头文件

#include <iostream>

Std命名空间

using namespace std;

注释

// 注释

/\* 注释 \*/

在Atom软件里面ctrl + / 可以快速（多行）注释/取消注释

主函数

int main()

{

cout << "Hello World";

return 0;

}

显示内容

cout << "Hello World"; // 打印Hello World之后不换行

cout << "Hello World" << endl; // 打印之后换行

每个内容之前都要有一个 <<

cout << "Hello " << “World！”;

cout << "Hello " << “World！” << "\n";

数据类型

布尔值 bool

字符串 char

整数 int

浮点数 float

双浮点数 double

无类型 void

宽字符 wchar\_t

类型定义 typedef

tpyedef int zhengshu

进而定义整数a就相当于

int a 相当于 zhengshu a

枚举类型：enum

enum 枚举名 {标识符1,标识符2...} 枚举变量

enum color {red, green, blue} c;

c = red;

其中 标识符分别对应 0, 1, ... 如果 cout << c; 则得到

0

如果

enum color {red, green = 5, blue} c;

则 他们分别对应0, 5, 6

定义变量

int a, b, c;

char d, e;

float f, g;

double h;

也可以同时赋值：

int a, b = 1, c;

例子：

#include <iostream>

using namespace std;

int main ()

{

// 变量定义

int a, b;

int c;

float f;

enum color { red, blue = 5, green } g;

g = blue;

// 实际初始化

a = 10;

b = 20;

c = a + b;

cout << c << endl;

f = 70.0/3.0;

cout << f << endl;

cout << g << endl;

return 0;

}

返回的结果

30

23.3333

5

定义函数

先声明函数，然后可以在任意位置定义函数

int func(); // 声明存在函数func

int main()

{

int i = func(); // 函数调用func

}

int func() // 函数定义

{

内容

return 结果;

}

局部变量、全局变量

局部变量：在函数内部声明，只在函数内部有效

全局变量：在函数外部声明

#include <iostream>

using namespace std;

int g; // 全局变量声明

int main ()

{

int a, b; // 局部变量声明

a = 10;

b = 20;

g = a + b;

cout << g;

return 0;

}

如果局部变量和全局变量冲突，则在此局部中，以局部变量为主

#include<iostream>

using namespace std;

int g = 20;

int func();

int main(int argc, char const \*argv[]) {

int k = func(); // 在func里k = g = 10

cout << g << endl; // 出来之后g = 20

cout << k << endl;

return 0;

}

int func(){

int g = 10;

return g;

}

常量

数字：

1, 121, 1.5, 2E-3

布尔值

true, false

不应该把true和false看作1和0,但是在cout打印时，是1和0

字符

|  |  |
| --- | --- |
| 转义序列 | 含义 |
| \\ | \ |
| \' | ' |
| \" | " |
| \? | ? |
| \a | 报警铃声 |
| \b | 退格键 |
| \f | 换页符 |
| \n | 换行符 |
| \r | 回车 |
| \t | 水平制表符 |
| \v | 垂直制表符 |

定义字符串

双引号内字符串

char a[] = "Hello, dear";

大括号里许多单引号字符，最后必须加\0！

char b[] = {'h','i','\0'};

定义常量

#define 常量名 常量值

例如

#define pi 3.14;

#define c "HALLO";

但是常量不能用 cout打印，例如 cout << pi << endl;

只能赋值后，打印变量

double p = pi;

char q[] = c;

cout << p << endl;

cout << q << endl;

修饰符类型

signed 有符号数

unsigned 无符号数

long 长

short 短

例如： short int a

类型限定符

const 常量

volatile 声明不需要优化

restrict 修饰指针，由 restrict 修饰的指针是唯一一种访问它所指向的对象的方式

static储存类

static 存储类指示编译器在程序的生命周期内保持局部变量的存在，而不需要在每次它进入和离开作用域时进行创建和销毁。因此，使用 static 修饰局部变量可以在函数调用之间保持局部变量的值。

例如

#include <iostream>

using namespace std;

void func(void);

int count = 10; /\* 全局变量 \*/

int main()

{

while(count--)

{

func();

}

return 0;

}

// 函数定义

void func( void )

{

static int i = 5; // 局部静态变量，定义之后，在func()里保存i的值

i++;

cout << "变量 i 为 " << i ;

cout << " , 变量 count 为 " << count << endl;

}

结果：

变量 i 为 6 , 变量 count 为 9

变量 i 为 7 , 变量 count 为 8

变量 i 为 8 , 变量 count 为 7

变量 i 为 9 , 变量 count 为 6

变量 i 为 10 , 变量 count 为 5

变量 i 为 11 , 变量 count 为 4

变量 i 为 12 , 变量 count 为 3

变量 i 为 13 , 变量 count 为 2

变量 i 为 14 , 变量 count 为 1

变量 i 为 15 , 变量 count 为 0

如果把

static int i = 5 改成 int i = 5，则

变量 i 为 6 , 变量 count 为 9

变量 i 为 6 , 变量 count 为 8

变量 i 为 6 , 变量 count 为 7

变量 i 为 6 , 变量 count 为 6

变量 i 为 6 , 变量 count 为 5

变量 i 为 6 , 变量 count 为 4

变量 i 为 6 , 变量 count 为 3

变量 i 为 6 , 变量 count 为 2

变量 i 为 6 , 变量 count 为 1

变量 i 为 6 , 变量 count 为 0

extern存储类

extern 修饰的变量、函数可以在调用另一个文件中的函数或值

例如文件 main.cpp

#include <iostream>

int count ;

extern void write\_extern(); //调用外部函数write\_extern()

int main()

{

count = 5;

write\_extern(); // 外部函数write\_extern()

}

文件support.cpp

#include <iostream>

extern int count; // 调用外部变量count = 5

void write\_extern(void)

{

std::cout << "Count is " << count << std::endl;

}

调用g++ main.cpp support.cpp -o write得到：

Count is 5

运算符

|  |  |  |  |
| --- | --- | --- | --- |
| + | 加法 | \*= | 乘后赋值 |
| - | 减法 | /= | 除后赋值 |
| \* | 乘法 | %= | 取余后赋值 |
| / | 除法 | ! | not 非 |
| % | 取余 | & | 二进制and |
| ++ | 自加 | | | 二进制or |
| -- | 自减 | ^ | 二进制异或 |
| == | 判断相等 | ~ | 二进制同或 |
| ！= | 判断不等 | << | 二进制左移 |
| > | 判断大小 | >> | 二进制右移 |
| < | 判断大小 | = | 赋值 |
| >= | 判断大于等于 | >>= | 同上 |
| <= | 判断小于等于 | &= | 同上 |
| && | and 与 | ^= | 同上 |
| || | or 或 | |= | 同上 |
| += | 加后赋值 | sizeof | 返回变量所占内存 |
| -= | 减后赋值 | , | 分隔 |
| \* | 指针运算，指向一个变量 | Condition?X:Y | Condition为真则返回X，假则Y |
| <<= | a<<=2相当于a=a<<2 | & | 指针运算，返回变量地址 |
| . 和 -> | 成员运算符用于引用类、结构和共用体的成员 |  |  |

循环

while

#include <iostream>

using namespace std;

int main()

{

int i = 5, s = 0;

while (i --) {

cout << i << endl;

s += (i + 1) \* (i + 1);

}

cout << i << endl;

cout << s << endl;

return 0;

}

i等于几，就循环几次，最后一次i等于0仍然进入循环，循环后变成-1

结果：

4

3

2

1

0

-1

55

for

int s1 = 0;

for (size\_t i = 0; i < 5; i++) {

/\* code \*/

s1 += (i + 1) \* (i + 1);

}

cout << s1 << endl;

可以用迭代器

int ar[5] = {3, 2, 5, 7, 9};

for (auto &x : ar) {

cout << x << endl;

}

此处 auto &x : ar 中的auto表示自动选择数据类型，&表示”引用“， 结果：

3

2

5

7

9

do...while

do

{

statement(s);

}while( condition );

与while的区别就是保证了至少进行一次循环，尽管条件不满足，例如

int k = 1;

do {

cout << k << endl;

} while(k > 10);

运行结果

1

控制语句： break, continue, goto

break 直接跳出当前循环到循环的括号外

continue 跳过循环中后面语句，进入该循环的下一次

判断语句

if(boolean\_expression 1)

{

// 当布尔表达式 1 为真时执行

}

else if( boolean\_expression 2)

{

// 当布尔表达式 2 为真时执行

}

else if( boolean\_expression 3)

{

// 当布尔表达式 3 为真时执行

}

else

{

// 当上面条件都不为真时执行

}

数字

short, int, long, float, double

数学运算（一些函数），需要引用数学头文件 <cmath>

cos(double); 该函数返回弧度角（double 型）的余弦。

sin(double); 该函数返回弧度角（double 型）的正弦。

tan(double); 该函数返回弧度角（double 型）的正切。

log(double); 该函数返回参数的自然对数。

pow(double, double); x 的 y 次方。

hypot(double, double); 两个数的第二范数。

sqrt(double); 该函数返回参数的平方根。

abs(int); 该函数返回整数的绝对值。

fabs(double); 该函数返回任意一个浮点数的绝对值。

floor(double); 该函数返回一个小于或等于传入参数的最大整数

举例

#include <iostream>

#include <cmath>

using namespace std;

int main()

{

double a = 3.1, b = 5.5, c = -3.6, ex;

int d = -9;

cout << cos(a) << endl;

cout << sin(a) << endl;

cout << log(a) << endl;

cout << pow(a, b) << endl;

cout << hypot(a, b) << endl;

cout << sqrt(a) << endl;

cout << floor(b) << endl;

cout << abs(d) << endl;

cout << fabs(c) << endl;

return 0;

}

有时候abs和fabs不那么严格

cout << fabs(d) << endl; // d是整数，但是用fabs

cout << abs(c) << endl; // c是浮点数，但是用abs

结果

9

3.6 // 自动变成了double

随机数

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

int main()

{

int i, j;

srand((unsigned)time(NULL));

for (i = 0; i < 10; i++)

{

j = rand();

cout << j << endl;

}

return 0;

}

还有其他的方法，有专门的#include <random>库

数组

声明数组

类型 数组名[元素个数]， 例如

double a[3];

或者用大括号直接初始化其中的元素

double a[3] = {3.0, 5.5, 4.9};

多维数组

int b[3][2]; // 4行3列

初始化

int b[3][2] =

{

{1, 3},

{2, 6},

{4, 5}

};

可以不换行写，不过不太直观

int b[3][2] = {{1, 3}, {2, 6}, {4, 5}};

进而可以去掉里面的大括号：

int b[3][2] = {1, 3, 2, 6, 4, 5};

按照行的顺序来写，与matlab的reshape相反

访问数组

double c = a[0]; // c = 3.0

int d = b[2][1]; // d = 5

字符串

定义字符串

大括号 + 字符

char a = {'H', 'i', '!', '\0'}; // '\0' 是必须的

双引号

char a = "Hi!";

或者指针法

const char \*a = "Hello, World!"; // const 必须要

一些函数

strcpy(s1, s2); 复制字符串 s2 到字符串s1

strcat(s1, s2); 连接字符串 s2 到字符串 s1 的末尾。

strlen(s1); 返回字符串 s1 的长度。

strcmp(s1, s2); 如果 s1 和 s2 是相同的，则返回 0；如果 s1<s2 则返回值小于 0；如果 s1>s2 则返回值大于 0。

strchr(s1, ch); 返回一个指针，指向字符串 s1 中字符 ch 的第一次出现的位置。

strstr(s1, s2); 返回一个指针，指向字符串 s1 中字符串 s2 的第一次出现的位置。

举例：

#include <iostream>

#include <cstring>

using namespace std;

int main()

{

char a[100] = "Hallo, World!";

char b[100] = "Hi!";

char c[100];

strcpy(c, a);

cout << a << endl; // "Hallo, World!"

cout << c << endl; // "Hallo, World!"

cout << strlen(c) << endl; // 13

cout << strlen(b) << endl; // 3

strcat(a, b);

cout << a << endl; // ”Hallo, World!Hi“

cout << strcmp(a, c) << endl; // 72 只看其正负零即可

cout << strcmp(a, b) << endl; // -8 只看其正负零即可

cout << strcmp(b, a) << endl; // 8 只看其正负零即可

return 0;

}

string类型

先调用#include <string>库

字符串的定义

string a = "Hello";

string b = "World";

string c, d;

字符串复制

c = a

字符串连接

d = a + b;

字符串长度

int l = d.size();

指针

地址： 在变量前面加一个&表示地址（指针：指针就是这个地址）：

#include <iostream>

using namespace std;

int main()

{

int a;

char b[3];

cout << &a << endl;

cout << &b << endl;

return 0;

}

结果

0x7ffe5d328790

0x7ffe5d328795

指针的定义

int \*i; /\* 一个整型的指针 \*/

double \*i; /\* 一个 double 型的指针 \*/

float \*i; /\* 一个浮点型的指针 \*/

char \*i; /\* 一个字符型的指针 \*/

关系：

&数 就是地址

\*地址 就是数， 所以定义指针的时候int \*i，其实就时int 变量

空指针指针

int \*ptr = NULL;

指针指向数字和数组的不同

int var[MAX] = {10, 100, 200};

int \*a = var; // 直接就指向数组的开始（第一个元素）

int \*b = &var[0]; // 指向第一个元素

cout << a << endl;

cout << b << endl;

结果：a和b一样、

因为数组的名称（此处var）本来就是它的地址的第一个，也就是var相当于&var[0]，例如

\*(var + 1) = 1，相当于var[1] = 1

另外

int \*i;

i = &a;

就等于

int \*i = &a;

C++ 指针的算术运算

对于指针有 ++, --, +, - 的运算

对于不同的数据类型，加的不同，例如字符char \*j，每次加因此，第一个声明1. 对于整数 int \*j，每次加4

int var[3] = {10, 100, 200};

int \*j; // 指针

// 指针中的数组地址

j = var; // 指向第一个元素

for (int i = 0; i < 3; i++)

{

cout << "Address of var[" << i << "] = ";

cout << j << endl; // 元素地址

cout << "Value of var[" << i << "] = ";

cout << \*j << endl; // 元素的值

// 移动到下一个位置

j++; // 对于整数每次加4

}

结果

Address of var[0] = 0x7fff1b11f46c

Value of var[0] = 10

Address of var[1] = 0x7fff1b11f470

Value of var[1] = 100

Address of var[2] = 0x7fff1b11f474

Value of var[2] = 200

指针数组：又许多指针构成的集合

#include <iostream>

using namespace std;

const int MAX = 3;

int main ()

{

int var[MAX] = {10, 100, 200};

int \*ptr[MAX]; // 3个指针构成的ptr

for (int i = 0; i < MAX; i++)

{

ptr[i] = &var[i]; // ptr每个指针对应var每个元素

}

for (int i = 0; i < MAX; i++)

{

cout << "Value of var[" << i << "] = ";

cout << \*ptr[i] << endl;

}

return 0;

}

指针的指针：

int var = 30;

int \*ptr;

int \*\*pptr;

ptr = &var;

pptr = &ptr;

引用

引用是让被引用和引用变量使用同一个地址。

创建引用

int& r = i; // & 读作引用。可以读作 "r 是一个初始化为 i 的整型引用"

int i = 17;

尽管 i在r之后赋值，r还是等于i等于17，因为r和i用的同一个地址。

使用例子：

#include <iostream>

using namespace std;

void swap(int& x, int& y);

int main ()

{

// 局部变量声明

int a = 100;

int b = 200;

swap(a, b);

cout << "交换后，a 的值：" << a << endl;

cout << "交换后，b 的值：" << b << endl;

return 0;

}

// 函数定义

void swap(int& x, int& y) // x和a用同一个地址，y和b也是

{

int temp;

temp = x; /\* 保存地址 x 的值 \*/

x = y; /\* 把 y 赋值给 x \*/ // x和y地址上的值交换了

y = temp; /\* 把 x 赋值给 y \*/ // 相当于a和b地址上的值交换了

return;

}

结构体

struct 结构名字 {

类型1 名字1;

类型2 名字2;

类型3 名字3;

} 变量名; // 变量名可以先不写

举例

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

} book;

使用时的举例：

#include <iostream>

#include <cstring>

using namespace std;

// 声明一个结构体类型 Books

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( )

{

Books Book1; // 定义结构体类型 Books 的变量 Book1

Books Book2; // 定义结构体类型 Books 的变量 Book2

// Book1 详述

strcpy( Book1.title, "C++ 教程");

strcpy( Book1.author, "Runoob");

strcpy( Book1.subject, "编程语言");

Book1.book\_id = 12345;

// Book2 详述

略

return 0;

}

指向结构的指针

定义结构指针

struct Books \*p; // struct Books 此处跟 int 类似

指向变量

p = & Book1;

指向结构成员

p -> title;

举例

#include <iostream>

#include <cstring>

using namespace std;

void printBook( struct Books \*book );

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

int main( )

{

Books Book1; // 定义结构体类型 Books 的变量 Book1

Books Book2; // 定义结构体类型 Books 的变量 Book2

// Book1 详述

略

// Book2 详述

略

printBook( &Book1 ); // Book1 的地址

printBook( &Book2 );

return 0;

}

// 该函数以结构指针作为参数

void printBook( struct Books \*book )

{

cout << "书标题 : " << book->title <<endl; 指针指向title，不用\*

cout << "书作者 : " << book->author <<endl;

cout << "书类目 : " << book->subject <<endl;

cout << "书 ID : " << book->book\_id <<endl;

}

函数

函数的定义：

返回类型 函数名(参数列表)

{

函数内容

return 返回值 // void类型可以不返回

}

函数声明

返回类型 函数名(参数列表); // 和定义函数的第一句一样

其中参数列表不用写变量名

举例：函数定义

int max(int num1, int num2)

{

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

函数声明

int max(int, int);

函数调用

int a, b, m;

m = max(a, b);

cout << m << endl;

函数的参数：

传值调用 该方法把参数的实际值赋值给函数的形式参数。

传递数组给函数

void myFunction(int \*param) //形式参数

void myFunction(int param[10]) // 已知数组大小

void myFunction(int param[]) // 未知大小的数组

函数返回数组

int \* myFunction()

必须定义返回一个指针，因为不能返回数组。（数组的名字就是它的地址），举例

int\* get()

{

...

}

int main()

{

int \*p;

p = get();

cout << \*p << endl;

return 0;

}

默认参数

可以在定义或声明函数的时候，设定默认参数

int sum(int a, int b=20);

其中默认参数必须在后，当有输入参数的时候先给前面的赋值

例如调用

sum(3); // 得到23

指针调用 该方法把参数的地址赋值给形式参数。

void swap(int \*x, int \*y)

{

int temp;

temp = \*x; /\* 保存地址 x 的值 \*/

\*x = \*y; /\* 把 y 赋值给 x \*/

\*y = temp; /\* 把 x 赋值给 y \*/

return;

}

该函数的调用：

int main ()

{

int a = 100;

int b = 200;

swap(&a, &b); 要注意此处int \*x = &a相当于int \*x; x = &a

cout << "交换后，a 的值：" << a << endl;

cout << "交换后，b 的值：" << b << endl;

return 0;

}

因为是a和b在电脑内存的地址上的值交换了，所以虽然swap内部是另一个局部，也能影响a和b的值

字符串作为函数的输入要用const char \*指针

void pr (const char \*a) // 这里

{

cout << a << endl;

}

int main()

{

char a[] = "Hello, World!";

pr(a);

return 0;

}

函数返回指针 （相当于函数返回数组）

引用调用 该方法把参数的引用赋值给形式参数。

void swap(int &x, int &y)

{

int temp;

temp = x; /\* 保存地址 x 的值 \*/

x = y; /\* 把 y 赋值给 x \*/

y = temp; /\* 把 x 赋值给 y \*/

return;

}

把引用当作返回值

#include <iostream>

using namespace std;

double vals[] = {10.1, 12.6, 33.1, 24.1, 50.0};

double& setValues( int i )

{

return vals[i]; // 返回第 i 个元素的引用

}

// 要调用上面定义函数的主函数

int main ()

{

cout << "改变前的值" << endl;

for ( int i = 0; i < 5; i++ )

{

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

setValues(1) = 20.23; // 改变第 2 个元素

setValues(3) = 70.8; // 改变第 4 个元素

cout << "改变后的值" << endl;

for ( int i = 0; i < 5; i++ )

{

cout << "vals[" << i << "] = ";

cout << vals[i] << endl;

}

return 0;

}

结构作为函数参数

定义结构

struct Books

{

char title[50];

char author[50];

char subject[100];

int book\_id;

};

写结构

Books Book1;

strcpy( Book1.title, "C++ 教程");

strcpy( Book1.author, "Runoob");

strcpy( Book1.subject, "编程语言");

Book1.book\_id = 12345;

定义函数

void printBook( struct Books book )

{

cout << "书标题 : " << book.title <<endl;

cout << "书作者 : " << book.author <<endl;

cout << "书类目 : " << book.subject <<endl;

cout << "书 ID : " << book.book\_id <<endl;

}

调用函数

printBook( Book1 );

Lambda匿名函数

auto 函数名 = [] (输入量) {return 输出量;};

例如 一个2维矩阵对应1维的index转换

#include <iostream>

using namespace std;

const int M = 5, N = 4;

int main()

{

auto index = [] (int a, int b) {return a + (b - 1) \* M;};

cout << index(3, 3) << endl;

return 0;

}

函数的重载

功能相似的函数，可以用相同的函数名例如：

double sum(double a, double b);

int sum(int a, int b);

int sum(int a, int b, int c);

#include <iostream>

using namespace std;

int sum(int, int);

int sum(int, int, int);

int main()

{

cout << sum(1,2) << endl;

cout << sum(1,2,3) << endl;

return 0;

}

int sum(int a, int b)

{

return a + b;

}

int sum(int a, int b, int c)

{

return a + b + c;

}

面向对象编程

类

类的定义

class Box

{

public:

double length; // 盒子的长度

double breadth; // 盒子的宽度

double height; // 盒子的高度

};

类的对象

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

类的使用

#include <iostream>

using namespace std;

class Box

{

public:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

int main( )

{

Box Box1; // 声明 Box1，类型为 Box

Box Box2; // 声明 Box2，类型为 Box

double volume = 0.0; // 用于存储体积

// box 1 详述

Box1.height = 5.0; // 赋值的方法

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 详述

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// box 1 的体积

volume = Box1.height \* Box1.length \* Box1.breadth; 调用的方法

cout << "Box1 的体积：" << volume <<endl;

// box 2 的体积

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Box2 的体积：" << volume <<endl;

return 0;

}

类里的成员还可以是函数

而且函数可以直接读取、赋值类里的成员。

class Box

{

public:

double length; // 长度

double breadth; // 宽度

double height; // 高度

double getVolume(void) // 函数，可以直接用类里的变量

{

return length \* breadth \* height;

}

};

在类外面也可以定义

double Box::getVolume(void)

{

return length \* breadth \* height;

}

举例

#include <iostream>

using namespace std;

class Box

{

public:

double length;

double breadth;

double height = 1; // 赋值了高度

void setlen(double len)

{

length = len; // 可以直接赋值类里的变量

}

};

int main()

{

Box box;

box.breadth = 2; // 直接赋值宽度（对于public成员）

box.setlen(3); // 利用函数setlen赋值长度

cout << box.length \* box.breadth \* box.height << endl;

return 0;

}

构造函数：和类同名的函数，所以调用类的时候会直接调用构造函数

class Line

{

public:

double length;

void setLength( double len );

double getLength( void );

Line(double len); // 这是构造函数，可以没有参数

};

Line::Line(double len) // 定义构造函数

{

cout << "Object is being created." << endl; // 类似于一个装饰器

length = len; // 同时赋值length

}

使用方法：

Line AB(10.0);

结果：

"Object is being created." // 同时AB的length也被设置为10.0

拷贝构造函数

通过使用另一个同类型的对象来初始化新创建的对象。

复制对象把它作为参数传递给函数。

复制对象，并从函数返回这个对象。

举例

class Line

{

public:

int getLength( void );

Line( int len ); // 简单的构造函数

Line( const Line &obj); // 拷贝构造函数， obj是一个对象引用

~Line(); // 析构函数

private

int \*ptr; // 后面有用

};

定义函数

Line::Line(int len)

{

cout << "调用构造函数" << endl;

// 为指针分配内存

ptr = new int;

\*ptr = len;

}

Line::Line(const Line &obj)

{

cout << "调用拷贝构造函数并为指针 ptr 分配内存" << endl;

ptr = new int;

\*ptr = \*obj.ptr; // 拷贝值

}

使用函数

int main( )

{

Line line1(10);

Line line2 = line1; // 这里也调用了拷贝构造函数

return 0;

}

友元函数

#include <iostream>

using namespace std;

class Box

{

double width;

public:

friend void printWidth( Box box ) // 友元函数

{

cout << "Width of box : " << box.width <<endl;

}

void setWidth( double wid )

{

width = wid;

}

};

// 程序的主函数

int main( )

{

Box box;

// 使用成员函数设置宽度

box.setWidth(10.0);

// 使用友元函数输出宽度

printWidth( box ); // 不用加box. 因为不属于Box类

return 0;

}

定义友元函数也可以在类外面定义，注意区别：

void Box::setWidth( double wid ) // 类成员函数，要加Box::

{ };

void printWidth( Box box ) // 友元函数，不用加Box::

{ };

类指针

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

Box \*ptrBox; // 类指针

ptrBox = &Box1; // 指向Box1

cout << ptrBox -> Volume() << endl; // 读取Volume()

this 指针

this指针是用来调用成员自己的函数，例如

class Box

{

public:

double Volume()

{ }

int compare(Box box2)

{

return this->Volume() > box2.Volume();

// 相当于box1.Volume() 和 box2.Volume()的比较

}

};

int main(void)

{

Box Box1(3.3, 1.2, 1.5); // Declare box1

Box Box2(8.5, 6.0, 2.0); // Declare box2

if(Box1.compare(Box2)) // box1.Volume() 和 box2.Volume()的比较

{

cout << "Box2 is smaller than Box1" <<endl;

}

else

{

cout << "Box2 is equal to or larger than Box1" <<endl;

}

return 0;

}

访问类修饰符public, private, protected

public的成员可以从外部赋值、读取，例如

cout << box.length << endl; // 读取

box.length = 3; // 赋值

对于private的成员，则必须使用public的成员函数读取和赋值

对于protected成员，在子类中可以访问

如果没定义public，private还是protected，默认时private

静态成员

静态变量

class Box

{

public:

static int objectCount; // 静态参数

index = objectCount; // 比如可以记录物体编号

Box(double l=2.0, double b=2.0, double h=2.0)

{

cout <<"Constructor called." << endl;

length = l;

breadth = b;

height = h;

objectCount++; // 每次创建对象时增加 1

}

double Volume()

{

return length \* breadth \* height;

}

private:

double length; // 长度

double breadth; // 宽度

double height; // 高度

};

使用：

int Box::objectCount = 0; // 初始化

int main(void)

{

Box Box1(3.3, 1.2, 1.5); // 声明 box1

Box Box2(8.5, 6.0, 2.0); // 声明 box2

cout << Box::objectCount << endl; // 读取Box类里的值

// cout << Box1.objectCount << endl; // 或者从任意一个对象读取

return 0;

}

继承：

有3种继承方式：public，private和protected继承

public继承，成员类型不变（例如public还是public）

protected继承，成员类型改变（public和protected变成protected，private不变）

private继承，成员全都变成private

继承

class Shape

{

......

};

例如Shape的子类circle

class circle: public Shape // public继承

{

public: // 继承之外还有新的属性（成员）

...

private:

...

};

比如

class 生物： 有寿命、重量等成员

class 动物: public 生物。就有腿的数量、是否长毛等成员

访问权限：

|  |  |  |  |
| --- | --- | --- | --- |
| 访问 | public | protected | private |
| 同一个类 | yes | yes | yes |
| 派生类 | yes | yes | no |
| 外部的类 | yes | no | no |

多态

比如已有类Shape

class Shape {

protected:

public:

Shape( int a=0, int b=0) // Shape有2个属性

{

width = a;

height = b;

}

int area() // Shape有一个方法

//这样不会随着子类而变化，要在前面加virtual

{

cout << "Parent class area :" <<endl;

return 0;

}

};

子类结构函数

class Rectangle: public Shape{

public:

Rectangle( int a=0, int b=0):Shape(a, b) { } // 子类的结构函数

// 结构函数Rectangle()有 :Shape(a,b)， 分别对应赋值

int area () // Rectangle有自己的面积公式（多态）

{

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

多态

错误示范：

class Shape {

int area() // 此处错误

{

cout << "Parent class area :" <<endl;

return 0;

}

};

Shape \*shape;

Rectangle rec(10,7);

shape = &rec;

cout << shape->area() << endl; // 结果还是Parent class area :0

int area() 改成虚函数

virtual int area()即可，结果变成

Rectangle class area :

70

纯虚函数：没有函数的定义，例如

virtual int area() = 0;

而不是

virtual int area ()

{

函数内容

}

多继承

一个子类可以有多个父类

class <派生类名>:<继承方式1><基类名1>,<继承方式2><基类名2>,…

{

<派生类类体>

};

数据封装

把与使用者无关的量设置成private，用public的函数进行修改和读取

接口

把基类的读取的函数写成纯虚函数