计算机图形学

C++简要回顾

陶钧
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

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

图形学常用编程语言

- C++：最流行
  - 常用于系统编程，高效
    - 大多数大型游戏都是基于C++开发
  - OpenGL支持最好
    - GLSL非常类似于C/C++
- Java：较流行
- HTML5, Javascript：新潮流
  - 随着网页编程流行而逐渐流行
- C#：新潮流
  - 随Unity而流行

C/C++概要

- 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`

C/C++概要

- 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`

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

- Alan Perlis

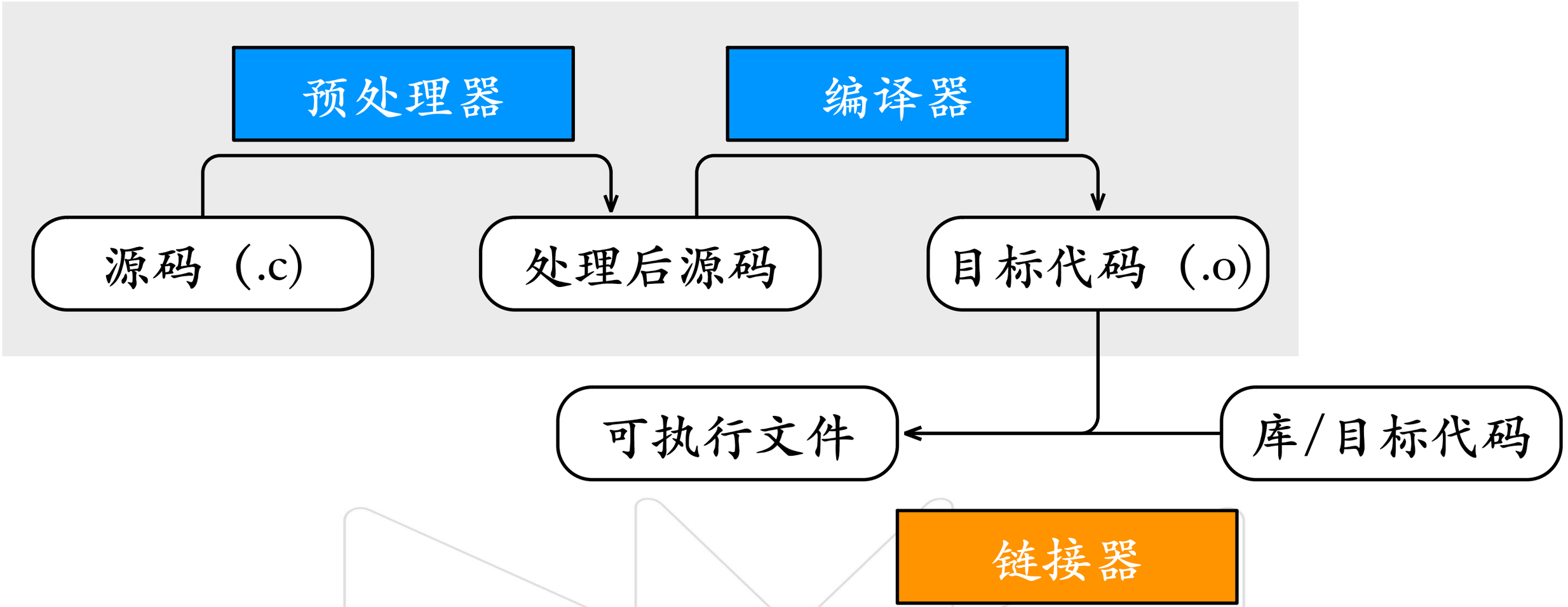
C/C++概要

- 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 vs 解释语言

- 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

C/C++程序编译流程



基本编程原则

- 编程应达到的目标
  - 鲁棒，简洁，高效，清晰易懂，可扩展，可测试，可移植
- 良好的编程习惯
  - 程序板式：缩进，换行，注释
  - 命名规则：清晰（看见名字能直观理解其含义）
    - https://en.wikipedia.org/wiki/Naming_convention_(programming)
  - 函数功能划分清晰
    - 功能尽可能单一，规模尽可能小
      - 1986年IBM在OS/360的研究结果：大多数有错误的函数都大于500行
      - 1991年对148,000行代码的研究表明：小于143行的函数比更长的函数更容易维护
      - debug往往比写程序需要的时间更长

C++简要回顾

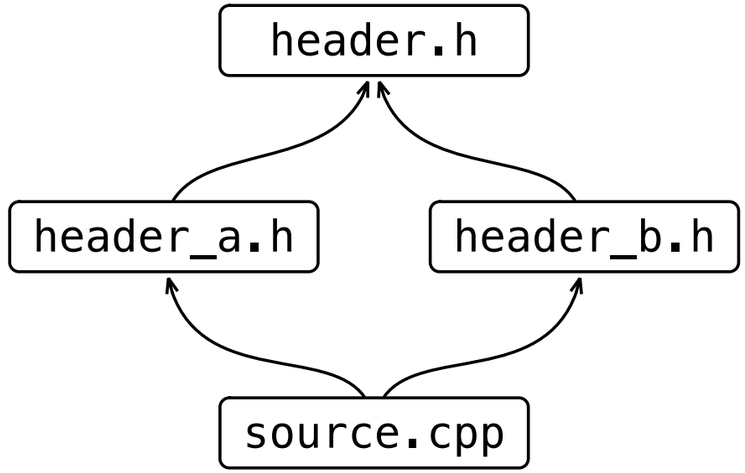
- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

预处理器

- 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
```



预处理器

- 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
```

预处理器

- 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！

预处理器

- 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*2]报错！
```

期望的代码：

```
float* array = new float[(20+10)*2];
```

预处理后的代码：

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

预处理器

- 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

//定义一维数组并分配内存空间
float* array = new float[TOTAL_SIZE*NUM_DIMENSION];
```

访问数组元素array[25*2]报错！

期望的代码：

```
float* array = new float[(20+10)*2];
```

预处理后的代码：

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

解决方案：

```
#define TOTAL_SIZE (ARRAY_A_SIZE+ARRAY_B_SIZE)
```

预处理器

- 容易出错的宏
  - 例子2

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

...

ABS(++n);
```

预处理器

- 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

预处理器

- 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)))

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

数据类型

- 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

数据类型

- 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

$$0\ 00000111\ 11000000000000000000000$$

$$+ \qquad 7 \qquad\qquad 0.5+0.25=0.75$$

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

数据类型

- C++中进行浮点运算时需要格外注意精度问题
  - 浮点型数据在图形学中最为常见
  - 精度损失举例

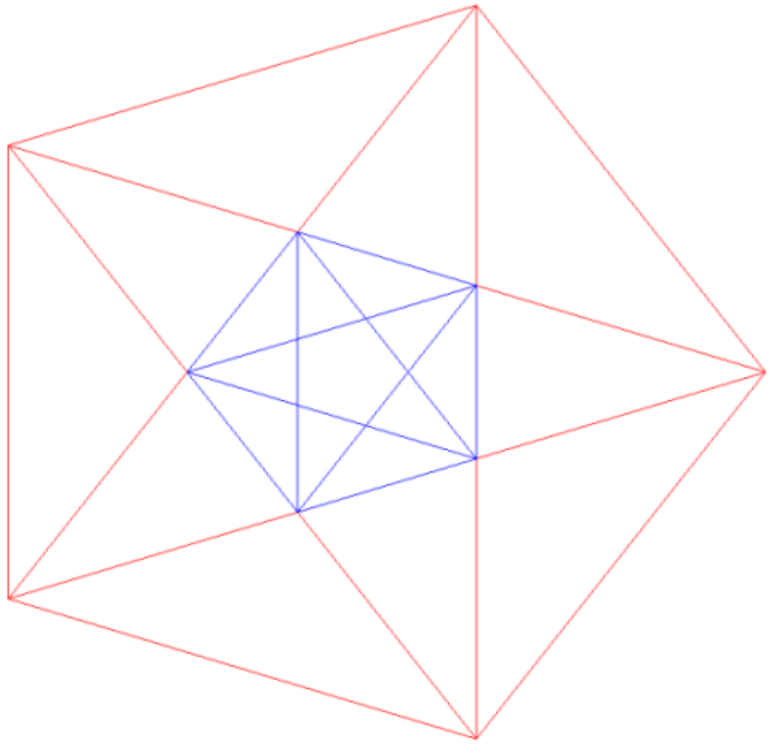
$$1.23456 \times 10^{30} + 9.87654 \times 10^{-30} - 1.23456 \times 10^{30}$$

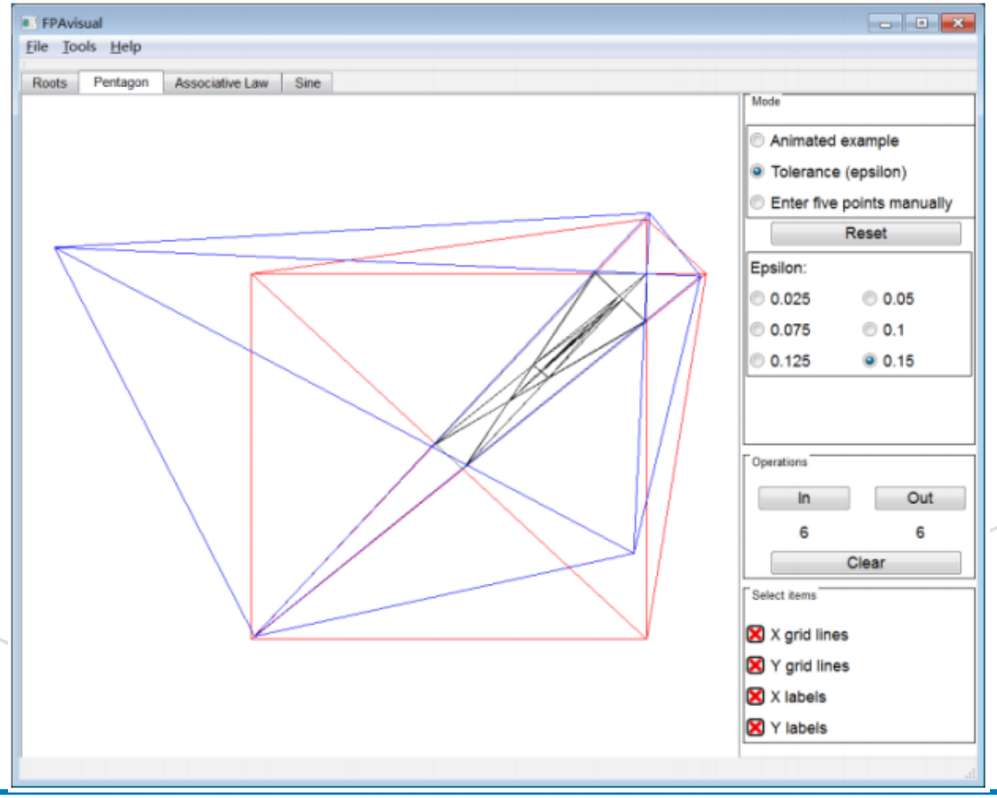
$$1.23456 \times 10^{30} - 1.23456 \times 10^{30} + 9.87654 \times 10^{-30}$$

  - 此外，在诸如求向量积（cross product），组合数等包含连乘运算的计算时，也要小心精度损失问题
    - $\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k(k-1)\ldots 1}$
  - 图形学中比较两个数是否相等时最为常用的语句为
    - if (**abs**(a-b)<e) 而非 if (a==b)

数据类型

- C++中进行浮点运算时需要格外注意精度问题
  - FPAvisual: Floating-Point Arithmetic Visualization
    - http://pages.mtu.edu/~shene/FPAvisual/
    - 举例：五边形的“in”与“out”操作





红色五边形经过6次“in”与6次“out”后得到的结果为蓝色五边形

类型转换

- 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?

类型转换

- 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?

```
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;
```

类型转换

- 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");
```

类型转换

- 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); //初始化字符串为n个空格
   ...
}

String s = 7;
```

- 等价于

```
String s("       ");
```

类型转换

- 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);
```

类型转换

- 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

类型转换

- 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;
```

类型与取值

- 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 : 1.316554×10^{-36}
    - int i : 65011712

类型与取值

- 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;
```

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

声明与定义

- 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);}
```

变量修饰词

- 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

变量修饰词

- 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);
}

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

程序输出：

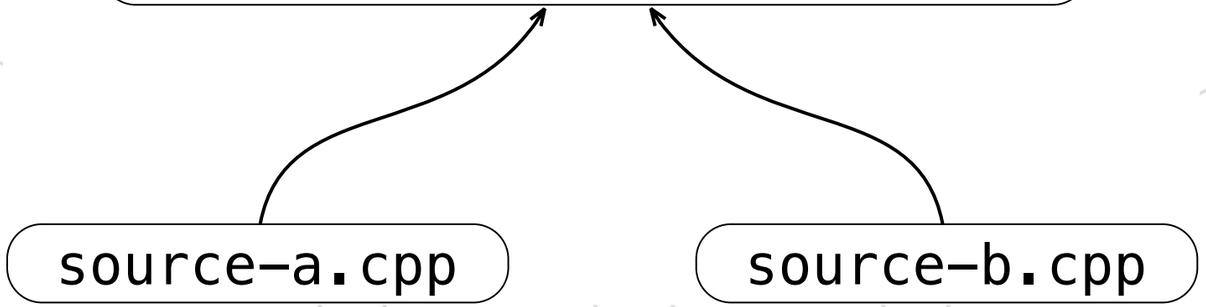
```
a = 2
a = 3
a = 4
a = 5
a = 6
```

函数修饰词

- 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

header.h

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



函数修饰词

- 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 vs 宏

- 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 vs 宏

- 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

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
```

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

指针与内存

- 指针
  - 记录变量内存地址的变量——在图形学编程中十分常见
  - 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)

指针与内存

- 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
```

指针与内存

- 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
```

指针与内存

指针数组

- 指向指针的指针
  - int a[10][10];
  - int *b[10];
- a是一个 10x10 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 x 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];
}
```

指针与内存

- 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);
```

  - 成对编码
    - 在写完分配内存（new/malloc）后，马上写释放内存代码（delete/free）
    - 最后在中间填上使用内存的代码

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);
}
```

指针与内存

- 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);
```

指针与内存

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);
```

- 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);
```

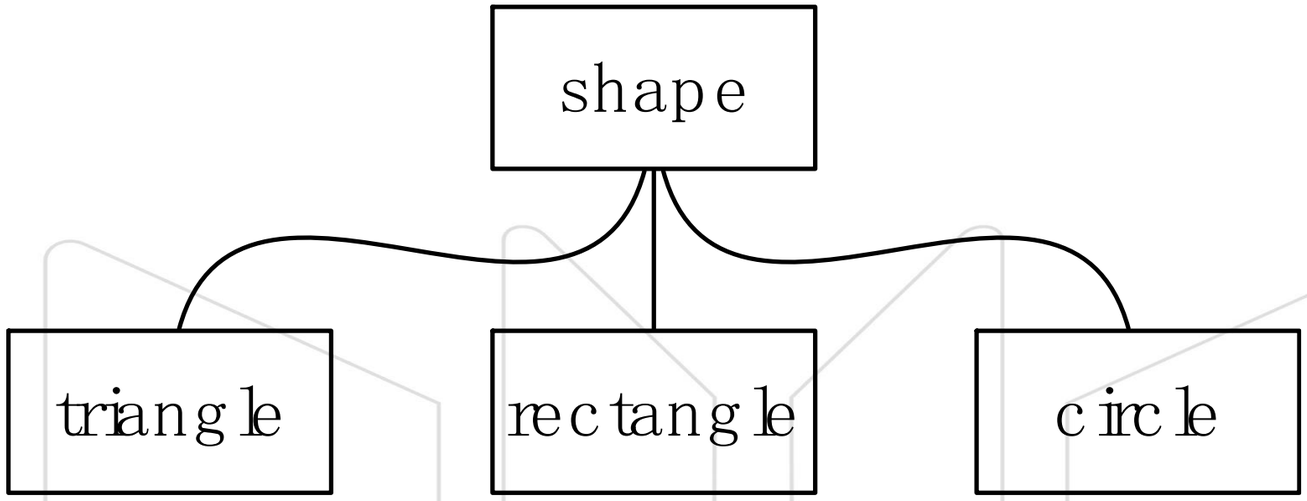
```
template<int mode>
void vector_template(int *a, int *b, int *c)
{
   for(int i=0; i<n; ++n){
      switch(mode){
         case 0: c[i] = add(a[i], b[i]);
               break;
         case 1: c[i] = sub(a[i], b[i]);
               break;
    }
}

//调用
vector_template<0>(a, b, c);
```

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

类、继承、多态

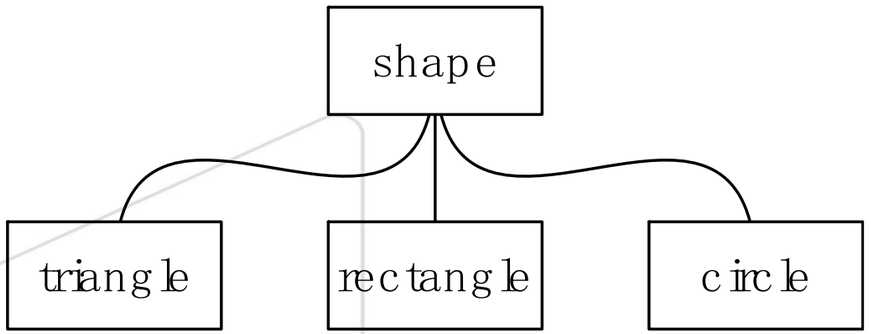
- 面向对象编程
  - 编写较大规模程序时尤为必要
    - 使代码模块化
    - 继承：实现代码重用
    - 派生：改造原有程序，解决其无法（完全）解决的问题
    - 多态：简化程序（同一操作对不同可以产生不同解释）
  - 举例



类、继承、多态

- 基类
  - 抽取共性方法、成员

```
class Shape {
public:
    Shape( ... ){}

    void draw(){
        //默认绘制方式
    }

    float area(){
        cout << "Shape area" << endl;
        return 0;
    }

protected:
    std::vector<vec2f> points;
};
```



类、继承、多态

- 派生类
  - 实现不同类型对象的具体功能

```
class Shape {
public:
   Shape( ... ){}

   void draw(){
      //默认绘制方式
   }

   float area(){
      cout << "Shape area" << endl;
      return 0;
   }

protected:
   std::vector<vec2f> points;
};
```

```
class Rectangle: public Shape{
public:
   Rectangle(...):Shape(...) { }

   void draw(){//画四边形}

   float area (){
      //长x宽
      cout << "Rectangle area:" ...
      return a;
   }
};

class Triangle: public Shape{
public:
   Triangle(...):Shape(...) { }

   void draw(){//画三角形}

   float area (){
      //海伦公式
      cout << "Triangle area: " ...
      return a;
   }
};
```

继承方式

类、继承、多态

- 应用举例：计算面积和

```
Triangle t1(...);
Triangle t2(...);
Rectangle r(...);
Circle c(...);


std::vector<Shape> all_objs;
//将t1, t2, r, c存入all_objs


float cal_area(){
    float sum = 0.0f;
    for (int i=0; i<all_objs.size(); ++i){
        sum += all_objs[i].area();
    }
}
```

应用举例：计算面积和

```
Triangle t1(...);
Triangle t2(...);
Rectangle r(...);
Circle c(...);


std::vector<Shape*> all_objs;
//将t1, t2, r, c存入all_objs


float cal_area(){
    float sum = 0.0f;
    for (int i=0; i<all_objs.size(); ++i){
        sum += all_objs[i]->area();
    }
}
```

输出：

```
Shape area
Shape area
Shape area
Shape area
```

错误原因：

早绑定：area()函数在编译期间已经设置好

更正方式：

基类函数声明更改为

```
virtual float area();
```

C++简要回顾

- 概要
- 预处理器与宏
- 数据类型
- 变量与函数
- 指针与内存
- 类、继承、多态
- 程序优化

- 基本原理
  - 了解你的程序
    - 哪部分耗费时间最多？每部分提升空间如何？
  - 性能优化
    - 改进算法、数据结构，降低时间复杂度
    - 在时间复杂度无法提高的情况下
      - 优化程序写法，减少不必要的运算，采用消耗较小的运算
      - 优化读写模式，充分利用缓存
    - 性能优化举例
      - 强度折减（Strength reduction）
      - 记录可重复使用的信息
      - 循环展开
      - 优化内存访问

强度折减

- 强度折减（strength reduction）
  - 将开销高的运算替换为开销较低的运算
  - 本小节内容整理自https://zhuanlan.zhihu.com/p/33638344
- 运算开销
  - comparison: 1 clock cycle
  - (u)int add, subtract, bitops, shift: 1 clock cycle
  - floating point add, sub: 3~6 clock cycles
  - indexed array access: cache effect
  - (u)int32 mul: 3~4 clock cycles
  - floating point mul: 4~8 clock cycles
  - floating point division: 14~45 clock cycles
  - (u)int division, remainder: 40~80 clock cycles

强度折减

类型转换

- int to short/char: 0~1 clock cycle
  - short/char通常使用32 bits存储，只是返回时进行截取
  - 只有在数组之类需要考虑内存大小时使用short/char
- int to float/double: 4~16 clock cycles
- 浮点型常量默认为双精度
  - 避免不必要的转换
  - 对于 float a, b 应该使用 a = b * 1.2 还是 a = b * 1.2f ？
- 使用乘法替代除法
  - 将常出现的除以常数替换成乘以其倒数
  - 对浮点型变量double y, a1, a2, b1, b2哪种写法更快？
    - y = a1/b1 + a2/b2
    - y = (a1*b2 + a2*b1) / (b1*b2)

记录可重复使用的信息

- 举例：计算Fibonacci数列
  - $f_0 = 1, f_1 = 1, f_i = f_{i-1} + f_{i-2}$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

```
int fibonacci(const int& i){
   if (i<2) {
      return 1;
   }
   return fibonacci(i-1)+fibonacci(i-2);
}
```

存在问题：重复计算

记录可重复使用的信息

- 举例：计算Fibonacci数列
  - $f_0 = 1, f_1 = 1, f_i = f_{i-1} + f_{i-2}$
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …

```
int fibonacci(const int& i){
   if (i<2) {
      return 1;
   }
   return fibonacci(i-1)+fibonacci(i-2);
}
```

输出斐波那契数列前40个数字：1. 412s

```
int fibo_data[100];

int fibonacci_mem(const int& i){
   if (fibo_data[i]>0){
      return fibo_data[i];
   }

   fibo_data[i] = fibonacci_mem(i-1)
                +fibonacci_mem(i-2);
   return fibo_data[i];
}
```

输出斐波那契数列前40个数字：0. 000s

空间换取时间

优化语句

- 展开循环（loop unrolling）
  - 使用编译器优化
    - gcc –funroll-loops
    - 某些编译器的-O3级别优化同样会展开循环
  - 手动展开

```
for(int i=0; i<n; ++i){
   do_something(i);
}
```

```
for(int i=0; i<n; ++i){
   do_something(i); ++i;
   do_something(i); ++i;
   do_something(i); ++i;
   do_something(i);
}
```

优化语句

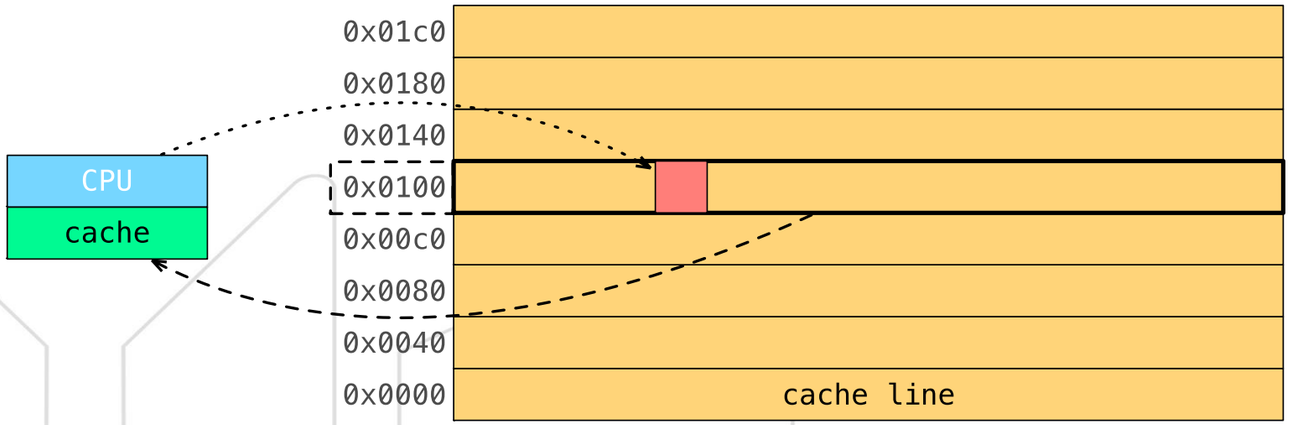
- 展开循环（loop unrolling）
  - 手动展开同时简化逻辑

```
for (int i=0; i<n; ++i){
    if (i%2==0){
        a(i);
    } else {
        b(i);
    }
    c(i);
}
```

```
for (int i=0; i<n; i+=2){
    a(i);
    c(i);
    b(i+1);
    c(i+1);
}
```

优化内存访问

- 对齐与合并访问
  - cache line: CPU每次从内存中缓存的数据大小
    - 通常为64B（字节）
    - 从64B的倍数起
  - 每次访问内存时，将加载包含该内存地址的一个cache line
    - 此后对同一cache line的访问将通过缓存进行



优化内存访问

- 对齐与合并访问
  - cache line意味着访问连续内存空间的重要性
    - C/C++中，数组为row-major（每行在内存中连续）
  - 试比较以下代码：

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[i][j];
    }
}
```

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[j][i];
    }
}
```

优化内存访问

- 对齐与合并访问
  - cache line意味着访问连续内存空间的重要性
    - C/C++中，数组为row-major（每行在内存中连续）
  - 试比较以下代码：

```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[i][j];
    }
}
```

运行时间：5. 414秒

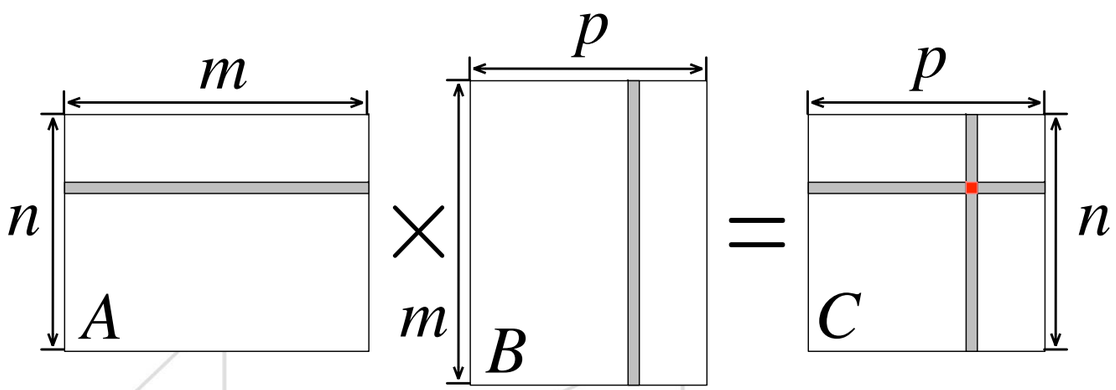
```
#define N 50000
int mat[N][N];

for(int i=0; i<N; ++i){
    for (int j=0; j<N; ++j){
        sum += mat[j][i];
    }
}
```

运行时间：35. 708秒

优化内存访问

- 对齐与合并访问：CPU
  - cache line意味着访问连续内存空间的重要性
    - C/C++中，数组为row-major（每行在内存中连续）
  - 在矩阵乘法中，常见写法容易不自觉地使用column-major的访问模式

```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
   for(int j = 0; j<N; ++j)
      for(int k = 0; k < N; ++k)
         C[i][j] += A[i][k]*B[k][j];
```



优化内存访问

- 对齐与合并访问：CPU
  - cache line意味着访问连续内存空间的重要性
    - C/C++中，数组为row-major（每行在内存中连续）
  - 在矩阵乘法中，常见写法容易不自觉地使用column-major的访问模式

```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
   for(int j = 0; j<N; ++j)
      for(int k = 0; k < N; ++k)
         C[i][j] += A[i][k]*B[k][j];
```

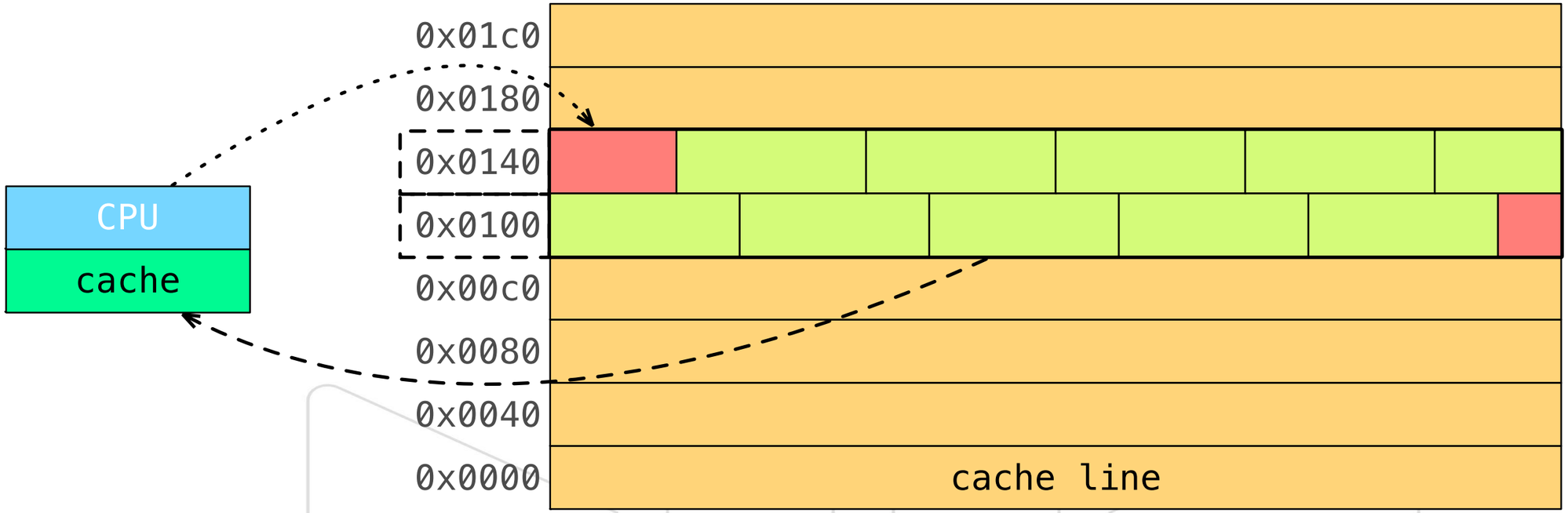
运行时间：231. 348秒

```
#define N 3000
int A[N][N], B[N][N], C[N][N];

for(int i=0; i<N; ++i)
   for(int k = 0; k<N; ++k)
      for(int j = 0; j<N; ++j)
         C[i][j] += A[i][k]*B[k][j];
```

运行时间：72. 557秒

优化内存访问

- 对齐与合并访问
  - 非对齐存储可能导致对一个数据的读取需要载入两个cache lines
    - 使用__align__(n)强制对齐
    - n必须为2的幂次方



参考

- C程序优化
  - Michael E. Lee, Optimization of Computer Programs in C
    - http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html
  - Lucas A. Wilson, Serial & Vector Optimization
    - https://portal.tacc.utexas.edu/documents/13601/1041435/06-Serial_and_Vector_Optimization.pdf/4eef1e1c-7592-4ac4-8608-1f0662553a88
  - ali4846j3o, C++性能优化（知乎专栏）
    - https://zhuanlan.zhihu.com/p/33638344

小结

- C++概要
  - 编译语言，执行效率高
  - 当前计算机图形学中最常见的语言
    - 在大型游戏、动画尤为常用
- 预处理器与宏
  - 本质为文字替换，使用灵活，但不安全易出错
- 数据类型
  - 类型声明决定如何理解二进制数据
  - 对浮点型数据的计算要尤其注意精度问题
- 指针与内存
  - 活用指针能极大提高程序效率，但要注意内存管理
- 类、继承、多态
  - 程序模块化的关键：基类提供共用属性、方法、接口；派生类实现多态
- 性能优化举例
  - 了解程序瓶颈所在，注重程序写法与实际运算、内存访问模式之间的关系

Questions?