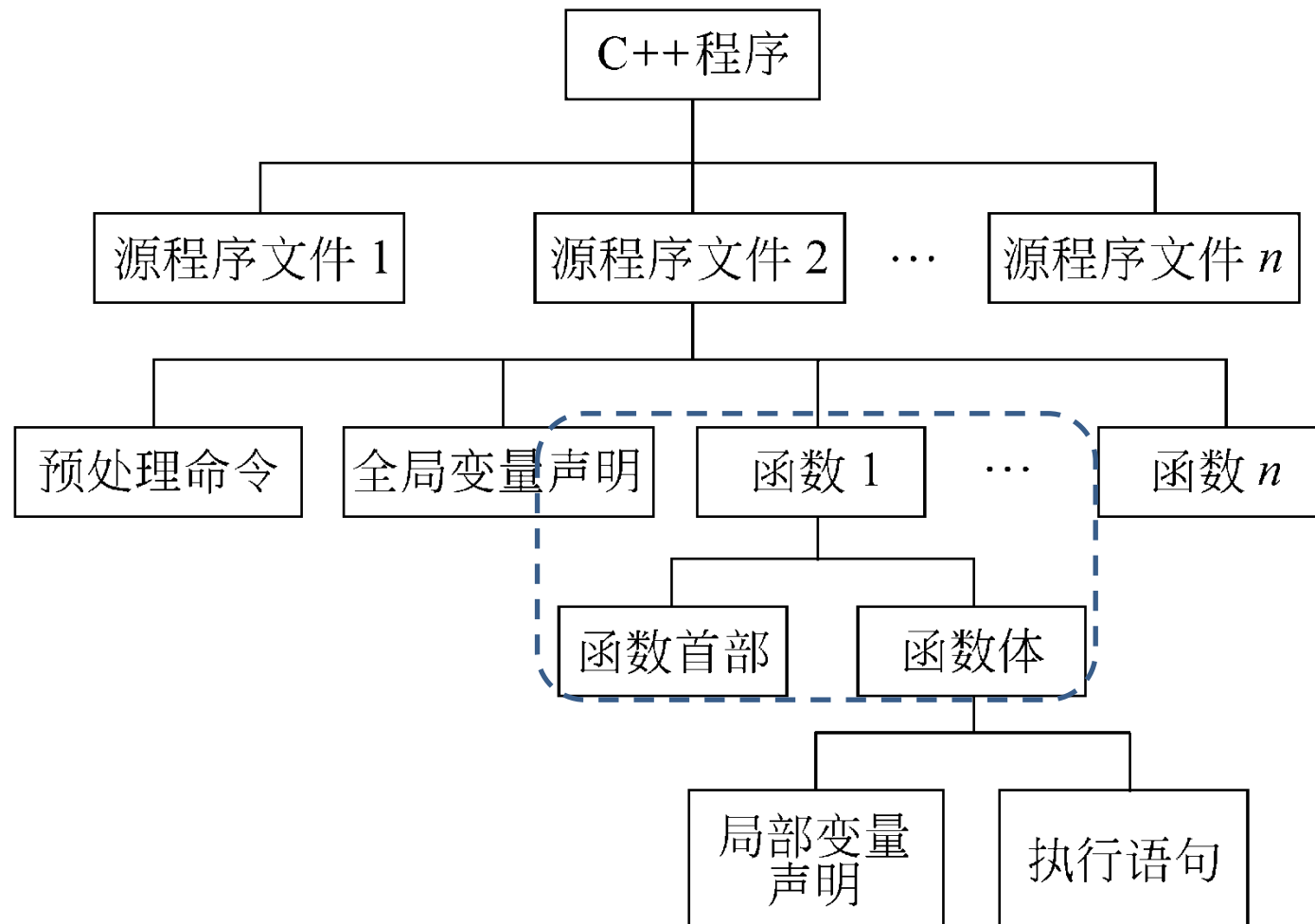


函数

C++程序设计

徐延宁 xyn@sdu.edu.cn

数字媒体技术教育部工程研究中心
山东大学软件学院



- **1、基础-函数的定义与调用**

递归函数

- 2、特殊函数-内联函数
- 3、特殊函数-重载与模板

- 4、变量存储类别、作用域和生命周期
- 5、多文件程序组织
- 6、预处理命令

- 函数遵循**定义**，**声明**，**调用**的次序：
- 函数定义包括**函数头（签名）**和**函数体**
 - 函数头的作用：函数实现与使用的桥梁
 - 函数头三个要素：名字 传入参数 返回结果
 - double sin(double angle)
 - 函数体对应函数的具体实现
- C/C++中，函数不一定要在类中定义

类型说明 **函数名（形式参数列表）**

```
{  
    函数体  
}
```

```
#include <iostream>  
using namespace std;  
void printstar(void)  
{  
    cout << "*****" << endl;  
}  
  
void print_message(int year)  
{  
    cout<< "\t" <<year << " Welcome " << endl;  
}  
  
int main(void)  
{  
    printstar();  
    print_message(2023);  
    printstar();  
    return 0;  
}
```

函数的定义，声明

函数的调用

• 函数的参数

- **形参**是定义函数时的参数**声明**;
- **实参**是运行函数时的参数**赋值**。
- 形参与实参必须类型相同，一一对应。
- 实参必需是运行时实际数值，数值可以由常量、变量、**表达式或者函数计算**产生。
- C++没有规定实参的计算次序
 - $\max(f(x), g(x))$, 先计算f?g?,
 - 如果要写类似上面的代码，注意编译器的具体处理方式

```
#include <iostream>
using namespace std;
int max(int x, int y) {
    int z;
    z = x > y ? x : y;
    return z ;
}
int main() {
    int a, b, c, d;
    cout << "enter two numbers:";
    cin >> a >> b;
    c = max(a, b);
    cout << "max=" << c << endl;
    return 0;
}
```

1-函数的定义与调用

- 函数声明：在一个文件(或函数) 中声明函数f，使得该函数可见可用。
 - 声明时，函数f的形参只要类型，不要名字
 - 函数的定义同时也是一次声明，函数可以被多次，到处声明
 - 同一个cpp代码文件中，定义在后面的函数如果希望被前面的函数调用，必须声明

```
float max (float,float); //声明

int main (void){
    //float max (float,float); // 声明
    float a,b, c;    cin>>a>>b;
    c=max (a,b) ; //使用
    cout<<“The max is”<<c<<endl;
}

float max (float x, float y) // 定义
{   float z;
    z=(x>y)? x : y ;
    return z;
}
```

- 找出左右两段代码错误的原因

```
int main (void){  
    float  a,b, c;    cin>>a>>b;  
    c=max (a,b) ;  
    cout<<“The max is”<<c<<endl;  
}  
  
float  max (float x, float y)  
{ float z;  
  z=(x>y)? x : y ;  
  return z;  
}
```

不声明，编译（语法）错误，IDE红线提示

```
float  max (float,float); //声明  
int main (void)  
{  
    float  a,b, c;    cin>>a>>b;  
    c=max (a,b) ;  
    cout<<“The max is”<<c<<endl;  
}
```

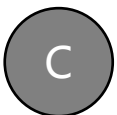
声明，但不存在，引发链接错误

左右两段代码，你喜爱的风格是

```
float max (float x, float y) //定义+声明
{ .....}
float min(float x,float y)
{.....}
int main (void){
    float  a,b, c;    cin>>a>>b;
    c=max (a,b) ;
    cout<<“The max is”<<c<<endl;
}
```

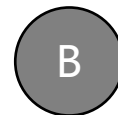


选我



无所谓，我无所谓

```
float  max (float,float); //声明
float min(float,float); //声明
int main (void){
    float  a,b, c;    cin>>a>>b;
    c=max (a,b) ;
    cout<<“The max is”<<c<<endl;
}
float max (float x, float y)
{ .....}
float min(float x,float y)
{.....}
```



选我

提交

C/C++函数的形参可以使用缺省值

```
void f(int x1, int x2 = 2){};
```

1.定义时不可以靠左边缺省

```
void f( int x1 = 5, int x2 , int x3 ) {}// 错
```


2.赋值时不可以中间缺省

```
cout<< f(10, , 8) <<endl; // 错
```

3. 不可以定义与声明同时赋缺省值，通常在函数声明中给出缺省值

```
int f( int x1, int x2 = 2, int x3 = 3);  
int main ( ){  
    f(1);    // 6  
    int f( int x1, int x2 = 3, int x3=4);  
    f(1,2);  // 7  
    return 0;  
}  
void f( int x1, int x2 , int x3 ) {  
    cout << x1 + x2 + x3 << endl;  
} // int
```

- C++中，所有函数的**定义**都是独立的，**不可嵌套定义**。



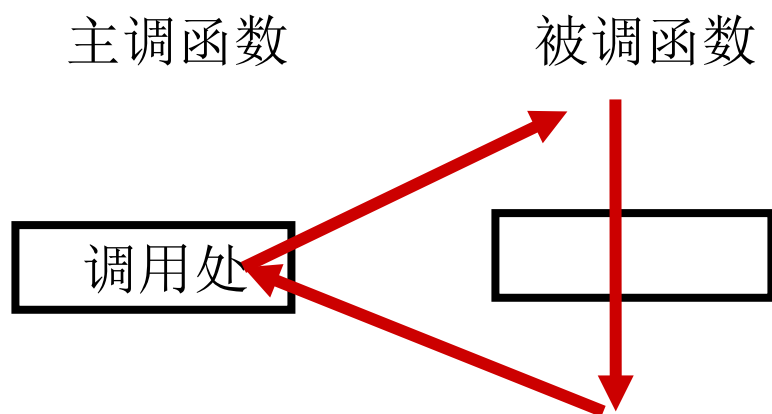
```
int max ( int a, int b)
{
    int c;
    int min ( int a, int b)
    { return ( a<b? a: b); }
    c=min(a,b);
    return ( a>b? a : b);
}
```

```
#include <iostream>
using namespace std;
void hanoi(int n, char one, char two, char three);
int main() {
    int m;
    cout << "input the number of disks:";
    cin >> m;
    cout << "The steps of moving " << m << " disks:" << endl;
    hanoi(m, 'A', 'B', 'C');
    return 0;
}
//将n个盘从one座借助two座，移到three座
void hanoi(int n, char one, char two, char three) {
    void move(char x, char y);
    if (n == 1) move(one, three);
    else {
        hanoi(n - 1, one, three, two);
        move(one, three);
        hanoi(n - 1, two, one, three);
    }
}
void move(char x, char y) {
    cout << x << "-->" << y << endl;
}
```

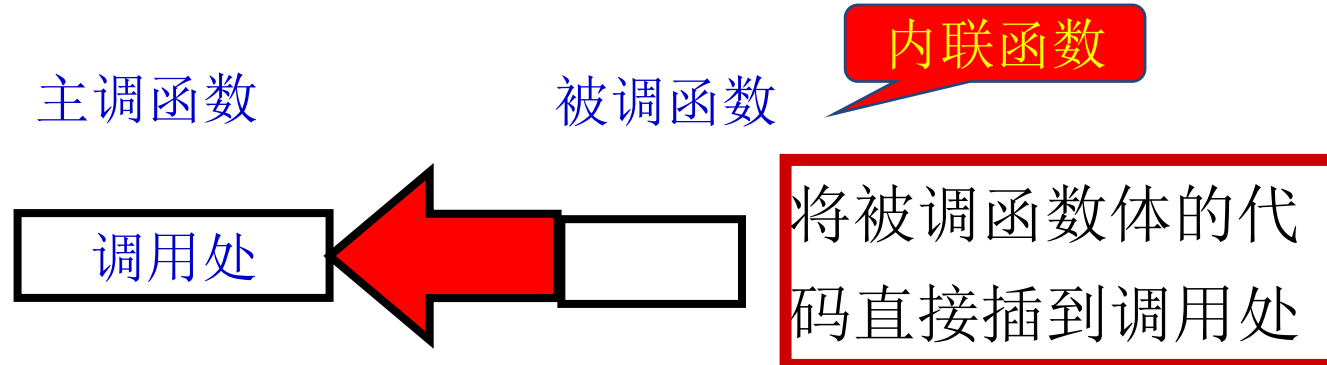
- 函数可以递归调用。汉诺塔，迷宫，全排列，**特别亲切**，确认你是会的，然后Pass

- 1、基础-函数的定义与调用
- 2、特殊函数-内联函数
- 3、特殊函数-重载与模板
- 4、变量存储类别、作用域和生命周期
- 5、多文件程序组织
- 6、预处理命令

- 内联函数 (inline) , 编译时, 将被调函数体的代码直接插到调用处
 - 实质是用目标程序的长度来换取执行时间。程序代码占用多份空间, 但免去调用的额外开销。
 - 一般把规模很小但频繁调用的函数声明成内联 (手机、跑步机、飞机是买来呢, 还是借用的?)



运行开销: 保留现场 (局部变量压栈), 前往被调函数;
恢复现场 (弹栈局部变量)



编译开销, 复制代码,
运行代码长度变长

- inline函数, C C++特色

- 内联函数的定义方法: 在函数类型前增加修饰词**inline**。

```
inline int max (int x, int y)
```

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

```
inline void printStar(){  
  cout << "*****"<<endl;  
}
```

```
void main (void )
```

```
{ int a,b,c;  
  cin>>a>>b;  
  c=max (a+b , a*b) ;  
  cout<<"The max is"<<c<<endl;  
  printStar();  
}
```

- 使用**inline**, 只是**请求**编译器内联该函数, 是否内联**由编译器决定**。

- C++ 11 用constexpr修饰函数，比inline更inline的函数
 - (对大部分人学习成本>收益，有一个概念，可以不过多关注)
 - constexpr修饰的函数，参数是常量时，**编译时运行函数**，返回一个确定的结果（编译常量）
 - constexpr double circleArea(double r) { return 3.14 * r * r; } // 参数必须是constexpr
 - **一般constexpr修饰的函数都非常简单，没有复杂逻辑（分支，循环），否则难以满足要求**

```
constexpr int f(int n) {return 2*n;}
```

```
int g(int n) {return 2*n;}
```

f是constexpr函数，遇到f(6+1)，**编译器**直接计算f的结果

g是普通函数，遇到g(6+1)，g在**程序运行阶段计算**，编译时不计算

```
void test(int n) {  
    int f5 = f(5);  
    int fn = f(n);  
    constexpr int f6 = f(6);  
    //因为f6必须赋值为编译常量，所以可以用f(6)赋值  
    //f(6)替换为f(n)是不行的，替换为g(6)也是不行的  
    char a[f(4)]; //数组长度必须是编译常量，所以可以用f(4)  
    // f(4)替换为f(n)是不行的，替换为g(4)也是不行的  
}
```

- 1、基础-函数的定义与调用
递归函数

- 2、特殊函数-内联函数
- **3、特殊函数-重载与模板**

- 4、变量存储类别、作用域和生命周期
- 5、多文件程序组织
- 6、预处理命令

- **重载函数**是相同的作用域，两个或更多具有**相同函数名**，**不同参数列表的函数**。编译器根据调用时的**实参个数以及类型**来确定调用谁。
- 共用一个函数名字，减轻**使用者（程序员）**的负担，再也不用记忆好多函数名了

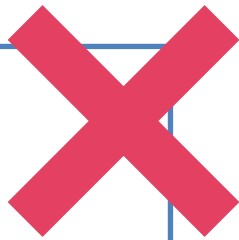
```
void print_int(int i) { ..... }  
void print_double(double d) { ..... }  
void print_char(char c) { ..... }
```

```
void print(int i) { ..... }  
void print(double d) { ..... }  
void print(char c) { ..... }
```

```
void main(void)  
{ print(1);  
  print(1.0);  
  print('c');  
}
```

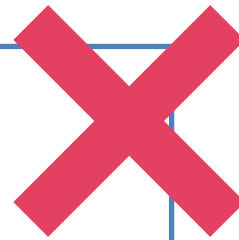

- 重载函数必须具有不同参数列表，理解为：不同的参数个数，或不同的参数类型。

```
void print(int i) { ..... }  
void print(int j) { ..... }
```



不能理解为不同的形参名字

```
void print(int i) { ..... }  
int print(int i) { ..... }
```



仅返回值类型不同时，不能定义为重载函数

```
print(5);
```

```
// 编译器无法链接到合适的实现
```

有趣的话题：当形参缺省多处声明+重载相遇，非常尴尬

```
int f(int x,int y=3);    int f(int);  
  
int main(){  
    int f(int x,int y=4);  
    cout << f(3) << endl; // 结果7  
    return 0;  
}  
  
int f(int x,int y){ return x+y;} // 重载  
int f(int x){ return x;} // 重载
```

```
int f(int x,int y=3);    int f(int);  
  
int main(){  
    cout << f(3) << endl;  
    //二义性导致编译错误,  
    return 0;  
}  
  
int f(int x,int y){ return x+y;} // 重载  
int f(int x){ return x;} // 重载
```

功能、逻辑相似的函数，能不能只写一个定义呢？ **模板函数，以类型作为变量**

```
int max(int a,int b,int c){  
    if(b>a) a=b;  
    if(c>a) a=c;  
    return a;  
}  
double max(double a,double b,double c){  
    if(b>a) a=b;  
    if(c>a) a=c;  
    return a;  
}  
long max(long a,long b,long c ){。 。 。 }
```

函数模板解决方案
类型作为变量

```
template<typename T>  
T max(T a,T b,T c){  
    if(b>a) a=b;  
    if(c>a) a=c;  
    return a;  
}
```

```
i=max(i1,i2,i3);  
d=max(d1,d2,d3);  
g=max<long>(1,2,3);
```

定义T是一个变量，变量T代表某一个类型，而不是数值

在编译时，可以允许编译器根据max的实参类型，推断T，也可以**指定T的类型**

- 函数模板(function template) :
 - 在函数头部前面, 使用template声明虚拟类型(T1, T2...)
 - 可以用虚拟类型 (如T1,T2) 定义该函数返回、形参、局部变量类型:
 - 编译器根据实际使用类型, 由模板编译生成一个或多个实例化函数
- 作用: 通用函数, 实现逻辑相同, 只变量类型有差异的函数, 用一个模板定义。
- 模板只提升了开发的效率, 对于执行效率没有提升, 并没有减少执行中函数的数量。

```
template<typename T>
T max(T a,T b,T c){
    if(b>a) a=b;
    if(c>a) a=c;
    return a;
}
```

```
i=max(i1,i2,i3);
d=max(d1,d2,d3);
g=max<long>(1,2,3);
```



编译生成

```
int max(int a,int b,int c){
    if(b>a) a=b;
    if(c>a) a=c;
    return a;
}
double max(double a,double b,double c){
    . . .
}
long max(long a,long b,long c ){. . . }
```

- 函数模板**f(T)**，编译时，编译器根据指定的类型`<double> f(1)` 推断虚拟类型T，
- 如果不指定类型**f(1)**，编译器自动推断，但可能推断出错。
 - 形参，可能语义（逻辑）错误
 - 返回值、局部变量类型无法推断，语法错误
- 函数模板**不能取代函数重载**，难以处理**参数个数不同，或逻辑实现不同**（实数、复数的加法）的情况。

```
template <class T>
T f(int n){
    T t = 1+n;
    return t;
}

f<double>(1); //常规使用方法，生成double版的函数f
f(1); //语法错误，无法推断
```

右边代码的VS Code给出的执行结果正确么?

- ☒ A 正确
- ☐ B 错误
- ☐ C 不知道

告诉编译器T是double

让编译器决定T1, T2
T1 double, T2 int

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

```
template <typename T>
T inc(int n){
    return 1 + n;
}
```

使用typename和class都可以

```
template <class T1, class T2>
T2 print(T1 arg1, T2 arg2){
    cout << arg1 << " " << arg2 << endl;
    return arg2;
}
```

```
int main(){
    cout << inc<double>(4) / 2 << endl;
    cout << print(3.1, 3)/2 << endl;
}
```

```
2.5
3.1 3
1
```

main函数中的四条cout语句，第几条有语法错误？

- ☐ A 1
- ☒ B 2
- ☐ C 3
- ☐ D 4

```
#include <iostream>
using namespace std;
template <class T>
T f(int n){      T t = 1+n;   return t; }

template <class T>
T f2(T n){      T t = 1+n;   return t; }

void main(){
    cout << f<double>(4) << endl;
    cout<< f(4) << endl;
    cout<< f2<double>(4) << endl;
    cout<< f2(4) << endl;
}
```

提交

- 1、基础-函数的定义与调用
递归函数

- 2、特殊函数-内联函数
- 3、特殊函数-重载与模板

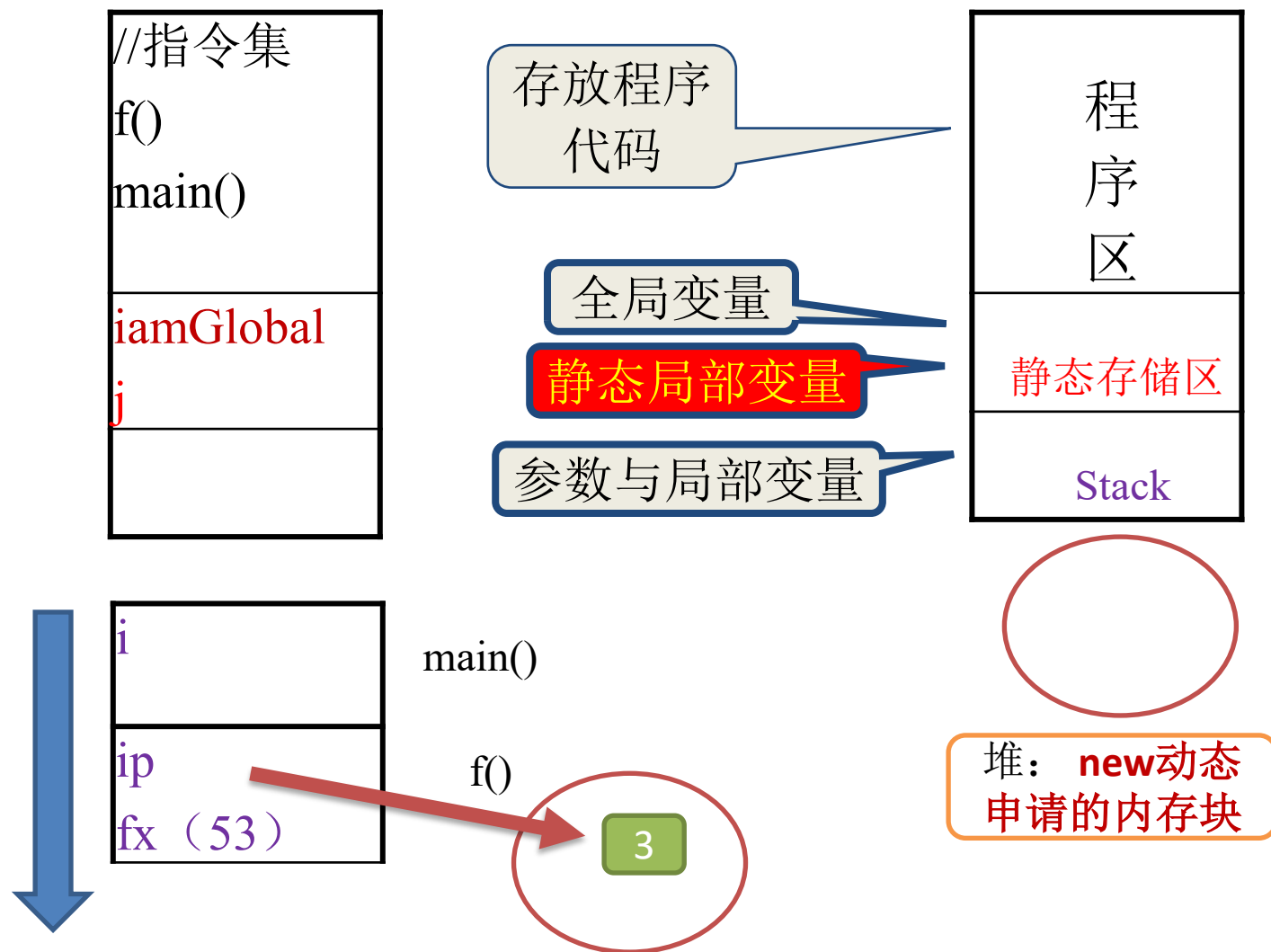
- **4、变量存储类别、作用域和生命周期**
- **5、多文件程序组织**
- 6、预处理命令



- 变量具有**生命周期**和**作用域**。函数具有**作用域**
- **生命周期**指变量的空间分配到释放的周期，为方便管理，C++将不同生命周期的变量存储在**不同性质**的**空间**中；大体有三类存储空间：
 - 1、静态空间：一旦分配，一直存在，直到程序结束被释放：不需要动态管理；
 - 2、栈空间：函数开始分配空间（压栈），函数结束释放空间（弹栈）；后进先出的方式自动维护；
 - 3、堆空间：new申请，delete释放（C语言malloc申请，free释放）；程序员主动维护。

变量的生命周期-存储类别

```
#include <iostream>
using namespace std;
int iamGlobal;
void f(){
    iamGlobal = 1;
    int *ip = new int;
    int fx = 53;
    *ip = 3;
    static int j = 1;
    cout << *ip << j++ << endl;
    delete *p;
}
int main(){
    iamGlobal = 2;
    for (int i = 0; i < 3; i++)    f();
    cout << iamGlobal << endl;
    return 0;
}
```

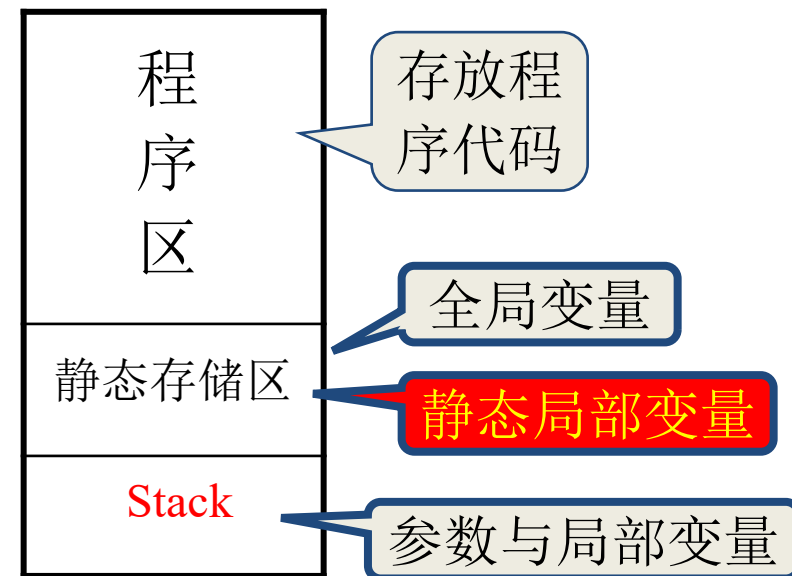


1. 程序代码区—存放函数体的二进制代码（忽略）
2. 静态区（static）：一旦分配，直到程序结束才回收；全局变量，静态局部变量

– 2.1 文字常量区，存放字符串常量

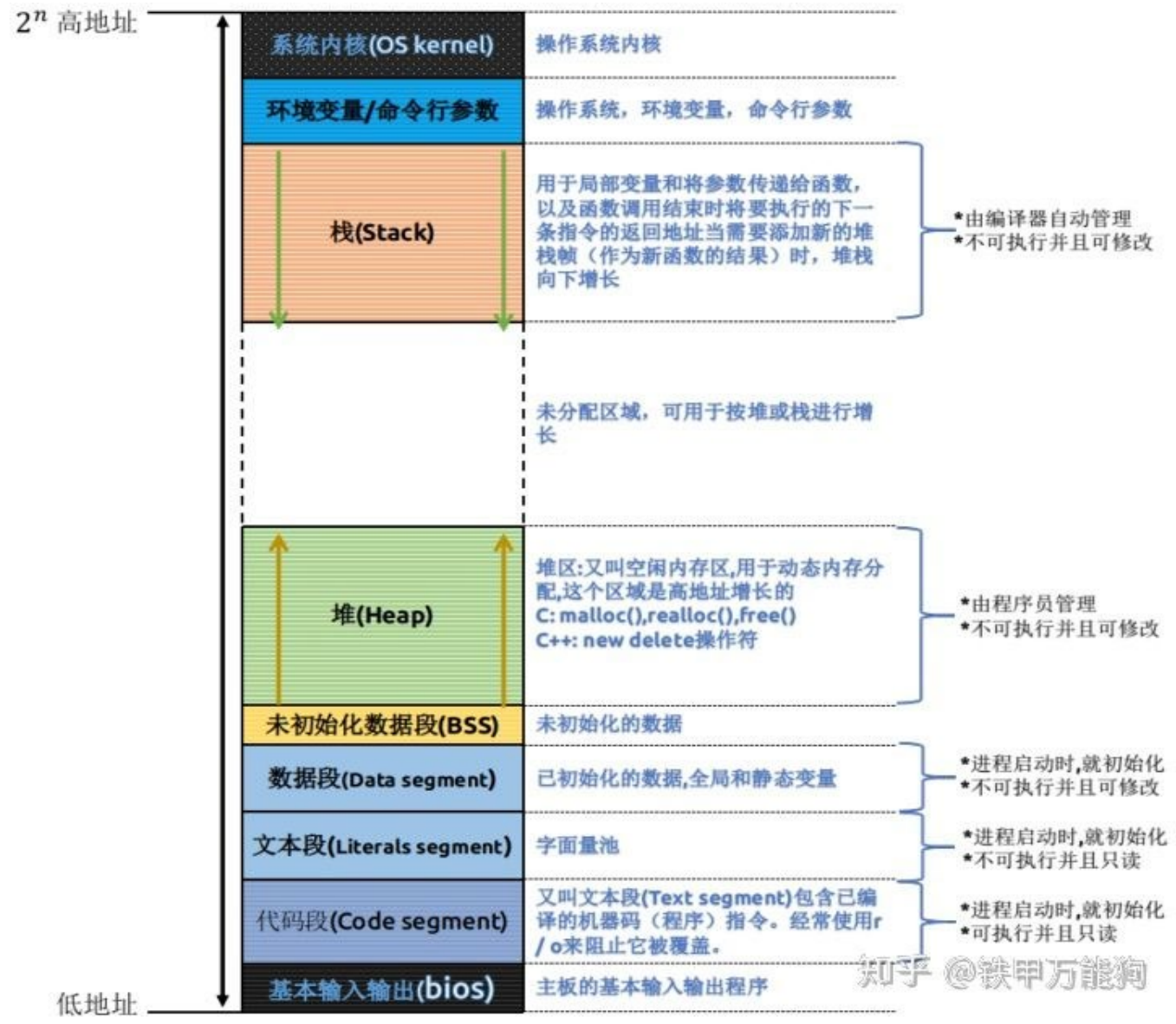
3. **栈（Stack）**：存储函数参数和局部变量，空间由系统**自动动态**分配，回收。函数被调用，分配，压栈；执行完，回收，弹出。特点：空间小（xMB量级），速度快

4. **堆（Heap）**：程序员**动态**申请的内存块，new申请，delete释放（Java自动释放）。特点：空间大（xGB量级），速度慢。对于C/C++，如果程序员忘记回收，即使APP关闭也还会驻留系统，成为垃圾。



new动态申请的内存块，通过引用访问

“堆” heap

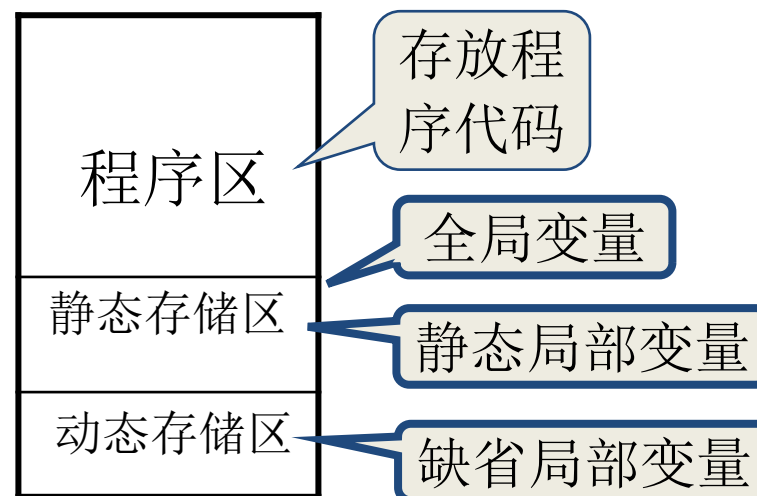


不同OS, 实现模型有所差别,

一个代表:

但主要模块相同: OS内核, 代码段, 文本段, 全局数据段, 分布在两侧。

栈, 堆 对向生长。栈严格有序, 快速高效。堆, 自由无序, 不断申请、释放, 造成很多碎片, 需要经常整理



- 变量被访问时，由内存拷贝到CPU寄存器中，如果变量被频繁访问，常驻寄存器非常有效率。
- 为提高执行效率，可以利用register声明一个变量为**寄存器变量**。
- 在程序中声明寄存器变量对编译系统只是**建议性**，不是强制性的。**通用**编程过程中，不必要用register声明变量，编译器会优化处理。

```
int fac(int n){
    int x;    static int y;
    register int i,f=1;
    //i, f被反复使用，建议其常驻寄存器
    for(i=1;i<=n;i++)    f=f*i;
    return f;
}
```

- 实际存储组织：多层、多类组织方式
- CPU 《-》 寄存器 《-》 多级Cache 《-》 内存 《-》 Buffer 《-》 外存、网络、键盘IO
- GPU 《-》 register, shared memory, local memory, global memory, constant memory, texture memory
- 什么样的数据送到什么特点的存储空间，影响因素：（流、随机）读写、数据量
- 程序员：我为什么要知道这么多， 厂商：好吧，编译器优化，降低通用编程门槛。
让编译器优化，可能得不到最好的性能，但提高了开发效率，降低了程序员的入门难度：不去关心硬件特性，专心描述业务逻辑。
- 但有时候差异巨大，编译器无法去除CPU与GPU之间的巨大鸿沟。

- **静态局部变量**：作用范围局部（函数内部），生命周期：静态。
- 在静态存储区内分配存储单元，在程序整个运行期间都不释放。占用更多内存。
- 在下一次该函数调用时，该变量保留上一次函数调用结束时的值。降低程序可读性。
 - `static int i=3;` **i只是最初是3；如果函数f执行时i被修改为4，则下次运行f, i的初值为4。**
- 静态局部变量或全局变量未赋初值时，**编译时自动使之0。**
- 静态局部变量一直存在，但对其他函数 **“不可见”**。可见范围仍是局部
- 开发实践中，避免使用静态局部变量，多见于：试卷，与以往的程序兼容。

```
int f(int a)
{ int b=0;
  static int c=3;
  b=b+1;
  c=c+1;
  return a+b+c;
}
```

只赋一
次初值

cout <<f(0);输出: 5

0 + 1 + 4

cout <<f(1); 输出: 7

1 + 1 + 5

cout <<f(2); 输出: 9

2 + 1 + 6

变量c是静态局部变量，在内存一旦开辟空间，就不会释放，空间值一直保留，局部变量的赋值初始化只在函数被首次调用时执行一次。

- **作用域**：标识符（变量、函数、类）在哪个区间有效：可见、可使用
 - 局部作用域：
 - **全局作用域（物理文件）**：
 - 当前文件，甚至其他文件
 - 文件作用域：当前文件
 - **名空间作用域（逻辑）**：
 - 类作用域：类变量（函数），实例变量（函数） 面向对象的封装可见性原则

- 局部作用域：在函数内部或在块（复合语句）中定义的变量都是局部变量，其作用域开始于声明处，结束于函数/块的结尾处。
 - 不同的函数可以使用相同名字的局部变量，同名局部变量分属不同函数，互不干扰。
 - 局部变量名相同时，以最内层定义的局部变量为准。java中，不允许。
C++，尽量避免

```
void main(void)
{  int x=10;
  {  int x=20; cout<<x<<endl; }
  cout<<x<<endl;
}
```

20
10

```
18
19
20 int x = 1;
21 {
22     int x = 5;
    System.out.println(x);
}
```

- 在函数**外**定义的变量称为全局变量，对应文件作用域。
- 其缺省的作用范围是：从定义位置开始到该源程序文件结束。

```
int p=1, q=5;
float f1( int a)
{ int b,c;
  ....
}
char c1,c2;
main( )
{ int m, n;
  ....
}
```

全局变量 (points to `int p=1, q=5;`)

局部变量 (points to `int b,c;`)

a,b,c有效 (points to the scope of `f1`)

p,q有效 (points to the global scope)

m,n有效 (points to the scope of `main`)

c1,c2有效 (points to the scope of `main`)

- 全局变量与局部变量同名时，局部变量优先。可以利用域作用符::标记全局变量

```
#include <iostream.h>
int i= 100;
void main(void)
{
    int i , j=50;
    i=18;      //访问局部变量i
    ::i= ::i+4; //访问全局变量i
    j= ::i+i;   //访问全局变量i和局部变量j
    cout<<"::i="<<::i<<"\n";
    cout<<"i="<<i<<"\n";
    cout<<"j="<<j<<"\n";
}
```

::i=104

i=18

j=122



- 全局变量增加了函数间数据联系的渠道，允许多个函数都来读取，修改它
 - 下面例子中，多了一个全局变量min，优点是？缺点是？函数maxmin给了我们一个惊喜

```
int min;
int maxmin (int x, int y){
    min=(x<y)?x : y;
    return (x>y)? x : y ;
}

void main (void)
{
    int a,b;
    cin>>a>>b;
    cout<<"The max is"<<maxmin(a,b)<<endl;
    cout<<" The min  is"<<min<<endl;
}
```

全局变量

maxmin函数，事实上有两个返回值
但是，极大的破坏了程序的健壮性

- 尽量克制使用全局变量的冲动，特别是当你和别人合作时
 - 全局变量的罪状：
 - 存储角度：全局变量在程序的全部执行过程中都占用存储单元，而不是仅在需要时才开辟单元。
 - 维护角度：使用全局变量过多，会降低程序的清晰性。在各个函数执行时都可能改变全局变量的值，程序容易出错。
 - 可重用角度：如果将一个函数移到另一个文件中，要将依赖的全局变量及其值一起移过去。
 -
 - 一般要求把程序中的函数做成一个封闭体，只通过“实参——形参”的渠道与外界发生联系，这样的程序移植性好，可读性强。

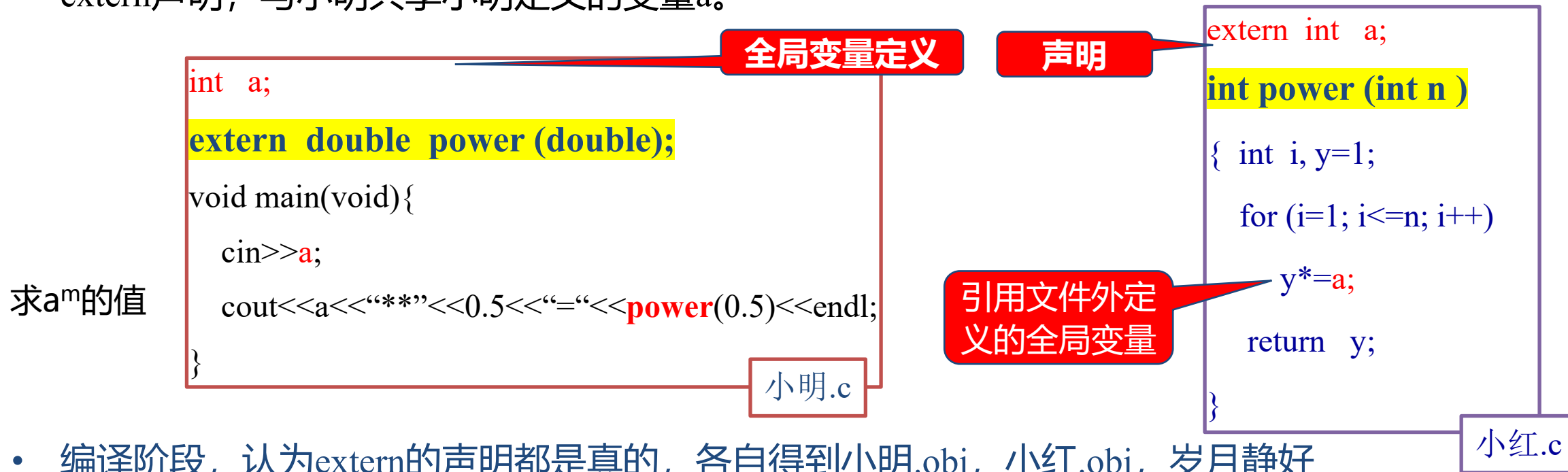
- 函数要访问当前源文件中定义的全局变量：
 - 通常在源文件前面定义全局变量，后面定义函数，函数可以访问全局变量
 - 如果在后面定义全局变量，需要在前面利用extern声明全局变量，声明之后可以使用
- 如果函数f要访问另一个文件中定义的全局变量x，则必须在f所在文件中添加x的extern声明

```
#include <iostream>
using namespace std;
int max(int,int); //函数声明
void main(){
    extern int a,b; //对全局变量a,b作提前引用声明，不是定义，不是定义，不是定义
    cout<<max(a,b)<<endl;
}

int a=15,b=-7; //定义全局变量a,b
int max(int x,int y){
    int z;
    z=x>y ? X:y;
    return z;
}
```

同一个文件中的情况

- 通常用extern声明全局变量是外部的，其他文件的。下面例子中，小明定义了变量a，小红通过extern声明，与小明共享小明定义的变量a。



- 编译阶段，认为extern的声明都是真的，各自得到小明.obj，小红.obj，岁月静好
- 链接阶段，兑现extern阶段，小明发现误解了小红power，并不能求解0.5次方，一拍两散
- 结论**，编译正确与链接正确还有很遥远的距离。
- 小红通过extern单方面共享小明的a，现在小明不想共享a了，他该怎么对小红说？



- static修饰全局变量：存储在静态存储区，在函数外部定义，只限在本文件中使用
- extern全局变量：存储在静态存储区，在其他文件中定义，在本文件中可以使用

```
int a;  
static int b;  
void m(void){  
    static int c;  
    int d;  
}
```

file1.c

```
extern int a;  
extern int b; // 错误  
extern int c; // 错误  
extern int d; // 错误
```

file2.c

另外：static局部变量：存储在静态存储区，在函数内部定义，只限在函数内部使用

- static 也可以修饰函数，描述其为内部函数
 - **内部函数**：函数只**局限**于在本文件中调用，其它文件不能调用，用**static** 定义该函数。
 - **外部函数**：是函数的默认形式，即可以被外部调用。
 - 在调用文件中用**extern 声明**，把该函数的作用域**扩展**到调用文件。

```
static float fac1( float n)
{ ..... }//该函数不能被其他文件使用
float fac(int n) {}
//该函数可以被其他文件使用
```

```
extern float fac( int n);
//这个函数的定义来自其他文件
```

关于cpp代码中的变量a,b,c,d, 下面说法正确包括:

- ☒ A 全局变量为a和b
- ☒ B 只有d是存放在栈空间中
- ☒ C 局部变量为c和d
- ☒ D 变量a,b,c的初始值均为0

```
int a;  
static int b;  
void m(void){  
    static int c;  
    int d;  
}
```

提交

file1.cpp中定义了变量a， File2.cpp中有声明extern int a，
会引发什么问题

- ☒ A 链接错误
- ☐ B 编译错误
- ☐ C 运行错误
- ☐ D 没有错误

```
static int a=3;
int main ( )
{
  |
}
```

file1.cpp

```
extern int a;
int fun (int n)
{
  |
  a=a*n;
  |
}
```

file2.cpp

提交

- 一个源文件中要用外部的 (extern) 的函数/全局变量如果有多个，会引发重定义错误，C++提供了命名空间 (namespace) 机制来解决名冲突/重定义问题。
 - 在一个命名空间中定义的全局标识符 (函数/全局变量)，其作用域为该命名空间。

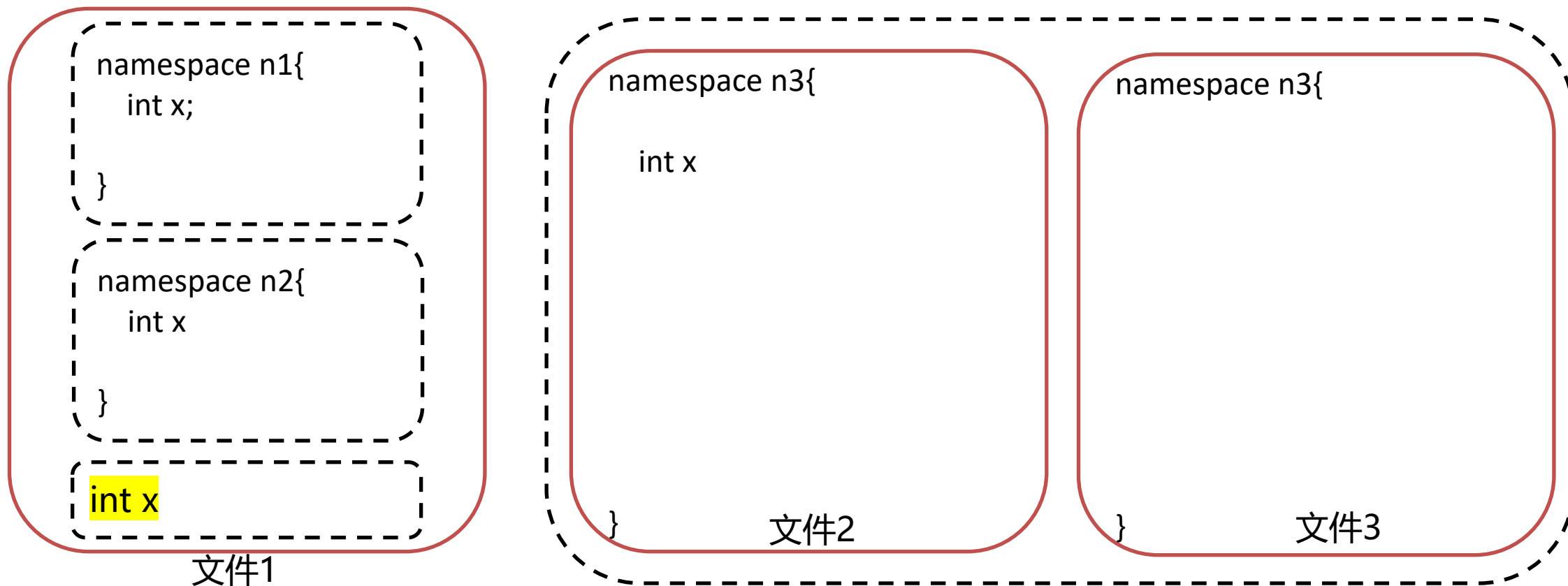
```
namespace A
{ int x=1;
  void f() { ..... }
}
```

```
namespace B
{ int x=0;
  void f() { ..... }
}
```

```
using namespace A;
... x ... //A中的x
f(); //A中的f
... B::x ... //B中的x
B::f(); //B中的f
```

```
using A::f;
... A::x ... //A中的x
f(); //A中的f
```

- 一个文件中可以定义多个命名空间，一个命名空间可以定义于多个文件中



- `n1::x` `n2::x` `n3::x` `::x`

- 1、基础-函数的定义与调用
递归函数
- 2、特殊函数-内联函数
- 3、特殊函数-重载与模板

- 4、变量存储类别、作用域和生命周期
- 5、多文件程序组织
- **6、预处理命令**

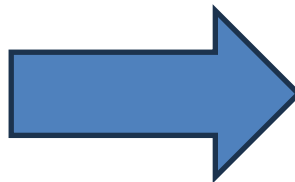
- **预处理编译**：源代码到目标代码的**编译之前的编译**，对源代码的处理
 - 预处理，编译，链接
- C/C++语言提供的编译预处理的功能有以下三种：
 - **文件包含** `#include`
 - 宏定义 `#define`
 - 条件编译 `#ifdef`
- **这些命令以 “#” 开头，末尾不包含分号**


```
#include <iostream>
#include "mathx.h"
using namespace std;
int main( ){
    ....., 可以直接使用max, 以及PI
}
```

```
// iostream
...
...
```

```
// mathx.h
...
...
```

预编译



```
// iostream
...
...

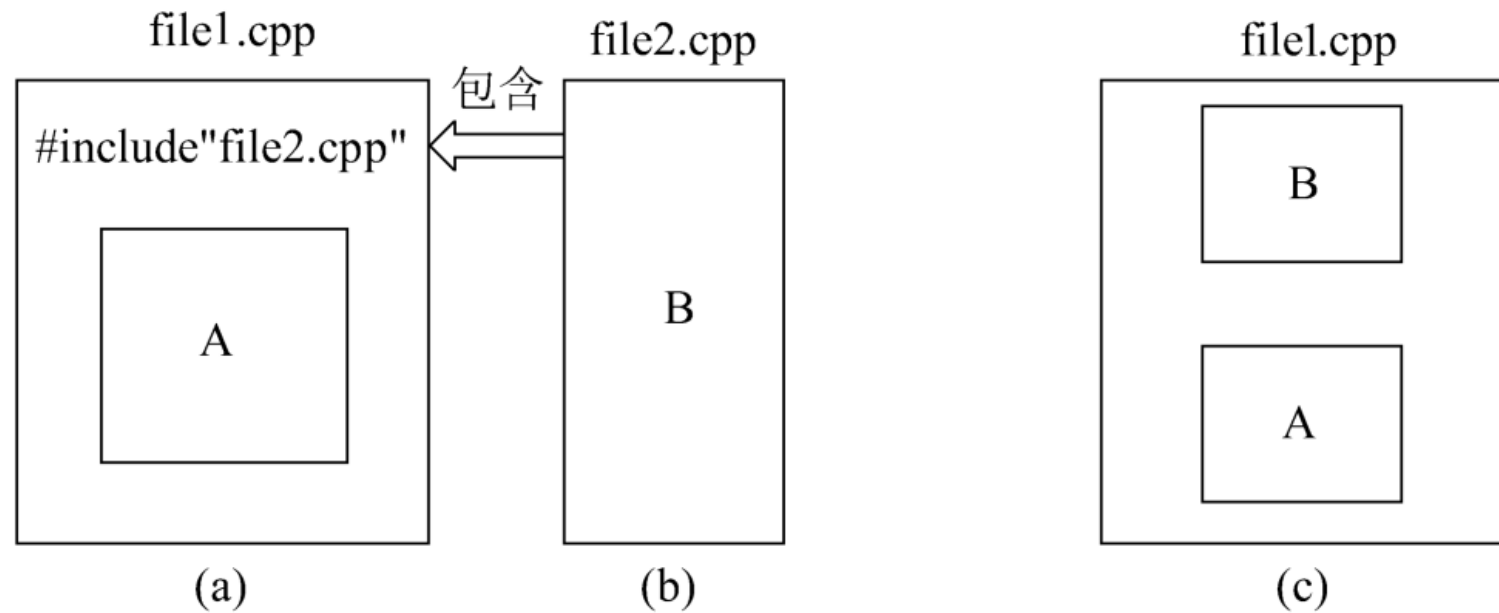
// mathx.h
...
...

using namespace std;
int main( ){
    ....., 可以直接使用max, 以及PI
}
```

暴力、简单、直接;

- 一个源文件也可以将另外一个源文件的全部内容包含进来，即将另外的文件包含到本文件之中。

include “文件名”



- 通常情况下，很少在一个cpp中include另一个cpp，而应当include其声明文件h，即包含file2的声明（引用），而不是其全部。（可以使用，但不拥有）

– 编译链接下面的file1.cpp，file2.cpp代码，会发生错误，myMath中的函数重定义

```
//File1.cpp  
#include "myMath.cpp"
```

```
//File2.cpp  
#include "myMath.cpp"
```

– 正确的做法：引入声明，而非定义，共同使用myMath.cpp中的定义

```
//File1.cpp  
#include "myMath.h"
```

```
//File2.cpp  
#include "myMath.h"
```

• C/C++ 标准套路:

- 头文件: 实现者与使用者之间的桥梁, 包括函数声明, 常量定义
- 函数库: 将函数在头文件中声明, 发布头文件源代码+库函数目标代码
- 使用者: 通过#include包含头文件, 知道如何调用, 不关心具体实现

```
#include <iostream>
#include "mathx.h"
using namespace std;
int main( ){
    ....., 可以直接使用max, 以及PI
}
```

file1.cpp (文件1)

```
const double PI = 3.14;
int max(int ,int );
double pow(double,double)
.....
```

```
#include "mathx.h"
int max(int x,int y){
    int z;
    z=x>y ? x : y;
    return z;
}
.....
```

mathx.h (文件2)

mathx.cpp (文件2)

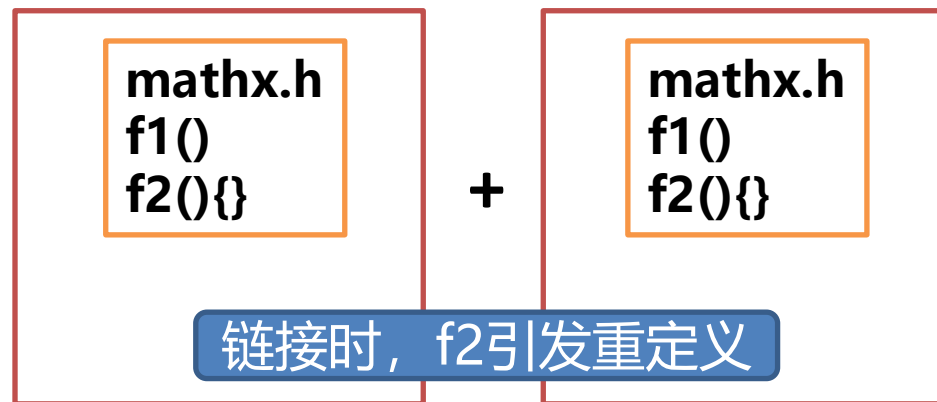
- 头文件（.h/.hpp header）一般包含以下几类内容：

- (1) 对类型的**声明**（struct/class还没讲到）
- (2) 函数**声明**
- (3) **内置(inline)**函数的**定义**。
- (4) **const**声明的**常变量**。用**#define**定义的宏
- (5) **不赋值**的外部变量声明。如**extern int a;**
- (6) 根据需要包含其他头文件。

- 由于重定义问题，头文件中**不要包含**
 - 函数的定义，例如f2(){}; 除非是inline函数
 - 全局变量的定义，例如double dd, 除非是const

mathx.h

```
#define G 9.8
const double PI = 3.14;
extern double aa; //在mathx.cpp中定义
void f1(); //在mathx.cpp中实现
inline void m(){}
double dd; //dd重定义
extern double cc = 1.0; //cc重定义，
//相当于double cc = 1.0; extern失去作用
void f2(){} //函数f会引发重定义
```



test1.h

```
int t1 = 1;
extern int t2;
void f1();
```

```
#include <iostream>
#include "test1.h"
using namespace std;
extern void f2();
extern int t4;
int main() {
    f1();    f2();
    cout << t2 << t3 << t4 << endl;
}
```

MainApp.cpp

```
#include <iostream>
#include "test1.h"
int t2 = 2;
int t3 = 3;
static int t4 = 4;
void f1() {}
void f2() {}
```

test1.cpp

关于左边代码，描述正确的选项包括

- ☒ A t1重定义导致链接错误
- ☒ B t3未定义导致编译错误
- ☒ C t4找不到导致链接错误
- ☒ D f1()规范， f2()没错但不规范
- ☒ E t2使用没有错误且规范

提交



- **#include <xx>** -到**系统目录**中寻找xx
 - 1.搜索-编译系统（例如gcc）自身的include目录，如果没找到
 - 2.搜索gcc的环境变量指定的CPLUS_INCLUDE_PATH
 - 3.最后搜索gcc的内定目录： /usr/include, /usr/local/include。
 - **不会搜索应用程序当前目录**
- **#include "myxx.h"** -先找**工程所在的（指定的）目录**，如果找不到，按照<myxx.h>定位
- 文件名可以包含目录，例如 #include <GL/GL.h>，寻找GL目录中的GL.h
- **建议**
 - 对于系统提供的头文件， 用尖括号形式。用户自己的头文件，用双引号形式。
 - **找不到**会发生编译错误，**找多了**更糟糕（好多地方都有xx.h，最终include哪一个）。很多时候我们会备份代码，但也会造成版本混乱，高效版本管理是复杂工程面临的难题。

- C/C++系统函数包含在C/C++编译器（开发包）中。
 - 包括一些常用数学计算函数（如sqrt()、exp()等）、字符串处理函数、标准输入输出函数、容器（List/Map）等。
 - C++标准库中的头文件一般不再包括后缀.h，但为了兼容C，许多C++编译系统保留了C头文件，提供两种方式，效果基本一样。
 - #include <math.h> //C形式的头文件
 - #include <cmath> //C++形式的头文件
 - 建议尽量用符合C++标准的形式，即不用后缀.h的模式。
- 用户自己编写的头文件，一般用.h为后缀。

```
#include <algorithm>
#include <bitset>
#include <complex>
#include <deque>
#include <exception>
#include <fstream>
#include <functional>
#include <iomanip>
#include <ios>
#include <iosfwd>
#include <iostream>
#include <istream>
#include <iterator>
#include <limits>
#include <list>
#include <locale>
#include <map>
#include <memory>
#include <new>
#include <numeric>
#include <ostream>
#include <queue>
#include <set>
```

。 。 。 。 。



C++标准库提供了万能函数头，`#include <bits/stdc++.h>`，其中include了所有的其他标准头，pta做题（竞赛）用但实用工程尽量少用，主要原因是：

- A 太简洁，不如罗列很多头文件有气势
- B 引入大量没有用到的代码，会降低程序的编译、执行效率
- C 引入大量没有用到的函数，当前工程中函数名、变量名同名冲突的可能性大增，存在潜在的危险。

提交

- 宏 (Macro) : 一种替换

- 用一个指定的标识符来代表一个串, 以后凡在程序中碰到这个标识符的地方都用串来代替。
- 标识符称为宏名, 替代过程称为“宏展开”。
- 宏展开只是一个预编译阶段的“物理”替换, 不做语法检查, 不是一条语句, 如果加分号“;”, 分号也是宏替换的一部分。
- 代码中双引号内容不做置换。

define 标识符 字符串

```
#define PRICE 30
void main(void)
{   int num, total; /*定义变量*/
    num=10;
    total=PRICE*num;
    cout<<“PRICE=” <<total<<endl;
}
```

编译前用30替代
如果30; 则语法错误

不做置换

- 可以用已定义的宏名，进行层层置换。
- #define命令出现在函数的外面，其有效范围为定义处至本源文件结束。可以用# undef命令终止宏定义的作用域。

```
#define G 9.8
```

```
void main(void )
```

```
{.....}
```

```
# undef G
```

```
int max(int a,int b)
```

```
{..... }
```

```
# define R 3.0
```

```
# define PI 3.1415926
```

```
# define L 2*PI*R
```

层层置换

```
# define S PI*R*R
```

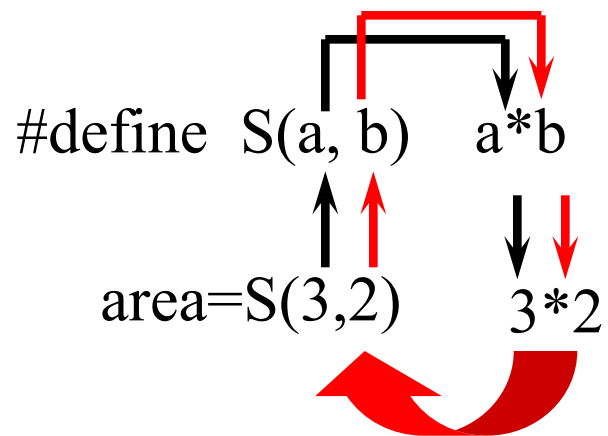
层层置换

```
void main(void)
```

```
{    cout<<"L="<<L<<" S="<<S<<endl;
```

```
}
```

- 带参数的宏，实现类似函数功能：形参用相应的实参代替，非形参保持不变。



S(a,b)等同于 a*b

S(3,2)等同于 3*2

宏名与括号间不能有空格

#define S (a,b) a*b

多了空格，CPP理解为：
将S替换为 (a,b) a*b

```
#define swap(x, y)\
```

```
x = x + y;\
```

```
y = x - y;\
```

```
x = x - y;
```

多行宏定义

swap(a,b);相当于

```
a = a + b;
```

```
b = a - b;
```

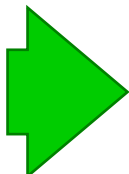
```
a = a - b;;
```

带参数宏的陷阱

```
#define PI 3.1415926
#define S(r) PI*r*r
void main(void)
{ float a, area, b;
  a=1; b=2;
  area=S(a+b);
  cout<<"r="<<a<<"\narea="<<area<<endl;
}
```

PI*a+b*a+b

```
#define S(r) PI*(r)*(r)
```

S(a+b)  PI*(a+b)*(a+b)

依次类推

```
#define MAX( a, b ) ( (a) > (b)? (a) : (b) )
```



- 带参数的宏与函数调用的相同之处
 - 有实参、形参，代入调用。
- 不同之处
 - 函数调用先求表达式的值，然后代入形参，而宏只是机械替换。
 - 函数调用时形参、实参进行类型定义，而宏不需要，只是作为字符串替代。#define swap(a,b)
 - 函数调用是在运行程序时进行的，其目标代码短，但程序执行时间长。而宏调用是在编译之前完成的，运行时已将代码替换进程序中，目标代码长，执行时间稍快。
 - 一般用宏表示短小的表达式。
- C++中，宏逐渐被inline，模板等替代

- C语言允许有选择地对程序的某一部分进行编译或跳过。作用，快速区分Debug版，Linux版。。。条件编译有以下几种形式：

1、 # ifdef 标识符

程序段1

else

程序段2

end if

2、 # ifndef 标识符

程序段1

else

程序段2

endif

define DEBUG

.....或者 #define DEBUG 1

ifdef DEBUG

cout<<x<<'\t'<<y<<endl;

endif

标识符

当标识符已被定义过（用 #define 定义），则对程序段1进行编译，否则编译程序段2.

调试完后去除
#define DEBUG,
则不输出调试信息。

- C语言允许有选择地对程序的某一部分进行编译。条件编译有以下几种形式：

```
# if  表达式1  
    程序段1  
  
# elif 表达式2  
    程序段2  
  
# else  
    程序段3  
  
# endif
```

当表达式1为真(非零)，编译程序段1，表达式2为真，编译程序段2，否则，编译程序段3。

```
#define DEBUG  
  
void main(void)  
{  int  a=14, b=15, c;  
   c=a/b;  
  
   # ifdef  DEBUG  
   cout<<"a="<<a<<" ,b="<<b<<endl;  
  
   # endif  
  
   cout<<"c="<<c<<endl;  
}
```

输出： a=14, b=15
c=0



- `#include <iostream>`
- `#include "myMath.h"`
- `#define PI 3.14`
- `#define Area(r) \`
 `- P *(r) * (r)`

`const double PI = 3.14;`

`inline double Area(double r)
{return PI * r * r;}`

逻辑上, 实质等效
编译器视角, 前者预编译 (正式编译时, 不存在预编译了)

```
# ifdef Win32  
    #define REAL float  
#else  
    #define REAL double  
# endif  
  
#define Win32  
  
void main() {  
    Real r = 10.0;  
    cout << Area(r);  
}
```

- C或者C++的头文件xx.h中，标准起手动作与结束动作

```
//xx.h  
  
#ifndef XX  
#define XX  
    ○ ○ ○  
    ○ ○ ○  
#endif
```

- 大家猜是做什么用的?
- 答案：小技巧，如果一个工程中多个cpp中#include xx.h，则只有第一次被include，之后的cpp不需要重复引入。

- C或者C++的头文件xx.h中，标准起手动作与结束动作的作用

```
//xx.h  
  
#ifndef XX  
#define XX  
  
    ○ ○ ○  
  
    ○ ○ ○  
#endif
```

- 小技巧，如果一个工程中多个cpp中#include xx.h，则只有第一次被include，之后的cpp不需要重复引入。

```
//a.cpp  
#include "xx.h"
```

```
//b.cpp  
#include "xx.h"
```



Next Chapter-数组与容器

- C/C++程序（工程）通常由多个源文件组成。
- 一个传统C的源文件，由一个或多个函数组成
- 一个C++源文件，可以由一个或多个类组成，类中包含函数/方法
- 可以执行的C/C++程序（工程）**必须有且只有一个**main()函数，
 - C/C++从main()函数开始执行和结束
 - main函数由OS(操作系统)调用
 - 有些例外，VC Win32程序对main函数进行了封装，winMain



- 1. 预处理 [预处理器cpp]
 - g++ -E c1-1.cpp > big.txt
- 2. 将预处理后的文件转换成汇编语言 [编译器egcs]
 - g++ -S c1-1.cpp
- 3. 由汇编**编译**目标代码(机器代码)生成.o的文件[汇编器as]
 - g++ -c c1-1.cpp
- 4. **连接**目标代码,生成可执行程序[链接器ld]
 - g++ -g c1-1.cpp -o c1-1.exe
- 为什么要知道这些,
 - 防止破译、代码优化、减少漏洞、安全防止被攻击
 - 好吧, 其实我也没接触, 听专家说的。

```
//#include <bits/stdc++.h>
#include <iostream>
using namespace std;
int main()
{
    char ch = -1;
    unsigned long long ss = ch;
    cout << "ss = " << hex << ss << endl;
    ss = long(-1);
    cout << "ss = " << hex << ss << endl;
    cout << sizeof(long long) << endl;

    system("pause");
    return 0;
}
```

cpp

```
18025 template<typename _CharT>
18026 int
18027 collate<_CharT>::_M_compare(const _CharT*, const _CharT*) const throw ()
18028 { return 0; }
18029
18030 template<typename _CharT>
18031 size_t
18032 collate<_CharT>::_M_transform(_CharT*, const _CharT*, size_t) const throw ()
18033 { return 0; }
18034
18035 template<typename _CharT>
18036 <_CharT>::
18037 are(const _CharT* __lo1, const _CharT* __hi1,
18038 st _CharT* __lo2, const _CharT* __hi2) const
18039 {
18040     string_type __one(__lo1, __hi1);
18041     string_type __two(__lo2, __hi2);
18042     return __one < __two;
18043 }
18044
18045 string_type __one(__lo1, __hi1);
18046 string_type __two(__lo2, __hi2);
18047 return __one < __two;
18048 }
```

Big.txt

```
289 call    _ZNSt8ios_base4InitC1Ev
290 leaq    __tcf_0(%rip), %rcx
291 call    atexit
292 .L20:
293 nop
294 addq    $32, %rsp
295 popq    %rbp
296 ret
297 .seh_endproc
298 .def     _GLOBAL__sub_I_main; .scl
299 .seh_proc _GLOBAL__sub_I_main
300 _GLOBAL__sub_I_main:
301 .LFB2052:
302 pushq   %rbp
303 .seh_pushreg %rbp
304 movq    %rsp, %rbp
305 .seh_setframe %rbp, 0
306 subq    $32, %rsp
307 .seh_stackalloc 32
308 .seh_endprologue
309 movl    $65535, %edx
```

.S