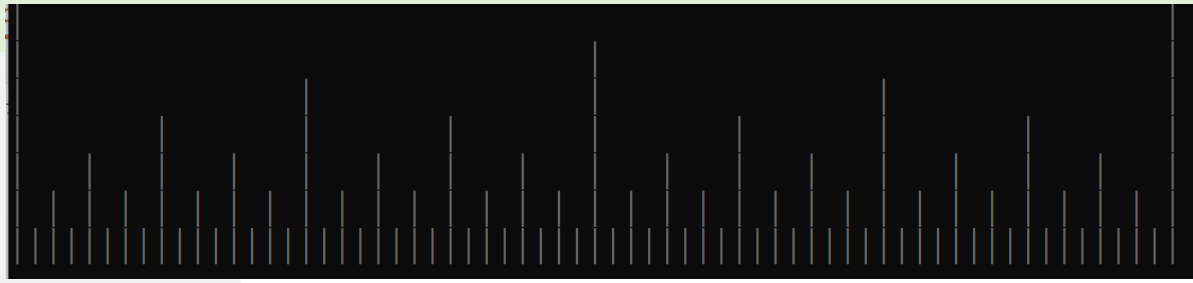
✓在需要将一项工作不断分为两项较小的、类似的工作时，递归非常有用

✓递归方法又是被称为分而治之策略（divide-and-conquer strategy）

✓需求：使用这种方法绘制标尺

- ❖标出两端，找出中点并将其标出
- ❖然后将同样的操作用于标尺的左半部分和右半部分

```
// ruler.cpp -- using recursion to subdivide a ruler
#include <iostream>
const int Len = 66;
const int Divs = 6;
void subdivide(char ar[], int low, int high, int level);
int main()
{   char ruler[Len];
    int i;
    for (i = 1; i < Len - 2; i++)
        ruler[i] = ' ';
    ruler[Len - 1] = '\0';
    int max = Len - 2;
    int min = 0;
    ruler[min] = ruler[max] = '|';
    std::cout << ruler << std::endl;
    for (i = 1; i <= Divs; i++)
    {
        subdivide(ruler, min, max, i);
        std::cout << ruler << std::endl;
        for (int j = 1; j < Len - 2; j++)
            ruler[j] = ' '; // reset to blank ruler
    }
    return 0;
}
void subdivide(char ar[], int low, int high, int level)
{
    if (level == 0)
        return;
    int mid = (high + low) / 2;
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}
```



3.2 包含多个递归调用的递归

- ✓ 需求：使用这种方法绘制标尺
  - ❖标出两端，找出中点并将其标出
  - ❖然后将同样的操作用于标尺的左半部分和右半部分
    - ❑subdivide()函数使用变量level来控制递归层
    - ❑subdivide()调用两次自己，一次针对左半部分，另一次针对右半部分
    - ❑调用次数将呈现几何级增长
    - ❑如果递归层较少，这是一种精致而简单的选择；如果递归层很多，这种递归方式将是一种糟糕的选择

目录

- 内联函数

4.1 内联函数

✓常规函数调用时，程序需要跳到函数的地址，并在函数结束时返回

✓执行到函数调用指令时，程序将在函数调用后立即存储该指令的内存地址，并将函数参数赋值到堆栈（为此保留的内存块），跳到标记函数起点的内存单元，执行函数代码（也许还需将返回值存放在寄存器中），然后跳回到地址被保存的指令处

✓来回跳跃并记录跳跃位置意味着以前使用函数时，需要一定开销

函数调用：保存现场→函数返回：恢复现场

C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）

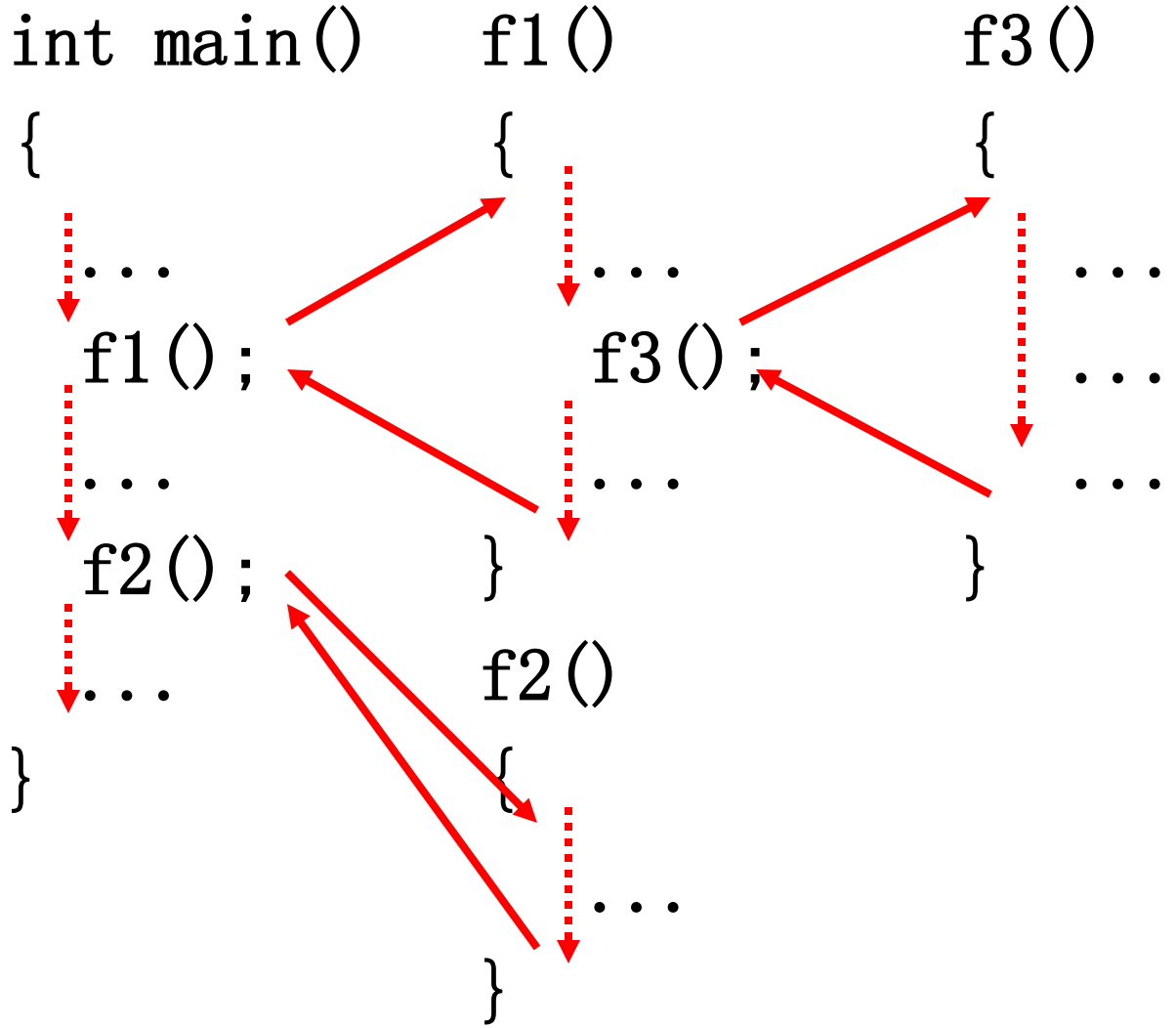
```
int main()
{
  ...
  f1();
  ...
  f2();
  ...
}
```

```
f1()
{
   ...
   f3();
   ...
}
f2()
{
   ...
}
```

```
f3()
{
   ...
   ...
   ...
}
```

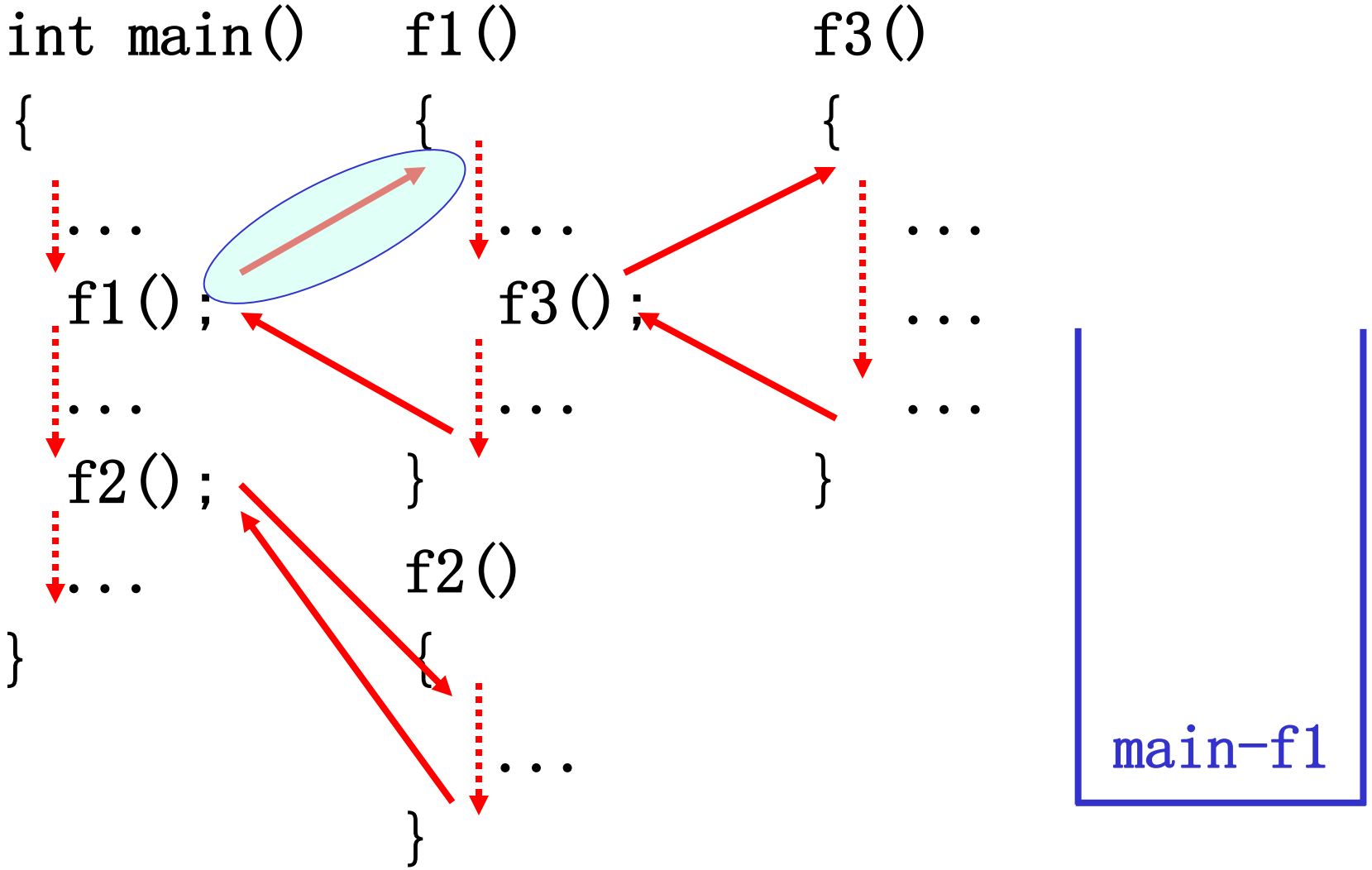
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



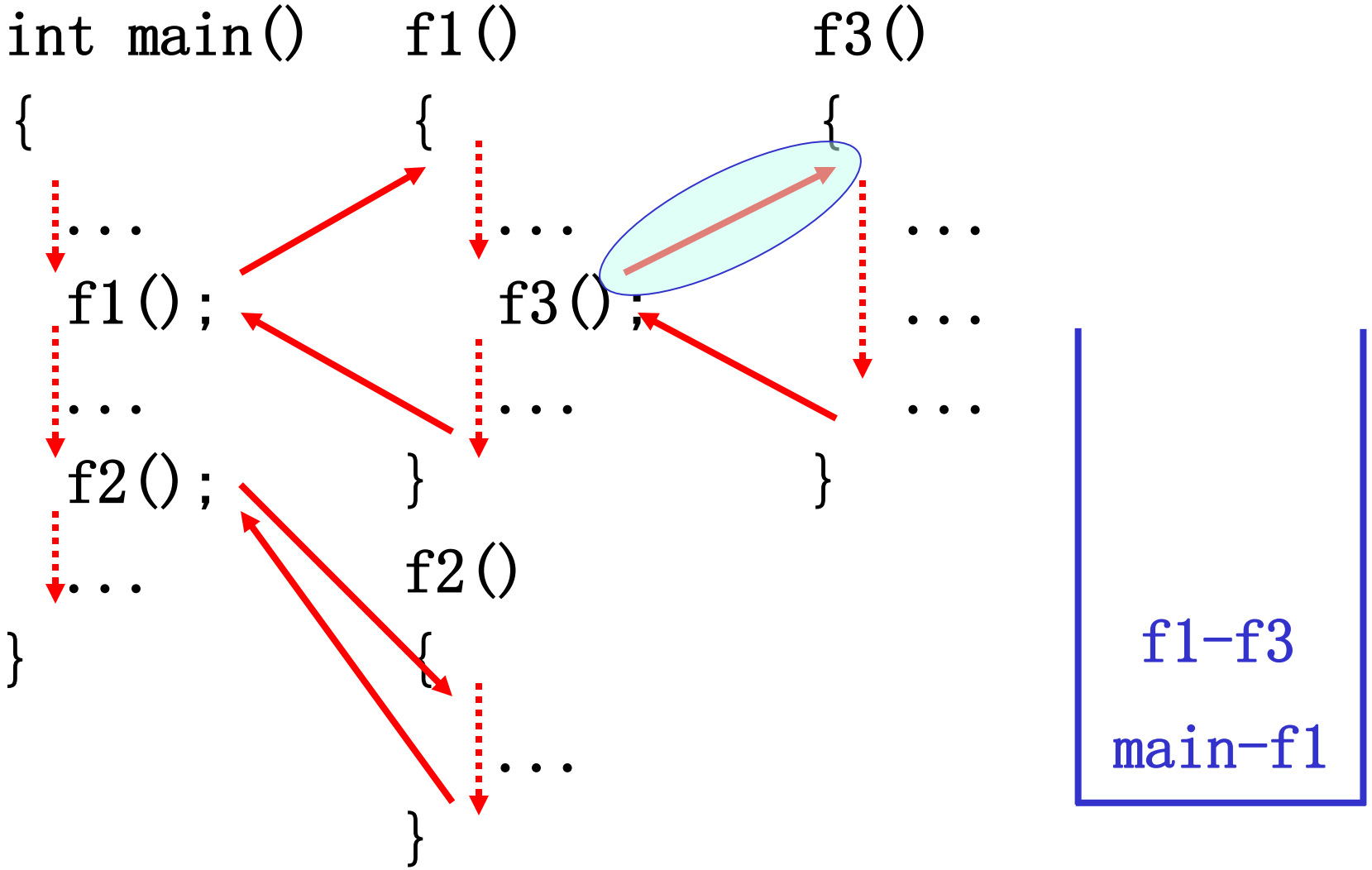
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



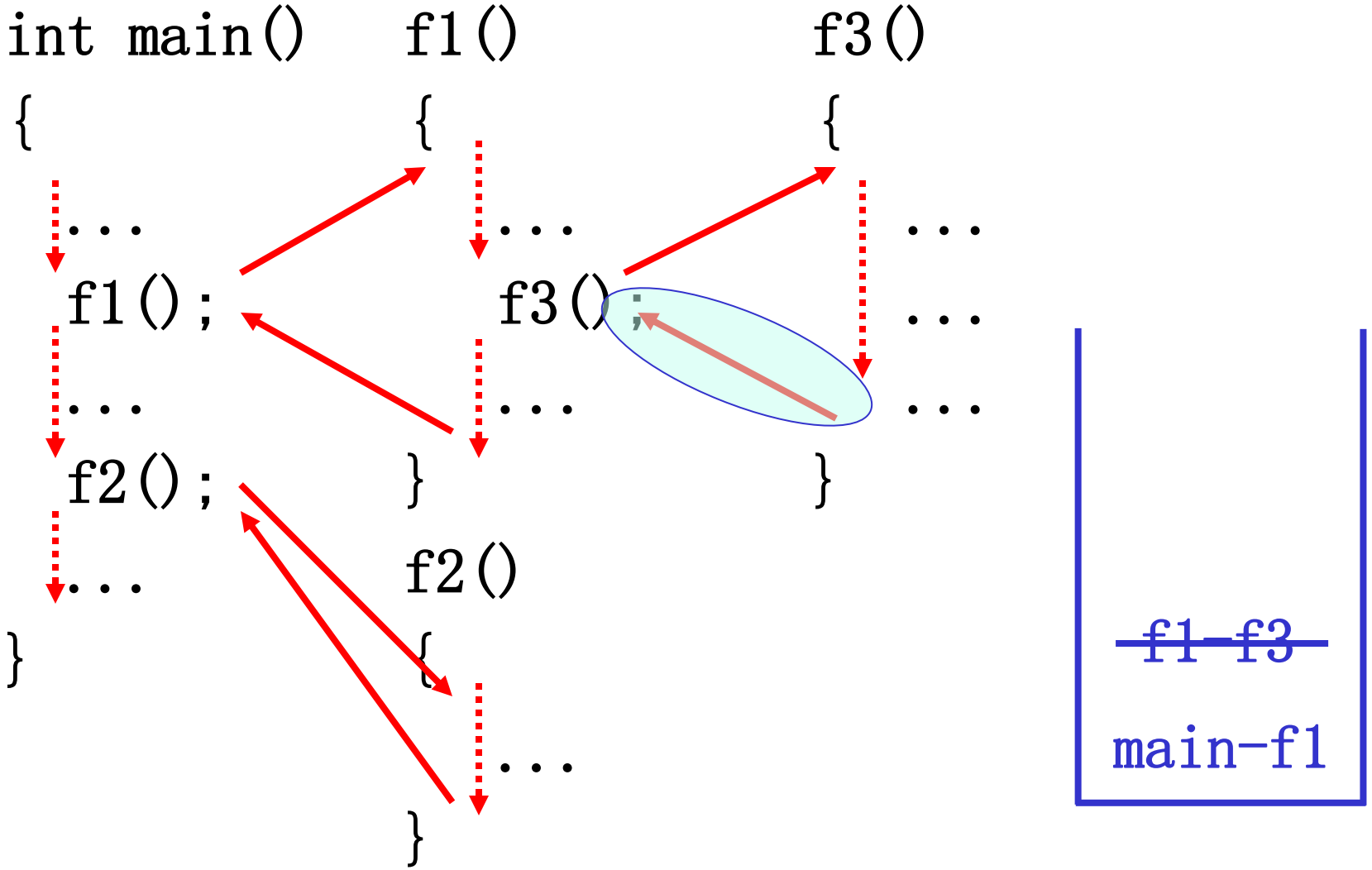
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



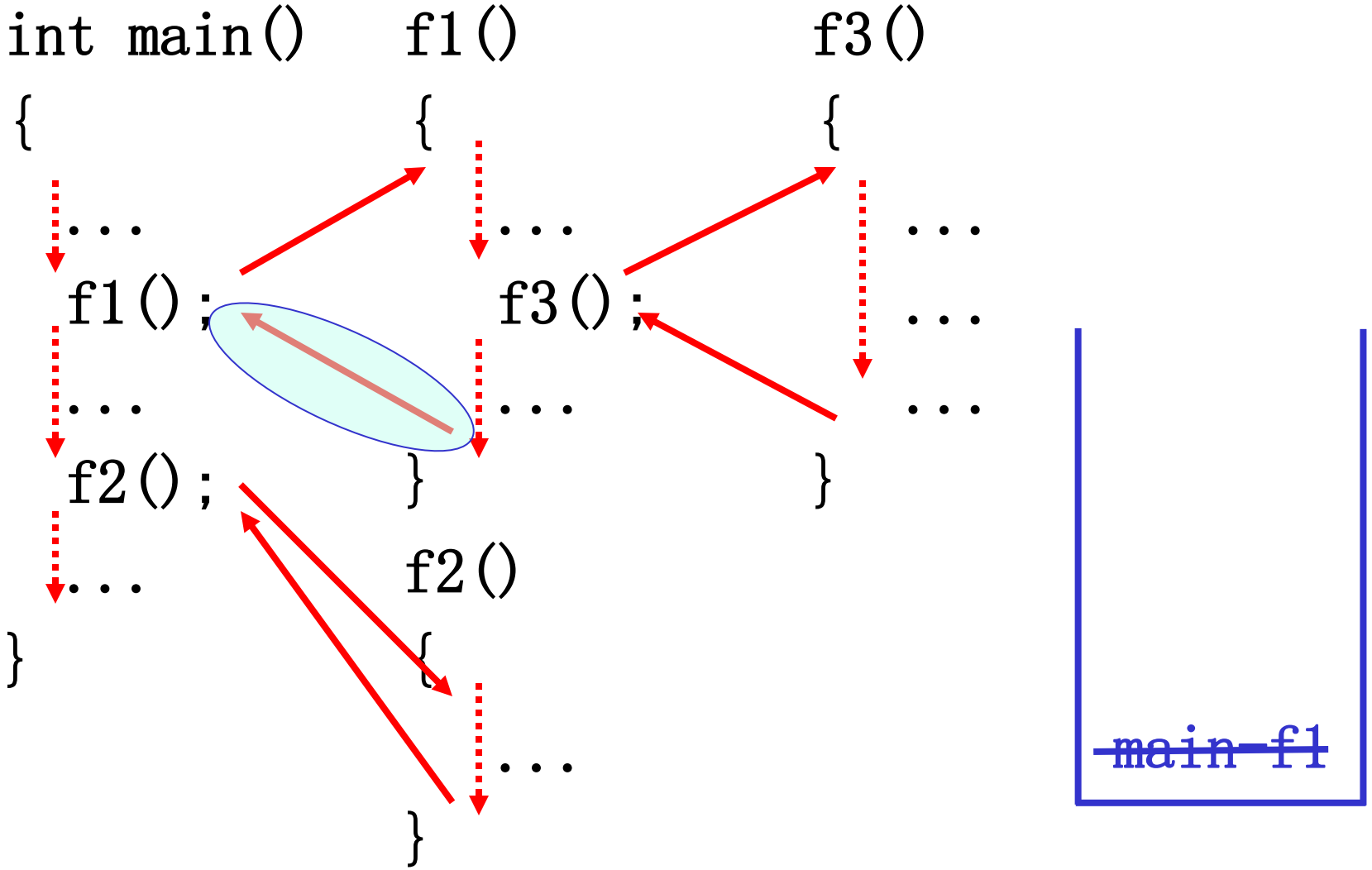
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



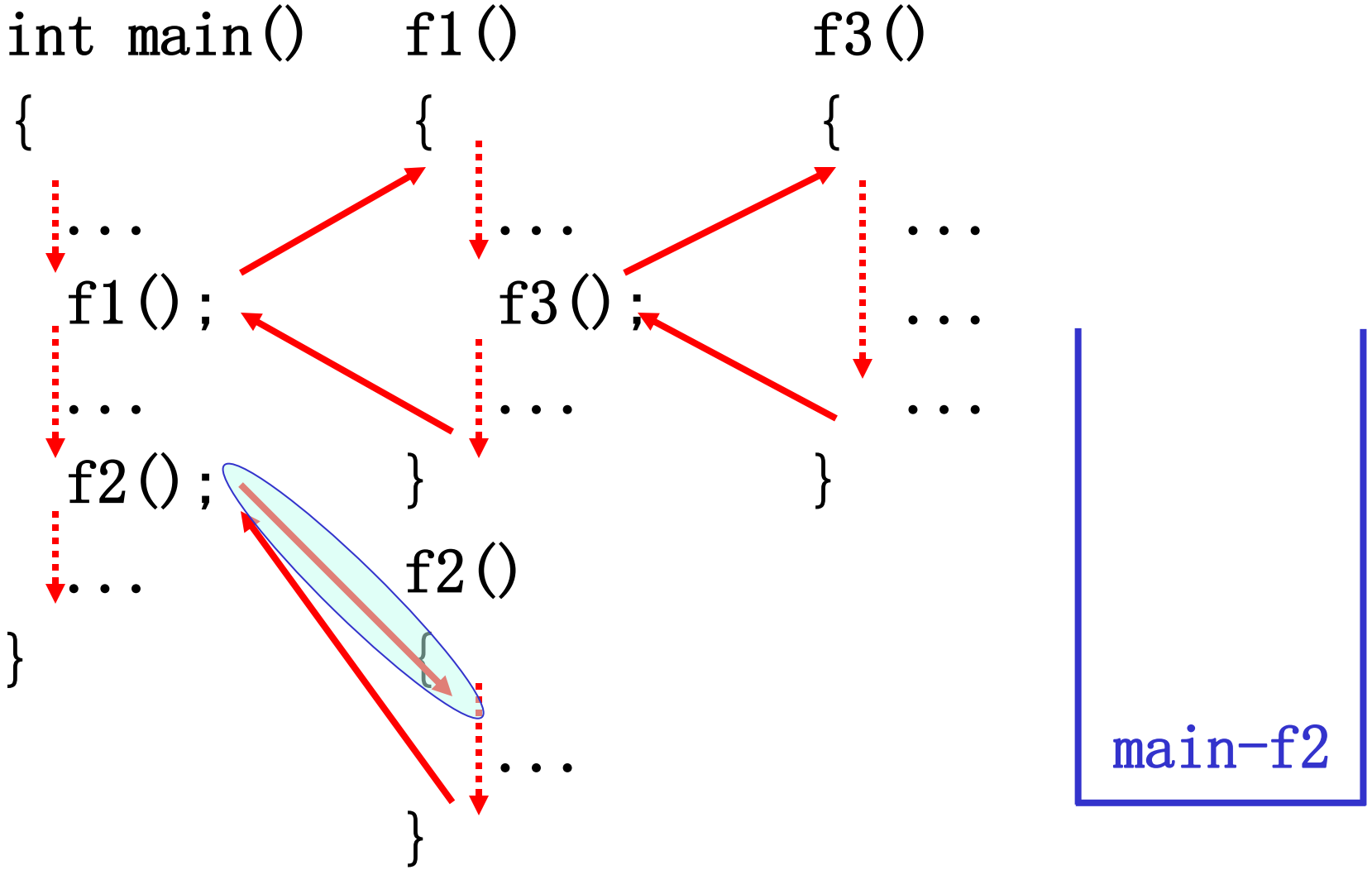
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



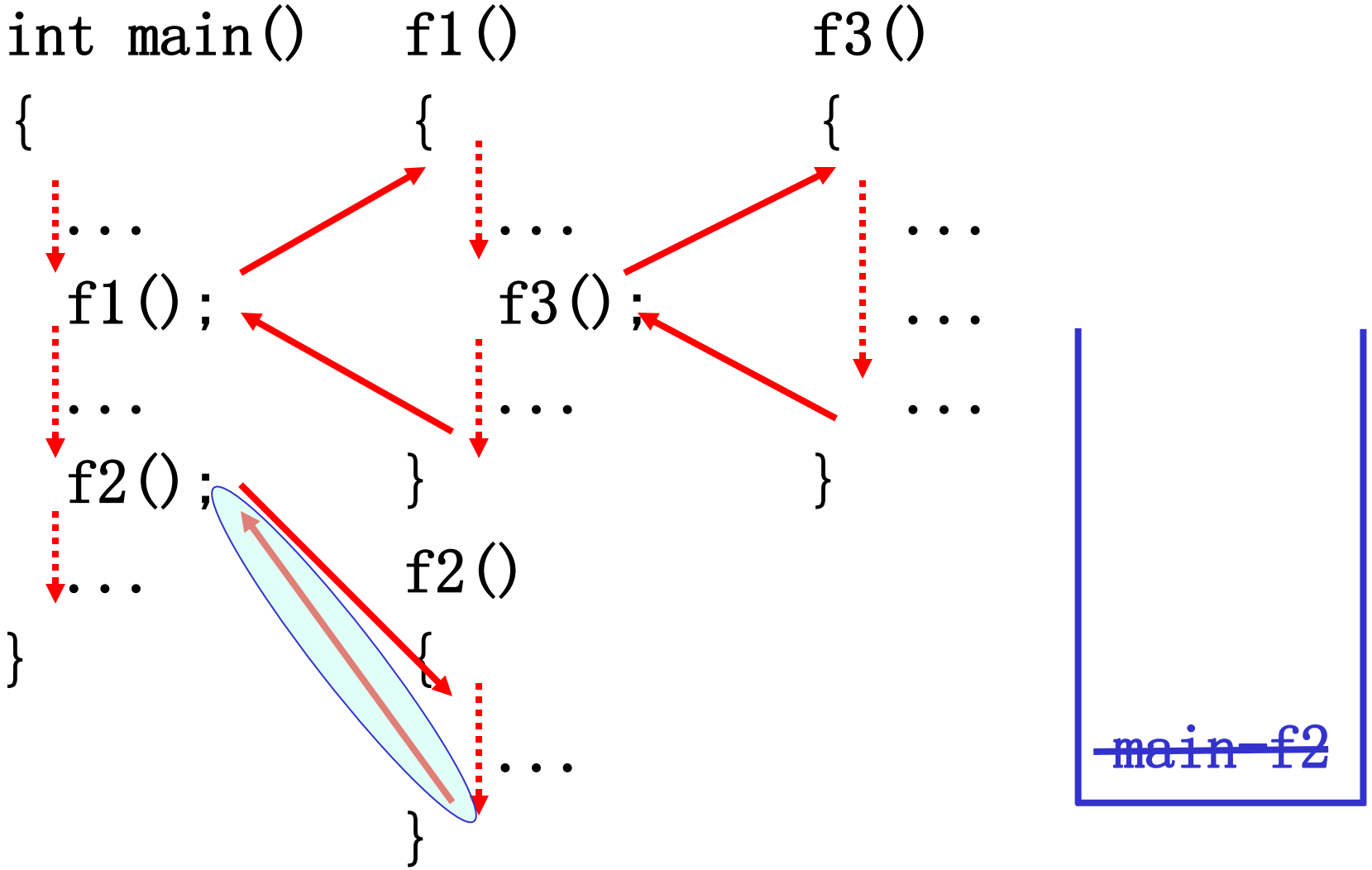
C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



C++常规函数的执行过程(栈方式理解)

函数调用：保存现场（进栈）→函数返回：恢复现场（出栈）



4.1 内联函数

✓内联函数的编译代码与其他程序代码“内联起来”

✓编译器将使用相应的函数代码替换函数调用

✓对于内联代码，程序无需跳到另一个位置处执行代码，再跳回来

✓内联函数的运行速度比常规函数稍快，但代价是需要占用更多内存

```
...
int main()
{
        ...
        hubba(2);
        ...
        hubba(4);
        ...
        hubba(10);
        ...
}
```

```
void hubba(int n)
{
    for (int i = 0; i < n; i++)
        std::cout << "hubba ";
    cout << '\n';
}
```

常规函数将程序流程转到独立的函数

```
...
int main()
{
        ...
        {   n = 2;
            for (int i = 0; i < n; i++)
                std::cout << "hubba ";
            cout << '\n';
        }
        ...
        {   n = 4;
            for (int i = 0; i < n; i++)
                std::cout << "hubba ";
            cout << '\n';
        }
        ...
        {   n = 10;
            for (int i = 0; i < n; i++)
                std::cout << "hubba ";
            cout << '\n';
        }
        ...
}
```

内联函数用内联代码替换函数调用

4.1 内联函数

✓定义内联函数

- ❖ 在函数声明前加上关键字inline
- ❖ 在函数定义前加上关键字inline

✓通常做法是，将整个定义（函数头和所有函数代码）放在本应提供原型的地方

```
// an inline function definition
inline double square(double x) { return x * x; }
int main()
 {}
```

4.1 内联函数

```
// inline.cpp -- using an inline function
#include <iostream>
// an inline function definition
inline double square(double x) { return x * x; }
int main()
{
    using namespace std;
    double a, b;
    double c = 13.0;
    a = square(5.0);
    b = square(4.5 + 7.5);    // can pass expressions
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c squared = " << square(c++) << "\n";
    cout << "Now c = " << c << "\n";

    return 0;
}
```

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

4.2 选择内联函数

✓如果执行函数代码的时间比处理函数调用的时间长，则节省的时间将只占整个过程的很小部分

❖如果代码执行时间很短，则内联调用就可以节省非内联调用使用的大部分时间

❖另一方面，由于这个过程相当快，因此尽管节省了该过程的大部分时间，但节省的时间并不大，除非该函数经常被调用

4.2 选择内联函数

注意：

✓尽管程序员请求将函数作为内联函数，但编译器并不一定会满足，如：

- ❖编译器认为该函数过大（如：函数定义占用太多行）
- ❖编译器发现函数调用了自己（内联函数不能递归）

✓有些编译器没有启用或实现内联函数特性

4.2 选择内联函数

✓内联函数与宏

- ❖ inline工具是C++新增特性（通过参数传递实现）
- ❖ C语言使用预处理器语句#define来提供宏（通过文本替换实现）

```
#define SQUARE(X) X*X
a = SQUARE(5.0); //is replaced by a=5.0*5.0 √
b = SQUARE(4.5+7.5); //is replaced by b=4.5+7.5*4.5+7.5 ×
c = SQUARE(c++); //is replaced by c=c++*c++ ×
```

- ❖如果使用C语言的宏执行类似函数的功能时，应该考虑将它们换成C++内联函数

目录

- 默认参数

5.1 默认参数

✓C++新内容-默认参数，是指当函数调用中省略了实参时自动使用的一个值

✓设置默认值，必须通过函数原型定义，函数定义不变

✓对于带参列表的函数，必须从右向左添加默认值。要为某个参数设置默认值，则必须为它右边的所有参数提供默认值

```
int harpo(int n, int m = 4, int j = 5);// prototype valid √
int chico(int n, int m = 6, int j ); // prototype invalid ×
// error: default arguments must be at the end
int groucho(int k = 1, int m = 8, int j = 9); // prototype valid √
```

5.1 默认参数

✓对于带参列表的函数，必须从右向左添加默认值。要为某个参数设置默认值，则必须为它右边的所有参数提供默认值

✓实参按从左到右的顺序依次被赋给相应的形参，不能跳过任何参数

```
int harpo(int n, int m = 4, int j = 5);// prototype valid √
beeps = harpo(1); // same as harpo(1, 4, 5) √
beeps = harpo(1, 2); // same as harpo(1, 2, 5) √
beeps = harpo(1, 2, 3); // no defaults arguments used √
beeps = harpo(3,,8); // error: cannot skip an argument ×
```

5.1 默认参数

```
#include <iostream>
using namespace std;
int add(int a = 0, int b = 0, int c = 0);
int main()
{
	cout << add(1) << endl;
	cout << add(1, 2) << endl;
	cout << add(1, 2, 3) << endl;
	return 0;
}
int add(int a, int b, int c)
{
	return a + b + c;
}
```

```
1
3
6
```

目录

- 函数重载
  - 函数重载基本概念
  - 函数重载与默认参数
  - 函数的名称修饰

6.1 函数重载基本概念

✓函数多态（函数重载overload）是C++新增功能

✓默认参数能够使用不同数目的参数调用同一个函数，而函数多态（函数重载）能够使用多个同名的函数

✓通过函数重载来设计一系列函数-它们完成相同的工作，但使用不同的参数列表

6.1 函数重载基本概念

✓函数重载的关键是函数的参数列表，也称为函数特征标（funciton signature）

✓C++允许定义名称相同的函数，条件是它们的特征标不同

✓函数重载的特点：

- ❖函数名相同
- ❖参数列表不同（参数个数不同，参数类型不同，参数顺序不同）
- ❖函数的返回类型可以相同也可以不相同

6.1 函数重载基本概念

✓函数重载的优点：

- ❖ 提高代码的复用性
- ❖ 提高代码的可读性
- ❖ 便于记忆

✓ 函数重载的注意事项：

- ❖ 函数重载是发生在同一个作用域内
- ❖ 函数重载是通过函数名和参数列表来区分的
- ❖ 函数重载是不能通过其他因素（如返回值、参数名等）来区分的（返回值类型不同不构成函数重载）

```
// 参数个数不同的重载函数
#include<iostream>
using namespace std;
void F() //无参
{
	cout << "F()" << endl;
}
void F(int a) //带参
{
	cout << "F(int a)" << endl;
}
int main()
{
	F();
	F(10);
	return 0;
}
```

```
F()
F(int a)
```

```
// 参数类型不同的重载函数
#include<iostream>
using namespace std;
void F(int a, int b)
{
      cout << "F(int a, int b)" << endl;
}
void F(char a, char b)
{
      cout << "F(char a ,cahr b)" << endl;
}
int main()
{
      F(10, 20);  //输出第一个函数结果
      F('A', 'B');//输出第二个函数结果
      return 0;
}
```

```
F(int a, int b)
F(char a ,cahr b)
```

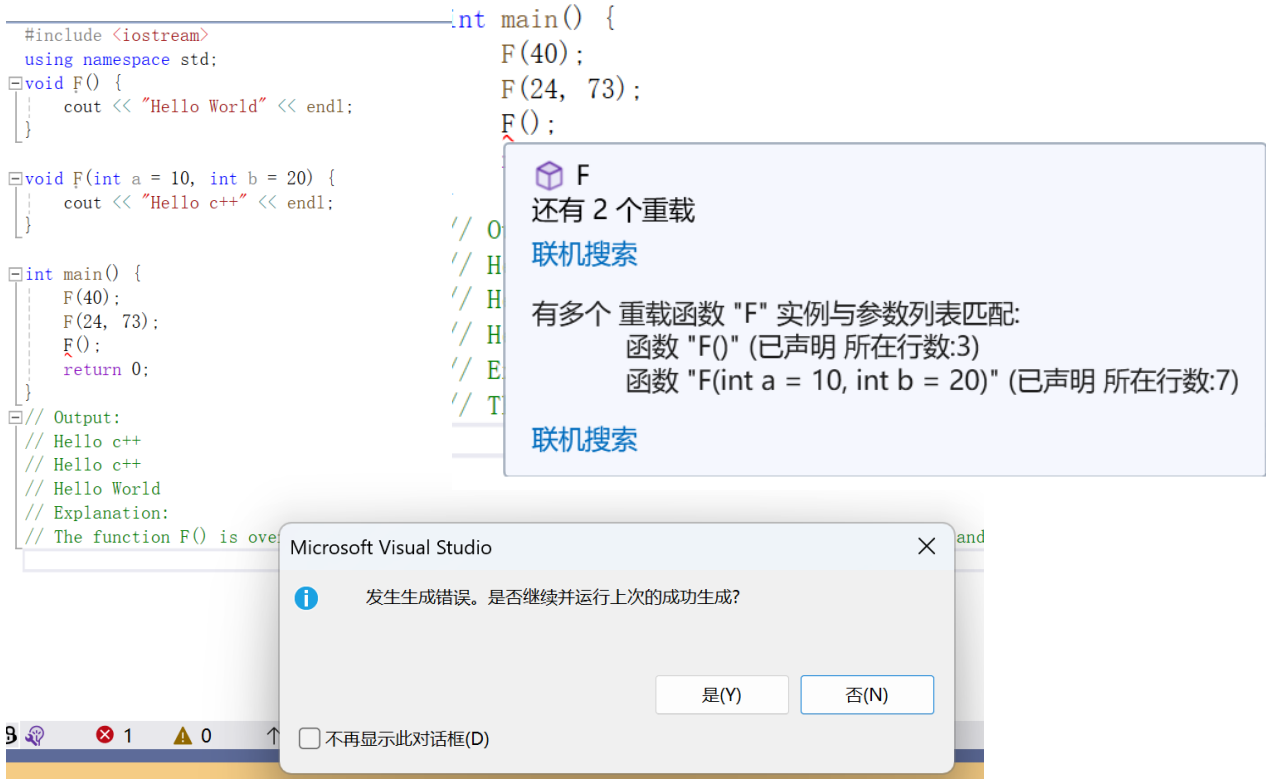
```
// 参数类型顺序不同的重载函数
#include<iostream>
using namespace std;
void F(int a, char b)
{
	cout << "F(int a, char b)" << endl;
}

void F(char a, int b)
{
	cout << "F(char a ,int b)" << endl;
}
int main()
{
	F(10, 'A'); //输出第一个函数结果
	F('A', 20); //输出第二个函数结果
	return 0;
}
```

```
F(int a, char b)
F(char a ,int b)
```

6.2 函数重载与默认参数

```
#include <iostream>
using namespace std;
void F() {
        cout << "Hello World" <<
endl;
}
void F(int a = 10, int b = 20) {
        cout << "Hello c++" << endl;
}
int main()
{
        F(40);
        F(24, 73);
        F();
        return 0;
}
```



解析：

- F(40)；会调用第二个F函数，那么a=40, b=20；
- F(24, 73)；会调用第二个F函数，a=24, b=73；
- F()； 有歧义×，它既符合第一个无参F函数的调用，也符合第二个全缺省F函数的调用(不放实参，默认使用缺省的形参值），这让系统为难，对重载函数调用不明确

6.2 函数重载与默认参数

- ✓仅当函数基本上执行相同的任务，但使用不同形式的数据时，才应采用函数重载
- ✓使用一个带默认参数的函数，只需编写一个函数（而不是两个），程序也只需为一个函数（而不是两个）请求内存；需要修改函数时，只需修改一个
- ✓如果需要使用不同类型的参数，则默认参数便不管用了，在这种情况下，应该使用函数重载

6.3 函数的名称修饰

✓C++开发工具中的编译器编写和编译程序时，C++编译器将执行一些操作-名称修饰（name decoration）或名称矫正（name mangling），它根据函数原型中指定的形参类型对每个函数名进行加密

long MyFunctionFoo(int,float)

编译器将名称转换为内部表示，来描述该接口：

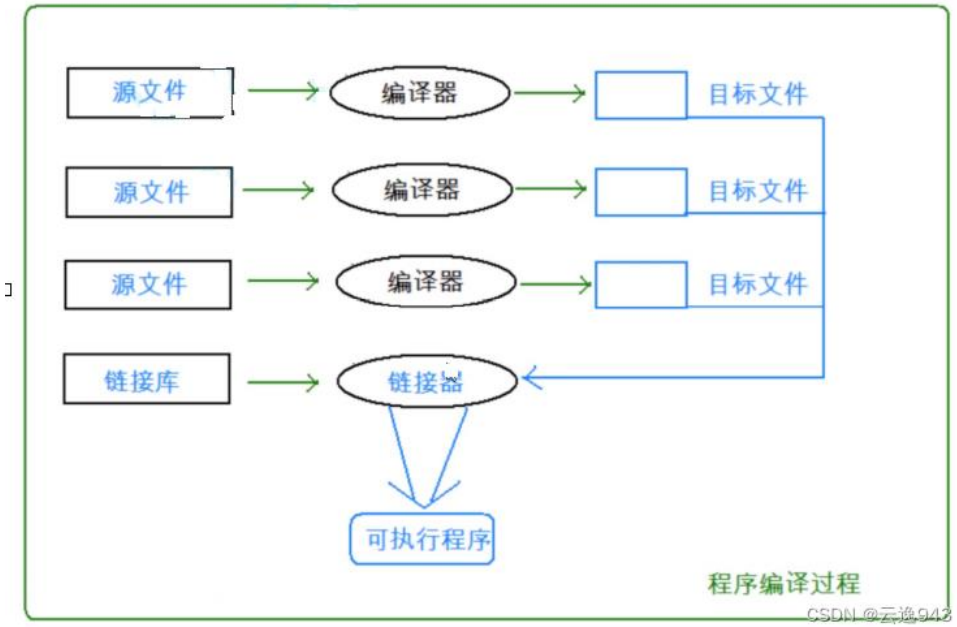
?MyFunctionFoo@@YAXH

不同编译器可能采用不同名称修饰

6.3 函数的名称修饰

- ✓ C语言不支持函数重载：当C语言在编译过程生成符号表时，它的函数在符号表中的名字就只是用自定义的函数名来表示，如果在C语言中写了函数重载，那么它会在符号表中生成两个名字一模一样的函数名，地址完全相同

1.预编译:在预编译阶段，编译器大致会做：头文件的展开，删除注释，条件编译，宏替换生成test.i文件
2.编译:在编译阶段，编译器会：将C语言代码转换成汇编代码(test.s)；进行语法分析，词法分析，语义分析，符号汇总
3.汇编:在汇编阶段，编译器会把汇编代码转换成二进制指令(test.o)，形成符号表
4.链接：在链接阶段，链接器会合并段表；符号表的合并和重定位；找调用函数的地址，链接对应上，合并到一起最终生成可执行文件test.exe



6.3 函数的名称修饰

- ✓ C++支持函数重载：设计了一个函数名修饰规则，用函数名+参数类型为标准去区分各个重载函数的不同。C语言对函数名没有任何修饰，只有一个地址；而C++对函数名做了一定修饰，参数不同，修饰出来的名字就不同

```
[xjh@localhost ~]$ ls
testc  test.cpp
[xjh@localhost ~]$ g++ -o testcpp test.cpp
[xjh@localhost ~]$ ls
testc  testcpp  test.cpp
[xjh@localhost ~]$ objdump -S testcpp
```

```
int Add(int a, int b)
{
        return a + b;
}
```

```
00000000004005ad <_Z3Addii>:
  4005ad:       55                      push   %rbp
  4005ae:       48 89 e5                mov    %rsp,%rbp
  4005b1:       89 7d fc                mov    %edi,-0x4(%rbp)
  4005b4:       89 75 f8                mov    %esi,-0x8(%rbp)
  4005b7:       8b 45 f8                mov    -0x8(%rbp),%eax
  4005ba:       8b 55 fc                mov    -0x4(%rbp),%edx
  4005bd:       01 d0                   add    %edx,%eax
  4005bf:       5d                      pop    %rbp
  4005c0:       c3                      retq
```

C++函数名修饰

```
void func(int a, double b, int* p)
{}
```

```
00000000004005c1 <_Z4funcidPi>:
  4005c1:       55                      push   %rbp
  4005c2:       48 89 e5                mov    %rsp,%rbp
  4005c5:       89 7d fc                mov    %edi,-0x4(%rbp)
  4005c8:       f2 0f 11 45 f0          movsd  %xmm0,-0x10(%rbp)
  4005cd:       48 89 75 e8             mov    %rsi,-0x18(%rbp)
  4005d1:       5d                      pop    %rbp
  4005d2:       c3                      retq
```


Linux系统中g++编译器(针对C++语言的)修饰方法

目录

7.1 函数模板

✓C++编译器新增特性-函数模板

✓使用泛型来定义函数，其中泛型可用具体的类型（如int或double）替换

✓泛型编程通过把通用逻辑设计为模板，摆脱了类型的限制极大地提升了代码的可重用性

✓通过将类型作为参数传递给模板，可使编译器生产该类型的函数

✓模板特性有时也被称为参数化类型（parameterized types）

7.1 函数模板

- ✓函数模板允许以任意类型的方式来定义函数
- ✓template－声明函数模板
- ✓typename－表明其后面的符号是一种数据类型，可以用class代替
- ✓T－通用的数据类型，名称可以替换，通常为大写字母

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
  AnyType temp;
  temp = a;
  a = b;
  b = temp;
}
```

```
template <typename T>
void Swap(T &a, T &b)
{
  T temp;
  temp = a;
  a = b;
  b = temp;
}
```

```cpp
// funtemp.cpp -- using a function template
#include <iostream>
// function template prototype
template <typename T>  // or class T
void Swap(T& a, T& b);  //引用做参数
int main()
{
    using namespace std;
    int i = 10;
    int j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i, j);  // generates void Swap(int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";
    double x = 24.5;
    double y = 81.7;
    cout << "x, y = " << x << ", " << y << ".\n";
    cout << "Using compiler-generated double swapper:\n";
    Swap(x, y);  // generates void Swap(double &, double &)
    cout << "Now x, y = " << x << ", " << y << ".\n";
    return 0;
}
```

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
Using compiler-generated double swapper:
Now x, y = 81.7, 24.5.
```

```cpp
// function template definition
template <typename T>  // or class T
void Swap(T& a, T& b)
{
    T temp;//temp a variable of type T
    temp = a;
    a = b;
    b = temp;
}
```

7.2 重载的模板

✓需要对多种不同类型使用同一种算法的函数时，可以使用模板

✓函数模板与普通函数之间可以重载。编译器会根据调用时提供的函数参数，调用能够处理这一类型的最特殊的版本。在特殊性上，一般按照如下顺序考虑：

- ❖普通函数
- ❖特殊模板（限定了T的形式的，指针、引用、容器等）
- ❖普通模板（对T没有任何限制的）

```
// twotemps.cpp -- using overloaded template functions
#include <iostream>
template <typename T>     // original template
void Swap(T& a, T& b);
template <typename T>     // new template
void Swap(T* a, T* b, int n);
void Show(int a[]);
const int Lim = 8;
int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i, j);   // matches original template
    cout << "Now i, j = " << i << ", " << j << ".\n";
```

Part1

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```

```
    int d1[Lim] = { 0,7,0,4,1,7,7,6 };
    int d2[Lim] = { 0,7,2,0,1,9,6,9 };
    cout << "Original arrays:\n";
    Show(d1);
    Show(d2);
    Swap(d1, d2, Lim);  // matches new template
    cout << "Swapped arrays:\n";
    Show(d1);
    Show(d2);
    return 0;
}
```

Part2

```
template <typename T>
void Swap(T& a, T& b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
```

```
void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << endl;
}
```

```
template<typename T>
void func(T& t) { //通用模板函数
    cout << "In generic version template " << t << endl;
}

template<typename T>
void func(T* t) { //指针版本
    cout << "In pointer version template "<< *t << endl;
}

void func(string* s) { //普通函数
    cout << "In normal function " << *s << endl;
}
```

```
int i = 10;
func(i); //调用通用版本，其他函数或者无法实例化或者不匹配
func(&i); //调用指针版本，通用版本虽然也可以用，但是编译器选择最特殊的版本

string s = "abc";
func(&s); //调用普通函数，通用版本和特殊版本虽然也都可以用，但是编译器选择最特殊的版本
func<>(&s); //调用指针版本，通过<>告诉编译器我们需要用template而不是普通函数
```

总结

- 复习函数的基本知识
- 函数参数和按值传递
- 递归
- 内联函数
- 默认参数
- 函数重载
- 函数模板