



中山大學  
SUN YAT-SEN UNIVERSITY



# 多核程序设计与实践

C/C++入门

陶 钧

[taoj23@mail.sysu.edu.cn](mailto:taoj23@mail.sysu.edu.cn)

中山大学 数据科学与计算机学院  
国家超级计算广州中心

- 概要
- 预处理器与宏
- 变量与函数
- 修饰词
- 指针与内存



## 发展

- C : 发行于1972年，面向过程，最新标准（C18）
- C++ : 发行于1983年，面向对象，最新标准（C++17）
- 相对JAVA等更接近于低级语言

## 编译语言

- 由编译器产生机器代码

## 弱类型语言

- 容忍隐式类型转换

## 基本控制流

- if, else, else if, switch
- do, while, for, break, continue

## 发展

- C : 发行于1972年，面向过程，最新标准（C18）
- C++ : 发行于1983年，面向对象，最新标准（C++17）

相对JAVA等更接近于低级语言

由编译器产生机器代码

弱类型语言

容忍隐式类型转换

"A programming language is **low level** when its programs require attention to **the irrelevant.**"

- Alan Perlis

## 基本控制流

- if, else, else if, switch
- do, while, for, break, continue

## ● 发展

- C : 发行于1972年，面向过程，最新标准（C18）
- C++ : 发行于1983年，面向对象，最新标准（C++17）
- 相对JAVA等更接近于低级语言

## ● 编译语言

- 由编译器产生机器代码

## ● 弱类型语言

- 容忍隐式类型转换

## ● 基本控制流

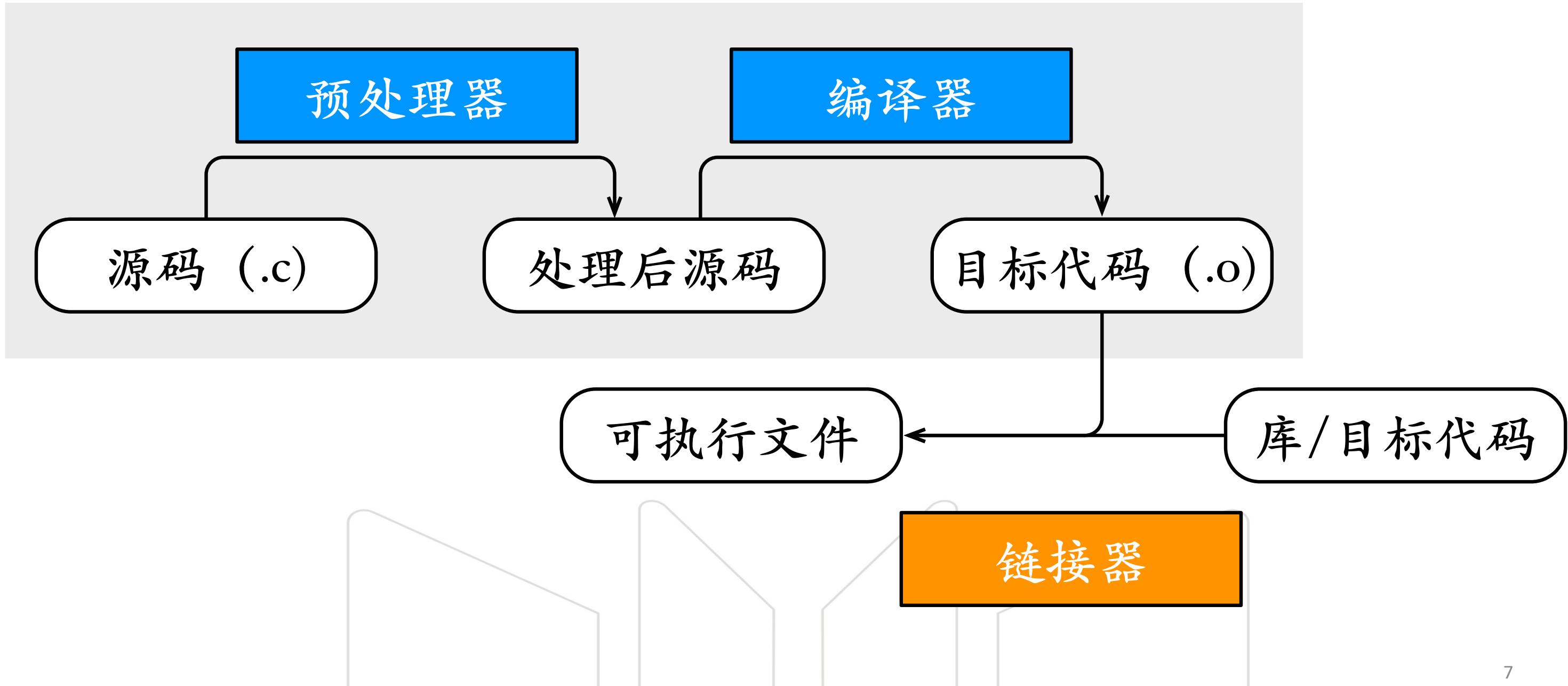
- if, else, else if, switch
- do, while, for, break, continue

## ● 编译语言

- C/C++
- 编译器将代码转换成机器指令
- 机器指令无法在不同架构上运行
- 通常执行速度更快

## ● 解释语言

- JAVA、Python
- 执行过程中由解释器转换成指令（但为实时优化提供可能性）
- 通常执行更慢
- 即时编译器技术 ( Just-in-Time ) : 实时编译部分代码



- 概要
- 预处理器与宏
- 变量与函数
- 修饰词
- 指针与内存



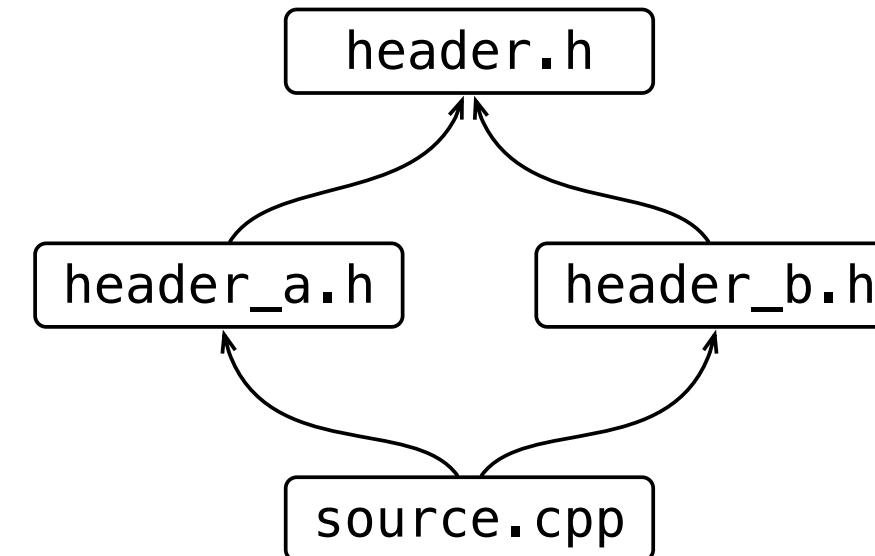
- 处理以“#”开头的预处理指令

- `#include <header.h>`
- 宏 ( macro )
  - 宏定义
    - 本质为文字替换
    - 常量宏定义 : `#define SOME_VALUE 1024`
    - 参数宏定义 : `#define ABS(n) ((n>0)?n:(-n))`
  - 条件编译
    - `#if, #elseif, #else, #endif`
    - `#ifdef, #ifndef, defined`

## ● 宏用途举例

- 例子1：避免头文件重复编译

```
#ifndef HEADER_H
#define HEADER_H
...
#endif //HEADER_H
```



## ● 宏用途举例

### - 例子2：根据平台编译

```
#if defined(__WINDOWS__) || defined (__WIN32)
|| defined (__WIN32__) || defined (__WIN64)
LARGE_INTEGER t;
QueryPerformanceCounter(&t);
#else
    struct timeval t;
gettimeofday(&t, NULL);
#endif
```



## ● 容易出错的宏

- 例子1：

```
#define ARRAY_A_SIZE      20 // 数组a大小为20
#define ARRAY_B_SIZE      10 // 数组b大小为10
// 数组a+b总大小
#define TOTAL_SIZE          ARRAY_A_SIZE+ARRAY_B_SIZE
// 数组中每个元素包含的浮点数数目
#define NUM_DIMENSION       2

// 定义array并分配内存空间
float* array = new float[TOTAL_SIZE*NUM_DIMENSION];
```

访问数组元素array[25\*NUM\_DIMENSION]报错！

## ● 容易出错的宏

- 例子1：

期望的代码：

```
#define ARRAY_A_SIZE      20 // 数组a大小为20
#define ARRAY_B_SIZE      10 // 数组b大小为10
#define TOTAL_SIZE         (ARRAY_A_SIZE+ARRAY_B_SIZE)
```

预处理后的代码：

```
#define NUM_DIMENSION    2
#define float* array = new float [20+10*2];
```

// 定义array并分配内存空间

```
float* array = new float [TOTAL_SIZE*NUM_DIMENSION];
```

访问数组元素array[25\*2]报错！

## ● 容易出错的宏

- 例子1：

```
#define ARRAY_A_SIZE      20 // 数组a大小为20
#define ARRAY_B_SIZE      10 // 数组b大小为10
// 数组a+b总大小
#define TOTAL_SIZE         ARRAY_A_SIZE+ARRAY_B_SIZE
// 数组A和B中的浮点数数目
#define NUM_DIMENSION      2
float* array = new float[(20+10)*2];

// 定义TOTAL_SIZE并分配内存空间
float* array = new float[TOTAL_SIZE*NUM_DIMENSION];
#define TOTAL_SIZE (ARRAY_A_SIZE+ARRAY_B_SIZE)

访问数组元素array[25*2]报错！
```

## 容易出错的宏

### - 例子2

```
#define ABS(n) ((n>0)?n:(-n))
```

...

```
ABS(++n);
```



- 容易出错的宏

- 例子2

```
#define ABS(n) ((n>0)?n:(-n))
```

...

```
ABS(++n);
```

调用宏前 : n=5

调用后期待结果 : n=6

调用后实际结果 : n=7

- 容易出错的宏

- 例子2

```
#define ABS(n) ((n>0)?n:(-n))
```

...

```
ABS(++n);
```

调用宏前 : n=5

调用后期待结果 : n=6

调用后实际结果 : n=7

实际执行代码 : ((++n>0)?++n:(-(++n)))

- 概要
- 预处理器与宏
- 变量与函数
- 修饰词
- 指针与内存



## ● 基本类型

- 字符类型: char (1)
- 整型: int (4) , short (2) , long (4) , long long (8)
- 浮点类型: float (4) , double (8)
- 实际长度依赖于具体编译器
  - 可用sizeof () 函数查看具体值

```
int a;  
sizeof(a); //4  
sizeof(int); //4
```

## ● 自定义类型

- struct, union, class

## ● 字符类型和整型的符号修饰词

- signed: 可表示正数和负数
- unsigned: 只可表示正数（范围是signed的两倍）

## ● 浮点数的二进制表示

- 单精度 (float)



符号

1 bit

指数

8 bits

有效数字

23 bits

0 00000111 11000000000000000000000000  
+ 7                   $0.5 + 0.25 = 0.75$

$$+1.75 \times 2^{(7-127)} = +1.316554 \times 10^{-36}$$

## ● 隐式类型转换 (implicit casting)

- 运算数类型不同时由编译器自动转换
- 基本类型遵循以下转换顺序从低向高转换
  - char < short < int < long < long long < float < double
  - 依照计算顺序依次转换

```
int a = 1, b = 2;  
float c = 0.5f;  
float d = (a+c)/b;  
float e = a/b+c/b;
```

d=e?



## ● 隐式类型转换 (implicit casting)

- 运算数类型不同时由编译器自动转换
- 基本类型遵循以下转换顺序从低向高转换
  - char < short < int < long < long long < float < double
  - 依照计算顺序依次转换

```
int a = 1, b = 2;  
float c = 0.5f;  
float d = (a+c)/b;  
float e = a/b+c/b;
```

d=e?

$$\begin{aligned}d &= (1+0.5f)/2 = (1.0f+0.5f)/2 = 1.5f/2.0f = 0.75f; \\e &= 1/2+0.5f/2 = 0+0.5f/2.0f = 0.0f+0.25f = 0.25f;\end{aligned}$$

- 隐式类型转换 (implicit casting)

- 运算数类型不同时由编译器自动转换
- 自定义类型使用构造函数进行转换

```
class String{  
public:  
    String(const char*);  
    ...  
}
```

```
String s = "some characters";
```

- 等价于

```
String s("some characters");
```

## ● 隐式类型转换 (implicit casting)

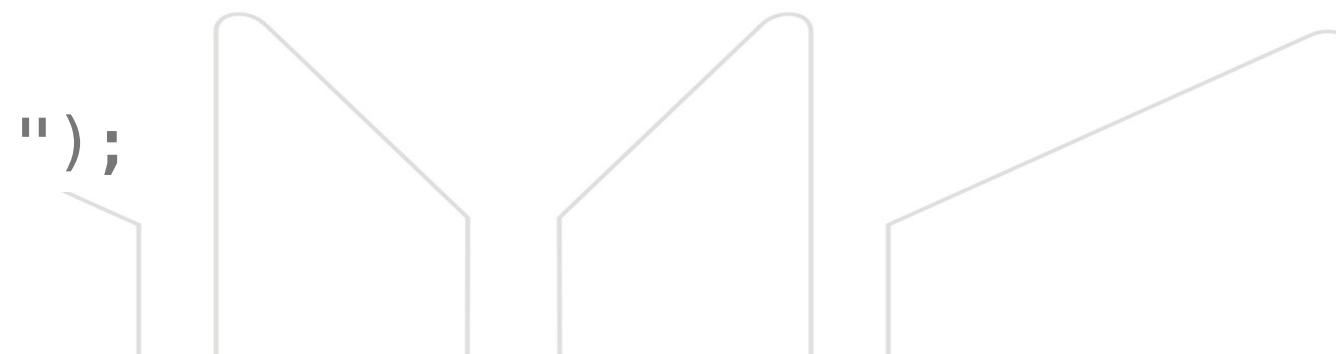
- 运算数类型不同时由编译器自动转换
- 自定义类型使用构造函数进行转换

```
class String{  
public:  
    String(const char*);  
    String(const int& n);  
    ...  
}
```

```
String s = 7;
```

- 等价于

```
String s("
```



## ● 隐式类型转换 (implicit casting)

- 运算数类型不同时由编译器自动转换
- 自定义类型使用构造函数进行转换

```
class String{  
public:  
    String(const char*);  
    String(const int& n);  
    ...  
}
```

```
String s = 'a';
```

- 等价于

```
String s((int)'a') -> String s(97);
```



## ● 隐式类型转换 (implicit casting)

- 运算数类型不同时由编译器自动转换
- 自定义类型使用构造函数进行转换

```
class String{  
public:  
    String(const char*);  
    explicit String(const int& n);  
    ...  
}
```

使用**explicit**禁止隐式类型转换

```
String s = 97;  
String s = 'a';
```

编译报错

## ● 强制类型转换 (explicit casting)

- 使用 (type) 运算符进行转换
- 从高精度向低精度转换时可能损失精度

```
(int) 1.23456 = 1
```

```
(char) 256 = 0
```

- 用途举例

```
int a = 1, b = 2;
```

```
float c = 0.5f;
```

```
float d = (a+c)/b;
```

```
float e = a/b+c/b;
```

```
float e = (float)a/b+c/b;
```

```
e = 1.0f/2+0.5f/2 = 0.5f+0.25f = 0.75f;
```

- 变量定义包括两层含义

- 指明数据位置
  - 表明如何理解数据

- 假设有 4 bytes 的数据: 00000011111000000000000000000000
    - float f :  $1.316554 \times 10^{-36}$
    - int i : 65011712



## 内联 (union)

- 内联的变量对应同一存储空间

```
union {
    int i;
    char c[4];
} INTEGER;
```

```
INTEGER value;
value.i = 0xdeadbeef;
swap(value.c[0], value.c[3]);
swap(value.c[1], value.c[2]);
```

```
value.i : 0xefbeadde;
```



## ● 函数定义

- 返回类型，函数名，参数表
- 返回类型为 **void** 时，不需要返回值

```
return_type function_name(optional-const argument_type argument_name)  
{  
    do_something;  
    return return_value or expression;  
}
```



## 变量作用域

### - 文件作用域与函数作用域

```
#include <cstdlib>
```

```
int a;  
  
int main(){  
    int b;  
    b = a;✓  
    b = c;✗  
}
```

```
int c;  
...
```

```
int a = 4;
```

```
void do_something(){  
    int a = 1;  
    ...  
}
```

## ● 变量作用域

### – 外部变量

- 文件外定义的变量

main.cpp

```
#include <cstdlib>

extern int a; // 外部变量声明

int main(){
    int b;
    b = a;
}
```

other.cpp

```
#include <cstdlib>

extern int a = 1; // 外部变量定义

void some_function(){
    ...
}
```

## ● 声明 (declaration)

- 声明向编译器表明变量（函数）的类型和名字

```
extern int a;  
int add(int a, int b);  
extern int add(int a, int b);
```

## ● 定义 (definition)

- 定义向链接器表明变量的实际存储空间或函数的具体实现

```
int a;  
extern int a = 1;  
int add(int a, int b){return (a+b);}  
extern int add(int a, int b){return (a+b);}
```

概要

预处理器与宏

变量与函数

修饰词

指针与内存



## ● const

- 表明该变量的值不可修改，否则编译将报错
  - 常用于全局常量定义或表明函数的参数内容不可修改

## ● volatile

- 表明程序每次读写该变量都需要直接对内存进行操作
  - 编译器不能对该变量读写进行优化（如缓存）
  - 为防止该变量被其他程序/设备修改导致数据不一致



## ● static

- 表明变量在程序运行中只定义一次

```
void static_inc()  
{  
    static int a = 1;  
    ++a;  
    printf("a = %d\n", a);  
}
```

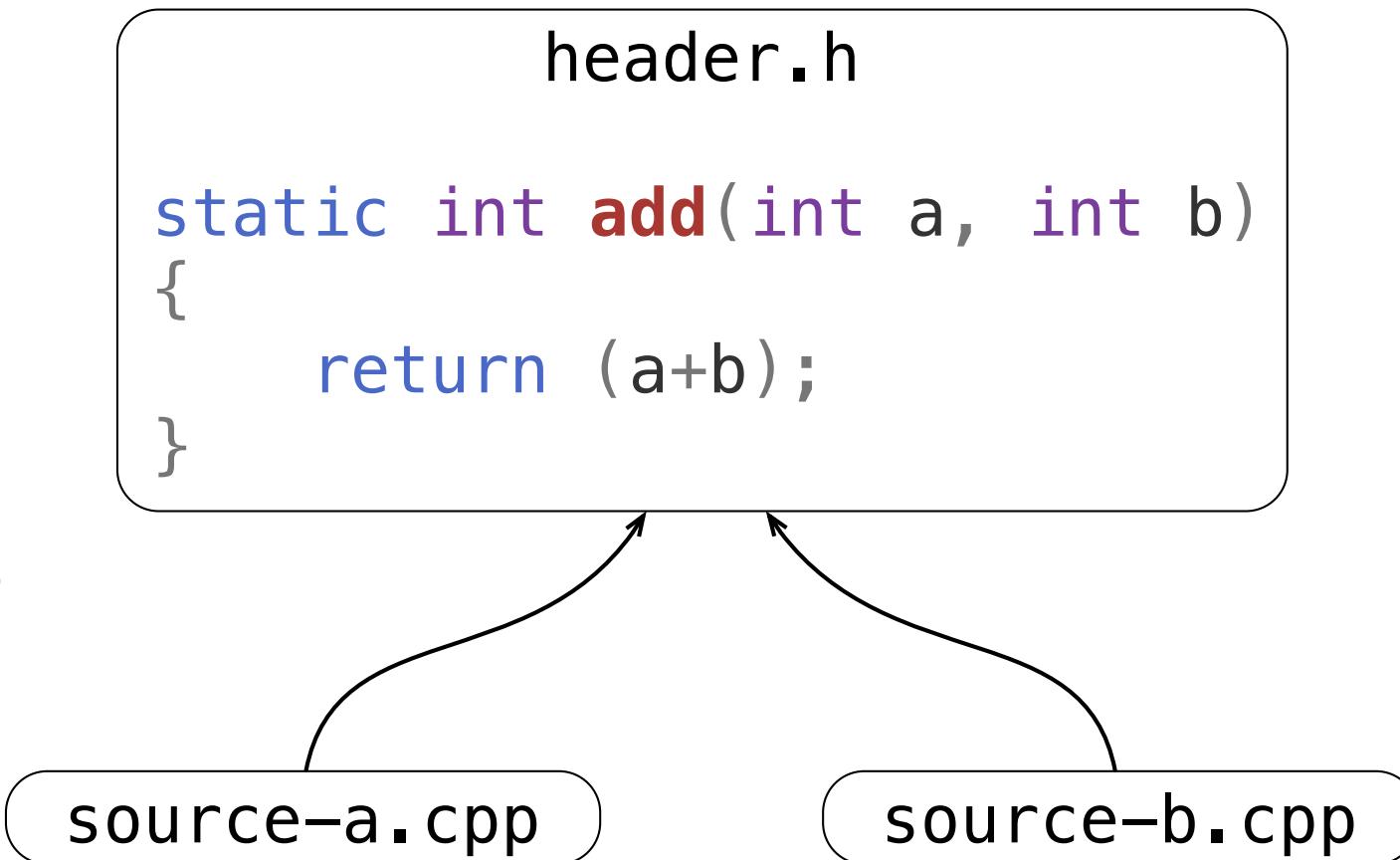
程序输出：

```
a = 2  
a = 3  
a = 4  
a = 5  
a = 6
```

```
for( int i = 0; i < 5; i++)  
{  
    static_inc();  
}
```

## ● static

- 表明函数只能被本文件使用
- 其他文件定义同名函数不会冲突
- 可用来在头文件中定义函数



- **inline**

- 表明函数代码将被置入调用行执行
- 避免实际调用函数产生的开销
  - 备份寄存器、分配栈空间、代码跳转等
- 常用于实现代码简短的函数
  - 否则将导致程序膨胀而降低缓存效率
- 添加在函数定义（而非声明）处
- 编译器可能自行决定**inline**某些函数

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



## ● inline

- 由编译器处理
- 参数中的语句只会被调用一次
- 可以作为类的成员函数
- 由 {} 决定函数体范围
- 需要匹配参数类型

## ● 宏

- 由预处理器处理
- 参数中的语句在宏中的每次出现都会被调用
- 不可以作为成员函数
- 换行表明宏定义结束
- 不需要匹配参数类型



- inline

- 由编译器处理
- 参数中的语句只会被调用一次
- 可以作为类的成员函数
- 由 {} 决定函数体范围
- 需要匹配参数类型

- 宏

- 由预处理器处理
- 参数中的语句在宏中的每次出现都会被调用
- 不可以作为成员函数
- 换行表明宏定义结束
- 不需要匹配参数类型

“Prefer **consts**, **enums**, and **inlines** to **#defines**.”

-Scott Meyers, Effective C++

## ● 函数模板

– 使用template关键字对不同类型数据创建通用函数

不使用模板：

```
int add(int a, int b){  
    return (a+b);  
}  
  
float add(float a, float b){  
    return (a+b);  
}  
  
double add(double a, double b){  
    return (a+b);  
}
```



## ● 函数模板

– 使用template关键字对不同类型数据创建通用函数

不使用模板：

```
int add(int a, int b){  
    return (a+b);  
}  
  
float add(float a, float b){  
    return (a+b);  
}  
  
double add(double a, double b){  
    return (a+b);  
}
```

使用模板：

```
template<typename T>  
T add(T a, T b){  
    return (a+b);  
}  
  
// 调用举例  
int a = add<int>(3, 5); // 8
```



- 概要
- 预处理器与宏
- 变量与函数
- 修饰词
- 指针与内存



## ● 指针

- 记录变量内存地址的变量
- 通过星号 (\*) 声明指针
  - `float* f;`
  - `char* c;`
- 通过取地址运算符 (&) 获得变量地址
  - `int i;`
  - `int* p = &i;`
- 不同类型的指针具有同样的大小
  - `sizeof(float*); //4 (32位程序)`
  - `sizeof(char*); //4 (32位程序)`

## ● 指针

- 可用星号 (\*) 取值
- 指针类型决定如何理解内存中的数据

0xaef034e5 : 00000011111000000000000000000000  
内存地址f   内存数据

```
float f_value = 1.316554e-36;  
float *f = &f_value;
```

```
int *i = (int*) f;//i指向f相同位置
```

```
int i_value = *i; //65011712  
int i_value_f = *f; //0
```



## 指针与数组

- 动态分配连续内存空间
- 具体运算的值依赖于指针类型

```
float* data = new float[1024];
```

```
char* c = (char*) data;
```

```
float* f = data;
```

c[0] //data中第 1 个 byte 对应的字符

f[0] //data中前 4 个 bytes 对应的浮点数

++c; // 指针向后移动 1 byte

++f; // 指针向后移动 4 bytes

## 指针数组

- 指向指针的指针
  - `int a[10][10];`
  - `int *b[10];`
- `a`是一个  $10 \times 10$  的二维数组
  - 程序将分配大小为100的连续空间
- `b`是一个长度为10的一维数组
  - 数组中每个元素为一个指针
  - 可分别为`b`中元素分配空间，大小不需要一致，空间上不一定连续

```
for(int i=0; i<10; ++i){  
    b[i] = new int[i+1];  
}
```

## 指针数组

- 分配空间连续的  $n \times m$  二维数组

```
int n = 10, m = 10;  
int **b, *b_data;
```

```
b_data = new int[n*m]; //为数据分配空间  
b = new int*[n]; //为一维指针分配空间
```

```
for(int i = 0; i < n; i++)  
{  
    //将b中每一个指针指向内存中相应位置  
    b[i] = &b_data[i*m];  
}
```



## 释放内存

- 用户需自行管理分配的内存
- 使用`delete[]`运算符和`free`函数

```
int *array = new int[100];
delete[] array;
```

```
int *array = (int*)malloc(sizeof(int)*100);
free(array);
```



## 指针用途举例

```
void smooth(float* org, float* ret, int n){  
    for(int i = 1; i < n-1; i++)  
    {  
        ret[i] = 0.5f*(org[i-1]+org[i]);  
    }  
}  
  
float* val = new float[10000000];  
float* smooth_val = new float[10000000];  
  
for(int i = 0; i < 128; i++)  
{  
    smooth(val, smooth_val, 10000000);  
    memcpy(val, smooth_val, sizeof(float)*10000000);  
}
```

## 指针用途举例

```
void smooth(float* org, float* ret, int n){  
    for(int i = 1; i < n-1; i++)  
    {  
        ret[i] = 0.5f*(org[i-1]+org[i]);  
    }  
}
```

```
float* val = new float[10000000];  
float* smooth_val = new float[10000000];  
  
for(int i = 0; i < 128; i++)  
{  
    smooth(val, smooth_val, 10000000);  
    swap(val, smooth_val);  
}
```

## 函数指针

- 声明函数指针

```
int (*func_ptr)(int ,int);
```

- 使用取地址运算符（&）函数获取函数地址

```
int add(int a, int b);  
int sub(int a, int b);
```

```
func_ptr = &add;  
func_ptr(5, 2); //7
```

```
func_ptr = &sub;  
func_ptr(5, 2); //3
```

## 函数指针用途举例

- 将数组中所有元素相加/相减

```
int add(int a, int b);
int sub(int a, int b);

void vector_if(int *a, int *b, int* c,
    int n, int mode)
{
    for(int i = 0; i < n; ++i){
        if(mode==0){
            c[i] = add(a[i], b[i]);
        } else {
            c[i] = sub(a[i], b[i]);
        }
    }
}
//调用
vector_if(a, b, c, n, 0);
```



## ● 函数指针用途举例

- 将数组中所有元素相加/相减

```
int add(int a, int b);
int sub(int a, int b);

void vector_if(int *a, int *b, int* c,
    int n, int mode)
{
    for(int i = 0; i < n; ++i){
        if(mode==0){
            c[i] = add(a[i], b[i]);
        } else {
            c[i] = sub(a[i], b[i]);
        }
    }
}

//调用
vector_if(a, b, c, n, 0);
```

```
void vector_fp(int *a, int *b, int* c,
    int n, int (*func_ptr)(int, int))
{
    for(int i=0; i<n; ++i){
        c[i] = func_ptr(a[i], b[i]);
    }
}

//调用
vector_fp(a, b, c, n, &add);
```

## 函数指针用途举例

- 将数组中所有元素相加/相减

```
int add(int a, int b);
int sub(int a, int b);
```

```
void vector_fp(int *a, int *b, int* c,
               int n, int (*func_ptr)(int, int))
{
    for(int i=0; i<n; ++i){
        c[i] = func_ptr(a[i], b[i]);
    }
}

//调用
vector_fp(a, b, c, n, &add);
```

```
#define VECTOR_OPERATOR add

void vector_macro(int *a, int *b,
                  int *c, int n)
{
    for(int i=0; i<n; ++n){
        c[i] = VECTOR_OPERATOR(a[i], b[i]);
    }
}

//调用
vector_macro(a, b, c, n);
```

# Questions?

