



§ 4. 函数

4.0. 为什么要引入函数

★ 目前为止的所讲的内容及作业都是只有一个main函数，负责完成一个程序的所有功能

例：输入两数求max

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

int main()
{
    int a,b,m;
    cin >> a >> b;
    m = a > b ? a : b;
    cout << "max=" << m;
    return 0;
}
```

```
//例：函数形式求两数最大值
#include <iostream>
using namespace std;
int my_max(int x, int y)
{
    int z;
    if (x>y) z=x;
    else z=y;
    return (z);
}
int main()
{
    int a,b,m;
    cin >> a >> b;
    m = my_max(a,b);
    cout << "max=" << m;
    return 0;
}
```

例：输出矩形多次
(不是连续多次，不能循环)

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

int main()
{
    ...;
    四句cout;
    ...;
    四句cout;

    return 0;
}
```

```
//例：函数形式求两数最大值
#include <iostream>
using namespace std;
void output_rect()
{
    cout << "*****"<<endl;
    cout << "* *"<<endl;
    cout << "* *"<<endl;
    cout << "*****"<<endl;
}
int main()
{
    ...;
    output_rect();
    ...;
    output_rect();
    return 0;
}
```

=> 如果分解为函数，则逻辑看的更清晰

=> 有效利用可以降低代码总量，并方便维护



§ 4. 函数

4.1. 概述

- ★ C/C++程序的基本组成单位
- ★ 一个函数实现一个特定的功能
- ★ 有且仅有一个main函数，程序执行从main开始
- ★ 函数平行定义，嵌套调用
- ★ 一个源程序文件由多个函数组成，一个程序可由多个源程序文件组成

```
//a1.cpp      //a2.cpp      //a3.cpp
int fun1( )    float fun5( )    double fun4( )
{
}
short fun2( )
{
}
long fun6( )
{
}

int main()
{
}
```

一个程序由3个源程序文件组成
共6个函数，有且仅有一个main

★ 函数的分类

用户使用角度 { 标准函数（库函数） 由系统提供（fabs/sqrt/strlen）
 使用时需要包含相应头文件
 自定义函数 用户自己编写

- 在使用上无任何的区别

函数形式 { 无参 调用函数无数据传递给被调用函数（getchar()）
 有参 调用函数有数据传递给被调用函数（putchar('A'））



§ 4. 函数

4.2. 函数的定义

4.2.1. 无参函数的定义

2.1. 无参函数的定义

函数返回类型 函数名 (**[void]**) {

{ ()
(void)

函数体 { 声明语句
 { 执行语句
 }

```
int fun()
{
    cout << "***" << endl;
    return 0;
}

int fun(void)
{
    cout << "***" << endl;
    return 0;
}
```

★ 函数名的命名规则同变量

★ 返回类型与数据类型相同

```
int fun(void) → long fun2()
{                {
    ...           ...
}                }
```

★ 返回类型可以是void，表示不需要返回类型

```
void fun3()
{
}
```



§ 4. 函数

4.2. 函数的定义

4.2.1. 无参函数的定义

★ C缺省返回类型为int(不建议缺省, int也写), C++不支持默认int, 必须写

=> C方式编译正确, C++报错

The screenshot shows a C file named 'c-demo.c' with the following code:

```
1 #include <stdio.h>
2
3 fun()
4 {
5     return 0;
6 }
7
8 int main()
9 {
10     fun();
11     return 0;
12 }
```

Red arrows point to the `fun()` definition and the `fun();` call in `main()`. The output window shows a successful build:

```
1>demo-c.vcxproj -> D:\Workspace\VS2022-demo\Debug\demo-c.exe
生成: 1 成功, 0 失败, 0 最新, 0 已跳过
生成 开始于 23:05, 并花费了 00.817 秒
```

The screenshot shows a C++ file named 'cpp-demo.cpp' with the following code:

```
1 #include <iostream>
2 using namespace std;
3
4 fun()
5 {
6     return 0;
7 }
8
9 int main()
10 {
11     fun();
12     return 0;
13 }
```

Red arrows point to the `fun()` definition and the `fun();` call in `main()`. The output window shows a compilation error:

```
error C4430: 缺少类型说明符 - 假定为 int。注意: C++ 不支持默认 int
```

=> 重要提醒: C方式不加头文件可能导致严重后果

The screenshot shows a C file named 'demo.c' with the following code:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("%f\n", sin(2));
6     return 0;
7 }
```

Red arrows point to the `sin(2)` call and the `printf` statement. The output window shows a warning and an incorrect result:

```
demo.c(5,17): warning C4013: "sin" 未定义; 假设外部返回 int
demo.c(5,9): warning C4477: "printf": 格式字符串 "%f" 需要类型 "double" 的参数, 但可变参数 1 拥有了类型 "int"
0.000000
```

Text annotations: "C方式: 结果有错, 因为未包含 <math.h>, sin默认返回int" and "=> 如果sin(2)存在于一个复杂的数学表达式中, 很难查错".

The screenshot shows a C++ file named 'demo.cpp' with the following code:

```
1 #include <iostream>
2
3 int main()
4 {
5     printf("%f\n", sin(2));
6     return 0;
7 }
```

Red arrows point to the `sin(2)` call and the `printf` statement. The output window shows a correct result:

```
0.909297
```

Text annotation: "C++方式1: 结果正确 因为iostream默认包含cmath".

The screenshot shows a C++ file named 'demo.cpp' with the following code:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("%f\n", sin(2));
6     return 0;
7 }
```

Red arrows point to the `sin(2)` call and the `printf` statement. The output window shows a compilation error:

```
demo.cpp(5,17): error C3861: "sin": 找不到标识符
```

Text annotation: "C++方式2: 报错, 因为无cmath则找不到sin".



§ 4. 函数

4.2. 函数的定义

4.2.1. 无参函数的定义

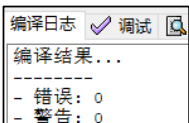
★ ANSI C++要求main函数的返回值只能是int并且不能缺省不写，否则编译会报错；但部分编译器可缺省不写；VS系列还允许void等其它类型 (建议唯一int)

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

```
main()
{
    return 0;
}
```

//VS报错
//Dev正确

error C4430: 缺少类型说明符 - 假定为 int。注意: C++ 不支持默认 int

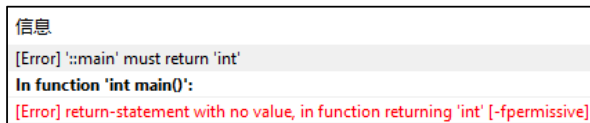


```
#include <iostream>
using namespace std;
```

```
void main()
{
    return;
}
```

//VS报warning
//Dev报错

warning C4326: “main”的返回类型应为“int”而非“void”



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

```
long main()
{
    return 0L;
}
```

//VS报warning
//Dev报错

warning C4326: “main”的返回类型应为“int”而非“long”





§ 4. 函数

4.2. 函数的定义

4.2.2. 有参函数的定义

函数返回类型 函数名 (**形式参数表**)

```
{  
    函数体 { 声明语句  
            执行语句  
}
```

```
int max(int x, int y)  
{  
    int z;          /* 声明语句 */  
    if (x>y)  
        z=x;  
    else  
        z=y;  
    return z;  
}
```

★ 函数名的命名规则同变量

★ 返回类型与数据类型相同

★ 返回类型可以是void，表示不需要返回类型

★ C缺省返回类型为int(不建议缺省，int也写)，C++不支持默认int，必须写



§ 4. 函数

4.3. 函数的嵌套调用

4.3.1. C++程序的执行过程 (一个具体的例子)

例：程序如下

```
void b()
{
    ...
}
void a()
{
    ...
    b();
    ...
}
int main()
{
    ...
    a();
    ...
    return 0;
}
```

//左例，9步

- (1) 执行main函数的开头部分
- (2) 遇到调用a函数的语句，流程转去a函数
- (3) 执行a函数的开头部分
- (4) 遇到调用b函数的语句，流程转去b函数
- (5) 执行b函数，如果再无其他嵌套的调用，则完成b函数的全部操作
- (6) 返回原来调用b函数的位置，即返回a函数
- (7) 继续执行a函数中尚未执行的部分，直到a函数结束
- (8) 返回main中调用a函数的位置
- (9) 继续执行main函数的剩余部分直到结束

如何返回？



§ 4. 函数

4.3. 函数的嵌套调用

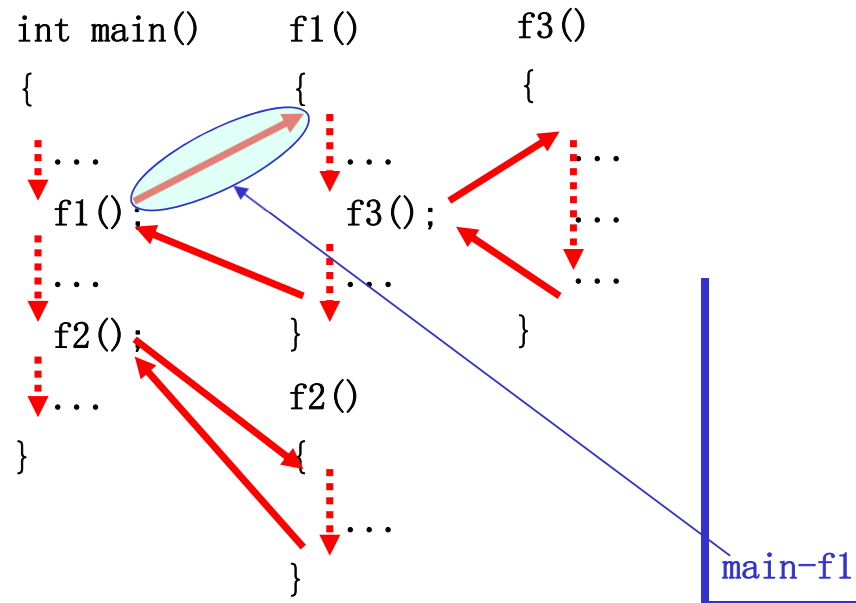
4.3.2. C++程序的执行过程 (通用描述)

- (1) 从main函数的第一个执行语句开始依次执行
- (2) 若执行到函数调用语句，则保存调用函数当前的一些系统信息 (保存现场 - 进栈)
- (3) 转到被调用函数的第一个执行语句开始依次执行
- (4) 被调用函数执行完成后，返回到调用函数的调用处，恢复调用前保存的系统信息 (恢复现场 - 出栈)
- (5) 若被调用函数中仍有调用其它函数的语句，则嵌套执行步骤(2)-(4) (保存和恢复现场的操作遵循栈的操作规则)
- (6) 所有被调用函数执行完后，顺序执行main函数的后续部分直到结束

4.3.3. C++程序的执行过程 (栈方式理解)

4.3.4. 特点

- ★ 嵌套的层次、位置不限
- ★ 遵循后进先出的原则 (栈)
- ★ 调用函数时，被调用函数与其所调用的函数的关系是透明的，适用于大程序的分工组织



自行画出调用过程中
栈的变化形式

图示: `main-f1`表示
保存main的现场,
转去f1函数执行

§ 4. 函数

4. 4. 函数参数与函数的值

4. 4. 1. 形式参数与实际参数

形式参数：在被调用函数中出现的参数

实际参数：在调用函数中出现的参数

★ 实参与形参分别占用不同的内存空间，实形参名称既可以相同，也可以不同

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

void fun(int x)
{
    cout << "fun " << &x << endl;
}

int main()
{
    int x = 10;
    cout << "main " << &x << endl;
    fun(x);
    return 0;
}
```

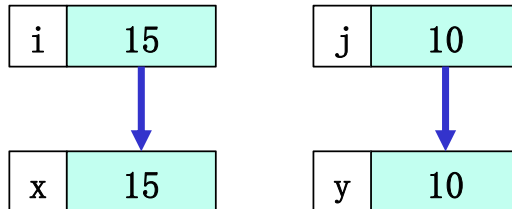
Microsoft Visual Studio 调试控制台

main	00EFD78
fun	00EFC44

本例证明了即使实形参名称相同，也占用不同的内存空间

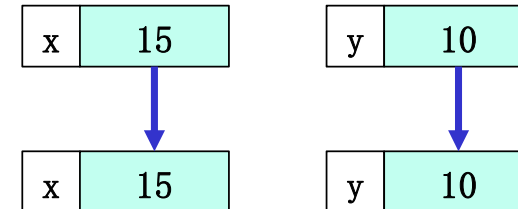
★ 参数的传递方式是“单向传值”，即将实参的值复制一份到形参中（理解为 形参=实参 的形式）

<pre>int main() { int i=15, j=10, m; m = max(i, j); cout<< "max=" << m; return 0; }</pre> <p>i, j为实参</p>	<pre>int max(int x, int y) { int z; z = x>y ? x : y; return z; }</pre> <p>x, y为形参</p>
--	--



<pre>int main() { int x=15, y=10, m; m = max(x, y); cout<< "max=" << m; return 0; }</pre> <p>x, y为实参</p>	<pre>int max(int x, int y) { int z; z = x>y ? x : y; return z; }</pre> <p>x, y为形参</p>
--	--

允许同名





§ 4. 函数

4. 4. 函数参数与函数的值

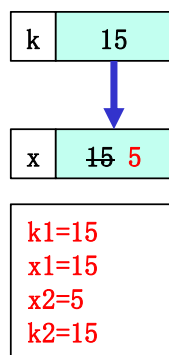
4. 4. 1. 形式参数与实际参数

★ 实参与形参分别占用不同的内存空间

★ 参数的传递方式是“单向传值”，即将实参的值复制一份到形参中 (理解为 形参=实参 的形式)

=> 推论：执行后，形参的变化不影响实参值

```
#include <iostream>
using namespace std;
void fun(int x)
{   cout << "x1=" << x << endl;
    x=5;
    cout << "x2=" << x << endl;
}
int main( )
{   int k=15;
    cout << "k1=" << k << endl;
    fun(k);
    cout << "k2=" << k << endl;
    return 0;
}
```



★ 实参可以是常量、变量、表达式，形参只能是变量

```
int main()
{   int k=10;
    fun(2+k*3);
    return 0;
}
```

```
void fun(int x)
{
    ...
}
x = 2+k*3
```

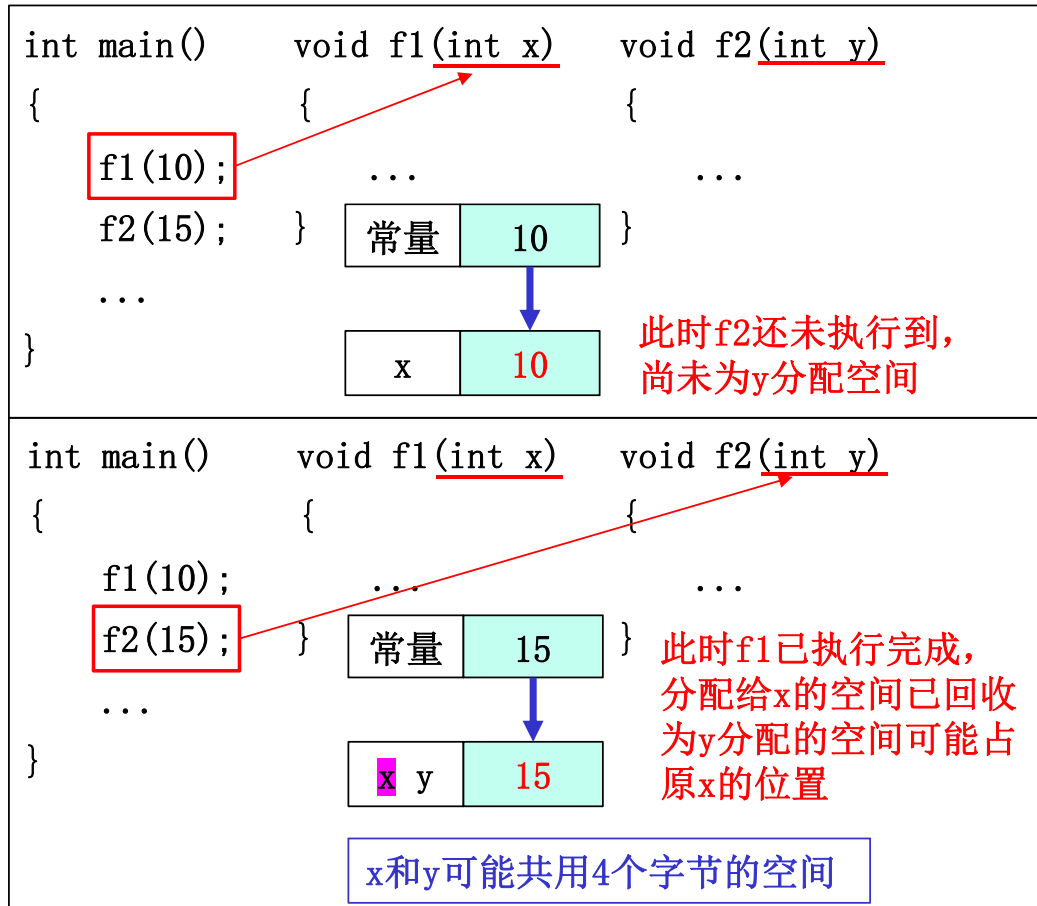


§ 4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

★ 形参在使用时分配空间，函数运行结束后释放空间



```
int main() {  
    ...  
    f1(..);  
    ...  
    f1(..);  
    ...  
}  
void f1(int x) {  
    ...  
}
```

- 1、假设main中调用10000次f1(), 则x的分配释放会重复10000次
- 2、每次x分配的4字节不保证是同一个空间

```
#include <iostream>  
using namespace std;  
void f1(int x)  
{  
    cout << "&x=" << &x << endl;  
}  
void f2(int y)  
{  
    cout << "&y=" << &y << endl;  
}  
int main()  
{  
    for (int i=0; i<2; i++) {  
        f1(10);  
        f2(15);  
    }  
    return 0;  
}
```

Microsoft Visual Studio 调试控制台

```
&x=00C2F6C4  
&y=00C2F6C4  
&x=00C2F6C4  
&y=00C2F6C4
```

本例：
x/y地址相同：证明了f1/f2调用结束后，x/y的空间会释放(x/y共用空间)
多个x(y)地址相同：证明了每次分配是同一空间(但仍然强调：不保证)



§ 4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

★ 实参、形参类型必须一致，否则结果可能不正确

```
#include <iostream>
using namespace std;
int fun(short x)
{
    cout << "x=" << x << endl;  x=4464
    return 0;
}
int main()
{
    long k=70000;
    fun(k); //编译有警告
    cout << "k=" << k << endl;  k=70000
    return 0;
}
```

实形参类型不一致时，
转换规则同赋值（形参 = 实参）

warning C4244: “参数”：从“long”转换到“short”，可能丢失数据



§ 4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 通过return语句获得，若return类型与返回类型定义不一致，以返回类型为准进行数据转换

```
... f(...)
{
    int k;
    ...
    k = fun(...);
    ...
}

int fun(...)
{
    int s;
    ...
    return s;
}
```

若s=10
则k=10

理解为
调用函数中值=return后值的形式

long fun2()	long fun2()	short fun3()
{	{	{
long a;	short a;	long a;
...
return a;	return a;	return a;
} 正确	} 正确	} 可能不正确

//问1: 运行结果(d的值是多少?)
//问2: 哪句会有warning错?

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

warning C4244: “参数”: 从“long”转换到“short”, 可能丢失数据

```
short fun3()
{
    long a = 70000;
    return a;
}

int main()
{
    long d;
    d = fun3();
    cout << d << endl;

    return 0;
}
```

0000000000000001 0001000101110000
↓
0001000101110000
↓
0000000000000000 0001000101110000



§ 4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ return后可以是变量、常量、表达式，有两种形式（带括号、不带括号）

```
return a;      return k*2;
return (a);    return (k*2);
```

★ 若函数不要求有返回值，则指定返回类型为void

<pre>void fun1() { ... return; }</pre>	<pre>int main() { ... return 0; }</pre>	<pre>int fun() { ... return 0; }</pre>
---	---	---

无return语句
空return语句

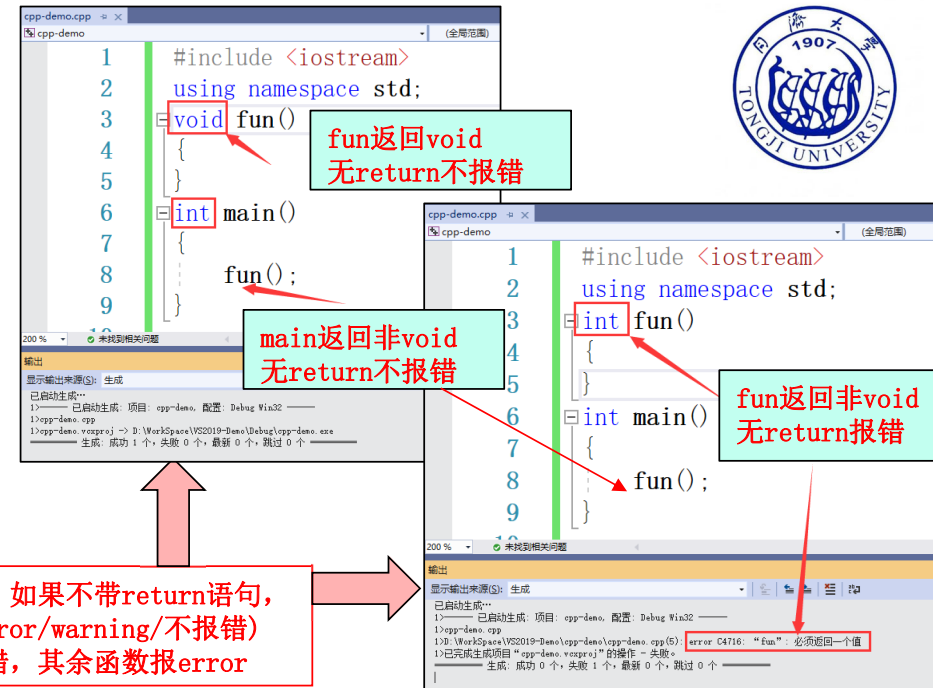
return int 型

return int型

返回类型非void的函数，如果不带return语句，不同编译器表现不同(error/warning/不报错)
VS: main无return不报错，其余函数报error

=> 推论：① 返回类型为void的函数不能出现在除逗号表达式外的任何表达式中
② 若逗号表达式要参与其它运算，则不能做为最后一个表达式出现

```
error C2186: “+”：“void”类型的操作数非法
error C2679: 二元“<<”: 没有找到接受“void”类型的右操作数的运算符(或没有可接受的转换)
```



```
#include <iostream>
using namespace std;
void f()
{   int x=10;
}
int main()
{
    int k=10;
    k=k+f(); //编译错
    k,f();   //可编译通过，无意义
    cout << (k, f()) << endl; //编译错
    cout << (k, f(), k+2) << endl; //可编译通过
    return 0;
}
```



§ 4. 函数

4. 4. 函数参数与函数的值

4. 4. 1. 形式参数与实际参数

4. 4. 2. 函数的值（函数的返回值）

★ 一个return只能带回一个返回值

★ 函数中可以有多个return语句，但只能根据条件执行其中的一个，执行return后，函数调用结束（return后的语句不会被执行到）

```
int fun(void)
{
    if (...)
        return ...;
    else
        return ...;
    ....; //无法被执行到
}
```



§ 4. 函数

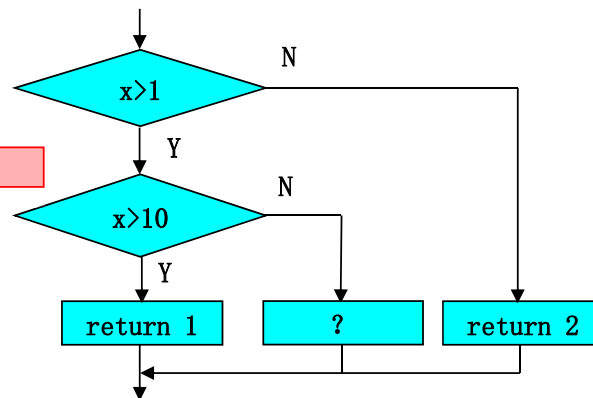
4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

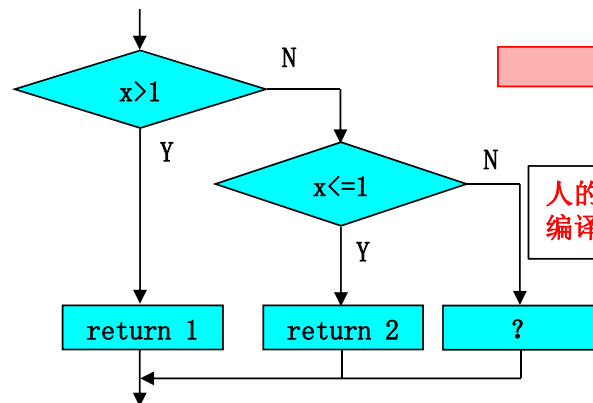
4.4.2. 函数的值（函数的返回值）

★ 如果函数中有分支/循环语句，但return未覆盖全部分支/出口，则VS会报warning错（无论判断条件是否覆盖!）

```
1  #include <iostream>
2  using namespace std;
3  int fun(int x)
4  {
5      if (x > 1) {
6          if (x > 10)
7              return 1;
8          }
9      else
10         return 2;
11 }
12 int main()
13 {
14     int x, r;
15     cin >> x;
16     r = fun(x);
17     return 0;
18 }
```



warning C4715: "fun": 不是所有的控件路径都返回值



人的思维：已全覆盖
编译器思维：缺else

```
1  #include <iostream>
2  using namespace std;
3  int fun(int x)
4  {
5      if (x > 1)
6          return 1;
7      else if (x <= 1)
8          return 2;
9      } //已覆盖int型的全部表示范围
10
11 int main()
12 {
13     int x, r;
14     cin >> x;
15     r = fun(x);
16     return 0;
17 }
```




§ 4. 函数

4.4. 函数参数与函数的值

4.4.1. 形式参数与实际参数

4.4.2. 函数的值（函数的返回值）

★ 如果函数中有分支/循环语句，但return未覆盖全部分支/出口，则VS会报warning错（无论判断条件是否覆盖!）

```
1  #include <iostream>
2  using namespace std;
3  int fun()
4  {
5      int i;
6      for (i = 0; i <= 100; i++) {
7          if (i >= 10)
8              return 0;
9      }
10 }
11 int main()
12 {
13     fun();
14     return 0;
15 }
```

warning C4715: "fun": 不是所有的控件路径都返回值

同理：
人的思维：
i<=100不可能被执行到，因此
只有i>=10这一个出口

编译器思维：
循环退出有两个出口
(1) i>=10 满足后 return 0
(2) 表达式2 (i<=100) 不满足，
结束循环(但无return)

```
1  #include <iostream>
2  using namespace std;
3  int fun()
4  {
5      int i;
6      for (i = 0; ; i++) {
7          if (i >= 10)
8              return 0;
9      }
10 }
11 int main()
12 {
13     fun();
14     return 0;
15 }
```

输出

显示输出来源(S): 生成

已启动生成...

1> 已启动生成: 项目: cpp-demo, 配置: Debug Win32

1>cpp-demo.cpp

1>cpp-demo.vcxproj -> D:\Workspace\VS2019-Demo\Debug\cpp-demo.exe

生成: 成功 1 个, 失败 0 个, 最新 0 个, 跳过 0 个



§ 4. 函数

4. 5. 函数的调用

函数的编写方法:

通过第2-3章的基本知识, 定义不同数据类型的变量,
采用顺序、分支、循环等基本结构, 按照函数的预期功能
来编写每个函数



§ 4. 函数

4. 5. 函数的调用

4. 5. 1. 基本形式

函数名() : 适用于无参函数

函数名(实参表列): 适用于有参函数, 用, 分开

与形参表的个数、顺序、类型一致

★ 若同一变量同时出现在一个函数的多个参数中, 且有自增、赋值、复合赋值等改变变量值的操作, 则不同编译器处理的方式可能不同 (不在讨论, 也不建议深入)

```
int i=3;
```

```
fun(i++, i)
```

从左至右: fun(3, 4)

不再讨论

从右至左: fun(3, 3)

也不建议深入

注意: fun(i++, --j) 这种不同变量是必须讨论的

printf/scanf等函数有参数个数、类型不等的情况出现, 称为可变参数方式, 本课程暂不讨论

```
printf("%d\n", a);           //2个参数
printf("%d %d\n", a, b);     //3个参数
scanf("%d", &a);             //2个参数
scanf("%d %d", &a, &b);      //3个参数
```

```
1  #include <iostream>
2  using namespace std;
3  void fun(int x, int y)
4  {
5      cout << x << ' ' << y << endl;
6  }
7  int main()
8  {
9      int i = 3;
10     fun(i++, i);
11     return 0;
12 }
```

Dev
VS

Microsoft Visual Studio 调试控制台

```
[root@RF5-X64 ~]# cat t2.cpp
#include <iostream>
using namespace std;
void fun(int x, int y)
{
    cout << x << ' ' << y << endl;
}
int main()
{
    int i = 3;
    fun(i++, i);
    return 0;
}

[root@RF5-X64 ~]#
[root@RF5-X64 ~]# c++ -Wall -o t2 t2.cpp
[root@RF5-X64 ~]#
[root@RF5-X64 ~]# ./t2
3 3
```

某Linux版本的编译器



§ 4. 函数

4. 5. 函数的调用

4. 5. 2. 调用方式

函数语句: 函数调用+;

```
printf("Hello. \n");  
putchar(' A');
```

函数表达式: 出现在某个表达式中

```
c=my_max(a, b)+4;  
k=sqrt(m);
```

函数返回类型
不能是void

函数参数: 作为另一个函数的参数

```
printf("max=%d", my_max(a, b));  
putchar( getchar() );  
sqrt( fabs(x) );
```



§ 4. 函数

4. 5. 函数的调用

4. 5. 2. 调用方式

★ 函数调用时，不能写返回类型

定义及实现时: <pre>long f1() { ... } int max(int x, int y) { ... }</pre>	调用时: <pre>k = f1(); ✓ k = long f1(); ✗ k = max(i, j); ✓ k = int max(i, j); ✗</pre>
---	--

```
1  #include <iostream>
2  using namespace std;
3  long f1()
4  {
5      cout << "f1" << endl;
6      return 0L;
7  }
8  int main()
9  {
10     int k;
11     k = long f1(); //调用f1函数
12     return 0;
13 }
```

error C2062: 意外的类型“long”



§ 4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

★ 无参函数调用时，参数位置不能写void

定义及实现时: <pre>int fun() //空 { ... } int fun(void) //写void { ... }</pre>	调用时: <pre>k = fun(); ✓ k = fun(void); ✗</pre>
---	--

```
1  #include <iostream>
2  using namespace std;
3  int fun(void)
4  {
5      cout << "fun" << endl;
6      return 0;
7  }
8  int main()
9  {
10     int k;
11     k = fun(void); //调用f1函数
12     return 0;
13 }
```

```
error C2144: 语法错误: “void” 的前面应有 “)”
error C2144: 语法错误: “void” 的前面应有 “;”
error C2059: 语法错误: “)”
warning C4091: “ ”: 没有声明变量时忽略 “void” 的左侧
```



§ 4. 函数

4.5. 函数的调用

4.5.2. 调用方式

- ★ 函数调用时，不能写返回类型
- ★ 无参函数调用时，参数位置不能写void
- ★ 有参函数调用时，实参不能写类型

定义及实现时：

```
int max(int x, int y)
{
    ...
}
```

调用时：

```
int i=10, j=15;
k=max(i, j); ✓
k=max(int i, int j); ✗
```

```
1  #include <iostream>
2  using namespace std;
3  int my_max(int x, int y)
4  {
5      return x > y ? x : y;
6  }
7  int main()
8  {
9      int i = 10, j = 15, k;
10     k = my_max(int i, int j);
11     cout << "max=" << k << endl;
12     return 0;
13 }
```

```
(10,13): error C2144: 语法错误: “int” 的前面应有 “)”
(10,17): error C2660: “my_max”: 函数不接受 0 个参数
(3,5): message : 参见 “my_max” 的声明
(10,13): error C2144: 语法错误: “int” 的前面应有 “;”
(10,17): error C2086: “int i”: 重定义
(9): message : 参见 “i” 的声明
(10,20): error C2062: 意外的类型 “int”
(10,25): error C2059: 语法错误: “)”
```



§ 4. 函数

4.5. 函数的调用

4.5.2. 调用方式

★ 函数调用时，不能写返回类型

定义及实现时: long f1() { ... }	调用时: k = f1(); ✓ k = long f1(); ✗
int max(int x, int y) { ... }	k = max(i, j); ✓ k = int max(i, j); ✗

问题：其它函数的返回值
可由调用函数使用，
main的返回值给谁？

★ 无参函数调用时，参数位置不能写void

定义及实现时: int fun() //空 { ... }	调用时: k = fun(); ✓ k = fun(void); ✗
int fun(void) //写void { ... }	

★ 有参函数调用时，实参不能写类型

定义及实现时: int max(int x, int y) { ... }	调用时: int i=10, j=15; k=max(i, j); ✓ k=max(int i, int j); ✗
---	--



§ 4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对库函数，加相应的头文件说明

<code>#include <stdio.h></code>	输入输出函数	}	C方式 仅此种	}	C++方式 两种均可
<code>#include <math.h></code>	数学运算函数				
<code>#include <string.h></code>	字符串运算函数				
<code>#include <cstdio></code>	输入输出函数	}	C++方式 两种均可		
<code>#include <cmath></code>	数学运算函数				
<code>#include <cstring></code>	字符串运算函数				

注意：<cstdio>和<cmath>这两个头文件在VS中缺省可以不加，其它编译器一般需要加



§ 4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前

两种方法：

返回类型 函数名(形参类型)；

返回类型 函数名(形参类型 形参表)；

<pre>int my_max(int, int); int main() { k=my_max(i, j); } int my_max(int x, int y) { ... }</pre>	<pre>int my_max(int x, int y); int main() { k=my_max(i, j); } int my_max(int x, int y) { ... }</pre>	<pre>int my_max(int p, int q); int main() { k=my_max(i, j); } int my_max(int x, int y) { ... }</pre> <p>pq不要求 与实现中的 xy一致</p>
--	--	---



§ 4. 函数

4.5. 函数的调用

4.5.1. 基本形式

4.5.2. 调用方式

4.5.3. 对被调用函数的说明

★ 对库函数，加相应的头文件说明

★ 对自定义函数，在调用前加以说明，位置在调用函数前/整个函数定义前

★ 若被调用函数出现在调用函数之前，可以不加说明 (有些编译器可能必须加)

//可以没有说明

```
float fun()
{
    ...
}
int main()
{
    float k;
    k=fun();
    return 0;
}
```

float fun(); //必须有说明

```
int main()
{
    float k;
    k=fun();
    return 0;
}
float fun()
{
    ...
}
```

问：编译器的思维是怎样的？
为什么实现后面必须加说明？



§ 4. 函数

4. 5. 函数的调用

4. 5. 3. 对被调用函数的说明

★ 调用说明可以在函数外，针对后面所有函数均适用；也可在函数内部，只对本函数有效

```
int my_max(int x, int y);
int main()
{
    ..my_max(...); ✓
}
int f1()
{
    ..my_max(...); ✓
}
int my_max(int x, int y)
{
    ....
}
```

```
int main()
{
    int my_max(int, int);
    ..my_max(...); ✓
}
int f1()
{
    ..my_max(...); ✗
}
int my_max(int x, int y)
{
    ....
}
```

error C3861: "my_max": 找不到标识符

```
int main()
{
    ..my_max(...); ?
}
int my_max(int x, int y);
int f1()
{
    ..my_max(...); ?
}
int my_max(int x, int y)
{
    ....
}
```

```
int main()
{
    int my_max(int, int);
    ..my_max(...); ?
}
int my_max(int x, int y)
{
    ....
}
int f1()
{
    ..my_max(...); ?
}
```



§ 4. 函数

4. 5. 函数的嵌套调用

4. 5. 4. 实例

例1：求四个整数的最大值

```
//方法1
int max2(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}

int max4(int a, int b, int c, int d)
{
    int m;
    m = max2(c, d);
    m = max2(m, b);
    m = max2(m, a);
    return m;
}

int main()
{
    int a, b, c, d, m;

    ...输入a/b/c/d四个数字
    m = max4(a, b, c, d);
    ...输出最大值

    return 0;
}
```

```
//方法2
int max2(int x, int y)
{
    return (x > y ? x : y);
}

int max4(int a, int b, int c, int d)
{
    int m1, m2, m;
    m1 = max2(a, b);
    m2 = max2(c, d);
    m = max2(m1, m2);
    return m;
}

int main()
{
    int a, b, c, d, m;

    ...输入a/b/c/d四个数字
    m = max4(a, b, c, d);
    ...输出最大值

    return 0;
}
```

```
//方法3
int main()
{
    ...
    m = max2( max2( max2(c, d), b), a);
    ...
}
```

```
//方法4
int main()
{
    ...
    m = max2( max2(a, b), max2(c, d) );
    ...
}
```

一个函数的返回值做为
另一个函数的参数
(本例中函数名相同)



§ 4. 函数

4.5. 函数的嵌套调用

4.5.4. 实例

例2: 写一个函数, 判断某正整数是否素数

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

int prime(int n)
{
    int i;
    int k = int(sqrt(n));

    for(i=2; i<=k; i++)
        if (n%i == 0)
            break;

    return i<=k ? 0 : 1;
}

int main()
{
    int n;
    cin >> n; //为简化讨论, 此处假设输入正确
    cout << n << (prime(n) ? "是":"不是") << "素数" << endl;
    return 0;
}
```

循环的结束有两个可能性:
1、表达式2 (i<=k) 不成立 (是素数)
2、因为 break 而结束 (不是素数)

```
//03模块例: 求100~200间的素数
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int m, k, i, line=0;

    for(m=101; m<=200; m+=2) {
        k=int(sqrt(m));

        for(i=2; i<=k; i++)
            if (m%i==0)
                break;

        if (i>k) {
            cout << setw(5) << m;
            line++;
            if (line%10==0)
                cout << endl;
        } //end of for

        return 0;
    }
}
```

改写为用
prime函数

```
//03模块例: 求100~200间的素数
#include <iostream>
#include <iomanip>
using namespace std;
int prime(int n)
{
    int i;
    int k = int(sqrt(n));

    for(i=2; i<=k; i++)
        if (n%i == 0)
            break;

    return (i<=k ? 0 : 1);
}

int main()
{
    int m, line = 0;
    for(m=101; m<=200; m+=2) {
        if (prime(m)) {
            cout << setw(5) << m;
            line++;
            if (line%10==0)
                cout << endl;
        }

        return 0;
    }
}
```



§ 4. 函数

4. 5. 函数的嵌套调用

4. 5. 4. 实例

例3: 验证哥德巴赫猜想

```
#include <iostream>
#include <cmath>
using namespace std;
int prime(int n)
{
    int i;
    int k = int(sqrt(n));
    for(i=2; i<k; i++)
        if (n%i == 0)
            break;
    return i<k ? 0 : 1;
}
void gotbaha(int even)
{
    int x;
    for (x=3; x<=even/2; x+=2)
        if ( prime(x) + prime(even-x) == 2) {
            cout << x << "+" << even-x << "=" << even << endl;
            break; //不要break则求出全部组合
        }
}
int main()
{
    int n;
    cin >> n; //为简化讨论, 此处假设输入正确
    gotbaha(n);
    return 0;
}
```

一道题目的解可用于另一题中
强调过程的积累、经验的积累

Microsoft Visual Studio 调试控制台

```
18
5+13=18
```

Microsoft Visual Studio 调试控制台

```
18
5+13=18
7+11=18
```



§ 4. 函数

4.6. 函数的递归调用

4.6.1. 含义

函数直接或间接地调用本身

直接递归

```
f1()
{
...
f1()
...
}
```

间接递归

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

★ 递归是指函数体中调用自己

★ 函数的返回值做本函数的参数，是嵌套，不是递归

```
int main()
{
...
m = max2( max2( max2(a, b), c), d );

m = max2( max2(a, b), max2(c, d) );
...
}
```

必然有条件判断是否进行下次递归调用!!!

4.6.2. 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

例：求解第5个学生的年龄

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；

回溯

```
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



§ 4. 函数

4. 6. 函数的递归调用

4. 6. 3. 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

=> 不要全面考虑1..n之间的变换关系，而应理解为只有n和n-1两层，且第n-1层数据已求得

例1：求解5个学生的年龄

```
int age(int n)
{
    if (n==1)
        return 10;
    else
        return age(n-1)+2;
}
```

```
int main()
{
    cout << age(5) << endl;
    return 0;
}
```

$\text{age}(5) = \text{age}(4) + 2;$

$\text{age}(4) = \text{age}(3) + 2;$

$\text{age}(3) = \text{age}(2) + 2;$

$\text{age}(2) = \text{age}(1) + 2;$

$\text{age}(1) = 10;$



§ 4. 函数

4. 6. 函数的递归调用

4. 6. 3. 如何写递归函数

例2: 采用非递归法和递归法两种方式求解n!

非递归法:

全面考虑1-n的关系,
可得出下列公式:

$$n! = 1*2*\dots*n;$$

```
int fac(int n)
{
    int s=1, i;
    for(i=1; i<=n; i++)
        s = s * i;
    return s;
}
```

递归法:

不全面考虑1-n的关系,
仅考虑n和n-1两层,
且假设n-1层已知

$$n! = n * (n-1)!$$

$$(n-1)! = (n-1) * (n-2)!$$

...

$$1! = 1$$

$$0! = 1;$$

```
int main() //也可以由键盘输入n值, 此处略
{
    int n = 5;
    cout << n << "!=" << fac(5) << endl;
}
```

```
int fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1) * n;
}
```

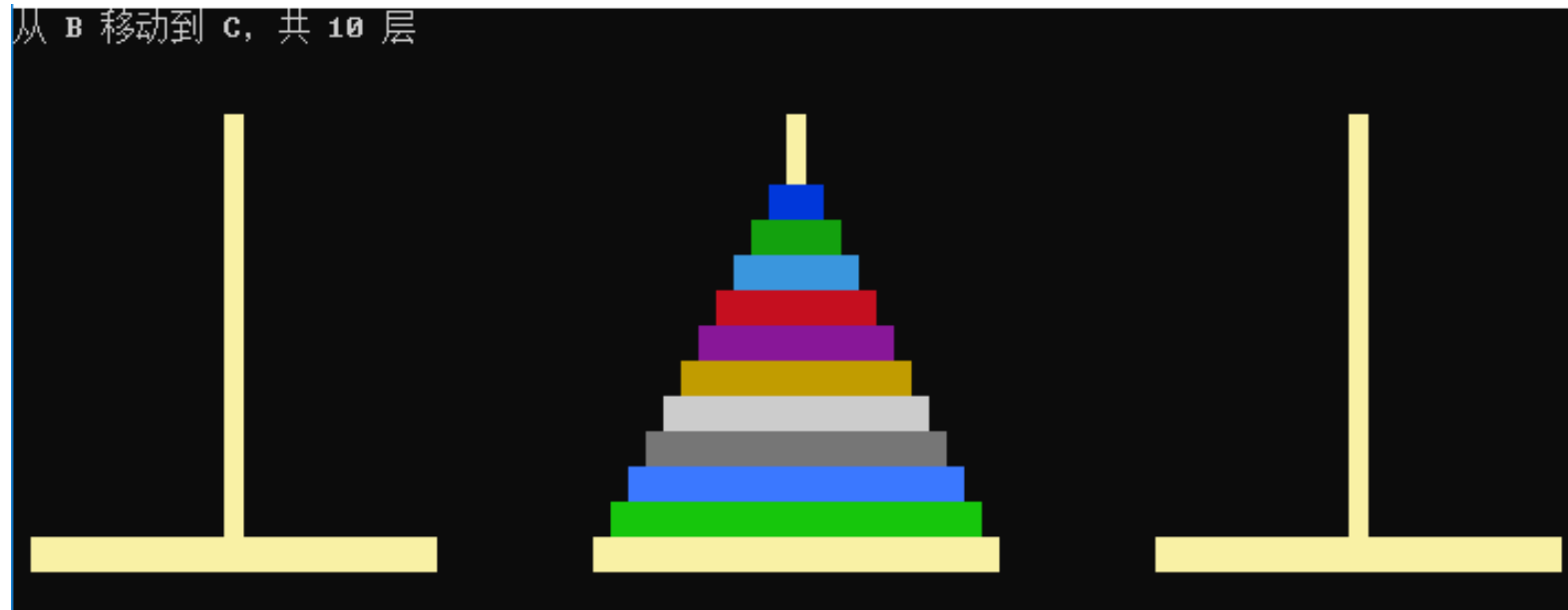


§ 4. 函数

4. 6. 函数的递归调用

4. 6. 3. 如何写递归函数

例3: 汉诺塔问题





§ 4. 函数

4. 6. 函数的递归调用

4. 6. 3. 如何写递归函数

4. 6. 4. 如何读递归函数

- ★ 每次递归调用时，借助**栈**来记录调用的层次
- ★ 栈初始为空，每次递归函数被调用时在栈中增加一项，递归函数运行结束后栈中减少一项
- ★ 本次调用结束后，返回上次的调用位置，继续执行后续的语句
- ★ 重复操作至栈空为止



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0||n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(5)



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(4)	5
fac(5)	

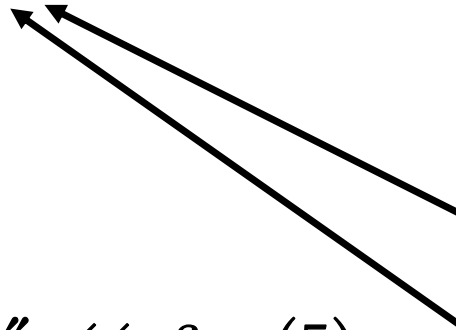


例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(3)	4
fac(4)	5
fac(5)	



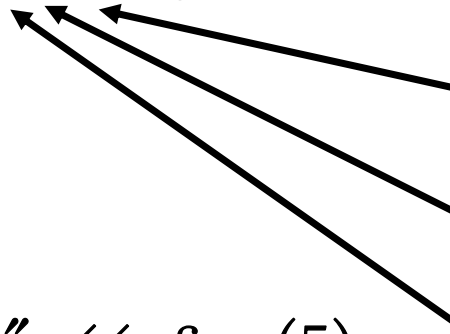


例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	





例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(1)	2
fac(2)	3
fac(3)	4
fac(4)	5
fac(5)	



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(1)	2	1
fac(2)	3	
fac(3)	4	
fac(4)	5	
fac(5)		



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(2)	3	2
fac(3)	4	
fac(4)	5	
fac(5)		

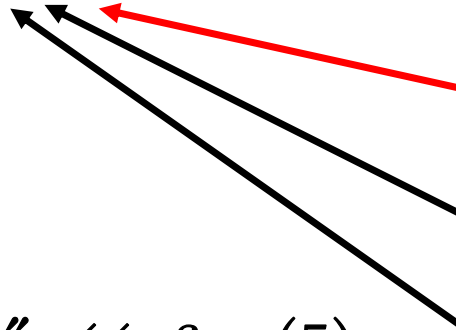


例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(3)	4	6
fac(4)	5	
fac(5)		



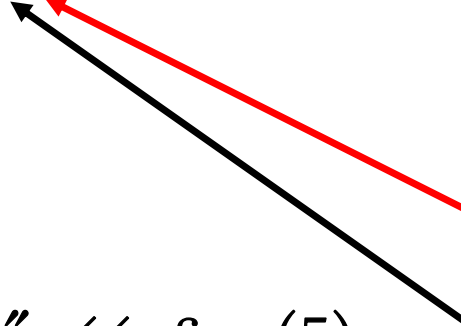


例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(4)	5	24
fac(5)		

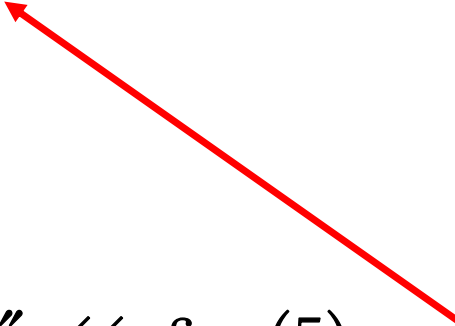




例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

A red arrow originates from the 'fac(5)' cell of the table and points to the 'return fac(n-1)*n;' line in the code block above.

fac(5)		120
--------	--	-----



例1：写出程序的运行结果及程序的功能

```
long fac(int n)
{
    if (n==0 || n==1)
        return 1;
    else
        return fac(n-1)*n;
}

int main()
{
    cout << "fac(5)=" << fac(5);
    return 0;
}
```

fac(5)=120

fac(1)	2	1
fac(2)	3	2
fac(3)	4	6
fac(4)	5	24
fac(5)		120



例2: 写出程序的运行结果

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

int main()
{
    f(7, 'k'); //VS中main无return不报错
}
```



例2: 写出程序的运行结果

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

int main()
{   f(7, 'k');
}
```

7, k



例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

5, k

7, k

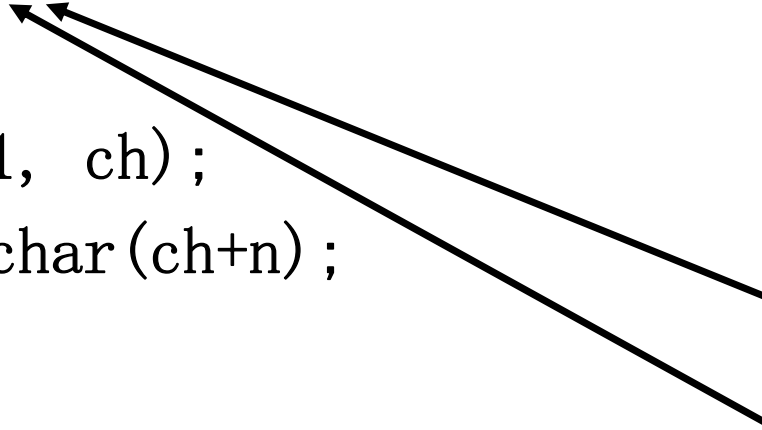


例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

3, k
5, k
7, k





例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

1, k
3, k
5, k
7, k



例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

2, k
1, k
3, k
5, k
7, k



例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

0, k
2, k
1, k
3, k
5, k
7, k



例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

0, k
2, k
1, k
3, k
5, k
7, k

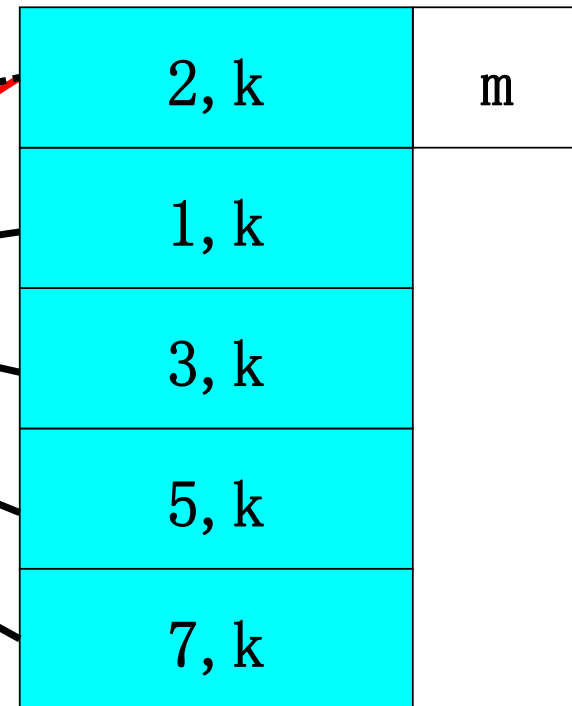


例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

黑虚：上次保存现场位置
红实：本次恢复现场位置





例2: 写出程序的运行结果

```
void f(int n, char ch)
```

```
{  if (n==0)
```

```
    return;
```

```
    if (n>1)
```

```
        f(n-2, ch);
```

```
    else
```

```
        f(n+1, ch);
```

```
    cout << char(ch+n);
```

```
}
```

```
int main()
```

```
{  f(7, 'k');
```

```
}
```

黑虚:上次保存现场位置
红实:本次恢复现场位置

1, k	1
3, k	
5, k	
7, k	

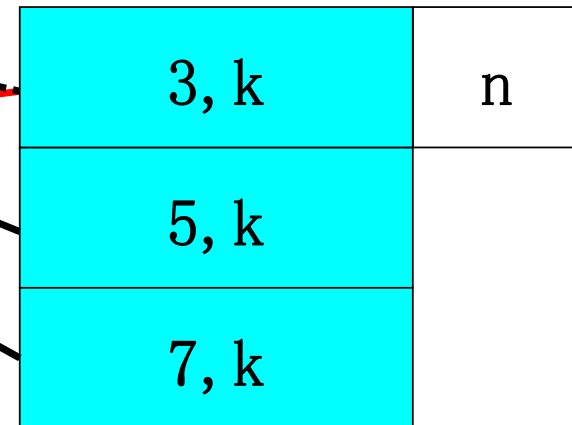


例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

黑虚：上次保存现场位置
红实：本次恢复现场位置



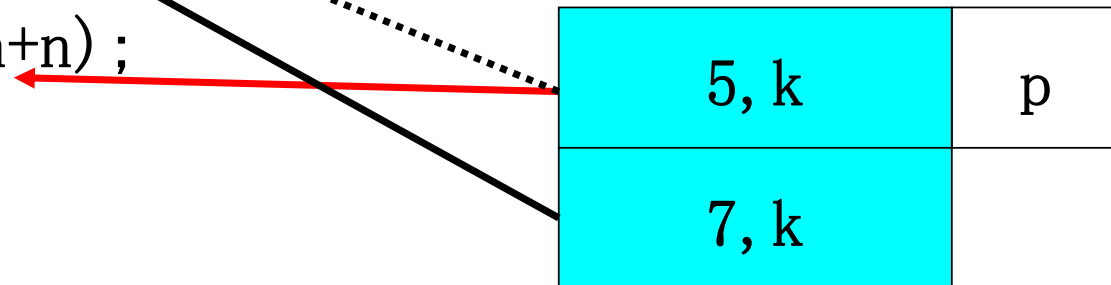


例2: 写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

黑虚:上次保存现场位置
红实:本次恢复现场位置



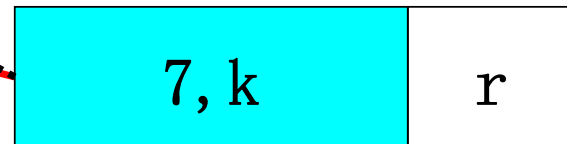


例2: 写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

黑虚:上次保存现场位置
红实:本次恢复现场位置





例2：写出程序的运行结果

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

int main()
{
    f(7, 'k');
}
```

mlnpr

0, k
2, k
1, k
3, k
5, k
7, k



例3：写出程序的运行结果及功能

```
void f(int n, int k)
{
    if (n>=k)
        f(n/k, k);
    cout << n%k;
}

int main()
{
    f(14, 2);      1110
    cout << endl;
    f(65, 8);      101
    return 0;
}
```

请用栈的方式
自行画图理解



§ 4. 函数

4. 6. 函数的递归调用

4. 6. 5. 不设定终止条件的递归函数 (错误的用法)

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4.11中详述
void fun()
{
    num++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{
    fun();
    return 0;
}
```

多编译器/多种模式, 观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4.11中详述
void fun()
{
    int a, b, c, d, e, f, g, h, i, j;
    a=b=c=d=e=f=g=h=i=j=10;
    num++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{
    fun();
    return 0;
}
```

多编译器/多种模式, 观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit

1、为什么会运行崩溃?

答:

2、不定义变量、定义10个int、10个double的情况下崩溃时打印的num值不同, 为什么?

答:

3、有兴趣自行研究各编译器如何改变堆栈大小

```
#include <iostream>
using namespace std;
int num = 0; //全局变量, 后面4.11中详述
void fun()
{
    double a, b, c, d, e, f, g, h, i, j;
    a=b=c=d=e=f=g=h=i=j=10;
    num++; //用于统计fun被调用了多少次
    if (num % 1000 == 0)
        cout << "num=" << num << endl;
    fun();
}
int main()
{
    fun();
    return 0;
}
```

多编译器/多种模式, 观察结果
VS2022 : x86 / x64
Dev C++ : 32bit / 64bit
Linux C++ : 64bit



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

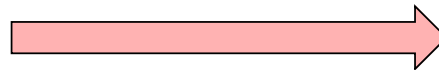
含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第02模块中：变量不能同名，不够准确)

int main()	int f1()	int f2()	int f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
f2();
f3();	a=15;	a=70000;	a=23;
}
	}	}	}

- f1()/f2()/f3() 中的三个a依次分配/释放，在不同时刻占用不同/相同(不保证)的内存空间，互不干扰



★ 形参等同于局部变量

int f1(int x)	int f2(long x)	int f3(int x)
{	{	{
...
}	}	}

可以自行构造打印
形参地址的测试程序

```
#include <iostream>
using namespace std;
void f1()
{
    int a=15;
    cout << "f1:" << &a << endl;
}
void f2()
{
    long a=70000;
    cout << "f2:" << &a << endl;
}
void f3()
{
    short a=23;
    cout << "f3:" << &a << endl;
}
int main()
{
    f1();
    f2();
    f3();
    return 0;
}
```

Microsoft Visual Studio 调试控制台

```
f1:003FFD34
f2:003FFD34
f3:003FFD34
```



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

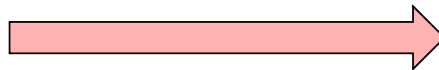
含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第02模块中：变量不能同名，不够准确)

int main()	void f1()	void f2()	void f3()
{ ... f1(); ... }	{ int a; ... a=15; f2(); }	{ long a; ... a=70000; f3(); }	{ short a; ... a=23; ... }

- f1()/f2()/f3() 中的三个a占用不同的内存空间，
当进入f1()时，分配int a;
当进入f2()时，分配long a;
(此时int a未释放，在哪里?)
当进入f3()时，分配short a;
(此时int a/long a均未释放，在哪里?)



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

void f3() //f3为什么要在f2的前面?
{
    short a=23;
    cout << "f3:" << &a << endl;
}
void f2() //f2为什么要在f1的前面?
{
    long a=70000;
    cout << "f2:" << &a << endl;
    f3();
}
void f1()
{
    int a=15;
    cout << "f1:" << &a << endl;
    f2();
}
int main()
{
    f1();
    return 0;
}
```

三个a的空间一定不同!!!

Microsoft Visual Studio 调试控制台

```
f1:001BFD74
f2:001BFC90
f3:001BFBAC
```



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

★ 不同函数内的局部变量可以同名(第02模块中：变量不能同名，不够准确)

int main()	void f1()	void f2()	void f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
...
}	a=15;	a=70000;	a=23;
	f2();	f3();	...
	}	}	}

- 在f3()执行时，三个a占用不同的内存空间(其中f1/f2中的int a/long a在“现场栈”中)，在f3中只能访问short a;

f2-f3
(a=70000)
f1-f2
(a=15)
main-f1

int main()	void f1()	void f2()	void f3()
{ ...	{	{	{
f1();	int a;	long a;	short a;
...
}	a=15;	a=70000;	a=23;
	f2();	f3();	...
	}	}	}

- 在f2()执行时，f1()/f2()的两个a占用不同内存空间(其中f1中的int a在“现场栈”中)，在f2中只能访问long a，而f3()中的a不占空间(调用f3前则未分配，调用f3后则已释放)

f1-f2
(a=15)
main-f1



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量使用：

★ 复合语句内的变量，只在复合语句中有效(包括循环)

允许多层嵌套下各自定义
属于自己作用范围的变量

```
void fun()
{
    int i,a;
    a=15;
    for(i=0;i<10;i++) {
        int y;
        y=11; ✓
        a=16; ✓
    }
    y=12; ✗(超出复合语句的范围)
    a=17; ✓
}
```

error C2065: “y” : 未声明的标识符

```
void fun()
{
    int i,a;
    a=15;
    {
        int y;
        y=11; ✓
        a=16; ✓
    }
    y=12; ✗(超出复合语句的范围)
    a=17; ✓
}
```

```
void fun()
{
    int i,a=15;
    {
        int y;
        y=11; ✓
        a=16; ✓
        {
            int w=10;
            y=12; ✓
            a=13; ✓
            w=14; ✓
        }
        w=15; ✗(超出复合语句的范围)
    }
    y=12; ✗(超出复合语句的范围)
    a=17; ✓
}
```



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量使用：

- ★ 不同函数内的局部变量可以同名
- ★ 形参等同于局部变量
- ★ 复合语句内的变量，只在复合语句中有效(包括循环)
- ★ 在该函数的被调用函数内也无效(不可访问)

```
void f1()
{
    a=14; ✗
}

int main()
{ int a;
  a=15;
  f1();
  a=16;
}
```

```
void f1()
{ int a;
  a=14; ✓
}

int main()
{ int a;
  a=15;
  f1();
  a=16;
}
```

两个a都是局部变量，
分占不同的内存空间，
当f1执行时，main中的a在____？



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

含义：在函数内部定义，只在本函数范围内有效(可访问)的变量

使用：

- ★ 不同函数内的局部变量可以同名
- ★ 形参等同于局部变量
- ★ 复合语句内的变量，只在复合语句中有效(包括循环)
- ★ 在该函数的被调用函数内也无效(不可访问)

=> 递归函数中的局部变量/形参只能在本层被访问

```
#include <iostream>
using namespace std;
void fun(int n)
{
    int x = 10;
    cout << &x << ' ' << ++x << endl;
    if (n > 0)
        fun(--n);
}
int main()
{
    fun(3);
}
```

Address	Value
005BF814	11
005BF72C	11
005BF644	11
005BF55C	11

局部变量x的地址不同：各层访问不同的局部变量
=> 每层只能访问自己的x，x初值为10，++x为11
=> 上层的x在递归栈中，下层的x尚未分配空间



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量使用：

★ 从定义点到源文件结束之间的所有函数均可使用

<pre>int f1() { a=15; ✗ } int a; int main() { a=16; ✓ } int f2() { a=17; ✓ }</pre>	<pre>int a; int f1() { a=15; ✓ } int main() { a=16; ✓ } int f2() { a=17; ✓ }</pre>
---	---



§ 4. 函数

4.7. 局部变量和全局变量

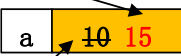
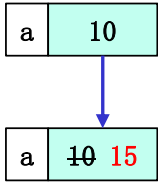
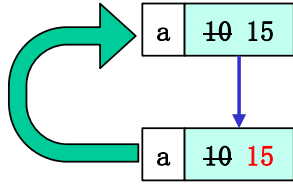
4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量使用：

★ 从定义点到源文件结束之间的所有函数均可使用

★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果

<pre>#include <iostream> using namespace std; int a; void f1() { cout << "f1 " << &a << endl; a=15; } int main() { cout << "main " << &a << endl; a=10; cout << "a=" << a << endl; f1(); cout << "a=" << a << endl; }</pre> 	<pre>#include <iostream> using namespace std; void f1(int a) { a=15; } int main() { int a =10; cout << "a=" << a << endl; a=10 f1(a); cout << "a=" << a << endl; a=10 return 0; }</pre> 	<pre>#include <iostream> using namespace std; int f1(int a) { a=15; return a; } int main() { int a =10; cout << "a=" << a << endl; a=10 a = f1(a); cout << "a=" << a << endl; a=15 return 0; }</pre> 
main()和f()访问的是同一个a，内存空间相同	main()和f()访问的是不同的a，占用不同的内存空间，单向传值	main()和f()访问的是不同的a，实参a是因为赋值语句而改变的



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量使用：

- ★ 从定义点到源文件结束之间的所有函数均可使用
- ★ 全局变量在某个函数中被改变后，其他函数再访问则得到改变后的结果
 - => 全局变量不在某函数被调用时被保存的“现场栈”中
 - => 递归函数的各层均可以访问同一全局变量

```
#include <iostream>
using namespace std;
int a = 10;
void fun(int n)
{
    int x = 10;
    cout << "&a=" << &a << " &x=" << &x << endl;
    cout << " a=" << ++a << " x=" << ++x << endl;
    if (n > 0)
        fun(--n);
}
int main()
{
    fun(3);
    return 0;
}
```

全局变量a的地址相同：各层访问相同的全局变量
=> ++a变化4次，值11~14
局部变量x的地址不同：各层访问不同的局部变量
=> 每层只能访问自己的x，x初值为10，++x为11
=> 上层的x在递归栈中，下层的x尚未分配空间

```
Microsoft Visual Studio 调试控制台
&a=0059C008 &x=00D3FD80
a=11      x=11
&a=0059C008 &x=00D3FC98
a=12      x=11
&a=0059C008 &x=00D3FB00
a=13      x=11
&a=0059C008 &x=00D3FAC8
a=14      x=11
```



§ 4. 函数

4.7. 局部变量和全局变量

4.7.1. 局部变量

4.7.2. 全局变量

含义：在函数体外定义，被多个函数所共用的变量
使用：

★ 在使用全局变量时应加以限制，提高程序的通用性和可靠性(别处的无意修改会导致结果变化)

=>本课程禁用全局变量(特别声明除外)

★ 若全局变量与局部变量同名，按“低层屏蔽高层”的原则处理(应尽量避免，以免理解错误)

全局变量和局部变量分别占用不同的内存空间

能否在f1()中访问全局变量a？


C：不能

C++：可以(后续模块内容)

```
#include <iostream>
using namespace std;
int a=10;
void f1()
{
    cout << "1.&a=" << &a << " a=" << a << endl;
    int a=5;
    cout << "2.&a=" << &a << " a=" << a << endl;
}
int main()
{
    f1();
    return 0;
}
```

全局a	10
-----	----

局部a	5
-----	---

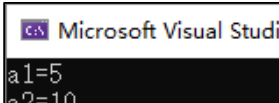


```
1. &a=0017C008 a=10
2. &a=0133FC88 a=5
```

```
#include <iostream>
using namespace std;
int a=10;
void f1()
{
    int a=5;
    cout << "a1=" << a << endl;
}
void f2()
{
    cout << "a2=" << a << endl;
}
int main()
{
    f1();
    f2();
}
```

局部a	5
-----	---

全局a	10
-----	----



```
a1=5
a2=10
```



§ 4. 函数

4.7. 局部变量和全局变量

使用:

★ 若全局变量与局部变量同名, 按“**低层屏蔽高层**”的原则处理 (应尽量避免, 以免理解错误)

=> “**低层屏蔽高层**”的规则同样适用于
局部变量和复合语句内的局部变量同名

=> 在多层次嵌套的情况下允许不同层次的变量
同名, 遵循的基本规则是“**低层屏蔽高层**”

```
void f1()
{
    int a=5, i;
    for(i=0;i<10;i++) {
        int a=10;
        cout << "a=" << a;    a=10
    }
    cout << "a=" << a;        a=5
}
```

```
inline int f()
{
    int a=5;
    cout << "fa=" << a << endl;
}
int main()
{
    int a=10;
    f();
    cout << "ma=" << a << endl;
}
```

问: inline应该这么理解吗?

```
int a=15; ←
void f1()
{
    int a=5, i; ←
    for(i=0;i<10;i++) {
        int a=10; ←
        if (i==5) {
            long a=20; ←
            cout << "a=" << a;    a=20
        }
        cout << "a=" << a;        a=10
    }
    cout << "a=" << a;            a=5
}
```

```
int main()
{
    int a=10;
    int a=5;
    cout << "fa=" << a << endl;
    cout << "ma=" << a << endl;
}
```



§ 4. 函数

4. 8. 变量的存储类别

4. 8. 1. 应用程序执行时的内存分布

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中, 变量占固定的存储空间
动态存储区	程序执行中, 变量根据需要分配不同位置的存储空间

4. 12. 2. 局部变量的存储

4. 12. 2. 1. 分类

自动变量: 函数进入后, 分配空间, 函数运行结束后, 释放空间 (重复进行)

- 1、假设main()中调用10000次f1(), 则x, a的分配释放会重复10000次
- 2、不保证每次x/a的空间与上次相同

- 1、假设main()中调用10000次f1(), 则a的分配释放只有1次(x仍为10000次)
- 2、每次进入f1中, a都保持上次的值不变

静态局部变量: 变量所占存储单元在程序的执行过程中均不释放 (无论函数体内外)

```
int main()      void f1(int x)
{ ...          {
  f1(..);      {
  ...          {
  f1(..);      {
  ...          {
} //假设调用10000次f1()
```

```
int main()      void f1(int x)
{ ...          {
  f1(..);      {
  ...          {
  f1(..);      {
  ...          {
} //假设调用10000次f1()
```



§ 4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.1. 分类

自动变量：函数进入后，分配空间，函数运行结束后，释放空间（重复进行）

★ 关于自动变量(auto)的新旧标准

- C++新标准中，缺省不写就是**自动变量**，而auto用来表示**自动存储类型**的变量
=>新标准中，自动变量/auto变量是**不同的变量**
- C++旧标准中，缺省不写就是自动变量，也可以加auto来表示
=>旧标准中，自动变量/auto变量是**相同的变量**

- 1、为适应多编译器，函数内的局部变量按正常定义，不加auto前缀
- 2、**不准使用**新标准的auto型变量(看得懂)
- 3、某些编译器默认使用旧标准，可通过加编译参数的方式使用新标准，具体方法略

```
#include <iostream>
using namespace std;
int main()
{
    auto int a;    //auto+类型
    int b=10;
    auto char c=2.1; //auto+类型

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

VS+Dev编译

```
#include <iostream>
using namespace std;
int main()
{
    auto a = 1;    //仅auto
    auto b = 'A';  //仅auto
    auto c=2.1;    //仅auto

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
    cout << sizeof(c) << endl;
    return 0;
}
```

VS+Dev编译

```
int main()
{
    auto int a;    //int 型自动变量
    int b=10;      //int 型自动变量(未加auto)
    auto char c=2.1; //char型自动变量
}
```

旧标准

//auto变量不允许跟类型，定义时必须初始化，根据初始化值决定类型

新标准

```
int main()
{
    auto int x; //错误
    auto a=1;   //int型(换为1U, 如何证明类型)
    auto b='A'; //char型
    auto f=1.0; //double型(换为1.0F)
}
```

如果想使用auto自动类型，要对1/1LU/1.0/1.0F等常量的含义非常清晰，因此本课程禁止使用

静态局部变量：变量所占存储单元在程序的执行过程中均不释放（无论函数体内外）



§ 4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量

★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均**重复赋初值**;

静态局部变量在第一次调用时执行, 以后每次调用**不再赋初值**, 保留上次调用结束时的值

<pre>#include <iostream> using namespace std; void f1() { int a=1; //正常写, 不加auto a++; cout << "a=" << a << endl; } int main() { f1(); a=2 f1(); a=2 f1(); a=2 }</pre>	自动变量	<pre>#include <iostream> using namespace std; void f1() { static int a=1; a++; cout << "a=" << a << endl; } int main() { f1(); a=2 f1(); a=3 f1(); a=4 }</pre>	静态局部变量
<div>若定义时赋初值, 自动变量在函数调用 时执行, 每次调用均 重复赋初值</div>		<div>静态局部变量赋初值在 第一次调用时执行, 以后每次调用不再赋初值, 而保留上次调用结束时的值</div>	
<div>1、a的分配/释放重复了3次 2、3次的a不保证分配同一空间</div>		<div>1、a在第一次调用时分配空间并进行初始化, 在3次退出/ 后2次调用中未再进行分配/释放 2、每次进入, a都是同一空间 3、在f1()内部, a可被访问, 在f1()外部, a不能访问(但存在)</div>	



§ 4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量

★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均**重复赋初值**;

静态局部变量在第一次调用时执行, 以后每次调用**不再赋初值**, 保留上次调用结束时的值

```
#include <iostream>
using namespace std;
int f(int n)
{
    int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

?

```
#include <iostream>
using namespace std;
int f(int n)
{
    static int fac=1;
    return fac*=n;
}
int main()
{
    int i;
    for(i=1;i<=5;i++)
        printf("%d!=%d\n",i, f(i));
    return 0;
}
```

?



§ 4. 函数

4.8. 变量的存储类别

4.8.2. 局部变量的存储

4.8.2.2. 使用

- ★ 自动变量占动态存储区, 静态局部变量占静态存储区, 缺省声明为自动变量
- ★ 若定义时赋初值, 自动变量在函数调用时执行, 每次调用均**重复赋初值**;
静态局部变量在第一次调用时执行, 以后每次调用**不再赋初值**, 保留上次调用结束时的值
- ★ 若定义时不赋初值, 则自动变量的值不确定, 静态局部变量的值为0 (' \0')

```
#include <iostream>
using namespace std;
int main()
{
    short a;
    static short b;
    static char c;
    cout << "a=" << a << endl;
    cout << "b=" << b << endl;
    cout << "c=" << (int)c << endl; //问: 为什么要int?
    return 0;
}
```

a值: VS : 编译报错
Dev: 不可预知值

b=0
c=0

VS error C4700: 使用了未初始化的局部变量 "a"

Dev

a=64
b=0
c=0

- ★ 函数的形参同自动变量



§ 4. 函数

4.8. 变量的存储类别

4.8.3. 寄存器变量

含义：对一些频繁使用的变量，可放入CPU的寄存器中，提高访问速度

(CPU访问寄存器比内存快一个数量级 $10^{-10}s$ vs $10^{-9}s$)

```
register int a;
```

★ 仅对自动变量和形参有效(隐含含义：不能长期占用)

★ 编译系统会自动判断(即使定义了register，最终是否放入寄存器中，仍需要编译系统决定)



§ 4. 函数

4.8. 变量的存储类别

4.8.4. 用extern扩展全局变量的使用范围

原因：全局变量从定义点到源文件结束之间的所有函数均可使用，为了能在其它部分使用变量，需要进行使用范围的扩展

方法：在定义范围外使用全局变量时，应加上extern的说明，**extern不分配存储空间**，只说明对应关系

```
int f1()
{
    a=15; ✗
}
int a;
int main()
{
    a=16; ✓
}

int f2()
{
    a=17; ✓
}
```

```
extern int a;
int f1()
{
    a=15; ✓
}
int a;
int main()
{
    a=16; ✓
}

int f2()
{
    a=17; ✓
}
```

不分配空间
说明对应关系

分配4字节空间



源程序文件ex1.cpp、ex2.cpp共同构成一个程序

ex1.cpp	ex2.cpp
<code>int a;</code>	
<code>int main()</code>	<code>int f1()</code>
<code>{</code>	<code>{</code>
<code> a=16; ✓</code>	<code> a=18; ✗</code>
<code>}</code>	<code>}</code>
<code>int f2()</code>	<code>int f3()</code>
<code>{</code>	<code>{</code>
<code> a=17; ✓</code>	<code> a=19; ✗</code>
<code>}</code>	<code>}</code>

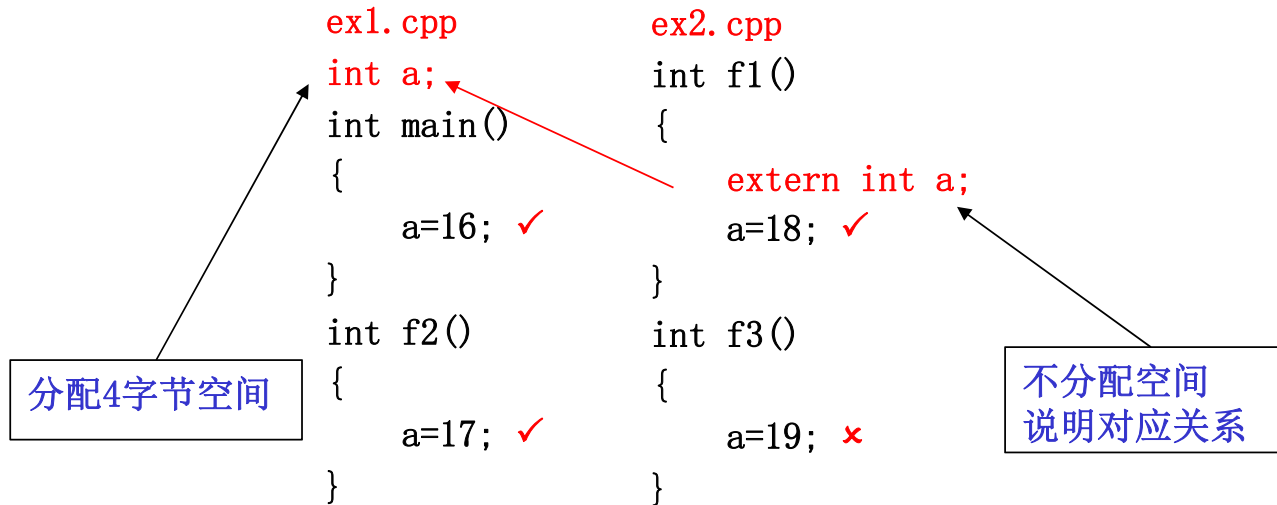
ex1.cpp	ex2.cpp
<code>int a;</code>	<code>extern int a;</code>
<code>int main()</code>	<code>int f1()</code>
<code>{</code>	<code>{</code>
<code> a=16; ✓</code>	<code> a=18; ✓</code>
<code>}</code>	<code>}</code>
<code>int f2()</code>	<code>int f3()</code>
<code>{</code>	<code>{</code>
<code> a=17; ✓</code>	<code> a=19; ✓</code>
<code>}</code>	<code>}</code>

分配4字节空间

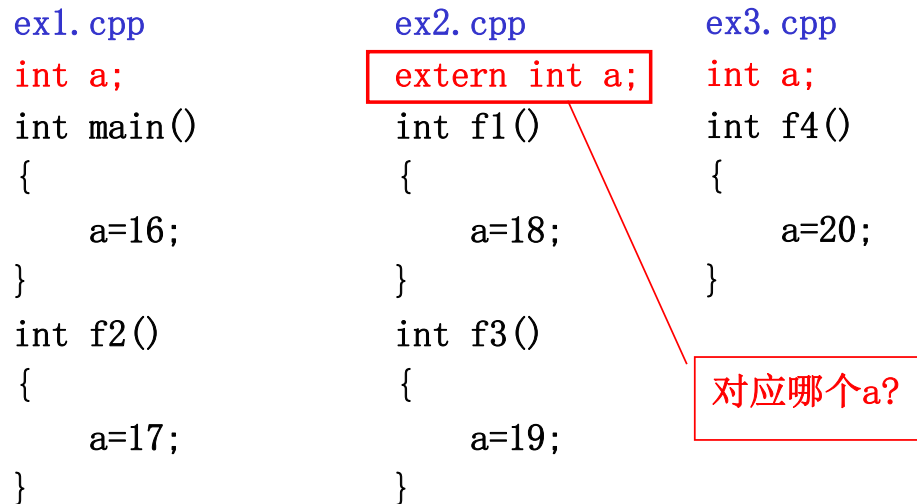
不分配空间
说明对应关系



例：源程序文件ex1.cpp、ex2.cpp共同构成一个程序



例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序





§ 4. 函数

4.8. 变量的存储类别

4.8.5. 全局变量的存储

外部全局变量：所有源程序文件中的函数均可使用

(其它源程序文件中加extern说明)

静态全局变量：只限本源程序文件的定义范围内使用

(static)

- ★ 两者均在静态数据区中分配，不赋初值则自动为0
- ★ 不同源程序文件中的静态全局变量允许同名
- ★ 静态全局变量可与其它源程序文件中的外部全局变量同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

static int a;

int main()

{

a=16; ✓

}

int f2()

{

a=17; ✓

}

ex2.cpp

extern int a;

int f1()

{

a=18; ✗

}

int f3()

{

a=19; ✗

}

ex3.cpp

int f4()

{

a=20; ✗

}

分配4字节空间

无法与静态全局建对应关系

外部源程序文件
无法访问静态全局
变量



例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
static int a;
int f1()
{
    a=18; ✓
}
int f3()
{
    a=19; ✓
}
```

ex3.cpp

```
int f4()
{
    a=20;
    x
}
```

不同源程序文件中的静态全局变量允许同名

例：源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

```
static int a;
int main()
{
    a=16; ✓
}
int f2()
{
    a=17; ✓
}
```

ex2.cpp

```
extern int a;
int f1()
{
    a=18; ✓
}
int f3()
{
    a=19; ✓
}
```

ex3.cpp

```
int a;
int f4()
{
    a=20; ✓
}
```

静态全局变量可与其它源程序文件中的外部全局变量同名

分配4字节空间

不分配空间
说明对应关系

分配4字节空间



例：源程序ex1.cpp-ex4.cpp共同构成一个程序

ex1.cpp	ex2.cpp	ex3.cpp	ex4.cpp	
<code>static int a;</code>	<code>extern int a;</code>	<code>int a;</code>	<code>int a;</code>	
<code>int main()</code>	<code>int f1()</code>	<code>int f4()</code>	<code>int f5()</code>	
<code>{</code>	<code>{</code>	<code>{</code>	<code>{</code>	
<code> a=16;</code>	<code> a=18;</code>	<code> a=20;</code>	<code> a=21;</code>	
<code>}</code>	<code>}</code>	<code>}</code>	<code>}</code>	
<code>int f2()</code>	<code>int f3()</code>			
<code>{</code>	<code>{</code>			
<code> a=17;</code>	<code> a=19;</code>			
<code>}</code>	<code>}</code>			情况1：正确/错误？

ex1.cpp	ex2.cpp	ex3.cpp	ex4.cpp	
<code>static int a;</code>	<code>extern int a;</code>	<code>int a;</code>	<code>static int a;</code>	
<code>int main()</code>	<code>int f1()</code>	<code>int f4()</code>	<code>int f5()</code>	
<code>{</code>	<code>{</code>	<code>{</code>	<code>{</code>	
<code> a=16;</code>	<code> a=18;</code>	<code> a=20;</code>	<code> a=21;</code>	
<code>}</code>	<code>}</code>	<code>}</code>	<code>}</code>	
<code>int f2()</code>	<code>int f3()</code>			
<code>{</code>	<code>{</code>			
<code> a=17;</code>	<code> a=19;</code>			
<code>}</code>	<code>}</code>			情况2：正确/错误？



§ 4. 函数

4.9. 变量属性小结

4.9.1. 变量的分类

按类型：字符型、整型、浮点型等

按作用域 { 局部变量
全局变量

按存储方式（生存期） { 动态存储变量
静态存储变量

按存储位置 { 内存变量
寄存器变量



§ 4. 函数

4. 9. 变量属性小结

4. 9. 2. 不同类型变量对应的存储区

程序(代码)区	存放程序的执行代码
静态存储区	程序执行中，变量占固定的存储空间
动态存储区	程序执行中，变量根据需要分配不同位置的存储空间

静态存储区	动态存储区	CPU寄存器
<ul style="list-style-type: none">● 外部全局变量● 静态全局变量● 静态局部变量● 常量/常变量	<ul style="list-style-type: none">● 自动变量● 函数形参● 堆(动态申请用，后续荣誉课)	<ul style="list-style-type: none">● 寄存器变量



§ 4. 函数

4.9. 变量属性小结

4.9.3. 变量的生存期、作用域与链接性

- ★ 生存期：在什么时间存在，也叫持续性（时间概念）
 - 存放在动态存储区的变量（动态存储）
 - 存放在静态存储区的变量（静态存储）
- ★ 作用域：在什么范围内可以访问（空间概念）
 - 只能在某个函数中被访问的变量（局部变量）
 - 能够在多个函数中被访问的变量（全局变量）
- ★ 链接性：全局变量如何在不同单元间共享（共享概念）
 - 在一个源程序文件的不同函数间共享（静态全局）
 - 在多个源程序文件的不同函数间共享（外部全局）

	生存期	作用域	存储区
自动变量	本函数	本函数	动态数据区
形参	本函数	本函数	动态数据区
寄存器	本函数	本函数	CPU的寄存器
静态局部	程序执行中	本函数	静态数据区
静态全局	程序执行中	本源程序文件	静态数据区
外部全局	程序执行中	全部源程序文件	静态数据区



§ 4. 函数

4. 10. 变量的声明与定义

定义：指定变量的类型，名称并分配存储空间

声明：指明变量的相互关系，不分配存储空间

`int a;` 定义

`extern int a;` 声明



§ 4. 函数

4.11. 内部函数和外部函数

- 内部函数：仅能在本源程序中被调用的函数
 - `static` 返回类型 函数名（形参表）
 - ★ 不同的源程序文件中可以同名
- 外部函数：可以在所有的源程序文件中被调用
 - ★ 本源程序文件中直接使用
 - ★ 其它源程序文件中加函数说明（可以加`extern`，也可以不加）

例：源程序文件ex1.cpp、ex2.cpp共同构成一个程序

外部源程序文件
无法访问内部函数

ex1.cpp

```
static float f2();  
int main()  
{  
    f2();    ✓  
}  
static float f2()  
{  
    ...  
}  
int f1()  
{  
    f2();    ✓  
}
```

ex2.cpp

```
int f3();  
{  
    f2();    ✗  
}
```



源程序ex1.cpp、ex2.cpp、ex3.cpp共同构成一个程序

ex1.cpp

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

ex2.cpp

```
int f3();  
{  
    f2();  
}
```

ex3.cpp

```
static char f2();  
int f4()  
{  
    f2();  
}  
static char f2()  
{  
    ...  
}
```

不同的源程序文件
中的内部函数可以
同名

ex1.cpp

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

ex2.cpp

```
int f3();  
{  
    f2();  
}
```

ex3.cpp

```
extern float f2();  
int f4()  
{  
    f2();  
}
```

在其它源程序文件
中加函数说明可以
访问外部函数

extern可要可不要



例：源程序ex1.cpp-ex3.cpp共同构成一个程序

ex1.cpp

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

ex2.cpp

```
static float f2()
int f3();
{
    f2();
}
float f2()
{
    ...
}
```

ex3.cpp

```
extern float f2();
int f4()
{
    f2();
}
```

情况1：正确/错误？

ex1.cpp

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

ex2.cpp

```
float f2()
int f3();
{
    f2();
}
float f2()
{
    ...
}
```

ex3.cpp

```
extern float f2();
int f4()
{
    f2();
}
```

情况2：正确/错误？



§ 4. 函数

4. 12. 头文件

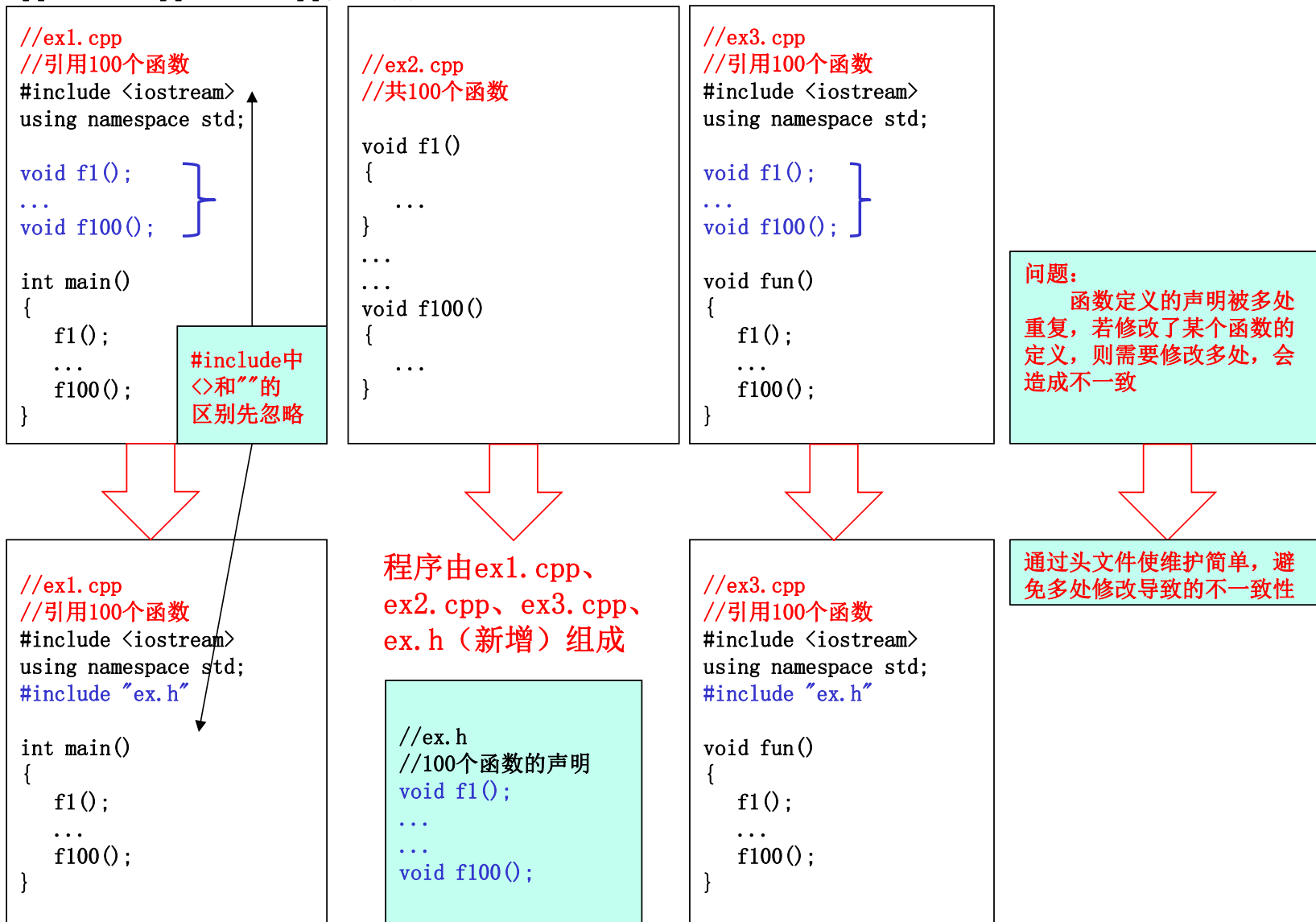
4. 12. 1. 头文件的内容及作用

头文件的内容:

- ★ 结构体类型 (**struct-后续模块**) 及类 (**class-后续模块**) 的声明
- ★ 函数的声明
- ★ inline函数的定义与实现
- ★ 符号常量的定义及常变量的定义
- ★ 全局变量的extern声明
- ★ 其它需要的头文件



例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

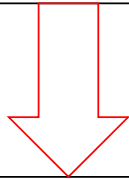




例：程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp
#include <iostream>
using namespace std;
inline void f1()
{
    ...
}

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



```
//ex1.cpp
#include <iostream>
using namespace std;

#include "ex.h"

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

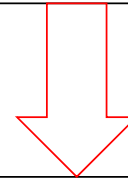
程序由ex1.cpp、
ex2.cpp、ex.h
(新增) 组成

```
//ex.h

inline void f1()
{
    ...
}
```

```
//ex2.cpp
#include <iostream>
using namespace std;
inline void f1()
{
    ...
}

void fun()
{
    f1();
    ...
    f1();
}
```

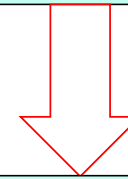


```
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
    f1();
    ...
    f1();
}
```

问题：
因为inline函数
必须和调用函数处在
同一个源文件中，
导致多处重复



通过头文件使维护简单，避
免多处修改导致的不一致性

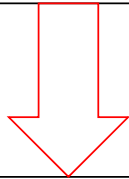


例：程序由ex1.cpp、ex2.cpp共同构成

```
//ex1.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

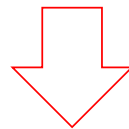
int main()
{
    ...pi...
    ...
    ...x...
}
```



```
//ex1.cpp
#include <iostream>
using namespace std;

#include "ex.h"

int main()
{
    ...pi...
    ...
    ...x...
}
```



程序由ex1.cpp、
ex2.cpp、ex.h
(新增) 组成

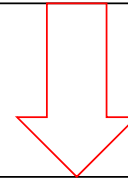
```
//ex.h

#define pi 3.14159
const int x=10;
```

```
//ex2.cpp
#include <iostream>
using namespace std;

#define pi 3.14159
const int x=10;

void fun()
{
    ...pi...
    ...
    ...x...
}
```

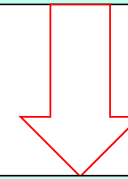


```
//ex2.cpp
#include <iostream>
using namespace std;

#include "ex.h"

void fun()
{
    ...pi...
    ...
    ...x...
}
```

问题：
符号常量及常变量在多处定义，导致重复定义以及维护困难



通过头文件使维护简单，避免多处修改导致的不一致性



例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp  
//定义全局变量
```

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

```
int x=10;
```

```
int main()  
{  
    ...x...  
}
```

```
//ex2.cpp  
//引用全局变量
```

```
extern int x;  
void f1()  
{  
    ...x...  
}
```

```
void f2()  
{  
    ...x...  
}
```

```
//ex3.cpp  
//引用全局变量
```

```
extern int x;  
void fun()  
{  
    ...x...  
}
```

```
//ex1.cpp  
//定义全局变量
```

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

```
int x=10;
```

```
int main()  
{  
    ...x...  
}
```

```
//ex2.cpp  
//引用全局变量
```

```
#include "ex.h"  
void f1()  
{  
    ...x...  
}
```

```
void f2()  
{  
    ...x...  
}
```

```
//ex3.cpp  
//引用全局变量
```

```
#include "ex.h"  
void fun()  
{  
    ...x...  
}
```

程序由ex1.cpp、
ex2.cpp、ex3.cpp、
ex.h（新增）组成

```
//ex.h  
//全局变量声明
```

```
extern int x;
```



例：程序由ex1.cpp、ex2.cpp、ex3.cpp共同构成

```
//ex1.cpp  
//定义全局变量
```

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

```
int x=10;
```

```
int main()  
{  
    ...x...  
}
```

```
//ex2.cpp  
//引用全局变量
```

```
extern int x;  
void f1()  
{  
    ...x...  
}
```

```
void f2()  
{  
    ...x...  
}
```

```
//ex3.cpp  
//引用全局变量
```

```
extern int x;  
void fun()  
{  
    ...x...  
}
```



程序由ex1.cpp、
ex2.cpp、ex3.cpp、
ex.h（新增）组成

```
//ex1.cpp  
//定义全局变量
```

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

```
#include "ex.h"
```

```
int main()  
{  
    ...x...  
}
```

```
//ex2.cpp  
//引用全局变量
```

```
#include "ex.h"  
void f1()  
{  
    ...x...  
}
```

```
void f2()  
{  
    ...x...  
}
```

```
//ex3.cpp  
//引用全局变量
```

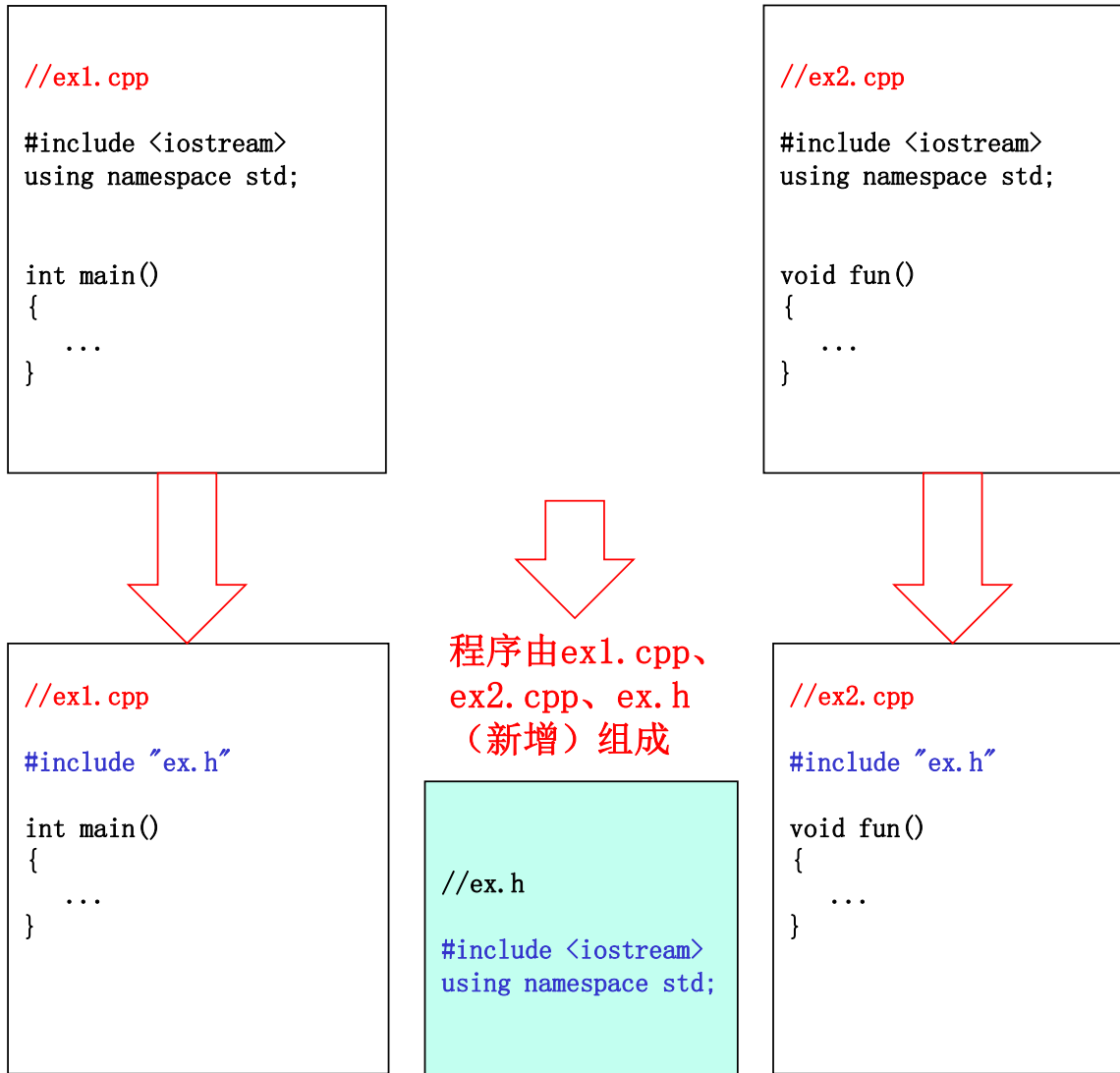
```
#include "ex.h"  
void fun()  
{  
    ...x...  
}
```

```
//ex.h  
//全局变量定义  
int x; //错误
```

注:1. 若头文件中包含全局变量
定义, 则被多个文件包含
会导致重复定义
2. 头文件中可包含静态全局/
只读变量, 但不建议
const int x = 10;
static int x = 15;



例：程序由ex1.cpp、ex2.cpp共同构成





§ 4. 函数

4. 12. 头文件

4. 12. 1. 头文件的内容及作用

头文件的作用：

- ★ 将编程者需要的在不同源程序文件传递的各种信息归集在一起，方便多次调用以及集中修改
- ★ 在一个源程序文件中包含头文件时，头文件的所有内容会被理解为包含到 `#include` 位置处，编译时（变量的定义及函数作用域等）均当作一个文件进行处理

头文件的包含方式：

`#include <文件名>`：直接到系统目录中寻找，找到则包含进来，找不到则报错

`#include "文件名"`：先在当前目录中寻找，找到则包含进来，
找不到则再到系统目录中寻找，找到则包含进来，找不到则报错

VS2022如果缺省安装，则头文件的目录为

64位Windows操作系统：

`C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\xx.xx.xxxxx\include`

具体版本号



§ 4. 函数

例1:理解<>和""的差别

例: 在当前目录下有
demo.h文件
内容:
int a=10;

源程序文件demo.c的内容
#include <iostream>
using namespace std;

```
#include <demo.h>
int main()
{
    cout << a << endl;
    return 0;
}
```

编译报错, 因为<>不寻找
当前目录中是否有demo.h

源程序文件demo.c的内容
#include <iostream>
using namespace std;

```
#include "demo.h"
int main()
{
    cout << a << endl;
    return 0;
}
```

编译正确

例2:理解<>和""的差别

例: 在当前目录下有
demo.h文件
内容:
int a=10;

例: 在系统目录下有
demo.h文件
内容:
int b=10;

源程序文件demo.c的内容
#include <iostream>
using namespace std;

```
#include <demo.h>
int main()
{
    cout << b << endl;
    return 0;
}
```

编译正确

源程序文件demo.c的内容
#include <iostream>
using namespace std;

```
#include "demo.h"
int main()
{
    cout << b << endl;
    return 0;
}
```

编译报错, 因为""方式找到
的是当前目录, 无b的定义



§ 4. 函数

4. 12. 头文件

4. 12. 1. 头文件的内容及作用

4. 12. 2. C++的标准库及头文件

C++包含系统头文件的两种形式:

`#include <math.h>` : C形式

`#include <cmath>` : C++形式

两种方式都是指编译系统的include目录的math.h

VS2022如果缺省安装, 则头文件的目录为

64位Windows操作系统:

C:\Program Files\Microsoft Visual Studio\2022\Community\VC\Tools\MSVC\xx.xx.xxxxx\include

具体版本号



附：关于全局变量使用的基本原则(实际工作中)

- 1、尽量不用
- 2、如果实在需要，尽量使用静态全局
- 3、如果静态全局不能满足要求，尽量在调用函数中进行extern声明

这三点，可理解为
权限最小化原则

<pre>//ex1.cpp int a; fun1() { ... } fun2() { ... }</pre>	<pre>//ex2.cpp f1() { ... f34() { extern int a; } ... f173() { extern int a; } ... f1000() { ... }</pre>
--	--

宁可多处重复，也不要直接放在最前面

4、给全局变量特殊的命名规则

例： ** //下划线开始
**_ //下划线开始+结尾
zs_** //特定串做前缀（假设姓名为张三）

- 5、#define的宏定义及const常变量不在限制范围内，鼓励多用(多处使用相同值时尽量使用)