

```

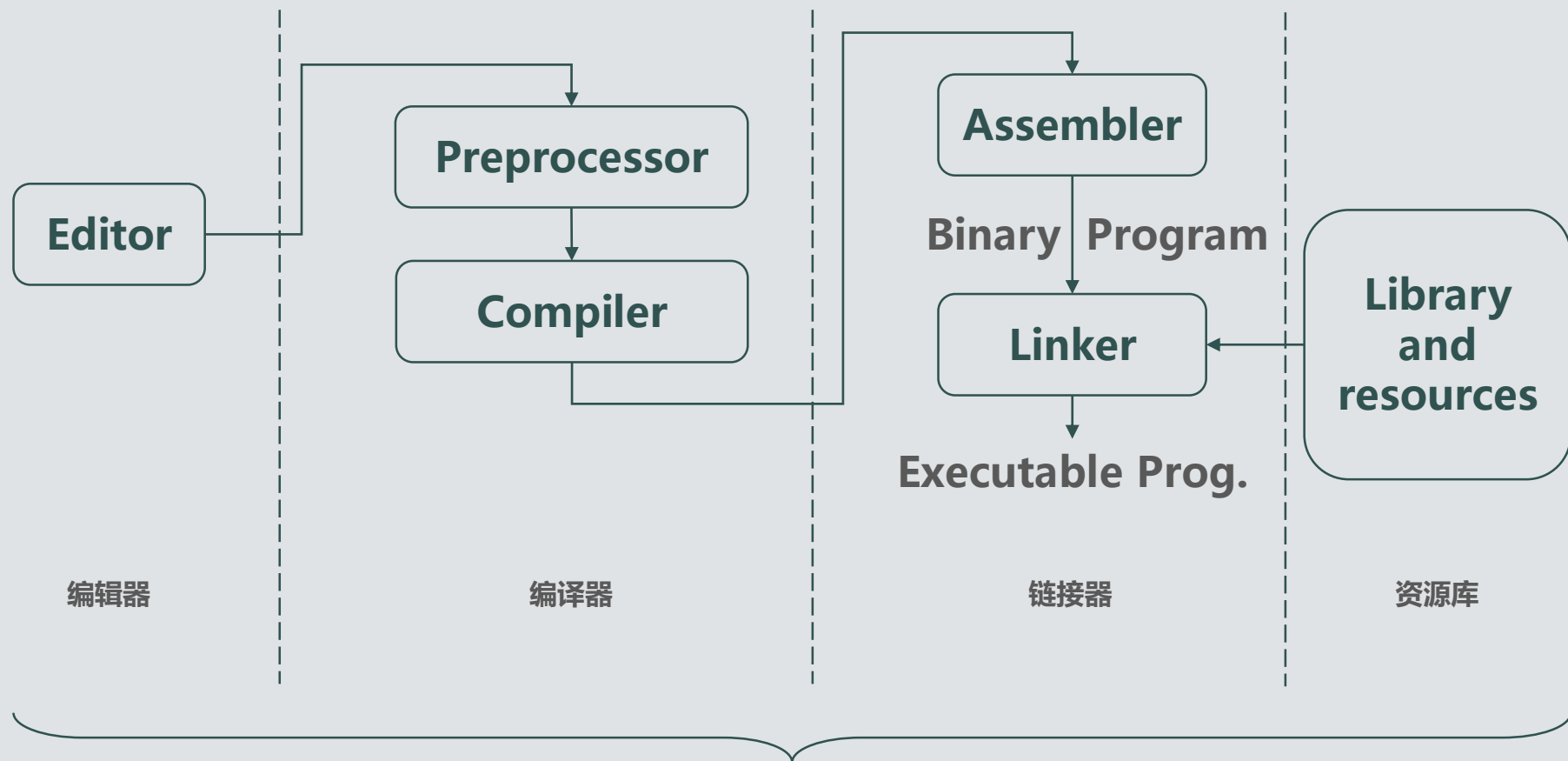
254 }
255 function updatePhotoDescription() {
256     if (descriptions.length > (page * 9) + (currentimage.substring(1) - 1))
257         document.getElementById("bigImageDesc").innerHTML += descriptions[page * 9 + currentimage.substring(1) - 1] + "  

258     }
259 }
260
261 function updateAllImages() {
262     var i = 1;
263     while (i < 10) {
264         var elementId = 'foto' + i;
265         var elementIdBig = 'bigImage' + i;
266         if (page * 9 + i - 1 < photos.length) {
267             document.getElementById(elementId).src = 'images/' + photos[page * 9 + i - 1] + '.jpg';

```

# Engineering Applications General C++

## > Something before C++ ...



**IDE, Integrated Development Environment**

## > Something before C++ ...

**Editor**      MS Office word, Text Editor, VS Code, ...

**Compiler**    ICC, GCC, Clang, VC++,... (C++)

**Linker**      (built-in)

**IDE**            Visual Studio (Microsoft, C/C++, C#, ...)  
                  Xcode (Apple, java, Object-C, C++, Swift, ...)  
                  Eclipse (Open Source, java, C++, Python, PHP, ...)

- > 各编译器对源码的处理有一定差别（“严格程度”不同），同时也与编译器版本有关。对于C++国际标准未规定的内容，各编译器会“自由发挥”。
- > 现行最新C++版本为C++17，而大多数编译器还未对最新版本进行支持，处于普及地位版本是C++11。

## > Something before C++ ...

面向过程  
Process-oriented  
偏向底层

C

C++

面向对象  
Object-oriented

偏向应用

超集  
class  
template

## Recommended references

- > **C++ Prime Plus, 6<sup>th</sup> Edition, Stephen Prata**  
(中文版, 英文原版)

**R 1.4.2** refers to ch. 1.4.2 of the Literature

### > Create Code-File and Compiling

1. 扩展名            `.cpp / .cxx`
2. 文件名            只能使用字母、数字和下划线\_, 不能以数字开头
3. 编辑器            可使用记事本(Windows), TextEdit(Mac OS)  
                        可使用VS Code(全平台)提高效率和保护视力  
                        可使用IDE包含的编辑器

#### R 1.4.2

对于编译器, Windows平台可下载Visual Studio或安装Linux子系统, 使用g++编译器编译, Mac OS平台已包含clang编译器, 可以直接使用命令行编译。

```
username/directory$ g++ codefile.cpp -o filename
username/directory$ ./filename
```

## > First C++ Program

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

特性      大小写敏感  
            空格/空行不敏感

```
EdwarddeMacBook-Pro:General_CPP edwardsue$ g++ helloworld.cpp -o hello_world
EdwarddeMacBook-Pro:General_CPP edwardsue$ ls -l
total 40
-rwxr-xr-x  1 edwardsue  staff   15788 Mar  9 23:20 hello_world
-rw-r--r--  1 edwardsue  staff     110 Mar  9 23:16 helloworld.cpp
EdwarddeMacBook-Pro:General_CPP edwardsue$ ./hello_world
hello world
```

## > First C++ Program

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

**//** 注释，行中//后内容不会被编译，用于帮助理解

**#include** 预处理指令，“添加包含库<iostream>”

**namespace** 名空间

**int main()** 主函数，程序的入口函数

**cout** 标准输出控制

**return** 结束一个需要返回值的函数

**{}** 引起一段程序段(Block)



## > First C++ Program: main() function

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

int main()

函数头，其中int表示函数返回(return)类型为整形(integer)的结果；

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

函数体，说明函数的执行过程，由数条语句构成；  
每一条具有实意的c++语句都由分号(英文)隔开；  
分号是编译器的断句符，而非空格或空行。

## > First C++ Program: notation

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

注释应简要描述语句功能，设计思想，便于纠错、阅读；  
编译器将忽略注释；

1. 以 “//” 开始，至行尾结束；
2. 以 “/\*” 开始，至 “\*/” 结束。

## > First C++ Program: pre-processor directive

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

以#开头的指令：预处理语句；

在进行主编译之前对源文件进行处理；

#include <iostream>  
将“iostream”的内容添加到代码中，  
io即输入输出in&out；

此程序中需要将语句输出到屏幕(标准输出)，需要使用cout语句，已包含在iostream内容中，因此可以调用(call)。

### > First C++ Program: namespace

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

“名空间”是一个存放内容的区域，用以区分不同封装的同名变量、函数等；

```
std::hello();
mySpace::hello();
```

使用双冒号访问符 “::” 访问名空间的内容；

```
using namespace std;
```

使用 “std” 名空间，std成为默认的名空间；

iostream的内容包含在std中，在以下内容中可以省略std::直接使用cout等。

## > First C++ Program: cout

```
#include <iostream>

using namespace std;

int main()
{
    // output "hello world"
    cout<<"hello world"<<endl;
    return 0;
}
```

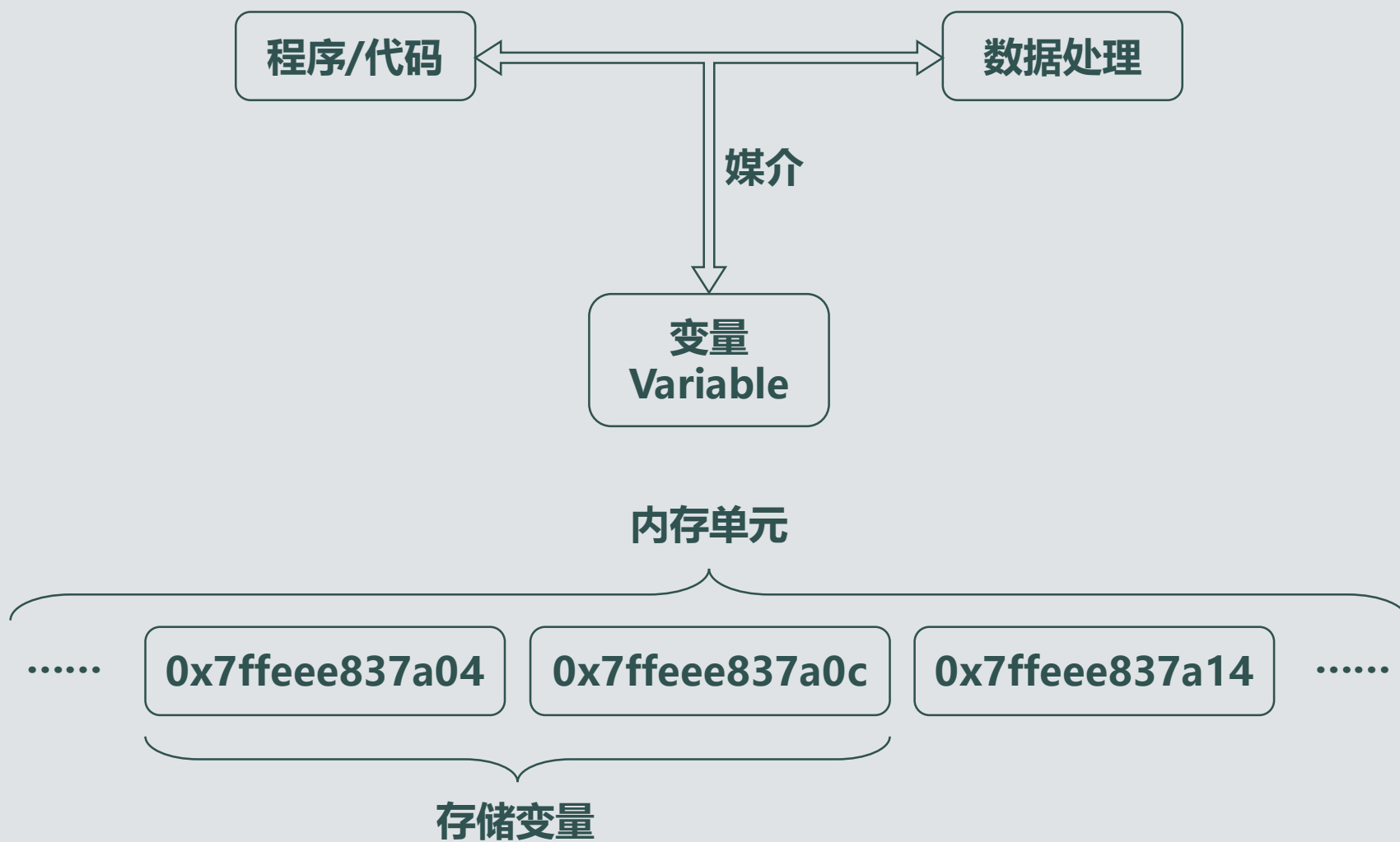
cout用以显示消息;

<<为流操作符, 显示了流的方向(输入/输出);

用双引号修饰的内容类型为字符串, cout可以接受字符串类型的参数;

endl表示换行(endline), 将屏幕光标移动到下一行开头。





## 基础格式

- 声明
- 定义

```
varType varName;  
varName = varValue;
```

\* 遵循先声明再使用规则

声明的本质是占取一片内存空间  
然后给内存空间一个合法的名称

先命名再存储值

```
varType2 varName2 = varValue2;
```

只能由字母、数字、下划线 ( \_ ) 组成;  
不能以数字开头;  
大小写敏感;  
不能使用C++关键字。

## 内存单元





## > Basic Variable Types

### 1. 整形 int (Integer)

用于表示整数，如：int score = 90;

可以用short, long, unsigned修饰，改变可表示值的取值范围。

R 3.1.3

R 3.1.4

### 2. 浮点数 float / double (Floating Point Number)

用于表示小数，如：float GPA = 1.3;

float为单精度——至少32位有效位，dec保证6位有效值；

double为双精度——至少48位有效位，通常64位，dec保证13位；

可以用long修饰。

### 3. 字符 (串) char / string (Character)

用于表示字符，如：char TshirtSize = 'M' ;

char \* name = "KIT" ; string major = "机械工程" ;

char值对应ASCII编码；

char \*为指针表示字符数组，部分情况下字符数组可与字符串混用；

string为C++引入的标准字符串变量。

### > Basic Variable Types

#### 4. 布尔类型 `bool` (Boolean)

用于表示二选一的值或情况，如：`bool pass = true;`  
`bool`变量只有`true`和`false`两种取值；  
也可以用数字来赋值/运算，非0表示`true`，0表示`false`；  
如：`bool start = -100;     // bool start = true;`  
      `bool stop = 0;         // bool stop = false;`

#### 5. 枚举类型 `enum` (Enumeration)

用于表示自定义的集合，如：`enum color {red, green, blue};`  
`color`成为新的变量名称，`red`、`green`和`blue`自动对应0,1,2；  
在枚举定义之后的使用中：  
`color desk; desk = red;`  
`color chair; chair = 0;`  
枚举将对集合中的值自动+1类推，也可显式指定对应的整数值。

R 4.6.1

## > Variable

### > Variable Operations

优先级	运算符	简介
1	. -> [ ] ( ) ++ --	各种访问符, 自增, 自减
2	! ~ & sizeof	取反, 取补, 取地址, 取长度
4	* / %	乘, 除, 取模
5	+ -	加, 减
6	<< >>	左移位, 右移位
7	< > <= >=	关系判断符
8	== !=	关系判断符
9 - 13	& ^   &&	按位与, 异或, 或; 逻辑与, 或
14	? :	三元操作符
15	= += -= etc.	赋值操作符
16	,	逗号分隔符

## > Complex Variable Types

### 1. 变量修饰 const (Constant)

用const修饰的变量，在第一次赋值之后，变量的值无法改变；  
如： `const int score = 59;` `score = 60;` 是非法的；

const变量值必须在声明时初始化，否则将会产生不确定值；  
如： `const int dis;` `dis = 120;` 是非法的；此处dis的值无法确定；

用const修饰的变量狭义上可称之为常量；  
在尽可能的情况下为变量加上const修饰（编程习惯）；

const能够加快程序编译（无需推断变量，直接从链接表访问变量）。

## > Complex Variable Types

### 2. 变量修饰 static

用static修饰的变量为全局变量，在代码任何位置都可以访问；  
如：static int count = 10;

在函数中进行初始化的全局变量只会在第一次调用赋值,如：

```
int func_count()    //返回函数调用次数
{
    static int n = 0;    //后续调用不会进行初始化
    n = n + 1;
    return n;
}
```

## > Complex Variable Types

### 3. 变量修饰 & (Reference)

用&修饰的变量称为引用变量，相当于赋予原变量一个别名；  
所有对引用变量进行的操作都会影响原变量；

```
int a = 100; int b = 200;  
int &r = a;           // r通过引用修饰赋值  
int nr = b;           // nr通过复制赋值  
  
a = 10;                // r的值随a变化, r = 10  
b = 20;                // nr的值不随b变化, nr = 200  
  
r = 1;                 // a的值随r变化, a = 1  
nr = 2;                // b的值不随nr变化, b = 20
```

## > Complex Variable Types

### 4. 数组变量 [] (Array)

数组是基于基础变量类型的序列，存放相同类型的一系列变量；  
如： `int scores[3] = {70, 80, 90};` // 声明同时初始化

数组的内容通过下标(index)访问，从0开始计数；  
如： `scores[0]`, `scores[1]`, `scores[2]`

声明数组但不初始化时必须包含元素个数；  
如： `int scores[50];` **`int scores[];`** //非法

声明同时初始化，可以让编译器推断元素个数；  
如： `int scores[] = {59, 60, 70};` //推断出元素个数为3

### > Complex Variable Types

#### 4. 数组变量 [] (Array)

字符数组需要空字符( `'\0'` )来形成字符串，因此实际长度多一位；  
如： `char name[4] = "wen" ;`

避免使用单字符形成字符数组（注意区分单双引号）；  
如： `char name[4] = { 'w' , 'e' , 'n' , '\0' }; //等价但麻烦`

字符数组同样可以让编译器推断元素数量；  
如： `char address[] = "durlach" ; //推断出元素个数为8`

数组可以嵌套，形成高维数组；  
如： `int area[2][3] = {1, 2, 3, 4, 5, 6}; //依次从最末下标赋值`  
通过 `area[m][n]` 访问数组成员。



### > Complex Variable Types

#### 4. 数组变量 [] (Array)

数组的局限性：必须显式指定元素数量，只能由常量指定。

考虑：char fullName[31];

此处认为30个字母长度的名称是“够用的”，但一旦给出超过30个字母长度的名称就会导致崩溃，但如果给出fullName[1000]则会导致资源浪费，并且也不是绝对保险；

→ 动态数组 → 指针，可以在使用时确定元素数量。

## > Complex Variable Types

### 5. 指针变量 \* (Pointer)

指针存储变量的地址，可以通过地址访问变量的值；  
此处&为取地址操作符；

声明形式形如：int \* pName;  
其中：pName表示地址；  
\*pName表示地址存放的值；

```
int day = 29;  
int *p_day;  
p_day = &day;
```

操作指针会影响原值；  
\*p\_day = 0; // day = 0

```
#include <iostream>  
  
using namespace std;  
  
int main()  
{  
    int room = 302;  
    cout<<"room value: "<<room<<endl;  
    cout<<" address: "<<&room<<endl;  
    return 0;  
}
```

```
room value: 302  
address: 0x7ffeee837a08
```

## > Complex Variable Types

### 5. 指针变量 \* (Pointer)

指针可以代替数组：

```
double *p;  
double scores[50];  
p = scores;
```

此处，指针p指向数组scores的第一个元素的**地址**；  
可以用\*(p+1)、\*(p+2).....访问数组中其他元素的**值**；

## > Complex Variable Types

### 5. 指针变量 \* (Pointer)

动态指针可以改变“先命名再存储”的模式；

R 4.7.4

→ 使用new来动态分配内存；  
如：int \*pd = new int;

此时一片无名的内存将被占用；  
pd为这片内存（开头）的地址；  
\*pd为值，但没有名称；

\* 遵循先声明再使用规则

声明的本质是占取一片内存空间  
然后给内存空间一个合法的名称

**先命名再存储值**

区别于：int \*p; int var; p = &var;

结束使用后必须使用delete pd; 来释放内存，否则将永久占用。

### > Complex Variable Types

#### 5. 指针变量 \* (Pointer)

R 4.7.6

动态指针可以管理动态数组：

```
int * pt = new int[10];
```

... ..

```
delete [] pt; //动态数组需要加上[]释放内存
```

同一内存不可以delete两次；  
如果未使用动态数组则不可以使用delete []释放内存。

## > Complex Variable Types

### 5. 指针变量 \* (Pointer)

注意：

```
int * a, b;    // a的类型为int指针, b的类型为int  
int * a, * b;  // a与b均为int指针
```

指针计算将自动转化地址单位：a+1将指向a地址的下一个地址；  
在数组指针中，若a为数组arr的指针，则a指向arr[0]的地址，  
a+1指向arr[1]的地址；

**！ 指针非常危险，谨慎地使用指针**

在使用\*varName赋值之前必须指定确切的、正确的指针地址；

如：char \*name; \*name = "zhang" ;

指针变量name地址未知，但赋值将会执行 → 内存覆盖

## > Type Conversion

### 1. 运算前提

**C++对于变量的基础运算基于相同的类型，如整形之间的加减乘除；  
如果混合计算整数变量和浮点数变量，是非法的；  
若要执行运算，则需要对变量进行类型转换。**

**\* 基础运算包含5种：加+，减-，乘\*，除/，求模(余数)%；  
求模运算只能对整数进行，浮点数求模编译器会报错；**

**\* 相同类型的变量运算结果也是该类型：  
(int)5/(int)2结果是2， 小数部分将被截去；**

## > Type Conversion

### 2. 显式(explicit)类型转换

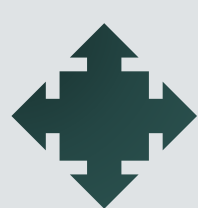
```
float f = 1.5;  
int i1 = (int)f;  // i = 1, 小数点后截去  
int i2 = int(f);  // i = 1, 小数点后截去
```

### 3. 隐式(implicit)类型转换

```
int i = 10.0/8;  // i = 1, 小数点后截去
```

\* 在除法中，如果 / 两侧的变量都是整数，执行整数除法，小数部分截去；  
如果 / 两侧有至少一个浮点数，则结果为浮点数。





# 分支语句

## Branch Statement

### > Branch Statement

分支语句用于对情况进行判断，以采取相应的操作，如：

当 $n$ 为正整数时， $\gamma(n) = (n - 1)!$

当 $n$ 为其他情况时， $\gamma(n) = \int_0^\infty \frac{t^{n-1}}{e^t} dt$

常用的分支语句有：

if-statement

switch-statement

三元操作符  $?:$

## > Branch Statement

### > Branch Statement: Logical Expression

关系/逻辑表达式的结果非true即false;

常用的关系表达式:

`>, <, >=, <=, ==, !=`

常用的逻辑表达式:

`||` 逻辑或, `&&` 逻辑与, `!` 逻辑非

如: `x > 5 || x < 0 && x > -3`

优先级	运算符	简介
4	<code>*</code> <code>/</code> <code>%</code>	乘, 除, 取模
5	<code>+</code> <code>-</code>	加, 减
9 - 13	<code>&amp;</code> <code>^</code> <code> </code> <code>&amp;&amp;</code> <code>  </code>	按位与, 异或, 或; 逻辑与, 或

## > Branch Statement

### > Branch Statement: if

句法:

```
if (test-condition)      //满足测试条件 → 执行statement(s)
    {statement(s);}      //否则跳过{...}
```

test-condition是一个关系/逻辑表达式;  
true → 满足, false → 不满足;

statement为单行时可省略花括号{ }

if可以配合else使用:

```
if (test-condition)      //满足测试条件 → 执行statement_1
    {statement_1;}
else                      //否则执行statement_2
    {statement_2;}
```

如: if(x>y) z = x;  
 else z = y;

## > Branch Statement

### > Branch Statement: if

if可以配合else if使用:

if (test-condition-1)	//满足测试条件1 → 执行statement_1
{statement_1;}	
else if (test-condition-2)	//满足测试条件2 → 执行statement_2
{statement_2;}	
...	
else if (test-condition-n)	//满足测试条件n → 执行statement_n
{statement_n;}	
...	
else	
{statement_else;}	//不满足上述任一条件, 可选

\* 一旦任一statement被执行, 下面的分支将会被跳过 (即使满足测试条件)

```
int x = 5;  
if(x > 3) x = 0;  
else if(x > 4) x = 1;  
else x = 2;
```

**x = ?**

## > Branch Statement

### > Branch Statement: switch

句法:

```
switch (integer-expression)
{
    case value_1:
        statement(s);
        break;
    case value_2:
        statement(s);
        break;
        ...
    case value_n:
        statement(s);
        break;
    default:
        statement(s);
}
```

//表达式的结果只能是整数

//结果与value相比较,  
//执行相应的statement;  
//跳出switch语句, 如果  
//没有break?

//可选, 如果不满足任一  
//value的值, 则执行

# > Branch Statement

## > Branch Statement: switch

### break的作用

```
int x = 3;
switch(x)
{
case 1: cout<<"x = 1"<<endl;
case 2: cout<<"x = 2"<<endl;
case 3: cout<<"x = 3"<<endl;
case 4: cout<<"x = 4"<<endl;
default: cout<<"x = ?"<<endl;
}
```

```
x = 3
x = 4
x = ?
```

```
int x = 3;
switch(x)
{
case 1: cout<<"x = 1"<<endl; break;
case 2: cout<<"x = 2"<<endl; break;
case 3: cout<<"x = 3"<<endl; break;
case 4: cout<<"x = 4"<<endl; break;
default: cout<<"x = ?"<<endl;
}
```

```
x = 3
```

跳出程序块(Block)，即跳出{}的范围，后续语句将被忽略。

## > Branch Statement

### > Branch Statement: operator ? :

称为三元运算符，需要三个操作数；

句法：

`expression ? statement_1 : statement_2`

如果expression为true，执行statement\_1，否则执行\_2；

如：

`a > b ? a : b`      //求a、b中较大的一个



## > Branch Statement

### > Loop Statement

循环用于重复执行任务，如：

将规模为1000的数组内容全部输出到屏幕 (array[1000])

根据不同的条件/要求，常用的循环语句有：

for-loop

while-loop

do-while-loop

# > Branch Statement

## > Loop Statement: for-loop

### 1. 循环初始化 $i = 0$

只能设置整数变量(int, short等);  
初始值可以不为0;

### 2. 循环测试 $i < 5$

检查变量是否满足条件;  
若满足则执行操作;

### 3. 循环体 {...}

实际进行的操作;  
变量*i*可代入循环体中, 也可不代入;

### 4. 循环更新 $i++$

更新用于控制循环的变量*i*;  
 **$i++$  区别于  $++i$** ;  
循环更新表达式可以是任意的。

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

int main()
{
    for(int i = 0; i<5; i++)
    {
        cout<<i<<" ". "<<"for-loop"<<endl;
    }
    cout<<"end for-loop"<<endl;
}
```

```
0. for-loop
1. for-loop
2. for-loop
3. for-loop
4. for-loop
end for-loop
```

# > Branch Statement

## > Loop Statement: for-loop

R 5.1.5

R 5.1.7

### 1. $i++$ 与 $++i$

运算结果:  $i=i+1$ ;

返回值:  $i++ \rightarrow i$ ,  $++i \rightarrow i+1$

### 2. 循环更新表达式

任意合法的表达式,  $i = i * i + 5$  .....

组合赋值运算符  $+=$ ,  $-=$ ,  $*=$ ,  $/=$   $\%=$

\* 注意{}的作用范围;  
单行循环体可以省略{};

\* 不进行操作时保留分号, 表达式省略: `for(;; i++)`  
循环测试语句为空, 意味始终满足条件。

# > Branch Statement

## > Loop Statement: while-loop

1. 循环初始化  $i = 0$   
只能设置整数变量(int, short等);  
初始值可以不为0;
2. 循环测试  $i < 5$   
检查变量是否满足条件;  
若满足则执行操作;
3. 循环体 {...}  
实际进行的操作;  
变量*i*可代入循环体中, 也可不代入;
4. 循环更新  $i++$   
更新用于控制循环的变量*i*;  
 **$i++$  区别于  $++i$** ;  
循环更新表达式可以是任意的。

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

int main()
{
    int i = 2;
    while(i<5)
    {
        cout<<i<<". while-loop"<<endl;
        i++;
    }
    cout<<"end while-loop"<<endl;
}
```

```
2. while-loop
3. while-loop
4. while-loop
end while-loop
```

# > Branch Statement

## > Loop Statement: do-while-loop

1. 循环初始化  $i = 0$   
只能设置整数变量(int, short等);  
初始值可以不为0;
2. 循环测试  $i < 5$   
检查变量是否满足条件;  
若满足则执行操作;
3. 循环体 {...}  
实际进行的操作;  
变量*i*可代入循环体中, 也可不代入;
4. 循环更新  $i++$   
更新用于控制循环的变量*i*;  
 **$i++$ 区别于 $++i$** ;  
循环更新表达式可以是任意的。

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

int main()
{
    int i = 2;
    do
    {
        cout<<i<<". while-loop"<<endl;
        i++;
    }while(i<5);
    cout<<"end do-while-loop"<<endl;
}
```

```
2. while-loop
3. while-loop
4. while-loop
end do-while-loop
```

## > Branch Statement

### > Loop Statement

**while-loop**首先判断循环条件，满足则执行循环体；

R 5.3

**do-while**首先执行循环体，再判断是否继续执行。

```
while(i<5)
{
    cout<<i<<" . while-loop"<<endl;
    i++;
}
```

```
do
{
    cout<<i<<" . while-loop"<<endl;
    i++;
}while(i<5);
```

循环可以嵌套、混合使用：

```
for(int i=0; i<5; i++)
{
    for(int j=0; j<4; j++)
    {
        while(...) {...}
    }
}
```

# > Branch Statement

## > Loop Statement

### continue的作用

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

int main()
{
    for(int i = 0; i < 10; i++)
    {
        if (i%2 == 0) continue;
        cout<<i<<" . loop, not be skipped"<<endl;
    }
}
```

```
1. loop, not be skipped
3. loop, not be skipped
5. loop, not be skipped
7. loop, not be skipped
9. loop, not be skipped
```

不再执行continue之后的语句;  
进行变量更新(for-loop), 执行下一次循环。





## > Function: Introduction

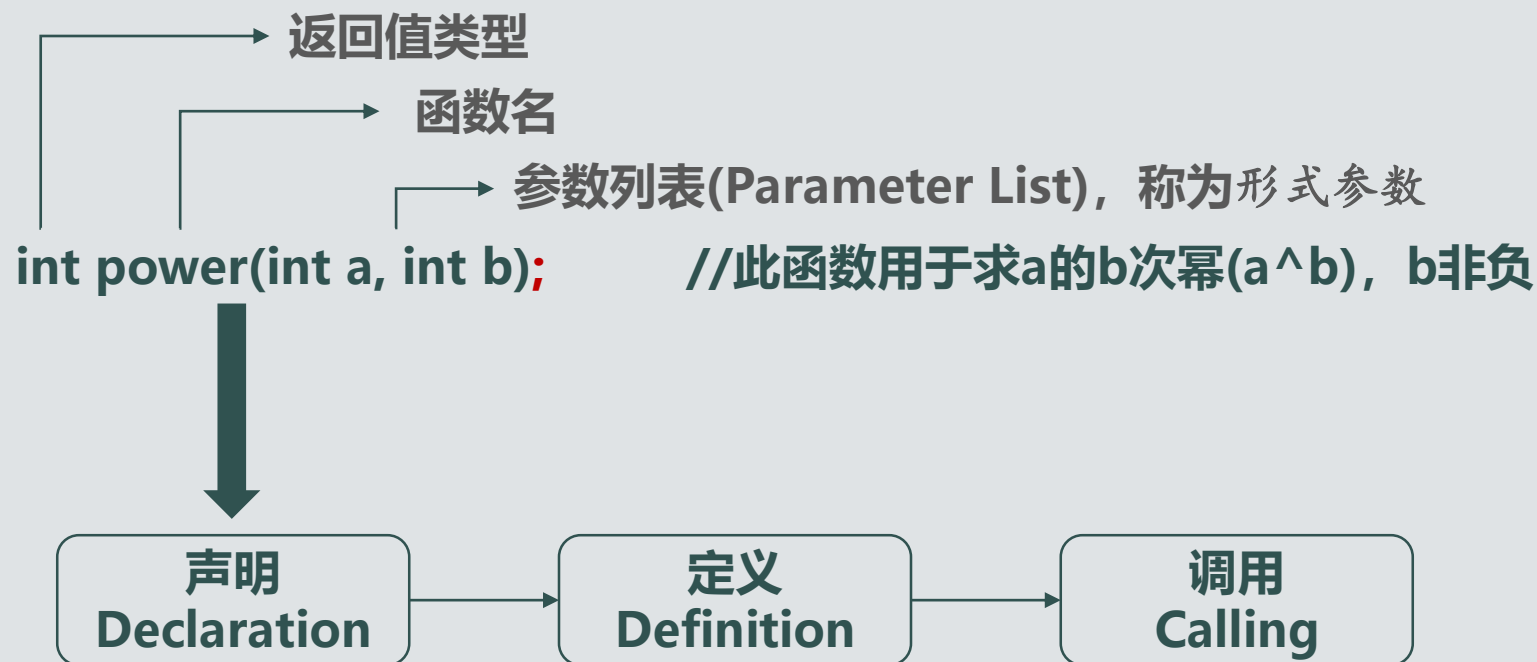
函数是完整程序的子程序，用于完成特定的任务，具有相对的独立性；

一般具有输入参数，并且有返回值，建立映射关系；

主函数main()是一个特殊的函数，它具有函数的所有特征；



## > Function: Introduction

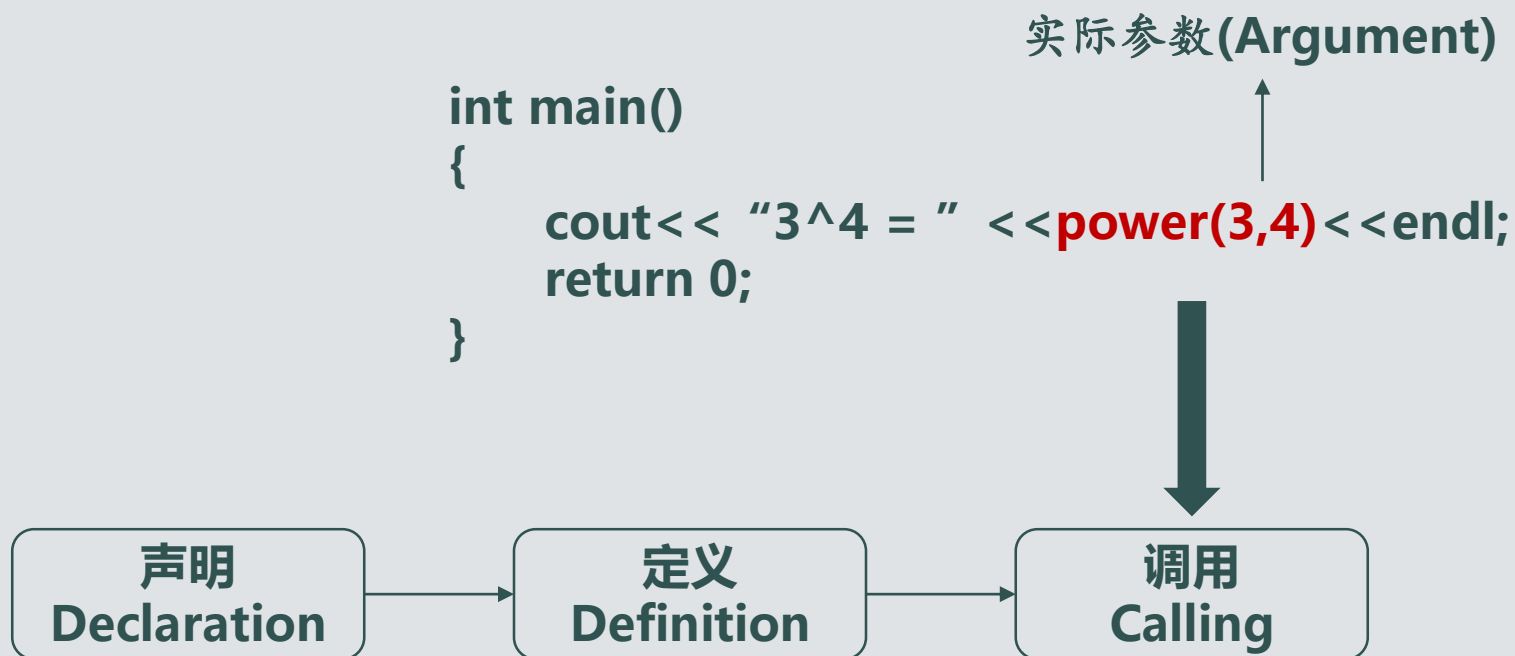


## > Function: Introduction

```
int power(int a, int b)
{
    if(b == 0 && a == 0) return 0;
    else if(b == 0 && a != 0) return 1;
    else {
        int i = 1;
        while(b-->0) i *= a;
        return i;
    }
}
```



## > Function: Introduction



## > Function: Return

返回值可以是任何变量类型，但**不可以是数组类型**；

特殊的返回值类型void：

“空”类型，不具有实际的值(Value)，无需写return语句；  
用于执行操作而不需要返回含有实际意义的值；  
如打印一个int数组的所有内容：

### R 7.3

方括号表示参数为数组，方括号为空表示数组长度任意（编写函数时将array[]看做数组即可，实际上这是一个指针，可用int \* arr替换，这个替换仅在函数声明中可以互换）

```
void print_all(int array[], int length)
{
    for(int i = 0; i < length; i++)
        cout << array[i] << " ";
    cout << endl;
}
```

```
int numbers[10];
for(int i = 0; i < 10; i++) numbers[i] = i;
int count = sizeof(numbers) / sizeof(int);
print_all(numbers, count);
```

用sizeof()可以求出变量的size，但是在函数中不可使用，因为函数中传值时，只传递了数组指针（只是一个地址，没有长度）

## > Function: Return

**递归(Recursion):** 以函数自身作为返回值, 如斐波那契数列:

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

- \* 递归非常消耗资源, 因为函数在内存中运行, 递归将使得数以千百计个函数同时占用内存。
- \* 将函数视为可展开的一条语句, 该语句的值即返回值: 函数可与变量一同代入计算或其他应用。

## > Function: Parameter and Argument

参数传递是函数的核心，在此之前需了解：

### 变量的作用域/生命周期

仅限于程序块(Block, {}), 一旦在{}范围以外，此变量即销毁；  
程序块可以嵌套；  
全局变量不会销毁，若有同名变量需要用::访问符。

```
#include <iostream>
using namespace std;
int a = 1, b = 2;
int main()
{
    int a = 10;
    cout<<"global a = "<<::a<<endl;
    cout<<"local a = "<<a<<endl;
    {
        int b = 20; int c = 30;
        cout<<"global b = "<<::b<<endl;
        cout<<"local b = "<<b<<endl;
    }
    cout<<"global b without :: = "<<b<<endl;
    // cout<<"c = "<<c<<endl; // c is not visible
}
```

```
global a = 1
local a = 10
global b = 2
local b = 20
global b without :: = 2
```

## > Function: Parameter and Argument

参数传递是函数的核心，在此之前需了解：

**变量的作用域/生命周期**

仅限于程序块(Block, {}), 一旦在{}范围以外，此变量即销毁；  
程序块可以嵌套；

全局变量不会销毁，若有同名变量需要用::访问符。

函数自动具有块标记{}, 因此变量在函数运行结束之后会销毁：

```
void swap_value(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

swap x,y by value pass: x = 3,y = 5

```
int x = 3, y = 5;
swap_value(x,y);
cout<<"swap x,y by value pass: "<<"x = "<<x<<"<<"y = "<<y<<endl;
```



### > Function: Parameter and Argument

参数传递是函数的核心，在此之前需了解：

变量的作用域/生命周期

仅限于程序块(Block, {}), 一旦在{}范围以外，此变量即销毁；  
程序块可以嵌套；

全局变量不会销毁，若有同名变量需要用::访问符。

函数自动具有块标记{}，因此变量在函数运行结束之后会销毁：

R 7.2

**复制传值**，又称为按值传递(Value Pass)，形参将实参复制了一份，在函数运行结束之后，为复制变量创建的内存将会释放 → 变量销毁。

如何实现交换两个变量的值(Swap)?

## > Function: Parameter and Argument

R 8.2

**引用传值**，又称为引用传递(Reference Pass)，形参不复制实参；通过在形参前加&符号形成引用传值，传入的是实参变量的地址；实参处于更大的生命周期中，在函数结束时实参变量不会销毁，对实参变量进行的操作则会保留：

```
void swap_ref(int &a, int &b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

调用时无需加上&符号

R 7.2

**复制传值**，又称为按值传递(Value Pass)，形参将实参复制了一份，在函数运行结束之后，为复制变量创建的内存将会释放 → 变量销毁。

✓ 如何实现交换两个变量的值(Swap)?

### > Function: Parameter and Argument

R 8.2

**引用传值**，又称为引用传递(Reference Pass)，形参不复制实参；通过在形参前加&符号形成引用传值，传入的是实参变量的地址；实参处于更大的生命周期中，在函数结束时实参变量不会销毁，对实参变量进行的操作则会保留：

R 7.2

**复制传值**，又称为按值传递(Value Pass)，形参将实参复制了一份，在函数运行结束之后，为复制变量创建的内存将会释放 → 变量销毁。

实际上还有第三种传值方式，称为指针传递(Pointer Pass)，即将变量的指针作为参数传入函数，声明如：

```
typeName funcName(typeName *varName);
```

数组作为函数参数时，就使用了指针传值（又称为**地址传递**）。

## > Function: Overload

### R 8.4 函数重载指的是：

对于相同的函数名，针对不同的参数（数量、类型），采取不同的操作，以达到不同的目的；

这种特性称之为函数的多态。

如计算几个“数”相加：

```
int add(int a, int b);  
int add(int a, int b, int c);  
double add(double a, double b);  
complex add(complex a, complex b);
```

这里每个函数都叫做add，在调用这些函数时，编译器将通过实参的类型和数量，自动选定相应的函数。

### > Function: Overload

函数重载可以实现逻辑合理的许多功能，不再需要使用多余的名称去命名一些功能重复的函数；

不仅仅是函数，各种**运算符也可以重载**，如：+ - \* / ^等；  
(后续会涉及到运算符重载)

C++已经内置重载了许多函数和运算符；

如果使用VS Code或者其他具有类似功能的编辑器，将鼠标悬置于运算符/函数上，可以看到此运算符/函数被重载了多少次。

## > Function: Array as Parameter (Special Topic)

R 7.3

```
int sum_arr(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++) total += arr[i];
    return total;
}
```

考虑以上代码，用于int数组元素求和；

**当且仅当**用于函数头(函数声明)时，int arr[]与int \*arr是等效的；

arr[]是表面，实际上是一个指针，指向传入数组的第一个元素；  
(如果数组十分庞大，传值时进行复制将消耗非常多的资源；  
因此使用指针指向数组第一个元素进行传值，避免资源浪费。)

因此**在函数中**使用sizeof()并不能正确得到**数组参数**的长度。  
(得到的是指针/地址的size)

### > Function: Array as Parameter (Special Topic)

R 7.3

```
int sum_arr(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++) total += arr[i];
    return total;
}
```

函数无法使用类似于sizeof这样的方法自行判断数组元素的个数，因此参数需要显式指定(此函数参数中的int n)；

除指定int n之外，还有一种方式用以判断函数数组结束的标识：采用**两个指针**作为参数，一个指向开始的位置，一个指向结束的位置；

声明类似于：

```
int sum_arr(int *arr_begin, int *arr_end);
```

## > Function: Array as Parameter (Special Topic)

R 7.3

```
int sum_arr(int * arr_begin, int * arr_end)
{
    int total = 0;
    while(arr_begin < arr_end) total += *arr_begin++;
    return total;
}
```

具体的实现方法如上;

请注意`total += *arr_begin++;`的运算符优先级;  
如果对优先级无把握, 可拆分为: `total+=*arr_begin; arr_begin++;`

调用方法如下:

```
int numbers[10];
for(int i = 0; i<10; i++) numbers[i] = i;
cout<<"sum of numbers = "<<sum_arr(numbers,numbers+10)<<endl;
```

此方法在处理字符串数组时十分实用 (指针指向字符串开头和结尾)。



## > Function: Array as Parameter (Special Topic)

R 7.3

数组无法作为函数的返回值类型，但是**指针可以** → 用指针替代数组；

```
double * vec_add(double * vec_2_1, double * vec_2_2)
{
    static double r[2];
    r[0] = vec_2_1[0] + vec_2_2[0];
    r[1] = vec_2_1[1] + vec_2_2[1];
    return r;
}
```

以上代码可以实现两个2x1向量相加，采用指针作为返回值类型；  
函数声明中的double \* vec\_2\_1等价于double vec\_2\_1[];

**作为返回对象的指针，必须能够在出函数之后不销毁：**  
使用static前缀 或 double \* r = **new** double[2];

```
double vec2[] = {PI/6,0.0};
double vec2_2[] = {3.0, 4.5};
double * vec = vec_add(vec2_2,vec2);
```

调用方式如上，PI为宏定义，#define PI 3.1415926

### > Function: Other Considerations

函数声明可以省略参数名（变量名），定义时不可省略，如：

```
int swap(int, int);
```

数组不可以作为返回值类型，以下写法是错误的：

```
int[] invert_arr(int []); //数组不可作为返回值类型
```

但是数组作为结构体/类的成员，可以作为返回值，在后续进行讲解；

使用const修饰的函数参数（形参），可以接受const或非const的实参；  
不用const修饰则只能接受非const实参，修饰如：

```
int print(const int []);
```

应尽量将具体功能用函数实现（代码模块化），main函数应尽量调用各个子函数。



类  
Class

## > How to describe vector?

描述一个2x1的向量:

数组: `double vec_2[2];`

向量之间的运算:

函数: `double * add(double * vec_1, double * vec_2);`

对比其他类型之间的运算:

```
double a = 1.3; double b = 2.4;  
double c = a + b;
```

如果“向量”能够成为double这样的变量类型?

→ 类

## > Class: Introduction

**类是一个自定义的数据类型：**

**能够将数据的表示和数据的操作整合在一起；**

**类声明：以数据成员的方式描述数据部分，以成员函数（称之为方法）的方式描述对数据的操作；**

**类定义：描述如何实现成员函数（对数据的操作）；**

**声明提出类的蓝图，定义则是实现的细节。**

## > Class: 2 Dimensional Vector

声明:

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    double get_x();
    double get_y();
    void show();
```

```
};
```

结尾的分号必不可少

对象，又称为实例 (Instance)，即由抽象的变量类型创建的实际变量，如：int x; vec2 vector1; x和vector1则被称为int和vec2的实例或对象。

访问，即通过“.”访问符实现，如通过vec2 vector1创建一个实例，则可以通过vector1.get\_x()访问成员函数。

访问控制，**类的对象可以直接访问所有的公有成员**，但无法访问类的私有成员；必须通过特殊的方式，或者公有成员方法访问；

除了private和public之外，还有第三种访问控制关键字：protected；protected的权限介于public与private之间，后续会详细讨论；

通过访问控制防止程序直接访问数据成员，称为数据隐藏，也是封装 (Encapsulation) 的一种形式。

## > Class: 2 Dimensional Vector

声明:

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    double get_x();
    double get_y();
    void show();
```

```
};
```

结尾的分号必不可少

访问控制是C++区别于C的核心，也是面向对象编程(Object-Oriented Programming)的核心；

用户无需了解数据是如何表示的，只需要知道成员函数的功能；

可以将具体的实现和接口 (Interface) 分隔开，修改实现而接口无需改动；

C++鼓励进行数据封装，因此类的成员默认为private属性(即private可省略)。

## > Class: 2 Dimensional Vector

通过域访问符::  
标识成员函数  
(方法)的归属

声明:



实现(定义):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    double get_x();
    double get_y();
    void show();
};
```

其他部分就如  
同函数定义  
(方法首先也  
是一个函数);

类的方法可以  
访问private  
成员;

R 10.2.3

```
double vec2::get_x()
{
    return m_x;
}
double vec2::get_y()
{
    return m_y;
}
void vec2::show()
{
    cout<<m_x<< " "
        <<m_y<<endl;
}
```

\* 类的声明可以放在一个单独的文件中, 与类的定义文件同名, 路径相同, 以.h为后缀名(头文件), 需要加上#include "className.h" 进行编译。



## > Class: 2 Dimensional Vector

小结:

```
class className
{
    private:
        data member declarations
    public:
        member function prototypes
};
```

\* 数据成员和成员函数，并无规定谁是公有谁是私有，一切按照需求设定；

\* 成员函数也可以在类中直接定义，无需在类外使用域访问符::定义；

\* 谁的数据成员？

R 10.2.3.3

```
class vec2
{
    double m_x;
public:
    double get_x();
};
```



```
double vec2::get_x()
{
    return m_x;
}
```



```
vec2 vector1, vector2;
double sum_x = vector1.get_x()
               + vector2.get_x();
```

对象/实例的成员，“我”  
的m\_x, **this->m\_x**

## > Class: 2 Dimensional Vector

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    double get_x();
    double get_y();
    void show();
};
double vec2::get_x() {return m_x;}
double vec2::get_y() {return m_y;}
void vec2::show()
{
    cout<<setw(4)<<m_x<<endl;
    cout<<setw(4)<<m_y<<endl;
}
```

```
int main()
{
    vec2 vector1;
    vector1.show();
    return 0;
}
```

```
0
6.95313e-310
```

变量的值由运行时内存随机赋予

如何对类中的数据成员进行赋值？ → 构造函数

## > Class: 2 Dimensional Vector

构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    double get_x();
    double get_y();
    void show();
};
```

vec2()是默认的构造函数，无参数；

vec2(double x, double y)是带参数的构造函数；

构造函数的作用就是在类创建对象/实例时，给对象一个初值，就如同默认变量类型赋初值：

```
double x = 5;
double x(5);
double x{5};    //C++11
```

默认构造函数不复制，不显式写出时，编译器会自动创建。

## > Class: 2 Dimensional Vector

### R 10.3.1

#### 构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    double get_x();
    double get_y();
    void show();
};
```

构造函数也是成员函数，同时必须是公有属性：public;

如果在类外定义，也需要域访问符::限定:

```
vec2::vec2()
{
    cout << "call default
constructor" << endl;
}
```

不需要返回值，即声明和定义前没有返回值类型;

**默认构造函数**可以省略或使用以下方式无需定义：(在类声明中)

```
vec2() = default; //C++11
```

\* 带有C++11标记的内容，需要支持C++11的编译器，编译语句加上：--std=c++11

## > Class: 2 Dimensional Vector

构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    double get_x();
    double get_y();
    void show();
};
```

类似的，对于带参数的构造函数的定义如下：

```
vec2(double x, double y)
{
    m_x = x;
    m_y = y;
    cout << "call x_y
constructor" << endl;
}
```

带参数的构造函数也可以成为默认构造函数，方法是声明中加上默认值：  
vec2(double x=1., double y=2.); //定义时无需默认值

构造函数名称即类的名称；  
不同构造函数实质上是函数的重载。

## > Class: 2 Dimensional Vector

构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    // vec2();
    vec2(double x=1.,
        double y=2.);
    double get_x();
    double get_y();
    void show();
};
```

类似的，对于带参数的构造函数的定义如下：

```
vec2(double x, double y)
{
    m_x = x;
    m_y = y;
    cout << "call x_y
constructor" << endl;
}
```

带参数的构造函数也可以成为默认构造函数，方法是声明中加上默认值：

```
vec2(double x=1., double
y=2.); //定义时无需默认值
```

**一个类只能有一个默认构造函数！**

## > Class: 2 Dimensional Vector

构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    double get_x();
    double get_y();
    void show();
};
```

具体调用哪一个构造函数，取决于实例化对象的构造方式:

```
vec2::vec2()
{
    cout<<"● call: constructor,
    default"<<endl;
}
vec2::vec2(double x, double y)
{
    m_x = x; m_y = y;
    cout<<"● call: constructor,
    double x double y"<<endl;
}
```

```
vec2 vector1;
vec2 vector2(2.,4.);
```

- call: constructor, default
- call: constructor, double x double y

## > Class: 2 Dimensional Vector

### 构造函数(Constructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    double get_x();
    double get_y();
    void show();
};
```

```
vec2 vector1;
vec2 vector2(2.,4.);
```

在构造对象/实例时，构造函数就会被调用；

对于带参数的构造函数，还可以使用以下方式调用：

```
vec2 vector1 = vec2(1., 2.);
```

**不要混淆构造函数中的参数与类中的数据成员。**



## > Class: 2 Dimensional Vector

### 析构函数(Destructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
};
```

在由类创建的对象“失去价值”(往往是程序运行结束、不再需要某一变量)的时候, 这个对象需要销毁, 把占用的内存空间释放出来:

→ 析构函数

用于完成清理工作;

声明: 在构造函数前加“~”符号;

无论有多少个构造函数, 都**只有一个**析构函数。

## > Class: 2 Dimensional Vector

### 析构函数(Destructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
};
```

与构造函数类似，析构函数也有编译器自动创建的默认版本，在不进行析构函数声明时，默认析构函数即会被调用；

一旦声明析构函数，就必须进行定义：

```
vec2::~~vec2()
{
    cout << "call default
destructor" << endl;
}
```

## > Class: 2 Dimensional Vector

析构函数(Destructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
};
```

析构函数在程序中无需显式调用，在内存释放时将会自动调用此函数(往往是程序运行结束时):

```
int main()
{
    vec2 vector1;
    vec2 vector2(2.,4.);
    vector1.show();
    vector2.show();
    return 0;
}
```

- call: constructor, default
- call: constructor, double x double y
- 0
- 6.95312e-310
- 2
- 4
- call: destructor, default
- call: destructor, default

## > Class: 2 Dimensional Vector

### 析构函数(Destructor):

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
};
```

析构函数一定没有参数的;

在没有使用new操作符的情况下, 空的析构函数/默认析构函数是可行的;  
(程序运行结束自动释放内存)

如果数据成员使用了new操作符, 则必须在析构函数中使用delete语句将相应的内存释放。

## > Class: 2 Dimensional Vector

### 小结:

```
class className
{
private:
    data member declarations
public:
    Constructor(s)
    Destructor
    member function prototypes
};
```

\* 构造函数和析构函数都是公有的成员函数，符合类方法的一切特性；

\* 构造函数在创建对象/实例时自动调用；

\* 析构函数在对象/实例销毁时自动调用；

\* 构造函数与析构函数都没有返回值类型（即使是void也不行）；

\* 类的函数与变量区别在于声明时有没有括号()。

## > Class: 2 Dimensional Vector

至此，我们完成了一个二维向量类的基本框架：

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
};
```

成为完善的二维向量“变量类型”还需要添加一些功能：

**向量**之间的加法;  
**向量**之间的点乘;  
**向量**与**实数**相乘（尺度变换）;

考虑这些计算的结果类型：

加法 → **二维向量**  
点乘 → double  
尺度变换 → **二维向量**

## > Class: 2 Dimensional Vector

至此，我们完成了一个二维向量类的基本框架：

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
    vec2 add(vec2 v);
    double dot_product(vec2 v);
    vec2 scale(double r);
};
```

加法 → **二维向量** → **vec2**

点乘 → **double**

尺度变换 → **二维向量** → **vec2**

新添加成员函数定义：

```
vec2 add(vec2 v);
double dot_product(vec2 v);
vec2 scale(double r);
```

考虑：为何这些函数只有一个参数？

## > Class: 2 Dimensional Vector

点乘函数定义:

```
double vec2::dot_product(vec2 v)
{
    return m_x * v.m_x + m_y * v.m_y;
}
```

向量相加函数定义:

```
vec2 vec2::add(vec2 v)
{
    m_x += v.m_x;
    m_y += v.m_y;
    return ??? ;
}
```

尺度变换函数定义:

```
vec2 vec2::scale(double r)
{
    m_x *= r;
    m_y *= r;
    return ??? ;
}
```

此处应返回(return)哪个值才是vec2类型的?



## > Class: 2 Dimensional Vector

向量相加函数定义:

```
vec2 vec2::add(vec2 v)
{
    m_x += v.m_x;
    m_y += v.m_y;
    return ??? ;
}
```

尺度变换函数定义:

```
vec2 vec2::scale(double r)
{
    m_x *= r;
    m_y *= r;
    return ??? ;
}
```

此处应返回(return)哪个值才是vec2类型的?

**return \*this;**

使用被称为this的特殊指针;  
this指针自动指向类的对象, 如:  
**vec2 vector;    //则this指针就是vector的地址**

## > Class: 2 Dimensional Vector

### this指针

每个成员函数/类的方法都包含一个this指针;

this指针指向调用的对象:

vec2 vector1;

vec2 vector2;

vector1.this指向vector1对象;

vector2.this指向vector2对象;

如果需要调用整个对象, 需要加上\*符号, 正如return \*this一样;

指针的内容通过 “->” 访问符获取, 在只涉及一个对象的成员函数中, this->可以省略 (m\_x实质上是this->m\_x的缩写):

```
double vec2::get_x()
{
    return m_x;
}
```

## > Class: 2 Dimensional Vector

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    vec2();
    vec2(double x, double y);
    ~vec2();
    double get_x();
    double get_y();
    void show();
    vec2 add(vec2 v);
    double dot_product(vec2 v);
    vec2 scale(double r);
};
```

```
vec2 vec2::add(vec2 v)
{
    m_x += v.m_x;
    m_y += v.m_y;
    return *this;
}
```



```
vec2 add_outClass(vec2 v1, vec2 v2)
{
    vec2 vR(v1.get_x()+v2.get_x(),
            v1.get_y()+v2.get_y());
    return vR;
}
```

为何这些函数只有一个参数？ —— 类的成员函数与普通函数的区别

## > Class: Other Considerations

类同样可以创建数组方式的对象：

```
vec2 vectors[5];           //这将调用默认构造函数  
vec2 vectors2[2] =  
    {vec2(1.,2.), vec2(3.,4.)} //调用带参数的构造函数
```

**this**指针无需在函数声明的参数列表中出现，可直接使用；

类的数据成员不可以在声明时赋值：

```
class vec2{double m_x = 1.0;}; //非法
```

这是因为类在没有创建对象/实例的时候是不占用内存空间的，变量的值也就无处存放；

例外：使用**static**前缀可以在数据成员声明时赋值：

```
class vec2{static double m_x = 1.0;};
```

## > Struct

结构(Structure)也是一种自定义的变量类型，句法与类相似：

```
struct structName
{
    varType1 varName1;
    varType2 varName2;
    ... ...
};
```

```
structName ins_1;    //通过结构名称创建对象/实例
... ins_1.varName1 ... //通过.访问符获取结构体的变量
```

### > Struct

#### 结构与类的异同：

结构是C时代的产物，默认成员为public属性；  
类是C++的产物，默认成员为private属性（数据封装）；

结构也可以通过添加public、protected、private等关键字改变访问权限；

结构在进行继承时，默认为公有(public)继承；  
类在进行继承时，默认为私有(private)继承。

## > Accessor

**直接成员访问符 .**  
用于访问类、结构的成员；

**间接成员访问符 ->**  
用于访问指针的成员；

**域访问符 ::**  
用于访问域、namespace的成员；  
在类外定义函数时，类名成为成员函数的域，需要使用::访问符。

### > Multiple Files Compiling

声明:

P. 72

```
class vec2
{
private:
    double m_x;
    double m_y;
public:
    double get_x();
    double get_y();
    void show();
};
```

\* 类的声明可以放在一个单独的文件中，与类的定义文件同名，路径相同，以.h为后缀名(头文件)，需要加上#include "className.h" 进行编译。

# How?



## > Multiple Files Compiling

声明放置于以类命名的头文件`vec2.h`:

```
#ifndef VEC2_  
#define VEC2_  
  
class vec2  
{  
private:  
    double m_x;  
    double m_y;  
public:  
    vec2();  
    vec2(double x, double y);  
    ~vec2();  
    double get_x();  
    double get_y();  
    void show();  
    vec2 add(vec2 v);  
    double dot_product(vec2 v);  
    vec2 scale(double r);  
};  
  
#endif
```

```
#ifndef x  
#define x  
    ... ..  
#endif
```

是预编译命令：如果未定义x，则对x进行定义；

用于控制一段代码只进行编译一次。

## &gt; Multiple Files Compiling

定义放置于以类命名的文件 **vec2.cpp**:

```
#include <iostream>
#include <iomanip>
#include "vec2.h"

using namespace std;

vec2::vec2() {cout<<"● call:
    constructor, default"<<endl;}
vec2::vec2(double x, double y)
{
    m_x = x; m_y = y;
    cout<<"● call: constructor,
    double x double y"<<endl;
}
vec2::~~vec2() {cout<<"● call:
    destructor, default"<<endl;}
double vec2::get_x() {return m_x;}
double vec2::get_y() {return m_y;}
void vec2::show()
{
    cout<<setw(4)<<m_x<<endl;
    cout<<setw(4)<<m_y<<endl;
}
vec2 vec2::add(vec2 v)
{
    m_x += v.m_x;
    m_y += v.m_y;
    return *this;
}
double vec2::dot_product(vec2 v)
{
    return m_x * v.m_x + m_y * v.m_y;
}
vec2 vec2::scale(double r)
{
    m_x *= r;
    m_y *= r;
    return *this;
}
```

## > Multiple Files Compiling

主程序使用**#include** “**vec2.h**” 包含这个类：

```
#include "vec2.h"

int main()
{
    vec2 vector1;
    vec2 vector2(2.,4.);
    vector1.show();
    vector2.show();
    return 0;
}
```

此时，在主程序中，vec2已经是一个已定义的自定义变量类型，可以直接使用、调用成员函数；

编译此主程序时，由于有多个源文件，因此需要在编译命令中添加所有源文件名称：

```
g++ mainFile.cpp vec2.cpp -o programName
```