



# 程序性能调优

徐枫

清华大学软件学院

fengxu@tsinghua.edu.cn

# C++ 性能优化策略

## ■ 优化算法

- 如果能够将复杂度为 $O(n^3)$ 的算法改为复杂度为 $O(n \log n)$ 的代码，那么将最有效的提高程序的性能

## ■ 优化代码

- 在数学上等价的方法，在计算机上运行起来，很多时候并不等价。这时候就需要我们了解计算机的工作机制，对代码进行优化调整。

## ■ 使用性能库

- 对于很多特殊的问题，有一些人专门编写了针对这个问题的性能库，能够对这些特定问题实现高速求解。比如一些有名的数学库。

# 什么是算法

- 算法是用于实现特定任务的有限步骤集合
- 算法代表着用系统的方法描述解决问题的策略机制. 也就是说, 能够对一定规范的输入, 在有限时间内获得所要求的输出.
- 如果一个算法有缺陷, 或不适合于某个问题, 执行这个算法将不会解决这个问题.
- 不同的算法可能用不同的时间、空间或效率来完成同样的任务.

# 算法的特点

---

- 一个算法应该具有以下五个重要的特征:
  - 有穷性
  - 确切性
  - 输入
  - 输出
  - 可行性(有效性)
  - 高效性
  - 健壮性

# 检测算法

---

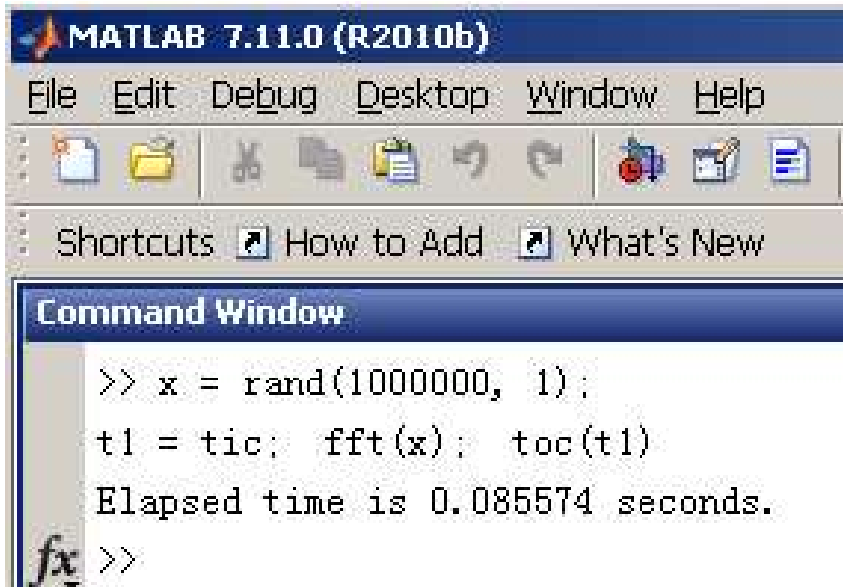
- 算法的**效率**用算法使用的处理器时间和内存空间来测量
- 有必要调整算法以**最佳**的方式使用可用资源, 如处理器时间和内存
- 要分析算法的复杂度, 理解**时间和空间**复杂度十分重要

# 算法时间复杂度的度量

- 算法执行时间需要通过依据该算法编制的程序在计算机上运行时所消耗的时间来度量. 而度量一个程序的执行时间通常有两种方法:
  - 事后统计的方法
  - 事先分析估算的方法

# 事后的统计方法

- 可以利用计算机的内部计时功能, 先把程序编写好运行一下进行计时. 不过这种方法有两个缺陷:
  - 必须先编制好程序并运行
  - 所得出的时间统计量依赖于计算机的软硬件等环境因素, 有时容易掩盖算法本身的优劣性

A screenshot of the MATLAB 7.11.0 (R2010b) Command Window. The window title is "MATLAB 7.11.0 (R2010b)". The menu bar includes "File", "Edit", "Debug", "Desktop", "Window", and "Help". The toolbar contains icons for file operations and execution. The Command Window shows the following text: ">> x = rand(1000000, 1);", "t1 = tic; fft(x); toc(t1)", "Elapsed time is 0.085574 seconds.", and "fx >>".

```
MATLAB 7.11.0 (R2010b)
File Edit Debug Desktop Window Help
Shortcuts How to Add What's New
Command Window
>> x = rand(1000000, 1);
t1 = tic; fft(x); toc(t1)
Elapsed time is 0.085574 seconds.
fx >>
```

- C++获得时间
  - **#include "time.h"**
  - **Clock () 函数**
- C++进行算法时间效率分析的工具
  - **Visual studio analyze上的 profiler**



# Profiler性能分析

Profiler是Visual Studio中提供的性能分析工具

可以在“调试”或“分析”中选择“性能探查器”来打开







# Profiler性能分析

Profiler能够对CPU、GPU、内存等多方面进行性能分析

**分析目标**

启动项目  
img\_pro

更改目标 ▼

---

**可用工具**

☐ CPU 使用率  
查看 CPU 执行代码时的时间耗费情况。当 CPU 遇到性能瓶颈时很有用

☐ GPU 使用情况  
检查 DirectX 应用程序中的 GPU 使用情况。这有助于确定性能瓶颈是 CPU 还是 GPU

☐ 内存使用率  
检查应用程序内存以查找内存泄漏等问题

☒ 性能向导  
CPU 采样、检测、.NET 内存分配和资源争用

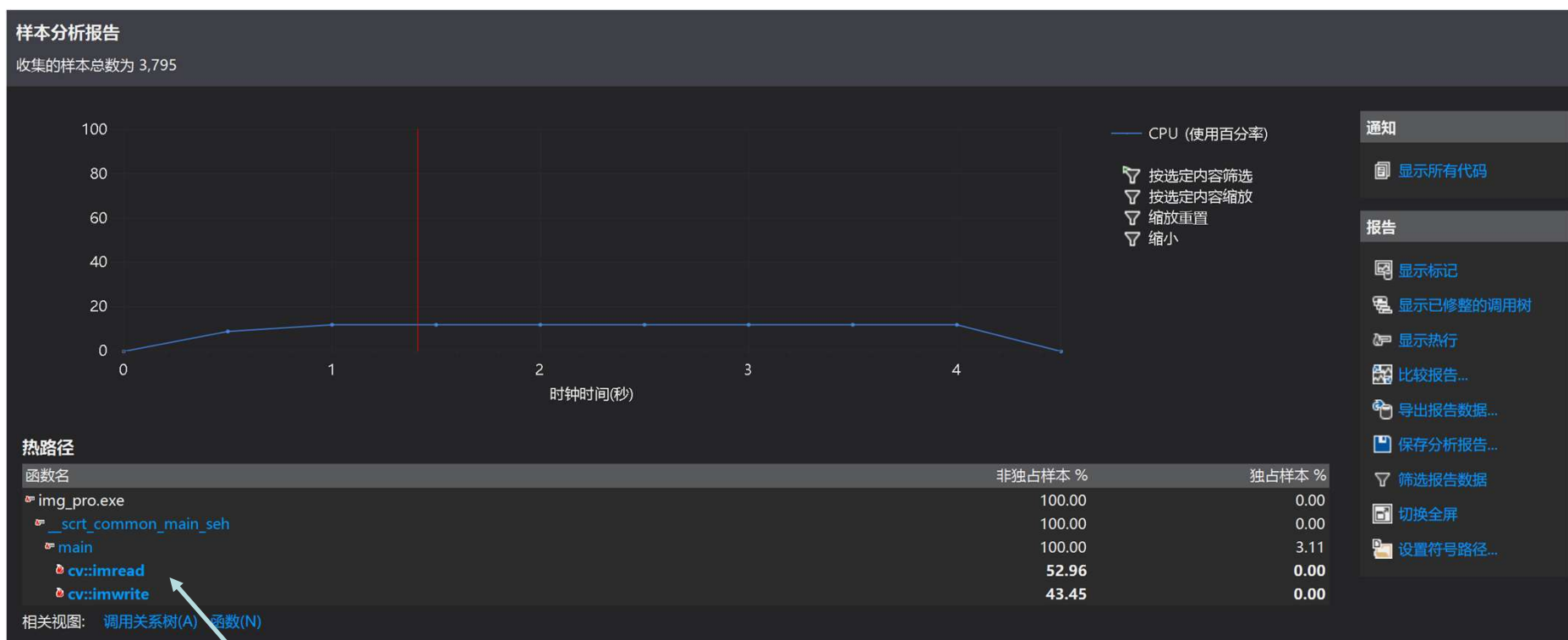
开始

选择要进行的项目，  
再点击“开始”



# Profiler性能分析

在程序运行之后会给出性能分析报告



可以看到图像读写的IO操作是性能瓶颈，  
提升性能要注重这部分

# 事先分析估计的方法

- 依据算法选用的策略
- 问题的规模 -例如求100以内还是10000以内的素数
- **书写程序的语言 -对于同一个算法, 实现语言的级别越高, 执行效率就越低**
- **编译程序产生的机器代码的质量**
- **机器执行指令的速度**

# 算法效率的度量

- 显然, 同一个算法用不同的语言实现, 或者用不同的编译器进行编译, 或者在不同的计算机上运行时, 效率均不相同
- 这表明使用绝对的时间单位衡量算法的效率是不合适的. 撇开这些与计算机硬件、软件有关的因素, 可以认为一个特定算法 “运行工作量” 的大小, 只依赖于问题的规模(通常用整数量 $n$ 表示), 或者说, 它是问题规模的函数 $f(n)$

# 算法时间复杂度

- 一个算法是由**控制结构**(顺序、分支和循环)和**原操作**(指固有数据类型的操作)构成的, 则算法时间取决于两者的综合效果
- 为了便于比较同一问题的不同算法, 通常的做法是, 从算法中选取一种对于所研究的问题(或算法类型)来说是**基本运算的原操作**, 以该基本操作重复执行的**次数**作为算法的时间量度

# 确定时间复杂度

- 计算算法的时间复杂度需要计算算法中包含的**步骤**的所需时间
- 在**最差**情况下执行所有步骤所需的时间是**默认**情况下所谓的算法的时间复杂度
- 步骤的执行取决于算法中指定的**条件**. 因此, 可以根据算法中涉及的条件来测量算法的时间复杂度

# 确定时间复杂度

- 计算斐波纳序列的第n个数:
- Step 1: start
- Step 2: input the value of n
- Step 3:     **if** ( $n \leq 1$ ) then go to step 14
- Step 4:              $x = 0$
- Step 5:              $y = 1$
- Step 6:     write( $x + " " + y$ )
- Step 7:     **for** ( $i = 0$  to  $n-2$ )
- Step 8:              $\{f = y + x$
- Step 9:              $x = y$
- Step 10:             $y = f$
- Step 11:             $i = i + 1$
- Step 12: write ( $f$ ) }
- Step 13: go to step 15
- Step 14: write ( $n$ )
- Step 15: stop

# 确定时间复杂度

- 根据 $n$ 的值, 在上述算法中可能存在两种情况
- 如果 $n$ 的值小于或等于 1, 则时间复杂度为常量并且不取决于输入( $n$ ), 如下表所示:

语句	执行频率
Step 2: input the value of $n$	1
Step 3: if ( $n \leq 1$ ) then go to step 14	1
Step 14: write ( $n$ )	1
执行的指令总数	3



# 确定时间复杂度

如果  $n$  的值大于 1, 则时间复杂度为  $4n+2$ , 如下表所示

语句	执行频率
Step 2: <i>input the value of <math>n</math></i>	1
Step 3: <i>if (<math>n \leq 1</math>) then go to step 14</i>	1
Step 4: $x = 0$	1
Step 5: $y = 1$	1
Step 6: <i>write(<math>x + \text{ " " } + y</math>)</i>	1
Step 7: <i>for <math>i = 0</math> to <math>n-2</math> repeat steps 8 to 11</i>	1
Step 8: $f = y + x$	$n-1$
Step 9: $x = y$	$n-1$
Step 10: $y = f$	$n-1$
Step 11: $i = i + 1$	$n-1$
Step 12: <i>write (<math>f</math>)</i>	1
执行的指令总数	$4n+2$

# 确定时间复杂度

- 计算两个矩阵之和:
- Step 1: start 1
- Step 2: for  $i = 0$  to  $m-1$  repeat steps 3 to 6
- Step 3:  $i = i + 1$  **m**
- Step 4: for  $j = 0$  to  $n-1$  repeat steps 5 to 6
- Step 5:  $c[i, j] = a[i, j] + b[i, j]$  **mn**
- Step 6:  $j = j+1$  **mn**
- Step 7: Step 15: stop

执行频率高的语句放在内循环

# 空间复杂度

- 是程序运行所需要消耗的存储空间
- 一般的递归算法要有 $O(n)$ 的空间复杂度
  - 简单说就是递归运算时通常是反复调用同一个方法, 递归 $n$ 次, 就需要 $n$ 个空间
  - 这个空间到底多大? 我们姑且把它当作每次调用时分配的内存大小, 到底多大, 它自己确定

# 确定空间复杂度

## ■ 参考以下算法:

- Step 1: start
- Step 2: function Add (k, m)
- Step 3:  $l = 0$
- Step 4: for  $j = 1$  to  $m$  repeat step 5
- Step 5:  $l = l + k[j]$
- Step 6: return the value of  $l$
- Step 7: stop

## ■ 对于以上算法:

- $k$  数组占用的空间为  $m$  个单元
- 其它变量  $m$ 、 $j$  和  $l$  将分别只占用一个单元的空间
- 因此, 您可以通过将所有变量占用的空间相加来获取此算法的空间复杂度

# 程序优化

- 我们在评价算法的优劣时, 往往是综合考虑时间复杂度和空间复杂度两个因素, 一般是希望能够找到两个都省的方法
- 但事实上我们往往需要牺牲时间复杂度来成全空间, 或牺牲空间来节省时间. 因此我们需要根据问题要求, 选择侧重节省的因素(权衡trade-off)
- 更多的时候由于空间的耗费我们一般可以容忍, 所以我们会更关注时间复杂度对算法的影响

# 代码优化技术分类

---

- 按照优化涉及的范围可分为:

- 局部优化
- 循环优化
- 全局优化

- 按照机器相关性, 可分为:

- 机器相关的优化
- 机器无关的优化

# 局部优化

- 局部优化是指在基本块内进行的优化, 考察一个基本块就可完成.
- 所谓基本块是指程序中顺序执行的语句序列, 其中只有一个入口语句和一个出口语句. 程序的执行只能从入口语句进入, 从出口语句退出
- 对基本块的优化技术包括局部公共子表达式删除、删除多余代码、交换语句次序、重命名临时变量

# 循环优化

- 所谓**循环**, 简单而言就是指程序中可能反复执行的代码序列. 因为循环中的代码会反复的执行, 所以循环的优化对提高整个代码的质量有很大的帮助
- 可以针对每个循环进行相应的优化工作. 主要有以下几种: **代码外提**、**删除归纳变量**、**强度削弱**.



# 全局优化

- 一个过程可以由多个基本块按照相应的流程来组成. **全局优化**就是基于这些基本块之间的优化. 为了进行全局代码优化, 必须在考察基本块之间的相互联系与影响的基础上才能完成
- 常用的全局优化技术有**复写传播**(copy propagation)、**常量折叠** (constant folding)、**删除全局公共子表达式**等

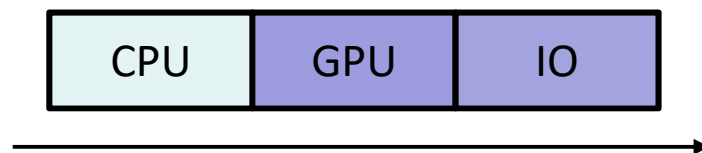
Pipeline 优化



# pipeline模式

某一类任务对于每一份数据都要依次进行CPU、GPU、IO（如硬盘读写）三步操作，每步耗时1s

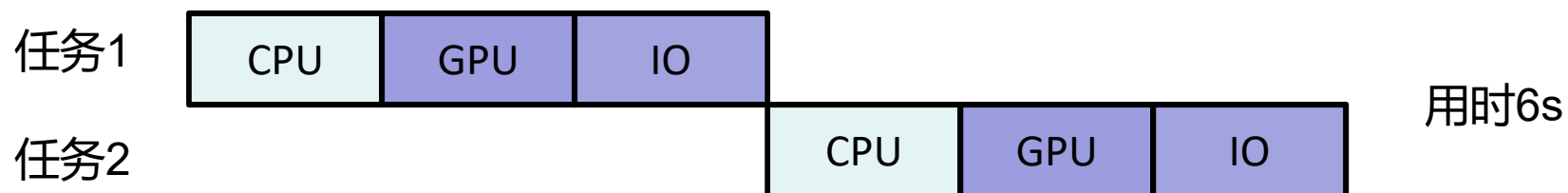
现在有两份数据要用该任务进行处理，如何让总处理时间最短？



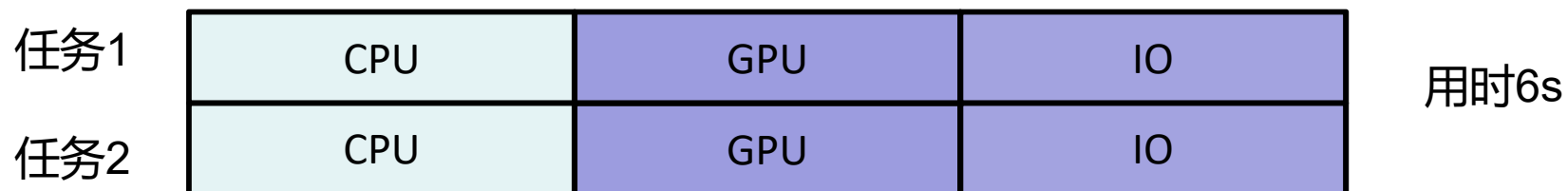


# pipeline模式

## ■ 方案1：简单串行



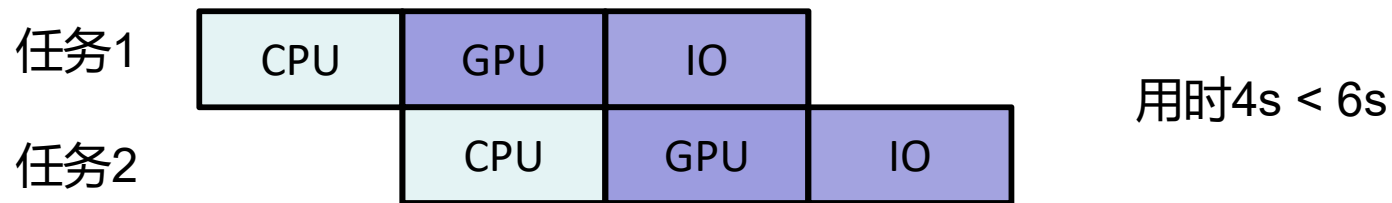
## ■ 方案2：简单并行，每次资源都被两个任务同时使用，会导致每步的耗时变为原来的两倍





# pipeline模式

- 方案3: pipeline (流水线) 模型, 将不同任务错开进行, 实现资源利用最大化



当任务数更多时, 该模型的优势会更加明显

# 与机器的关系

## ■ 机器相关优化

- 针对机器语言, 依赖于目标机器的结构和特点. 例如, 寄存器优化、多处理器优化、特殊指令优化等

## ■ 机器无关优化

- 针对中间代码, 不依赖于目标机器的结构和特点. 例如, 合并常量优化, 消除公共子表达式, 代码外提, 删除归纳变量, 强度削弱和删除无用代码等

# 检测编程结构

- 使用任何编程语言(例如 C、C++、C# 或 Java) 开发的应用程序都基于算法
- 每个算法包括若干编程结构, 如循环、判断和函数
- 应用程序的性能与应用程序中使用的编程结构相关
- 要实现所需级别的优化, 必须检测程序中使用的循环、分支语句和函数调用

# 检测循环

- ✓ 删除不想要的循环部件
- ✓ 合并循环
- ✓ 循环并行
- ✓ 减少循环内的工作
- ✓ 使用 sentinel 值
- ✓ 查看循环顺序
- ✓ 强度削弱（查看运算符）

- 在此技术中, 您需要首先确定循环内存在的**决策制定**步骤
- 确定这些步骤后, 您需要确定循环是否会影响（改变）这些步骤
- 如果循环**不会**影响步骤, 则从循环中删除这些**没有影响的步骤**
- 循环内操作如果与循环次数无关, 可并行加速

- 当多个循环使用**相同变量**时, 您可以合并循环
- 这有助于减少计算时间, 因为减少了执行的指令总数
- 某些表达式、变量或常量的值在循环内**不会更改**, 并且会不必要地占用处理器时间
- 因此, 更好的方法是将其放置在循环外部
- sentinel 值是放在搜索范围**末尾**的值
- sentinel 值使您不必执行额外的任务来检查输入搜索字符串结尾
- 您可以通过更改循环的**顺序**来提高程序的效率
- 在效率方面, 诸如乘和除之类的运算要比诸如加之类的运算更耗费资源
- 您应尝试将所有耗费资源的运算转换为**较容易**执行的运算

# 删除不想要的循环部件

## ■ 优化前的代码

```
for (i=0; i<n; i++)  
{  
    if (a==b)  
    {  
        c = c + d[i];  
    }  
    else {  
        e = e + d[i];  
    }  
}
```

## ■ 优化后的代码

```
if (a==b)  
{  
    for (i=0; i<n; i++)  
        c = c + d[i];  
}  
else  
{  
    for (i=0; i<n; i++)  
        e = e + d[i];  
}
```



# 合并循环

## ■ 优化前的代码

```
for (i=0; i<n; i++)  
{  
    c = c + d[i];  
}  
for (i=0; i<n; i++)  
{  
    e = e + d[i];  
}
```

## ■ 优化后的代码

```
for(i=0;i<n;i++)  
{  
    c = c + d[i];  
    e = e + d[i];  
}
```

# 循环并行

## ■ 优化前的代码

```
for (i=0; i<500; i++)  
{  
    g();  
}
```

## ■ 优化后的代码

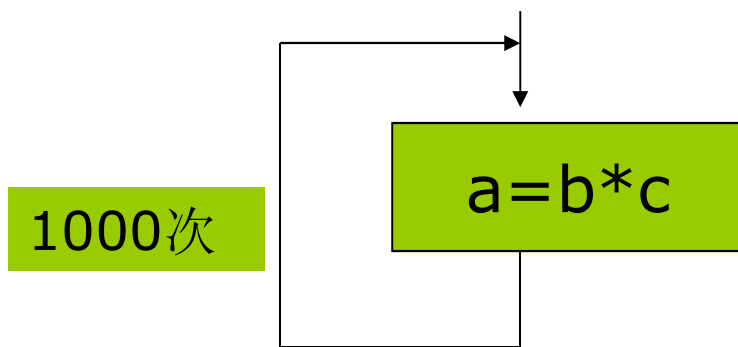
```
#pragma omp parallel for  
for (i=0; i<500; i++)  
{  
    g();  
}
```

时空权衡 trade-off

# 减少循环内的工作(循环不变外提)

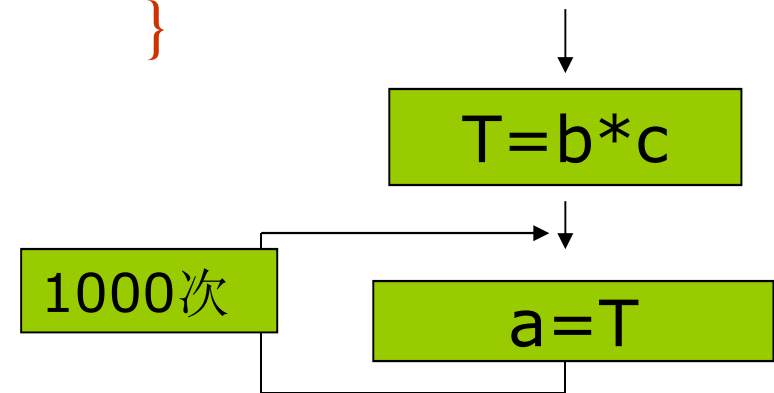
## ■ 优化前的代码

```
for (i=0; i<n; i++)  
{  
    a[i] = b[i] + t.p;  
}
```



## ■ 优化后的代码

```
temp = t.p  
for (i=0; i<n; i++)  
{  
    a[i] = b[i] + temp;  
}
```



扩展：删除多余运算

不必重复计算形式、值相同的变量

# 循环不变外提

## ■ 循环不变式的例子

- 计算半径为 $r$ , 从10度到360度的扇形的面积
  - `for(n=1; n<=36; n++)`
  - `{S:=10/360*pi*r*r*n; printf(" Area is %f", S); }`
- 由于表达式 $10/360 \cdot \pi \cdot r \cdot r$ 中的各个量在循环过程中不改变, 所以, 可以修改程序如下
  - `C= 10/360*pi*r*r;`
  - `for(n=1; n<36; n++)`
  - `{S:=C*n; printf(" Area is %f", S); }`
- 修改后的程序中,  $C$ 的值仅需计算一次, 而原来的程序需要计算35次

# 循环不变外提

## ■ 循环不变式的相对性

- 对于多重嵌套的循环, 循环不变四元式是相对于某个循环而言的. 可能对于更加外层的循环, 它就不是循环不变式
- 例子

```
for(i = 1; i<10; i++)  
  for(n=1; n<360/(5*i); n++)  
    {S:=(5*i)/360*pi*r*r*n;...}
```

- $(5*i)/360*pi*r*r$  对于  $n$  的循环(内层循环)是不变表达式, 但是对于外层循环, 它们不是循环不变表达式

```
for(i = 1; i<10; i++)  
  C=(5*i)/360*pi*r*r;  
  for(n=1; n<360/(5*i); n++)  
    {S:=C*n;...}
```

```
    T=5/360*pi*r*r;  
    for(i = 1; i<10; i++)  
      C=T*i;  
      for(n=1; n<360/(5*i); n++)  
        {S:=C*n;...}
```

# 删除多余运算

- 如果有表达式 $e_1$ 和 $e_2$ , 且它们的值始终相同, 则 $e_2$ 的计算部分可以省略, 只要用 $e_1$ 的值即可

例:  $m = a * b + 1;$

$n = a * b - 1;$

$x = a * b + 2;$

其中三个 $a * b$ 的子表达式始终一致, 因此, 可以优化为:

$temp = a * b;$

$m = temp + 1;$

$n = temp - 1;$

$x = temp + 2;$

# 删除多余运算

- 注意: 只有形式相同还不行, 必须值也要相同, 即形式相同并不能保证值相同

例:  $m = a * b;$

$a = a * b;$

$x = a * b;$

三个式子中的 $a * b$ 形式相同, 但第三个式子中的 $a * b$ 与前两个式子中的 $a * b$ 的值不同, 因此, 第三个  $a * b$ 不能省。

可优化:  $temp = a * b;$

$m = temp;$

$a = temp;$

$x = a * b;$

# 使用sentinel值

## ■ 优化前的代码

// Returns index of value, -1 for no result

```
int find(int* a, int l, int v)
{
    int i;
    for (i = 0; i < l; i++)
        if (a[i] == v)
            return i;
    return -1; // -1 means "no result"
}
```

## ■ 优化后的代码

```
int find(int* a, int l, int v)
{
    int i;
    // add sentinel item:
    a[l] = v; // prepare it with sentinel value
    for (i = 0; ; i++)
        if (a[i] == v) {
            if (i != l) // real result
                return i;
            // was sentinel value, not real result:
            return -1;
        }
}
```

对C++11有更好的写法:for\_each



# 查看循环顺序

## ■ 优化前的代码

```
for (j=1; j<100; j++){  
    for (i=1; i<5; i++){  
        a = a + b[i][j];  
    }  
}
```

执行的判断指令总数：

第一行 for = 100

第二行 for =  $5 \times 99 = 495$

Total =  $100 + 495 = 595$

## ■ 优化后的代码

```
for (i=1; i<5; i++){  
    for (j=1; j<100; j++){  
        a = a + b[i][j];  
    }  
}
```

执行的判断指令总数：

第一行 for = 5

第二行 for =  $4 \times 100 = 400$

Total =  $5 + 400 = 405$

把计算更多的放在内循环

Cache的命中率也要考虑！



# 内存的cache机制

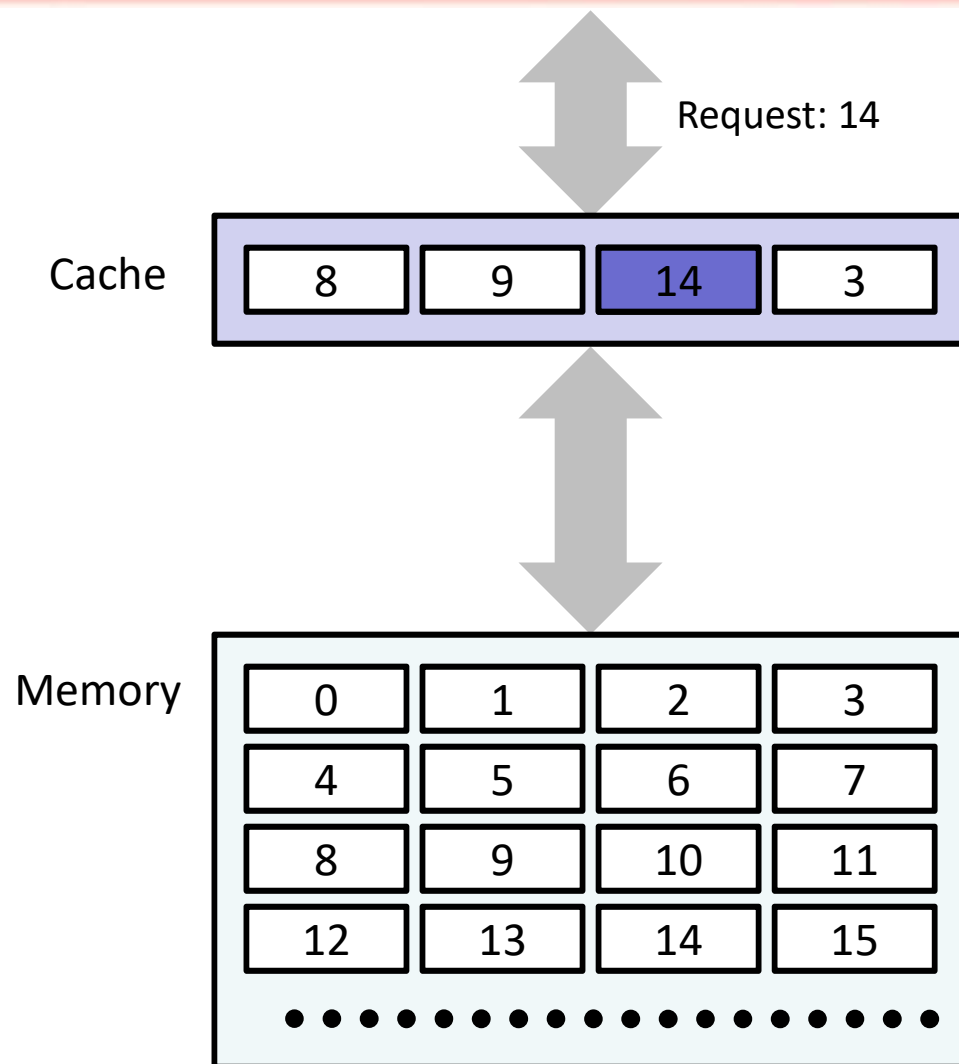
计算机访问内存时，会先访问硬件中的缓存，缓存容量小但读取速度快

缓存中会保存着近期使用过的数据



# 内存的cache机制

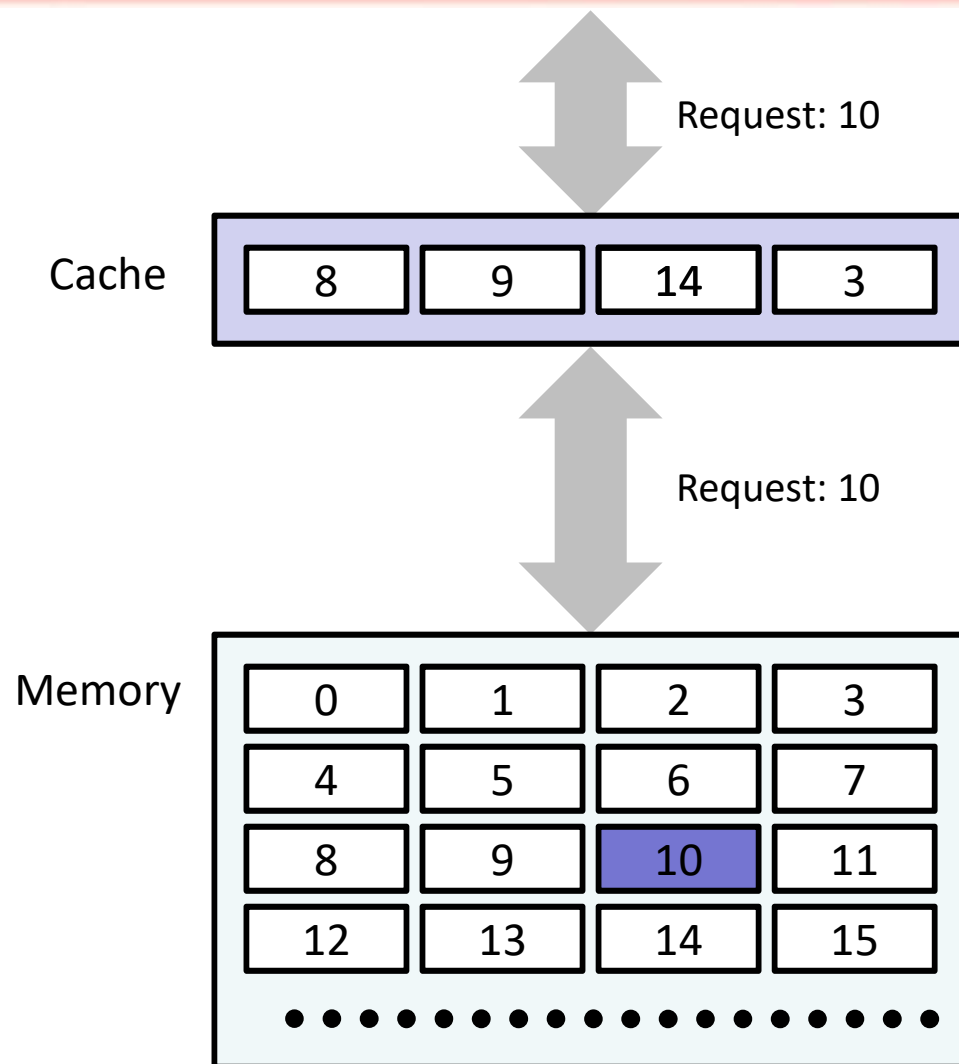
如果缓存中有要访问的内容，则**缓存命中**，直接读取缓存





# 内存的cache机制

如果缓存中没有要访问的内容，则**缓存未命中**，需要从内存中读取该内容

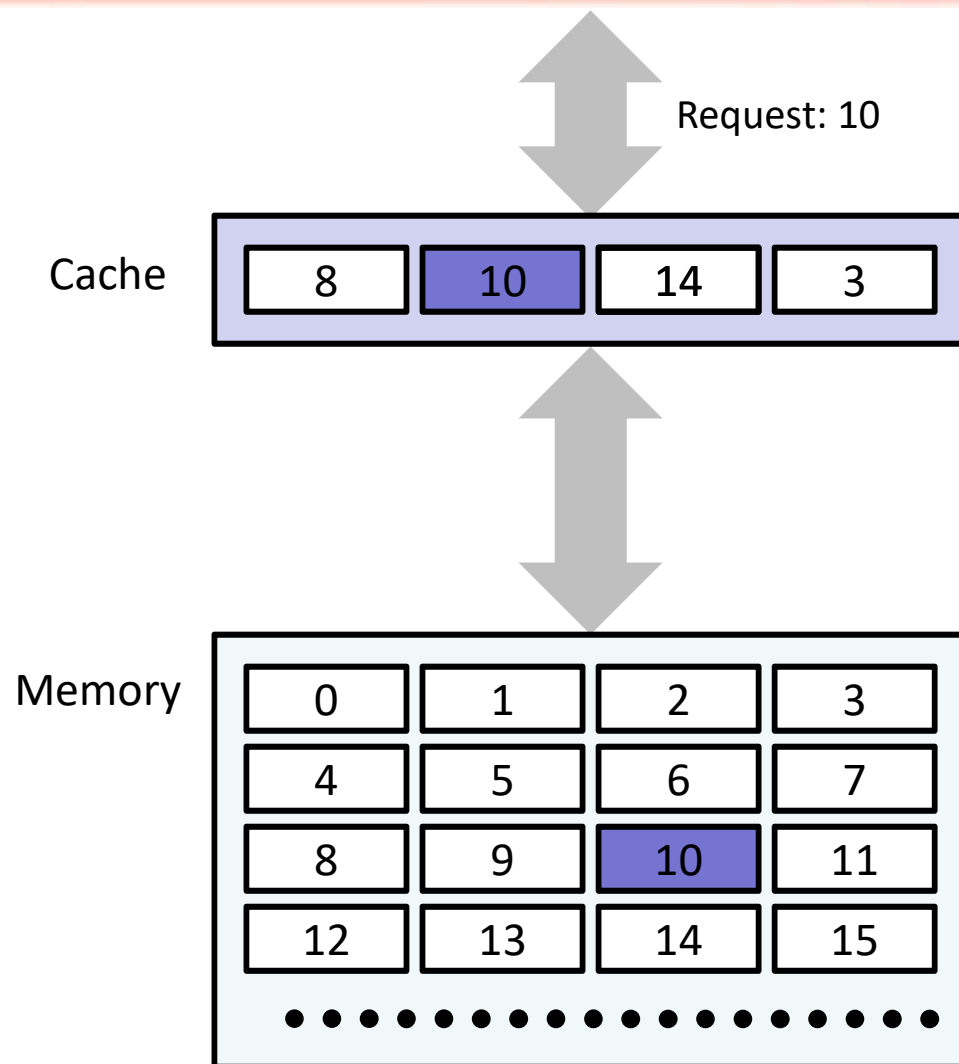




# 内存的cache机制

缓存未命中时，从内存中读取的内容会被写入缓存

如果缓存容量已满，则会将长期未使用的内容移出缓存





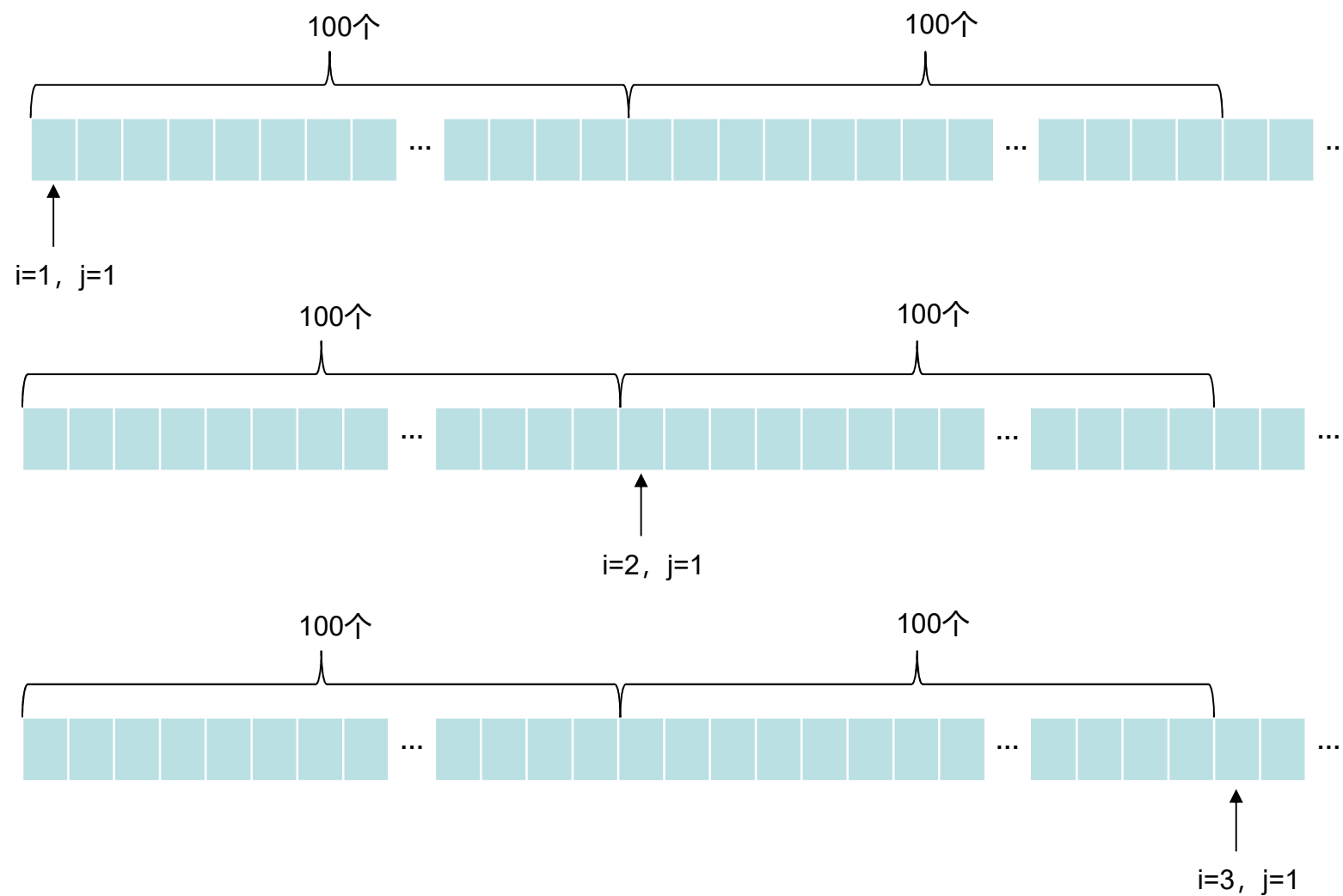
# 内存的cache机制

将内存中的内容读入缓存时，会将附近的一块数据都进行读入

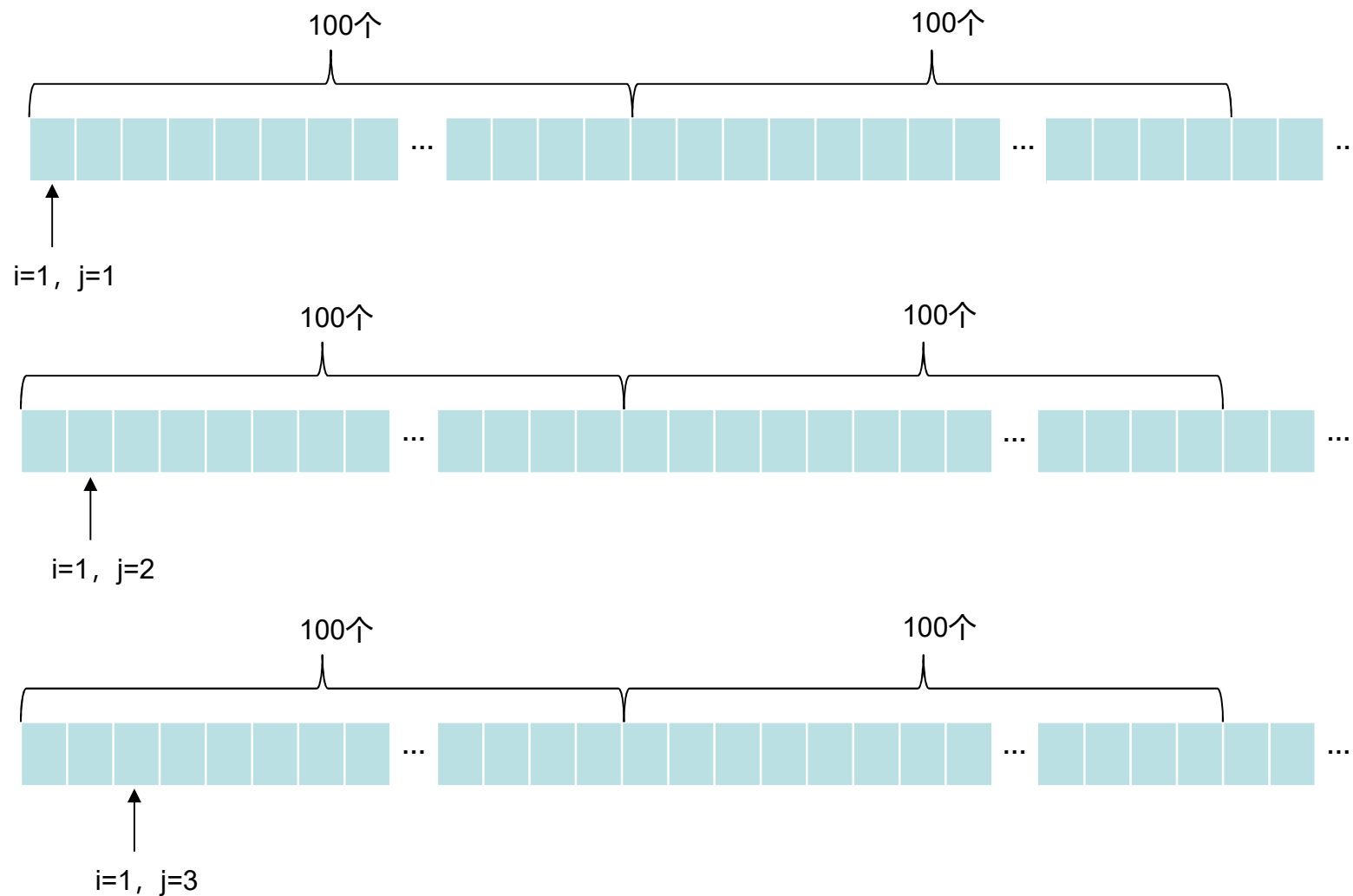
因此在编程时需要注意：

- 按照数据存储的顺序、以尽量小的步长来连续读取数据
- 一旦读取了某一个数据，就先尽量多地使用它

# 查看循环顺序（优化前）



# 查看循环顺序（优化后）



为什么同样是 $O(n \log n)$ 的算法，快速排序的效率会更高（相比堆排序）？



# 强度削弱

## ■ 优化前的代码

```
for ( i=0; i<n; i++ )  
{  
    a[i] = i * a * b * c;  
}
```

## ■ 优化后的代码

```
temp = a * b * c;  
final = temp;  
for (i=0; i<n; i++)  
{  
    a[i] = final;  
    final = final + temp;  
}
```

# 强度削弱

- 高强度的运算改为低强度的运算
- 实现同样的运算可以有多种方式, 用计算较快的运算代替较慢的运算
- $2*x$ 或 $2.0*x$  变成  $x+x$
- $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$  变成
$$(((\dots(a_n x + a_{n-1})x + a_{n-2})\dots)x + a_1)x + a_0$$

# 移位实现乘除法

$x = x * 8;$

$y = y / 8;$

可以改为:

$x = x \ll 3;$

$y = y \gg 3;$



$x = x * 9$

可以改为:

$x = (x \ll 3) + x$

# 避免整除

---

```
int i, j, k, m;
```

```
x = i / j / k;
```

可以改为:

```
int i, j, k, m;
```

```
x = i / (j * k);
```

# 自增、自减 / 复合赋值表达式

`x=x+1;或x++;`                      `++x;`



`x=x+y;`                      `x+=y;`

现代编译器已可以自动优化

对于内建数据类型，两个表达式效率相同

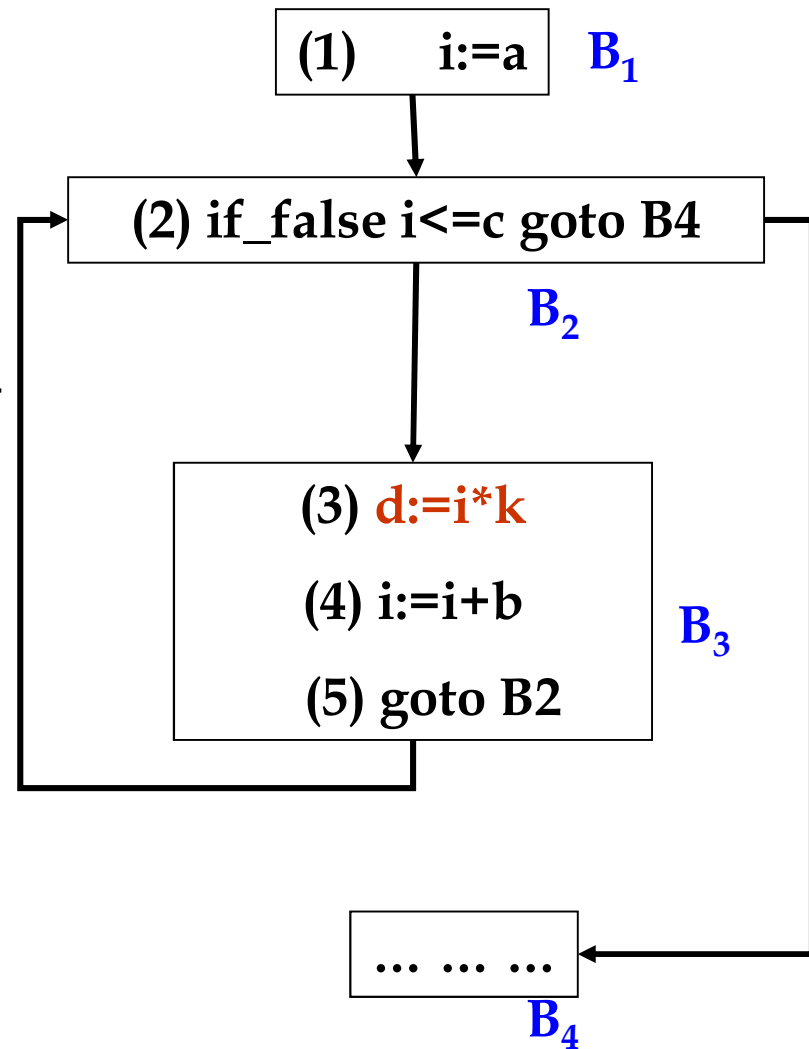
对于自定义数据类型，后者不借助中间变量进行赋值

# 归纳变量删除

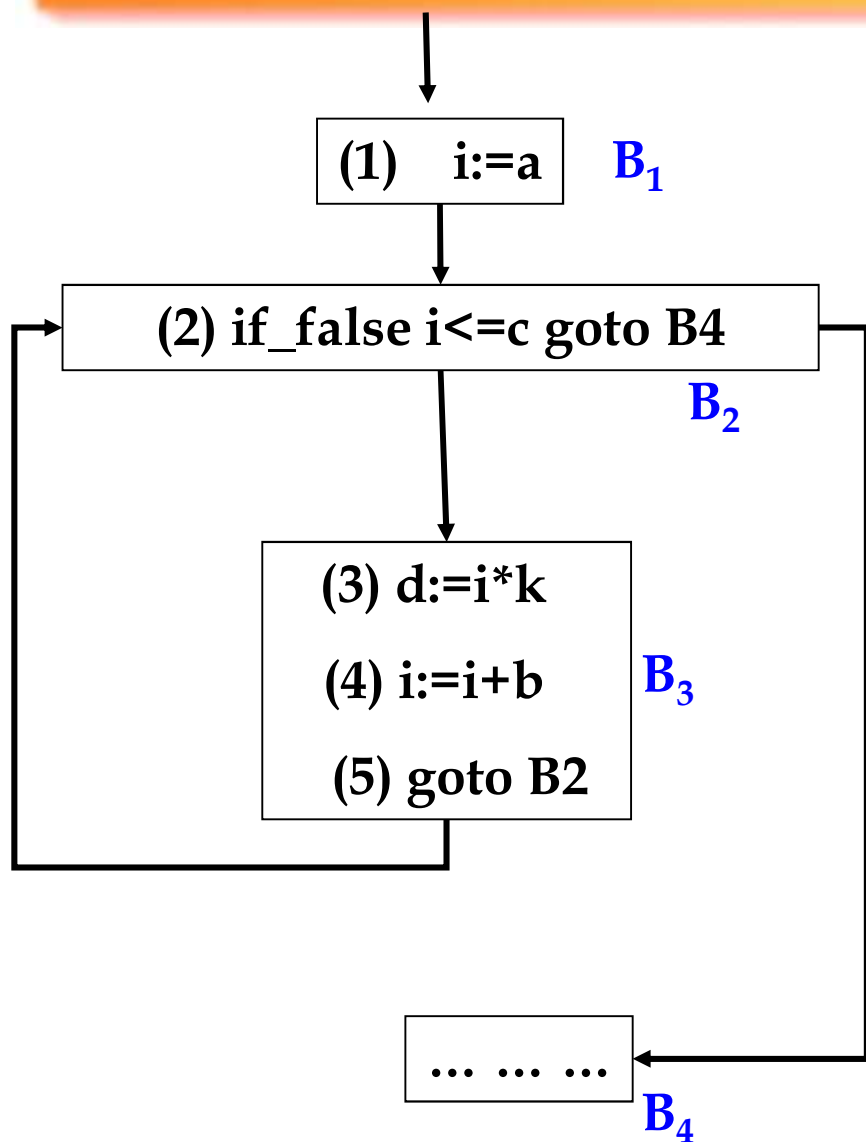
- 在循环中, 如果变量 $I$ 的值随着循环的每次重复都固定地增加或者减少某个常量, 则称 $I$ 为循环的归纳变量
- 如果循环中对 $I$ 只有唯一的形如 $I = I + c$ 的赋值, 且其中 $c$ 为循环不变量, 则称 $I$ 为循环中的基本归纳变量。若 $J := c_1 * I + c_2$ , 且其中 $c_1, c_2$ 为循环不变量, 即 $J$ 在循环中的值总是可归化为 $I$ 的同一线性函数, 则称 $J$ 是归纳变量, 并称它与 $I$ 同族
- 如果在一个循环中有多个归纳变量, 归纳变量的个数往往可以减少, 甚至减少到1个. 减少归纳变量的优化称为归纳变量的删除

# 归纳变量删除

- 设某程序中有如下循环  
i=a;  
while(i<=c)  
{...; d=i\*k; ...; i+=b;}  
■ d与i为归纳变量, d=i\*k在循环内恒真, 故条件i<=c等价于d<=c\*k
- 若i只用于控制循环, 则可省略, 将循环控制条件改为  
T2:=c\*k;  
if\_false d<=T2 goto B4
- 有关i的四元式均可删除

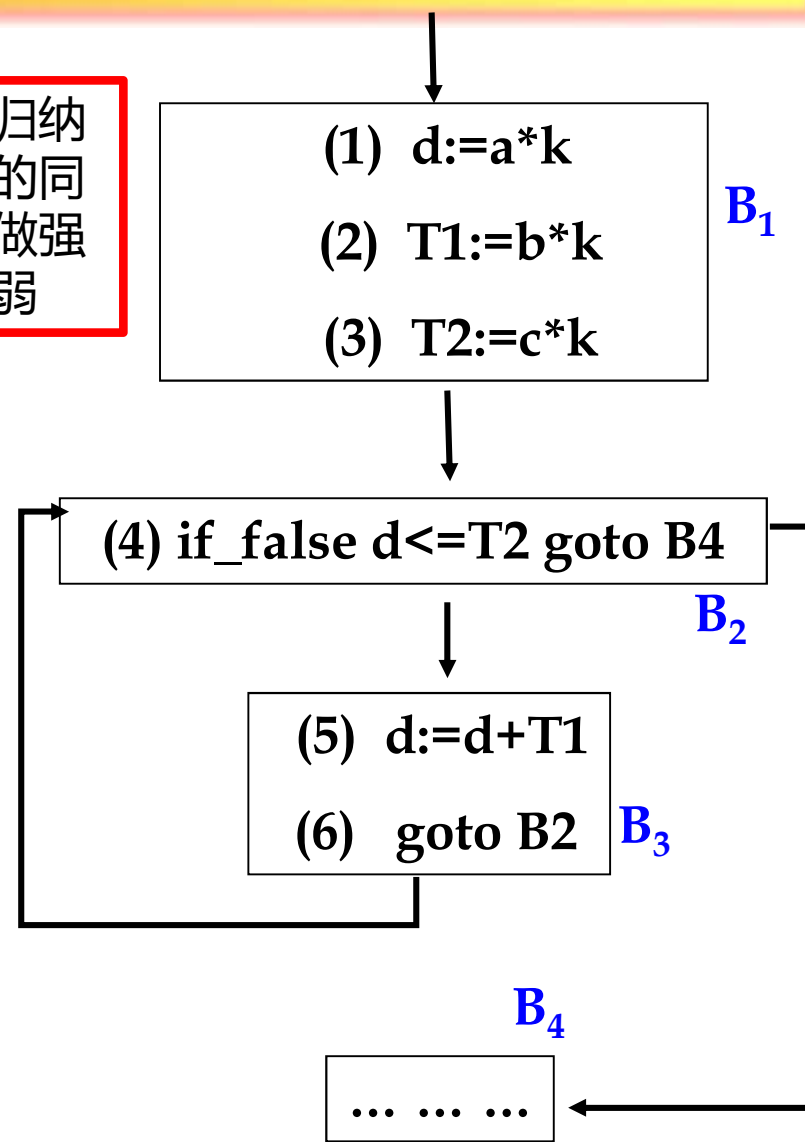


# 归纳变量删除



归纳变量*i*删除前

删除归纳变量的同时  
要做强度削弱



归纳变量*i*删除后

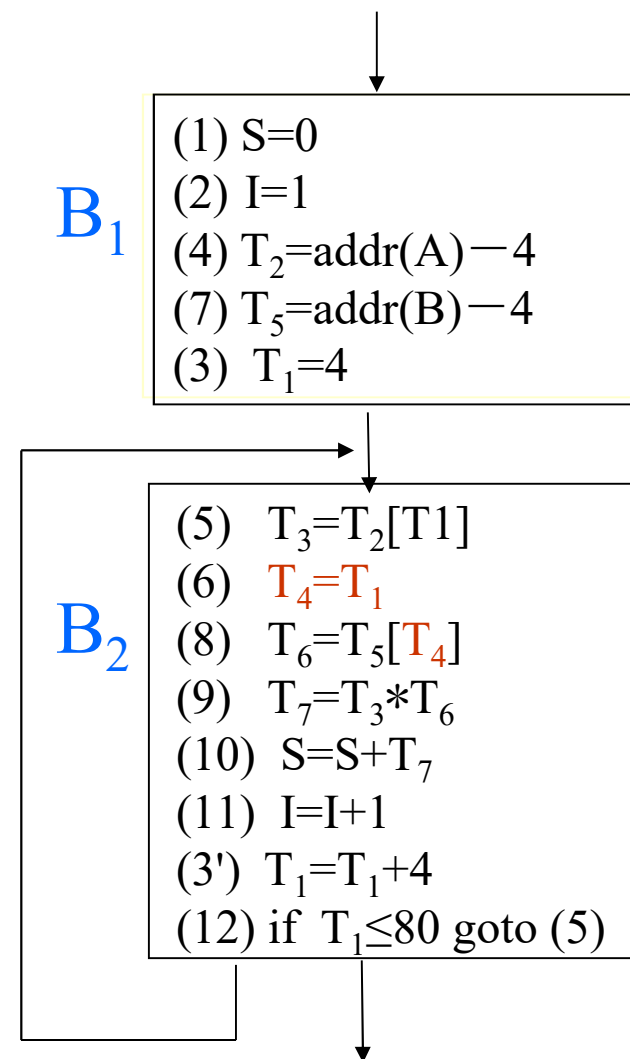


# 删除无用赋值

- 如果四元式 $z = x \text{ op } y$ 之后, 对 $z$ 赋予的该值再也没有被使用到, 那么这个四元式是无用的
- 例如:
  - $T0 = a; T0 += 5; X = T0; \dots; T0 += 1; \dots T0 = b; Y = b;$
  - 赋值 $T0 += 1$ 可被删除
  - $a = x + y; b = a; c = a + z;$
  - $a = x + y; c = a + z;$

# 复写传播

- 复写传播是指尽量不引用那些在程序中仅仅只传递信息而不改变其值, 也不影响其运行结果的变量



# 过程内嵌

- 过程内嵌是指针对源程序中的某些过程调用, 找到被调过程的过程体, 如果该过程体短小而且没有循环, 则将它拷贝到调用处, 从而消除过程调用的开销, 增大指令调度的可能性. 过程内嵌增加了程序的空间开销

内联函数 inline function

空间换时间

时空权衡 trade-off

# 常量合并

- 常量合并又称为常数表达式求值(constant expression evaluation), 是指在**编译时刻**就对已知操作数的值为**常数**的表达式求值, 并且用该结果值来替代这部分表达式

- 例如:

- 优化前:

- $x = 5; b = x + y;$

- 优化后:

- $b = 5 + y;$

- 例如:

- $a + 2 * 3$ 可翻译成 $a + 6$

- 表达式 $T1 = a + 1 + 3$ 在翻译阶段生成的代码为:  $T1 = a;$   $T1 += 1;$   $T1 += 3;$  由于对 $T1$ 的两次定值未被引用, 可将其修改为:  $T1 = a;$   $T1 += 4;$

**编译器优化**

# 检测函数

- 由于调用函数的频率的增加, 程序执行的时间也会增加
  - 检测函数对于性能优化来说很重要
  - 可通过以下方式优化函数:
    - ✓ 使用较快的函数
    - ✓ 了解数学函数
    - ✓ 了解标准函数
    - ✓ 将本地函数声明为静态函数
- 请尝试仅使用较快的函数
  - 通过了解与函数关联的时间复杂度, 您可以确定较快的函数
  - 使用标准数学方法来计算在每个阶段使用复杂计算的结果
  - 这使您能够更有效的解决问题
  - 使用静态函数, 能够更快的进行求值并且提高效率
  - 您需要选择性的在程序中使用数学函数, 如平方根

静态函数会被自动分配在一个一直使用的存储区, 直到退出应用程序实例, 避免了调用函数时压栈出栈, 速度快很多

# 检测分支

- 可通过分支来将代码某部分的控制转移到其它部分
  - 可以使用各种技术来使分支过程更有效并提高代码效率。
  - 可通过以下方式来检测分支:
    - ✓ 删除 else 子句
    - ✓ 使用有效的 Case 的语句
- 为每个 if 循环使用 else 子句会导致分支效率下降
  - 因此, 在可能情况下, 您应尝试删除 else 子句
  - 需要使用高效的 case 语句, 以使选项的顺序基于使用此选项的频率

**Switch语句击中第三个选项的时间跟if/else if语句击中第三个选项的时间相同。**

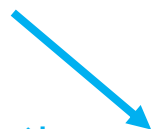
**击中第一, 第二选项的速度if语句快, 击中第四以及第四之后的选项的速度switch语句快。**

# 删除else语句

## ■ 优化前的代码

```
if (condition) {  
    Case A;  
}  
else{  
    Case B;  
}
```

如果condition为true的概率很低



```
if (!condition){  
    Case B;  
}  
else{  
    Case A;  
}
```

## ■ 优化后的代码

```
Case B;  
if (condition){  
    Undo Case B;  
    Case A;  
}
```



如果undo Case B  
的开销极小

# 以频率对case语句排序

```
switch (days_in_month)
{
    case 28:
    case 29:
        short_months ++;
        break;
    case 30:
        normal_months ++;
        break;
    case 31:
        long_months ++;
        break;
    default:
        cout << "month has fewer
than 28 or more than 31 days" <<
endl;
        break;
}
```



```
switch (days_in_month)
{
    case 31:
        long_months ++;
        break;
    case 30:
        normal_months ++;
        break;
    case 28:
    case 29:
        short_months ++;
        break;
    default:
        cout << "month has fewer than 28
or more than 31 days" << endl;
        break;
}
```



# 变量定义

- 在变量定义时避免类型不匹配造成临时变量的产生
  - `int x; x = static_cast<int>(1.0);`
- 变量延时定义，当某个变量需要提前定义时，定义为指针，初始化为空指针
  - 避免多调用构造函数
- 对象的复合（一个类包含另一个类的对象）时，可以使用包含对象的指针
  - 避免多调用构造函数

是否与上一章讲的声明就初始化矛盾？

# std::move、右值引用与 &&

- 在深度拷贝时使用move操作，或者在参数传递时使用move，可以减少消耗
- 使用move提高内存的使用效率

```
class A {  
    public: A() { std::cout << "Constructor" << std::endl; }  
    A(const A&) { std::cout << "Copy Constructor" << std::endl; }  
    A(A&&) { std::cout << "Move Constructor" << std::endl; }  
    ~A() {}  
};  
  
A getA() {  
    A aa;  
    return aa;  
}  
  
int main() {  
    A a = getA(); //移动构造函数减少了一次copy, 局部变量的资源不再被释放  
    return 0;  
}
```

# 函数参数

---

- 当函数参数过多时，应该使用对象来打包参数，减少参数过多带来性能消耗
- 使用const引用代替值传递
- 函数初始化可以利用初始化列表进行初始化

# 内存

---

- 减少频繁申请和释放内存
- 如果事先确定需要使用多少内存，可以事先进行分配
- 比如使用： `std::allocate`

**如果频繁申请和释放内存，其实相当于用时间换取空间。频繁申请释放内存的时间会大大降低程序运行的效率**

# 常规优化规则

---

- 以下是编写代码时要遵循的一些准则：
  - 确定优化区域
  - 确定优化的深度
  - 确定正确的备选方法
  - 确定需要

# 有关优化的常见误解

- 以下是编码和优化时通常容易引起的误解:
  - 认为程序无需优化, 因为该程序似乎速度很快
  - 认为编译器执行的优化就已足够 (有些确实被编译器考虑了)
  - 认为短代码更有效
  - 认为特定的解决方案将十分有效而不需要验证性能结果
  - 认为在编程时进行优化是一个很好的实践
    - 在设计时就应该进行优化

# 并行优化

## ■ 工具

- thread
- OpenMP
  - #pragma omp parallel for
- MPI
  - 进程级别的，使用MPI是在不同进程之间进行通信，通过进程进行并行计算
- CUDA
  - 使用NVIDIA的GPU进行优化
- OpenCL
  - 可以用于多种厂商的显卡

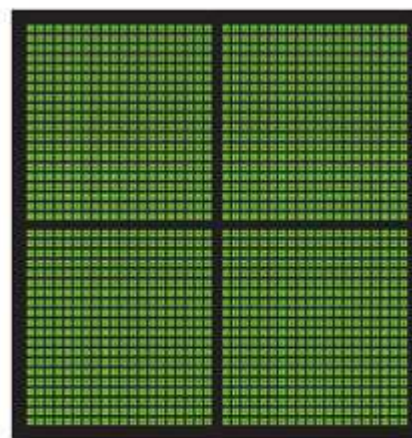
# Why GPU?

## ■ 核心数

- Core i7-6950X      10核20线程
- NVIDIA TITAN V   5120 CUDA Cores



CPU  
MULTIPLE CORES



GPU  
THOUSANDS OF CORES



# CUDA学习资料

---

- NVIDIA官方网站
- [https://www.youtube.com/watch?v=dJ\\_D7JjOe8s](https://www.youtube.com/watch?v=dJ_D7JjOe8s) 该频道下有CUDA的视频学习教程
- 有兴趣可以自学

# 对应用程序使用性能库

- 每个软件开发员在编写代码时都有自己的风格. 编写代码时可能效率很低
- 软件的某些部分, 如菜单栏, 可能在大多数软件中十分常用
- 重复编写这些部分的代码非常耗时并且容易引起错误
- 在这种情况下, 您可能希望使用现有的代码, 这些代码称为性能库
- 已经对这部分代码检查过错误. 此外, 该代码在某个时间段内达到很高的优化级别

# 使用性能库的优势

---

- 以下是使用性能库的一些优势：
  - 使程序员使用更少的时间来开发代码
  - 提供无错误代码
  - 能够最佳利用资源
  - 帮助改进性能
  - 使软件应用程序的功能更加稳定
  - 有些性能库利用GPU进行加速

# 了解性能库的类型

## ■ 根据任务, 性能库可以分为以下类别

- ✓ 工程和科学库
- ✓ 数学库
- ✓ 图形库
- ✓ 音频/视频库
- ✓ 图像处理库
- ✓ 其他库

- 这些库主要用于科学和工程应用程序
- 这些库可能包括用于对各种表达式进行搜索、排序和求值的功能

- 这些库用于对复杂数学函数进行求值, 如向量和矩阵计算
- 这些库有助于正确和精确地绘制图形、饼图、图表和条形关系图。
- 这些库可提高处理图像的速度
- 这些库有助于优化与音频-可视数据相关的各个函数
- 其它库可能包括用于执行各种任务的功能, 如语音识别、信号处理和密码



# C++多文件与库



# 目录

---

## C++多文件编程

## C++动态库与静态库

- 静态库
- 动态库



# C++多文件编程



# 多文件的优势

多文件编程将程序代码结构化，分为多个模块

- 模块之间独立，方便复用、迭代更新
- 不同模块独立开发，利于分工协作
- 项目层次结构清晰，可读性强
- .....

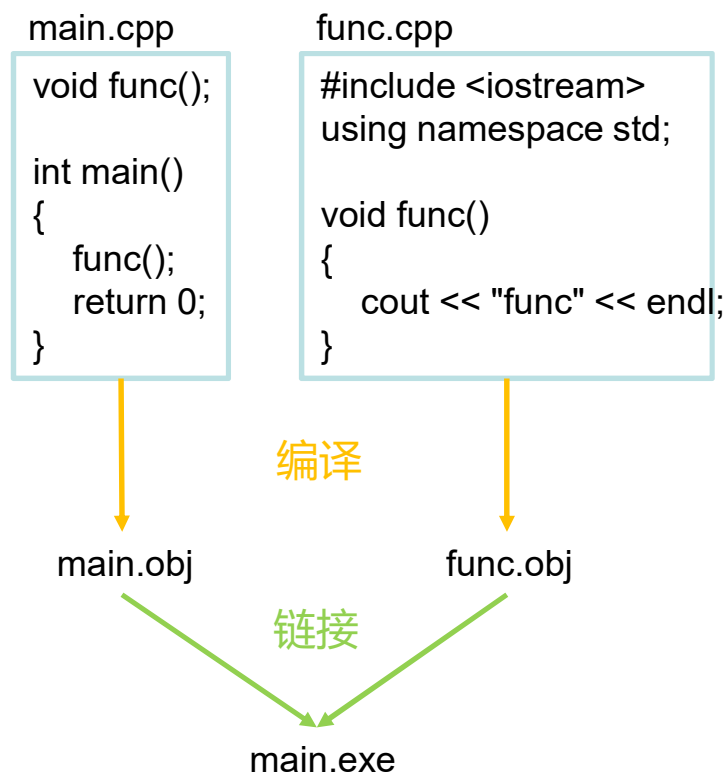




# 多文件项目的编译链接

## 不包含头文件项目的编译链接

- 编译过程检查调用的函数在**本文件**中是否有声明
- 链接过程检查函数在**所有源文件**中是否有定义且无重复定义





# 多文件项目的编译链接

当一个函数被多个文件用到，一个文件用到多个函数时，如果每次都进行声明的话会十分繁琐

怎么办？

使用头文件！

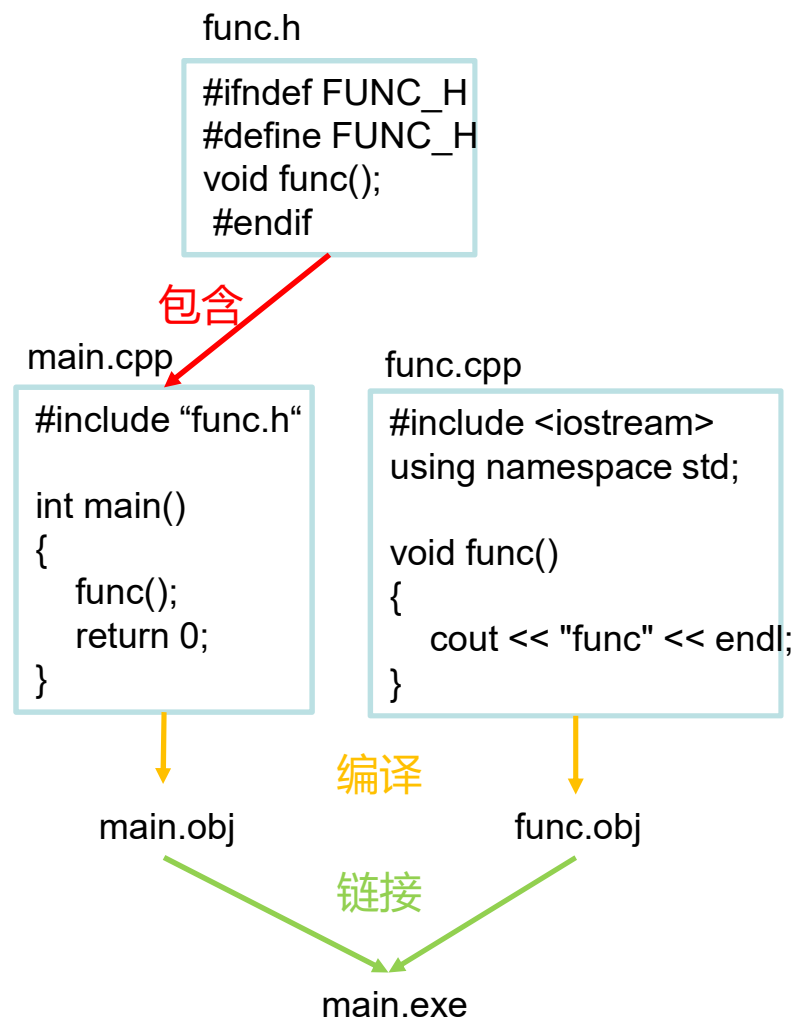
在头文件中进行声明，需要使用到某个函数，就把它所在的头文件包含进来，**头文件的内容会在编译前被粘贴到源文件中**



# 多文件项目的编译链接

## 包含头文件项目的编译链接

- 头文件的内容一般都会使用条件编译预处理语句包住，防止由于依赖关系多次被包含  
(也可以使用#program once)
- 其他文件再需要使用func()时，只需要包含该头文件即可





# 多文件项目编写

根据上面的编译链接原理，.cpp和.h文件各自应该写入程序的哪些部分？

- **.h文件**：函数声明，类或结构体的定义，函数模板，内联函数
- **.cpp文件**：函数与类函数的定义，变量定义（否则会有重复定义）



# C++动态库与静态库



# 什么是库?

库是已经写好的，可以复用的代码

我们常用的<iostream>、<vector>、<string>等就是C++提供的标准库，此外还有opencv等需要手动配置的常用库，也可以将自己的代码编成库

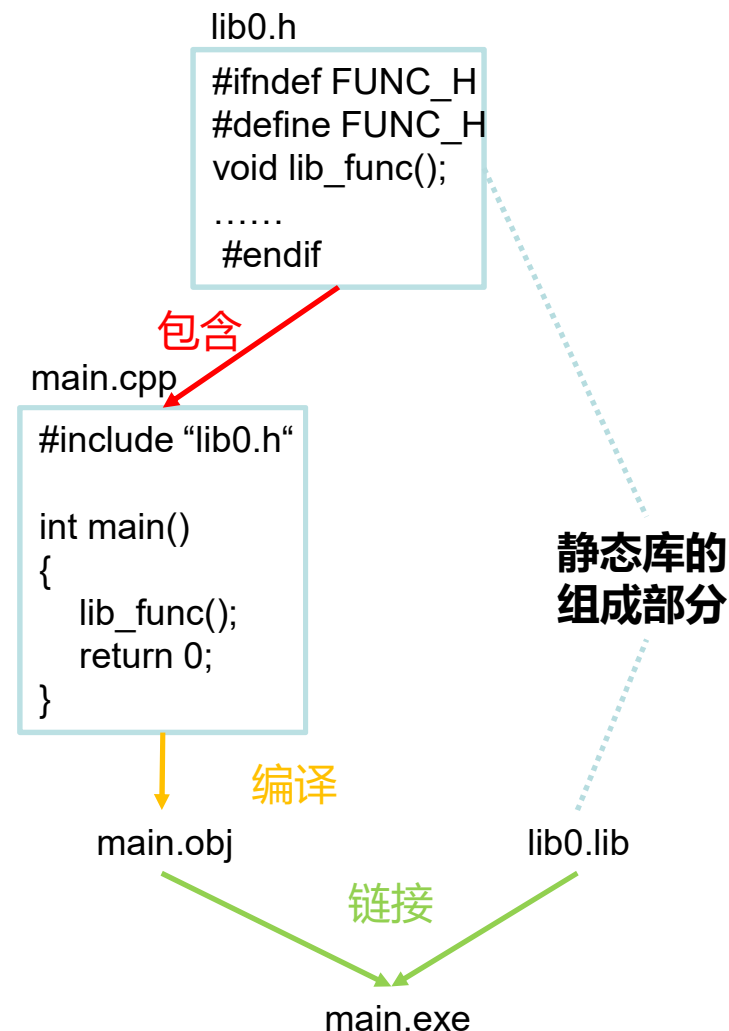
库通常包含头文件与编译好的库文件（有的库只有头文件）



# 静态库

静态库在链接阶段，将编译生成的目标文件与库文件一起链接到可执行文件中

- 静态库头文件在使用静态库时需要进行引用
- 静态库文件.lib可以看成是一组编译目标文件.obj的集合





# 静态库的特点

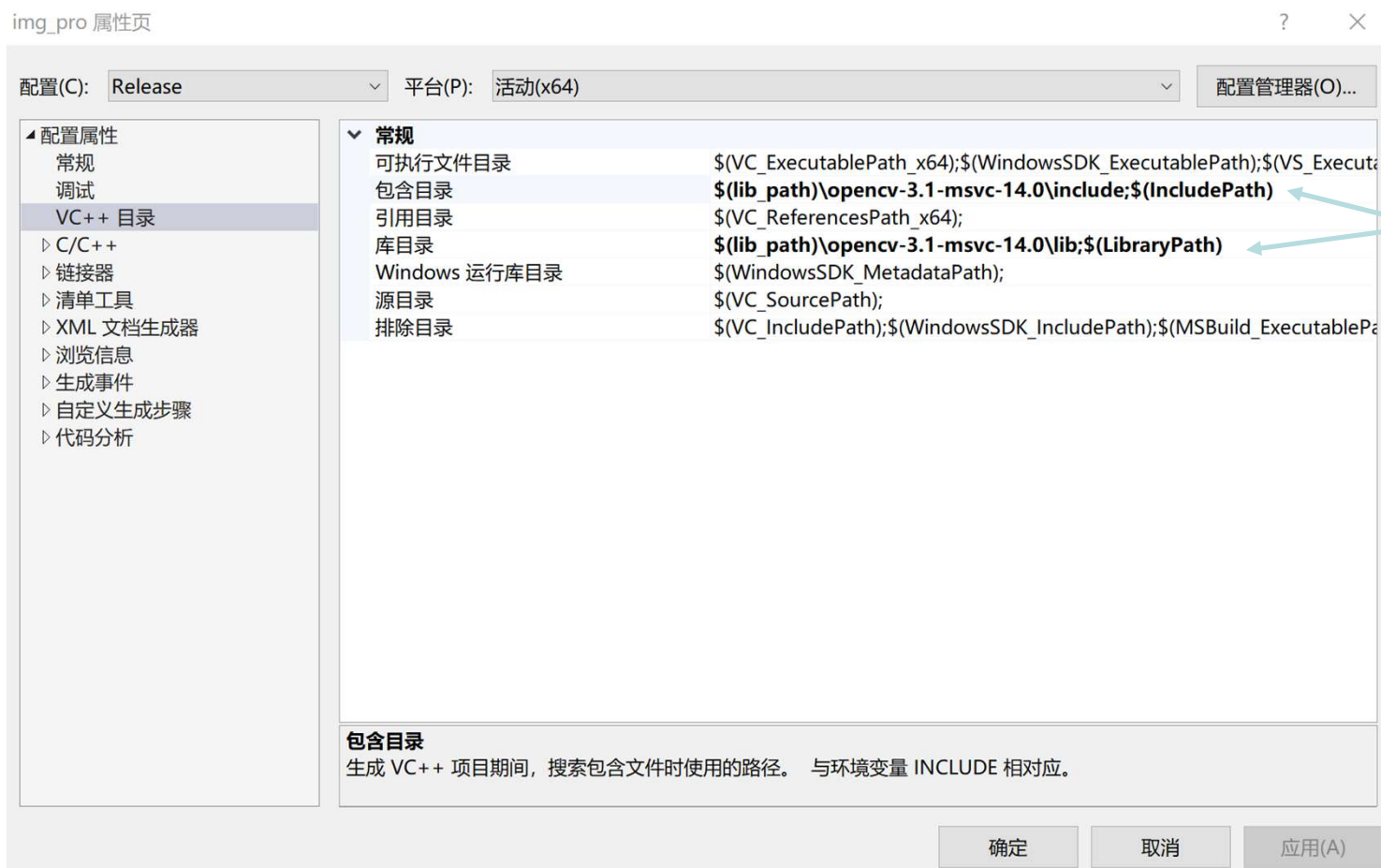
- 静态库对库文件的使用是在链接时期完成的
- 程序在运行时与函数库再无关系，方便移植
- 所有相关的编译目标文件与库文件被链接成一个可执行文件





# VS中静态库的配置使用

属性->VC++目录

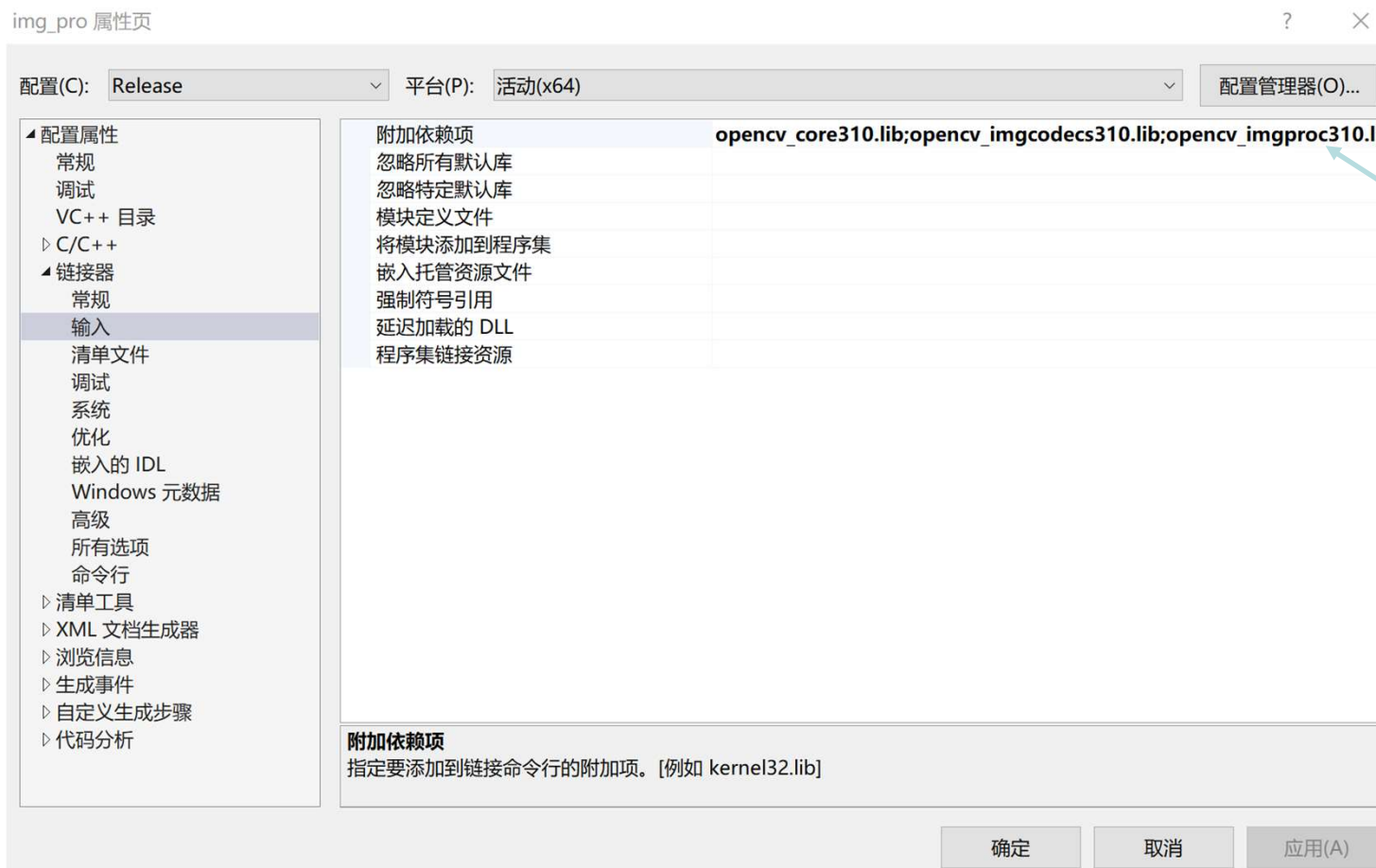


编辑添加头  
文件和库文  
件所在目录



# VS中静态库的配置使用

属性->链接器->输入

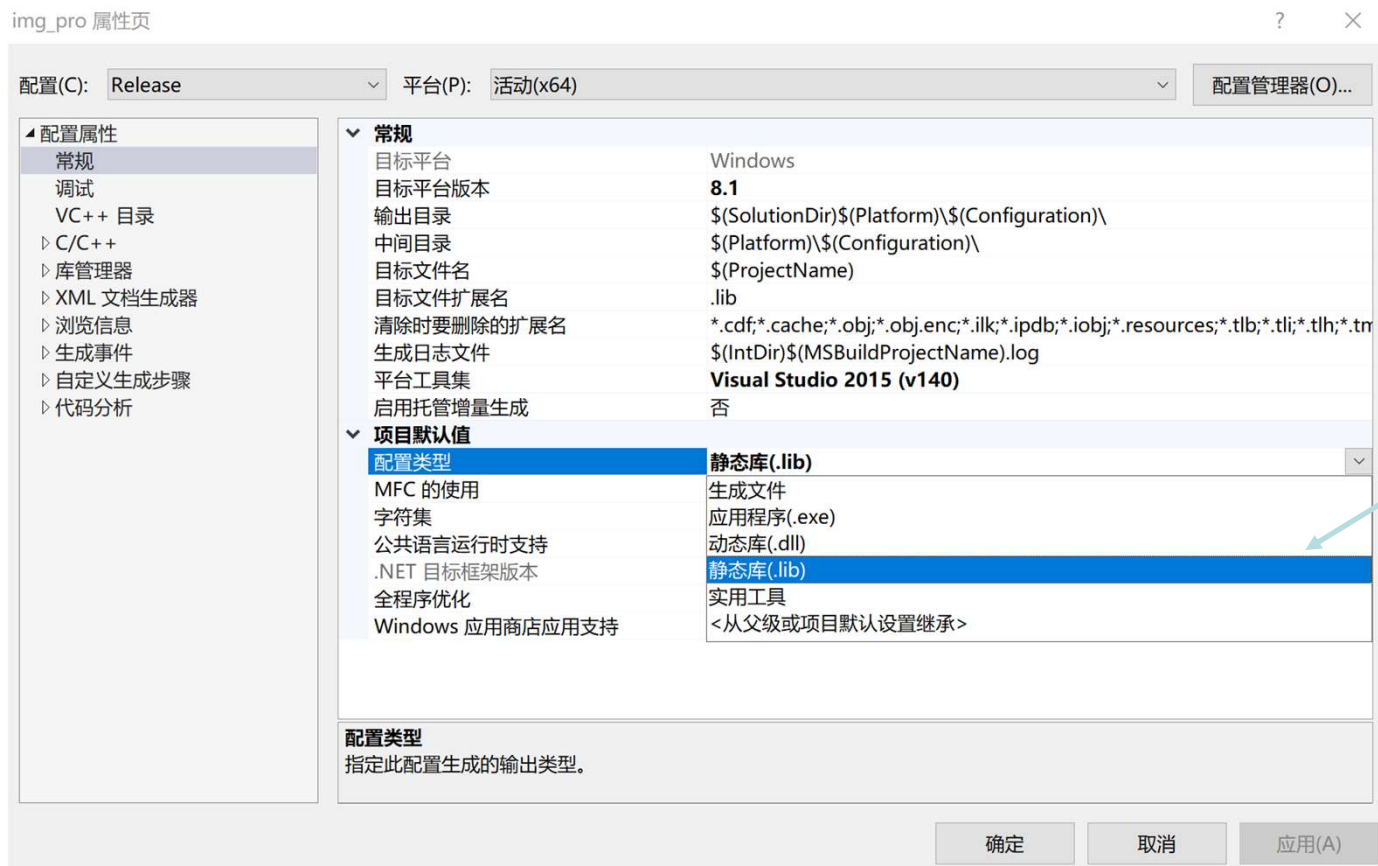


编辑添加库  
文件名



# VS中静态库的生成

除了使用现有的静态库，也可以将自己的代码生成静态库  
属性->常规



由生成应用程序  
改为生成静态库

之后进行生成可  
得到静态库文件

img\_pro.lib



# 动态库

有静态库了为什么还要有动态库？

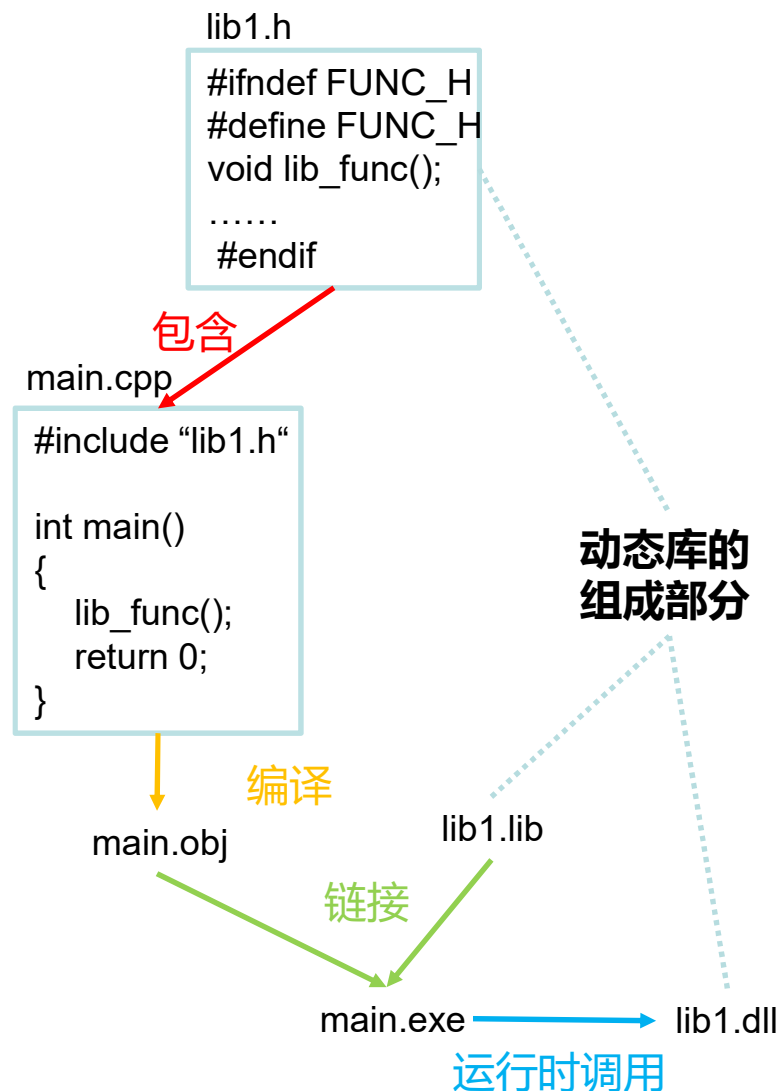
- 静态库存在空间浪费问题，不同程序使用同一个库时，静态库会被单独链接到每个程序中，空间代价较大
- 静态库对程序的更新和发布也会带来麻烦。如果静态库更新了，使用它的应用程序都需要重新编译链接、发布给用户



# 动态库

动态库文件.dll在程序编译时并不会被链接到可执行程序中，而是在程序运行时才被载入

动态库也会有.lib文件（导入库），不过比静态库的.lib文件要小的多





# 动态库的特点

- 动态库把一些库函数的载入推迟到程序运行时期
- 不同的应用程序如果调用相同的库，则只需要有一份该库的实例，可以实现进程之间的资源共享。（因此动态库也称为共享库）
- 动态库独立于可执行程序，更新与发布比静态库更加简单
- 可以在程序代码中显示调用来控制库的载入
- 库文件.dll与可执行文件分离，容易出现文件丢失





# VS中动态库的配置使用

与静态库相同，也需要在属性页面中添加头文件和库文件所在目录、添加库文件名

此外还要将动态库特有的.dll文件放在运行程序目录或者系统环境变量目录中



# VS中动态库的生成

新建项目时选择DLL，VS会自动补充DllMain函数

Win32 应用程序向导 - Win32Project1



应用程序设置

概述

应用程序设置

应用程序类型:

☐ Windows 应用程序 (W)

☐ 控制台应用程序 (C)

☒ DLL (D)

☐ 静态库 (S)

附加选项:

☐ 空项目 (E)

☐ 导出符号 (X)

☒ 预编译头 (P)

☒ 安全开发生命周期 (SDL) 检查 (C)

添加公共头文件以用于:

☐ ATL (A)

☐ MFC (M)

< 上一步

下一步 >

完成

取消

```
BOOL APIENTRY DllMain( HMODULE hModule,
                      DWORD ul_reason_for_call,
                      LPVOID lpReserved
                      )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```



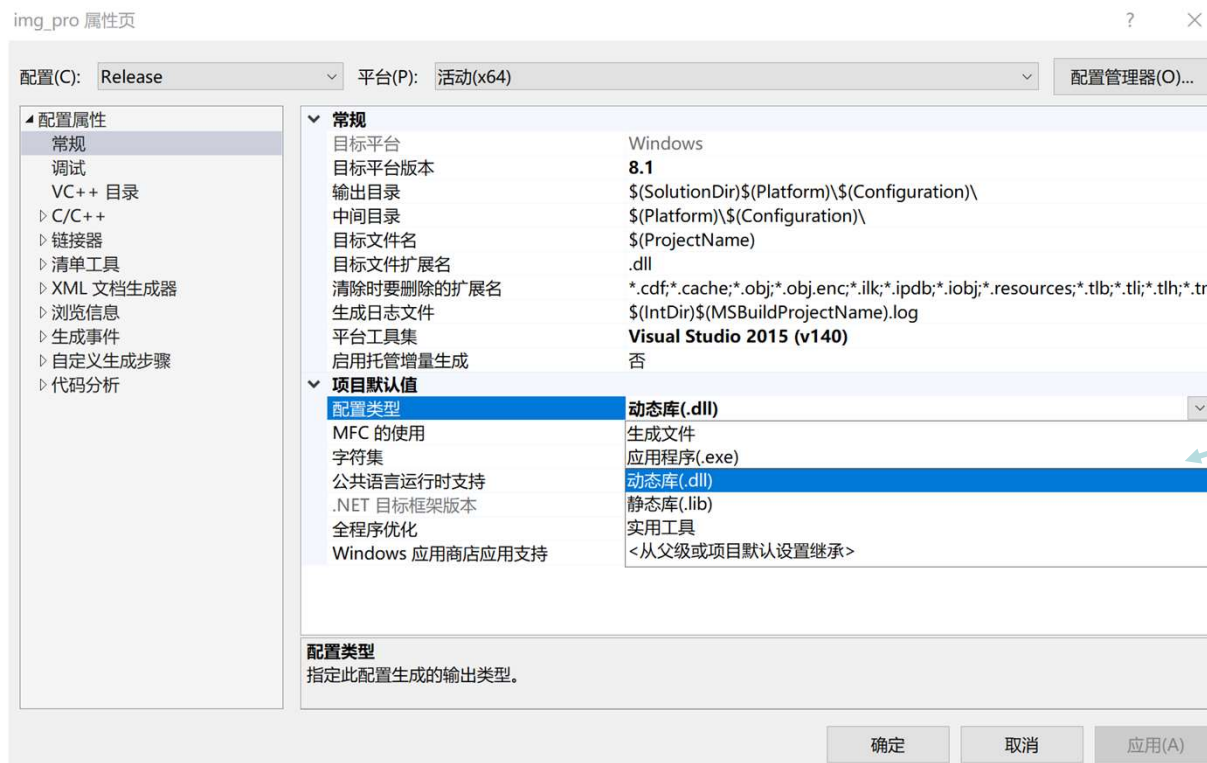


# VS中动态库的生成

库中可以被外部使用的函数需要声明关键字

`__declspec(dllexport)`

```
__declspec(dllexport) void func();
```



与之前生成静态库的方法相同，选择生成动态库

# 小结(1)

---

在本章中, 主要学习的内容有:

- 时间复杂度是算法步骤执行所需的时间
- 空间复杂度是对算法各组件利用的空间的测量
- 确定算法中使用的数据结构, 然后将其相加以测量控件复杂度
- 循环通常是程序中最耗时的构件
- 以下是用于优化循环的一些技术:
  - 删除不想要的循环部件
  - 合并循环
  - 循环并行
  - 减少循环内的工作

## 小结(2)

---

- 使用 sentinel 值
- 查看循环顺序
- 查看运算符
- 使用右值
- 并行优化

### ■ 以下是用于优化函数的一些技术:

- 使用较快的函数
- 了解数学函数
- 了解标准函数
- 将本地函数声明为静态函数

## 小结(3)

- 可通过分支来将代码某部分的控制转移到其它部分
- 以下是用于提高分支过程效率的一些技术:
  - 删除 else 子句
  - 使用有效 Case 的语句
- 在编写代码时, 您应记住以下优化准则:
  - 确定优化区域
  - 确定优化的深度
  - 确定正确的备选方法
  - 确定需要

## 小结(4)

---

- 以下是一些有关优化的常见误解:
  - 速度快的程序无需优化
  - 编译器执行的优化就已足够
  - 短代码更有效
  - 在编程时执行优化是一个好的做法
- 性能库可以大大提高应用程序的性能
- 有可用于各种用途的各种性能库, 如数学、图形和基于任务的功能