```
//从C到C++的快速教程

1. C++头文件不必是.h结尾，C语言中的标准库头文件如math.h,stdio.h在C++
被命名为cmath,cstdio.
#include <cmath>
#include <cstdio>
int main(){
   double a = 1.2;
   a = sin (a);
   printf("%lf\n",a);
}

2 除了C的多行注释，C++可以使用单行注释
/*
  CC的多行注释`
  用于注释一块代码
*/
#include <cmath>
#include <cstdio>
int main(){     //程序执行的入口，main主函数
   double a = 1.2;  //定义一个变量a
   a = sin (a);
   printf("%lf\n",a); //用格式符%lf输出a: lf表示是double型
}


3. 名字空间namespace.
为防止名字冲突(出现同名),C++引入了名字空间( namespace),
通过::运算符限定某个名字属于哪个名字空间
//如  “计算机1702”::“李平”
//如  “信计1603”::“李平”
#include <cstdio>
namespace first
{
   int a;
   void f(){/*...*/}
   int g(){/*...*/}
}

namespace second
{
   double a;
   double f(){/*...*/}
   char g;
}

int main ()
{
   first::a = 2;
   second::a = 6.453;
   first::a = first::g()+second::f();
   second::a = first::g()+6.453;

   printf("%d\n",first::a);
   printf("%lf\n",second::a);

```

```
    return 0;
}

通常有3种方法使用名字空间X的名字name：
/*
using namespace X; //引入整个名字空间
using X::name ; //使用单个名字
X::name; //程序中加上名字空间前缀，如X::
*/

4. C++的新的输入输出流库(头文件iostream)将输入输出看成一个流，并用
输出运算符 << 和输入运算符 >> 对数据(变量和常量进行输入输出);

其中有cout和cin分别代表标准输出流对象(屏幕窗口)和标准输入流对象(键盘);

标准库中的名字都属于标准名字空间std.

#include <iostream>
#include <cmath>
using std::cout; //使用单个名字
int main()
{
    double a;
    cout << "从键盘输入一个数" << std::endl; //endl表示换行符，并强制输出
    std::cin >> a; //  通过“名字限定”std::cin,
                   //cin是代表键盘的输入流对象，>>等待键盘输入一个实数
    a = sin(a);

    cout << a;    //cout是代表屏幕窗口的输出流对象
    return 0;
}


#include <iostream>  //标准输入输出头文件
#include <cmath>
using namespace std; //引入整个名字空间std中的所有名字
                     //cout cin都属于名字空间std;
int main() {
    double a;
    cout << "从键盘输入一个数" << endl;
    cin >> a;
    a = sin(a);
    cout << a;

    return 0;
}


5. 变量“即用即定义”，且可用表达式初始化

#include <iostream>
using namespace std;

int main (){
   double a = 12 * 3.25;
   double b = a + 1.112;
```

```
113
114     cout << "a contains : " << a << endl;
115     cout << "b contains: " << b << endl;
116
117     a = a * 2 + b;
118
119     double c = a + b * a; //“即用即定义”，且可用表达式初始化
120
121     cout << "c contains: " << c << endl;
122 }
123
124
125 6. 程序块{}内部作用域可定义域外部作用域同名的变量，在该块里就隐藏了外部变量
126 #include <iostream>
127 using namespace std;
128
129 int main ()
130 {
131     double a;
132
133     cout << "Type a number: ";
134     cin >> a;
135
136     {
137         int a = 1; // "int a"隐藏了外部作用域的“double a"
138         a = a * 10 + 4;
139         cout << "Local number: " << a << endl;
140     }
141
142     cout << "You typed: " << a << endl; //main作用域的“double a"
143
144     return 0;
145 }
146
147 7. for循环语句可以定义局部变量。
148
149 #include <iostream>
150 using namespace std;
151
152 int main (){
153     int i = 0;
154     for (int i = 0; i < 4; i++)
155     {
156         cout << i << endl;
157     }
158
159     cout << "i contains: " << i << endl;
160
161     for (i = 0; i < 4; i++)
162     {
163         for (int i = 0; i < 4; i++)        // we're between
164         {                                  // previous for's hooks
165             cout << i<< " ";
166         }
167         cout << endl;
168     }
```

```
    return 0;
}
```

8.访问和内部作用域变量同名的全局变量，要用全局作用域限定 ::

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

double a = 128;

int main (){
    double a = 256;

    cout << "Local a: " << a << endl;
    cout << "Global a: " <<::a << endl; //::是全局作用域限定

    return 0;
}
```

9.C++引入了“引用类型”，即一个变量是另一个变量的别名

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

int main ()
{
    double a = 3.1415927;

    double &b = a;                              // b 是 a的别名，b就是a

    b = 89;             //也就是a的内存块值为89

    cout << "a contains: " << a << endl;     // Displays 89.

    return 0;
}
```

引用经常用作函数的形参，表示形参和实参实际上是同一个对象，
在函数中对形参的修改也就是对实参的修改

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

void swap(int x, int y) {
     cout << "swap函数内交换前：" << x << " " << y << endl;
     int t = x; x = y; y = t;
     cout << "swap函数内交换后：" << x << " " << y << endl;
}
```

```
225 int main(){
226     int a = 3, b = 4;
227 
228     swap(a, b);
229     cout << a << ", " << b << endl;        // Displays 100, 4.
230 
231     return 0;
232 }
233 
234 /*
235 x,y得到2个int型变量的指针,x,y本身没有修改
236 修改的是x,y 指向的那2个int型变量的内容
237 */
238 void swap(int *x, int *y) {
239     cout << "swap函数内交换前：" << *x << " " << *y << endl;
240     int t = *x; *x = *y; *y = t;
241     cout << "swap函数内交换后：" << *x << " " << *y << endl;
242 }
243 
244 int main() {
245     int a = 3, b = 4;
246 
247     swap(&a, &b);  // &a赋值给x,&b赋值给y,
248                    //x,y分别是int*指针，指向a,b
249                    //*x, *y就是a和b
250     cout << a << ", " << b << endl;        // Displays 100, 4.
251 
252     return 0;
253 }
254 
255 
256 //x,y是实参的引用
257 void swap(int &x, int &y) {
258     cout << "swap函数内交换前：" << x << " " << y << endl;
259     int t = x; x = y; y = t;
260     cout << "swap函数内交换后：" << x << " " << y << endl;
261 }
262 
263 int main(){
264     int a = 3, b = 4;
265 
266     swap(a, b); //x,y将分别是a,b的引用，即x就是a,y就是b
267     cout << a << ", " << b << endl;        // Displays 100, 4.
268 
269     return 0;
270 }
271 
272 当实参占据内存大时，用引用代替传值（需要复制）可提高效率，
273 如果不希望因此无意中修改实参，可以用const修改符。如
274 #include <iostream>
275 using namespace std;
276 
277 void change (double &x, const double &y,double z){
278    x = 100;
279    y = 200;  //错！ y不可修改，是const double &
280    z = 300;
```

```
}

int main (){
   double a,b,c;//内在类型变量未提供初始化式，默认初始化为0

   change(a, b, c);
   cout << a << ", " << b << ", " << c << endl;        // Displays 100, 4.

   return 0;
}
```

10.对于不包含循环的简单函数，建议用inline关键字声明 为"inline内联函数"，
编译器将内联函数调用用其代码展开，称为“内联展开”，避免函数调用开销，
提高程序执行效率

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

inline double distance(double a, double b) {
    return sqrt(a * a + b * b);
}

int main() {
    double k = 6, m = 9;
    // 下面2行将产生同样的代码:
    cout << distance(k, m) << endl;
    cout << sqrt(k * k + m * m) << endl;

    return 0;
}
```

11. 通过 try-catch处理异常情况
正常代码放在try块，catch中捕获try块抛出的异常

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

int main (){
   int a, b;

   cout << "Type a number: ";
   cin >> a;
   cout << endl;

   try {
      if (a > 100) throw 100;
      if (a < 10)  throw 10;
      throw "hello";
   }
   catch (int result) {
      cout << "Result is: " << result << endl;
      b = result + 1;
   }
```

```
337     catch (char * s) {
338         cout << "haha " << s << endl;
339     }
340 
341     cout << "b contains: " << b << endl;
342 
343     cout << endl;
344 
345     // another example of exception use:
346 
347     char zero[] = "zero";
348     char pair[] = "pair";
349     char notprime[] = "not prime";
350     char prime[] = "prime";
351 
352     try {
353        if (a == 0) throw zero;
354        if ((a / 2) * 2 == a) throw pair;
355        for (int i = 3; i <= sqrt (a); i++){
356           if ((a / i) * i == a) throw notprime;
357        }
358        throw prime;
359     }
360     catch (char *conclusion) {
361         cout << "异常结果是:  " << conclusion << endl;
362     }
363     catch (...) {
364         cout << "其他异常情况都在这里捕获 " << endl;
365     }
366 
367     cout << endl;
368 
369     return 0;
370 }
371 
372 
373 
374 12. 默认形参:  函数的形参可带有默认值。必须一律在最右边
375 
376 #include <iostream>
377 using namespace std;
378 
379 double test(double a, double b = 7) {
380     return a - b;
381 }
382 
383 int main() {
384     cout << test(14, 5) << endl;
385     cout << test(14) << endl;
386 
387     return 0;
388 }
389 
390 /*错:  默认参数一律靠右*/
391 double test(double a, double b = 7, int c) {
392     return a - b;
```

```
}



13. 函数重载：C++允许函数同名，只要它们的形参不一样(个数或对应参数类型)，
调用函数时将根据实参和形参的匹配选择最佳函数，
如果有多个难以区分的最佳函数，则变化一起报错！
注意：不能根据返回类型区分同名函数

#include <iostream>
using namespace std;

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

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


//错：编译器无法区分int add (int a, int b),void  add (int a, int b)
void  add(int a, int b) {
    return a - b;
}


int main() {
    double   m = 7, n = 4;
    int      k = 5, p = 3;

    cout << add(m, n) << " , " << add(k, p) << endl;

    return 0;
}


14.运算符重载

#include <iostream>
using namespace std;

struct Vector2{
   double x;
   double y;
};

Vector2 operator * (double a, Vector2 b){
    Vector2 r;

   r.x = a * b.x;
   r.y = a * b.y;

   return r;
}

```

```
449 Vector2 operator+ (Vector2 a, Vector2 b) {
450     Vector2 r;
451 
452     r.x = a.x + b.x;
453     r.y = a.y + b.y;
454 
455     return r;
456 }
457 
458 int main (){
459    Vector2 k, m;             // C++定义的struct类型前不需要再加关键字struct: "struct
       vector"
460 
461    k.x =  2;                 //用成员访问运算符.访问成员
462    k.y = -1;
463 
464    m = 3.1415927 * k;        // Magic!
465 
466    cout << "(" << m.x << ", " << m.y << ")" << endl;
467 
468    Vector2 n = m + k;
469    cout << "(" << n.x << ", " << n.y << ")" << endl;
470    return 0;
471 }
472 
473 
474 
475 
476 #include <iostream>
477 using namespace std;
478 
479 struct Vector2 {
480    double x;
481    double y;
482 };
483 
484 ostream& operator << (ostream& o, Vector2 a){
485    o << "(" << a.x << ", " << a.y << ")";
486    return o;
487 }
488 
489 int main (){
490    Vector2 a;
491 
492    a.x = 35;
493    a.y = 23;
494    cout << a << endl; // operator <<(cout,a);
495    return 0;
496 }
497 
498 
499 
500 
501 
502 15. 模板template函数：厌倦了对每种类型求最小值
503 
```

```cpp
#include <iostream>
using namespace std;
int minValue(int a, int b) {//return a<b?a:b
    if (a < b) return a;
    else return b;
}
double minValue(double a, double b) {//return a<b?a:b
    if (a < b) return a;
    else return b;
}

int main() {
    int i = 3, j = 4;
    cout << "min of " << i << " and " << j << " is " << minValue(i, j) << endl;
    double x = 3.5, y = 10;
    cout << "min of " << x << " and " << y << " is " << minValue(x, y) << endl;

}

//可以转化成:  模板函数
#include <iostream>
using namespace std;

//可以对任何能比较大小(<)的类型使用该模板让编译器
//自动生成一个针对该数据类型的具体函数
template<class TT>
TT minValue(TT a, TT b) {//return a<b?a:b
    if (a < b) return a;
    else return b;
}

int main() {
    int i = 3, j = 4;
    cout << "min of " << i << " and " << j << " is " << minValue(i, j) << endl;
    double x = 3.5, y = 10;
    cout << "min of " << x << " and " << y << " is " << minValue(x, y) << endl;

    //但是,不同类型的怎么办?
    cout << "min of " << i << " and " << y << " is " << minValue(i, y) << endl;
}



//可以对任何能比较大小(<)的类型使用该模板让编译器
//自动生成一个针对该数据类型的具体函数
#include <iostream>
using namespace std;

template<class T1, class T2>
T1 minValue(T1 a, T2 b) {//return a<b?a:b
    if (a < b) return a;
    else return (T2)b; //强制转化为T1类型
}

int main() {
    int i = 3, j = 4;
```

```
560     cout << "min of " << i << " and " << j << " is " << minValue(i, j) << endl;
561     double x = 3.5, y = 10;
562     cout << "min of " << x << " and " << y << " is " << minValue(x, y) << endl;
563
564     //但是,不同类型的怎么办?
565     cout << "min of " << i << " and " << y << " is " << minValue(i, y) << endl;
566 }
567
568
569
570
571 //堆存储区
572 16. 动态内存分配:  关键字 new 和 delete 比C语言的malloc/alloc/realloc和free更好,
573 可以对类对象调用初始化构造函数或销毁析构函数
574
575 #define _CRT_SECURE_NO_WARNINGS   //windows
576 #include <iostream>
577 #include <cstring>
578 using namespace std;
579 int main() {
580     double d = 3.14;          // 变量d是一块存放double值的内存块
581     double *dp;               // 指针变量dp: 保存double类型的地址的变量
582                               // dp的值得类型是double *
583                               // dp是存放double *类型值 的内存块
584
585     dp = &d;                 //取地址运算符&用于获得一个变量的地址,
586                              // 将double变量d的地址(指针)保存到double*指针变量dp中
587                              // dp和&d的类型都是double *
588
589     *dp = 4.14;             //解引用运算符*用于获得指针变量指向的那个变量(C++中也称为对象)
590                             //*dp就是dp指向的那个d
591     cout << "*dp= " << *dp << "  d=:" << d << endl;
592
593     cout << "Type a number: ";
594     cin >> *dp;                    //输出dp指向的double内存块的值
595     cout << "*dp= " << *dp << "  d=:" << d << endl;
596
597     dp = new double;           // new 分配正好容纳double值的内存块(如4或8个字节)
598                                // 并返回这个内存块的地址, 而且地址的类型是double *
599                                //这个地址被保存在dp中, dp指向这个新内存块, 不再是原来d那个内存块了
600                                // 但目前这个内存块的值是未知的
601
602                                // 注意:
603                                // new 分配的是堆存储空间, 即所有程序共同拥有的自由内存空间
604                                //而d,dp等局部变量是这个程序自身的静态存储空间
605                                // new会对这个double元素调用double类型的构造函数做初始化, 比如初始化为0
606
607
608     *dp = 45.3;                //*dp指向的double内存块的值变成45.3
609
610     cout << "Type a number: ";
611     cin >> *dp;                    //输出dp指向的double内存块的值
```

```
612     cout << "*dp= " << *dp << endl;
613
614     *dp = *dp + 5;                  //修改dp指向的double内存块的值45.3+5
615
616     cout << "*dp= " << *dp << endl;
617
618     delete dp;                          // delete 释放dp指向的动态分配的double内存块
619
620
621     dp = new double[5];                  //new 分配了可以存放15个double值的内存块，
622                                          //返回这块连续内存的起始地址，而且指针类型是
623                                          //double *，  实际是第一个double元素的地址
624                                          // new会对每个double元素调用double类型的构造函数做初始化，比如初始化为0
625
626     dp[0] = 4456;                    // dp[0]等价于  *(dp+0)即*dp，也即是第1个double元素的内存块
627     dp[1] = dp[0] + 567;             // dp[1]等价于  *(dp+1)，也即是第2个double元素的内存块
628
629     cout << "d[0]=: " << dp[0] << "   d[1]=: " << dp[1] << endl;
630
631     delete[] dp;                         // 释放dp指向的多个double元素占据的内存块，
632                                          // 对每个double元素调用析构函数以释放资源
633                                          // 缺少[]，只释放第一个double元素的内存块，这叫“内存泄漏”
634
635
636     int n = 8;
637
638     dp = new double[n];                  // new 可以分配随机大小的double元素，
639                                          // 而静态数组则必须是编译期固定大小，即大小为常量
640                                          // 如 double arr[20];
641                                          //通过下标访问每个元素
642     for (int i = 0; i < n; i++) {
643         dp[i] = i;
644     }   //通过指针访问每个元素
645
646     double *p = dp;
647     for (int i = 0; i < n; i++) {
648         cout << *(p + i) << endl; //p[i]或dp[i]
649     }
650     cout << endl;
651
652     for (double *p = dp, *q = dp + n; p < q; p++) {
653         cout << *p << endl;
654     }
655     cout << endl;
656
657     delete[] dp;
658
659     char *s;
660     s = new char[100];
661
662     '\0'
```

```
663     strcpy(s, "Hello!"); //将字符串常量拷贝到s指向的字符数组内存块中
664
665     cout << s << endl;
666
667     delete[] s;  //用完以后，记得释放内存块，否则会“内存泄漏”！
668
669     return 0;
670 }
671
672
673
674
675
676 17. 类：是在C的struct类型上，增加了“成员函数”。
677 C的strcut可将一个概念或实体的所有属性组合在一起，描述同一类对象的共同属性，
678 C++使得struct不但包含数据，还包含函数(方法)用于访问或修改类变量(对象)的这些属性。
679
680
681 #include <iostream>
682 using namespace std;
683
684 struct Date {
685     int d, m, y;
686     void init(int dd, int mm, int yy) {
687         d = dd; m = mm; y = yy;
688     }
689     void print() {
690         cout << y << "-" << m << "-" << d << endl;
691     }
692 };
693
694 int main (){
695     Date day;
696     day.print();          //通过类Date对象day调用类Date的print方法
697     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
698     day.print();          //通过类Date对象day调用类Date的print方法
699
700    return 0;
701 }
702
703
704
705 // 成员函数 返回 “自引用”  (*this)
706 #include <iostream>
707 using namespace std;
708
709 struct Date {
710     int d, m, y;
711     void init(int dd, int mm, int yy) {
712         d = dd; m = mm; y = yy;
713     }
714     void print() {
715         cout << y << "-" << m << "-" << d << endl;
716     }
717     Date& add(int dd) {
718         d = d + dd;
```

```
719         return *this;    //this是指向调用这个函数的类型对象指针，
720                          // *this就是调用这个函数的那个对象
721                          //这个成员函数返回的是“自引用”，即调用这个函数的对象本身
722                          //通过返回自引用，可以连续调用这个函数
723                          // day.add(3);
724                          // day.add(3).add(7);
725     }
726 };
727
728 int main() {
729     Date day;
730     day.print();          //通过类Date对象day调用类Date的print方法
731     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
732     day.print();          //通过类Date对象day调用类Date的print方法
733     day.add(3);
734     day.add(5).add(7);
735     day.print();
736
737     return 0;
738 }
739
740 //成员函数重载“运算符函数”
741 #include <iostream>
742 using namespace std;
743
744 struct Date {
745     int d, m, y;
746     void init(int dd, int mm, int yy) {
747         d = dd; m = mm; y = yy;
748     }
749     void print() {
750         cout << y << "-" << m << "-" << d << endl;
751     }
752     Date& operator+=(int dd) {
753         d = d + dd;
754         return *this;    //this是指向调用这个函数的类型对象指针，
755                          // *this就是调用这个函数的那个对象
756                          //这个成员函数返回的是“自引用”，即调用这个函数的对象本身
757                          //通过返回自引用，可以连续调用这个函数
758                          // day.add(3);
759                          // day.add(3).add(7);
760     }
761 };
762
763 int main() {
764     Date day;
765     day.print();          //通过类Date对象day调用类Date的print方法
766     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
767     day.print();          //通过类Date对象day调用类Date的print方法
768     day += 3;            // day.add(3);
769     (day += 5) += 7;       //day.add(5).add(7);
770     day.print();
771
772     return 0;
773 }
774
```

18. 构造函数和析构函数

构造函数是和类名同名且没有返回类型的函数，在定义对象时会自动被调用，而不需要在单独调用专门的初始化函数如init，
构造函数用于初始化类对象成员，包括申请一些资源，如分配内存、打开某文件等

析构函数是在类对象销毁时被自动调用，用于释放该对象占用的资源，如释放占用的内存、关闭打开的文件

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

struct Date {
    int d, m, y;

    Date(int dd, int mm, int yy) {
        d = dd; m = mm; y = yy;
        cout << "构造函数" << endl;
    }
    void print() {
        cout << y << "-" << m << "-" << d << endl;
    }
    ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
        //目前不需要做任何释放工作，因为构造函数没申请资源
        cout << "析构函数" << endl;
    }
};

int main(){
    Date day;     //错：会自动调用构造函数，但没提供3个参数
    Date(4, 6, 1999);     //会自动调用构造函数Date(int dd, int mm, int yy)
//  day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
    day.print();          //通过类Date对象day调用类Date的print方法

    return 0;
}
```

执行上述代码，看看构造函数和析构函数执行了吗？

假如想如下调用构造函数构造对象，是不是要定义多个同名的构造函数（即重载构造函数）？

```
Date day;
Date day1（2）;
Date day2(23, 10);
Date day3(2,3,1999);
```

当然可以的

```
struct Date {
    int d, m, y;
    Date() {
        d = m = 1; y = 2000;
        cout << "构造函数" << endl;
    }
    Date(int dd) {
        d = dd;  m = 1; y = 2000;
        cout << "构造函数" << endl;
```

```
829     }
830     Date(int dd, int mm) {
831         d = dd;  m = mm;  y = 2000;
832         cout << "构造函数" << endl;
833     }
834     Date(int dd, int mm, int yy) {
835         d = dd; m = mm; y = yy;
836         cout << "构造函数" << endl;
837     }
838     void print() {
839         cout << y << "-" << m << "-" << d << endl;
840     }
841     ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
842             //目前不需要做任何释放工作，因为构造函数没申请资源
843         cout << "析构函数" << endl;
844     }
845 };
846
847 为什么不用默认参数呢？
848 #include <iostream>
849 using namespace std;
850
851 using namespace std;
852 struct Date {
853     int d, m, y;
854     Date(int dd = 1, int mm = 1, int yy = 1999) {
855         d = dd; m = mm; y = yy;
856         cout << "构造函数" << endl;
857     }
858     void print() {
859         cout << y << "-" << m << "-" << d << endl;
860     }
861     ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
862             //目前不需要做任何释放工作，因为构造函数没申请资源
863         cout << "析构函数" << endl;
864     }
865 };
866
867
868 int main(){
869     Date day;
870     Date day1(2);
871     Date day2(23, 10);
872     Date day3(2, 3, 1999);
873
874     day.print();
875     day1.print();
876     day2.print();
877     day3.print();
878     return 0;
879 }
880
881
882 //析构函数示例
883 #define _CRT_SECURE_NO_WARNINGS //windows系统
884 #include <iostream>
```

```
885 #include <cstring>
886 using namespace std;
887
888 struct student {
889     char *name;
890     int age;
891
892     student(char *n = "no name", int a = 0) {
893         name = new char[100];                 // 比malloc好!
894         strcpy(name, n);
895         age = a;
896         cout << "构造函数，申请了100个char元素的动态空间" << endl;
897     }
898
899     virtual ~student(){                                 // 析构函数
900         delete name;                            // 不能用free!
901         cout << "析构函数，释放了100个char元素的动态空间" << endl;
902     }
903 };
904
905 int main() {
906     cout << "Hello!" << endl << endl;
907
908     student a;
909     cout << a.name << ", age " << a.age << endl << endl;
910
911     student b("John");
912     cout << b.name << ", age " << b.age << endl << endl;
913
914     b.age = 21;
915     cout << b.name << ", age " << b.age << endl << endl;
916
917     student c("Miki", 45);
918     cout << c.name << ", age " << c.age << endl << endl;
919
920     cout << "Bye!" << endl << endl;
921
922     return 0;
923 }
924
925
926 19. 访问控制、类接口
927 将关键字struct换成class
928
929 #include <iostream>
930 #include <cstring>
931 using namespace std;
932
933 class student {
934     char *name;
935     int age;
936
937     student(char *n = "no name", int a = 0) {
938         name = new char[100];                 // 比malloc好!
939         strcpy(name, n);
940         age = a;
```

```
941         cout << "构造函数，申请了100个char元素的动态空间" << endl;
942     }
943
944     virtual ~student() {                                    // 析构函数
945         delete name;                                    // 不能用free!
946         cout << "析构函数，释放了100个char元素的动态空间" << endl;
947     }
948 };
949
950 int main() {
951     cout << "Hello!" << endl << endl;
952
953     student a; //编译出错:无法访问 private 成员(在“student”类中声明)
954     cout << a.name << ", age " << a.age << endl << endl; //编译出错
955
956     student b("John"); //编译出错
957     cout << b.name << ", age " << b.age << endl << endl;//编译出错
958
959     b.age = 21;  //编译出错
960     cout << b.name << ", age " << b.age << endl << endl;//编译出错
961
962     return 0;
963 }
964
965  class定义的类的成员默认都是私有的private，外部函数无法通过类对象成员或类成员函数
966 #include <iostream>
967 #include <cstring>
968 using namespace std;
969
970  class student {
971 //默认私有的，等价于 private:
972      char *name;
973      int age;
974  public: //公开的
975      student(char *n = "no name", int a = 0) {
976          name = new char[100];                    // 比malloc好!
977          strcpy(name, n);
978          age = a;
979          cout << "构造函数，申请了100个char元素的动态空间" << endl;
980      }
981
982      virtual ~student() {                                    // 析构函数
983
984          delete name;                                    // 不能用free!
985          cout << "析构函数，释放了100个char元素的动态空间" << endl;
986      }
987  };
988
989  int main() {
990      cout << "Hello!" << endl << endl;
991
992      student a;  //OK
993      cout << a.name << ", age " << a.age << endl ; //编译出错: 无法访问 private 成员
         (在“student”类中声明)
994
995      student b("John");
```

```
996         cout << b.name << ", age " << b.age << endl ;//编译出错
997
998         b.age = 21;
999         cout << b.name << ", age " << b.age << endl 1;//编译出错
1000        return 0;
1001    }
1002
1003    a.name，a.age仍然不能访问， 如何进一步修改呢？
1004
1005    #include <iostream>
1006    #include <cstring>
1007    using namespace std;
1008
1009    class student {
1010        //默认私有的，等价于 private:
1011        char *name;
1012        int age;
1013    public: //公开的
1014        char *get_name() { return name; }
1015        int get_age() { return age; }
1016        void set_age(int ag) { age = ag; }
1017        student(char *n = "no name", int a = 0) {
1018            name = new char[100];                 // 比malloc好!
1019            strcpy(name, n);
1020            age = a;
1021            cout << "构造函数，申请了100个char元素的动态空间" << endl;
1022        }
1023
1024        virtual ~student() {                              // 析构函数
1025            delete name;                             // 不能用free!
1026            cout << "析构函数，释放了100个char元素的动态空间" << endl;
1027        }
1028    };
1029
1030    int main() {
1031        cout << "Hello!" << endl << endl;
1032
1033        student a;
1034        cout << a.get_name() << ", age " << a.get_age() << endl ; //编译出错
1035
1036        student b("John");
1037        cout << b.get_name() << ", age " << b.get_age() << endl 1;//编译出错
1038
1039        b.set_age(21);
1040        cout << b.get_name() << ", age " << b.get_age() << endl ;//编译出错
1041
1042        return 0;
1043    }
1044
1045
1046    接口：public的公开成员（一般是成员函数）称为这个类的对外接口，外部函数只能通过这些接口访问类对象，
1047        private等非public的包含内部内部细节，不对外公开，从而可以封装保护类对象！
1048
1049    定义一个数组类array
1050
```

```
1051 #include <iostream>
1052 #include <cstdlib>
1053 using namespace std;
1054
1055  class Array {
1056      int size;
1057      double *data;
1058  public:
1059      Array(int s) {
1060          size = s;
1061          data = new double[s];
1062      }
1063
1064      virtual ~Array() {
1065          delete[] data;
1066      }
1067
1068      double &operator [] (int i) {
1069          if (i < 0 || i >= size) {
1070              cerr << endl << "Out of bounds" << endl;
1071              throw  "Out of bounds";
1072          }
1073          else return data[i];
1074      }
1075  };
1076
1077  int main() {
1078      Array t(5);
1079
1080      t[0] = 45;                          // OK
1081      t[4] = t[0] + 6;                    // OK
1082      cout << t[4] << endl;               // OK
1083
1084      t[10] = 7;                          // error!
1085      return 0;
1086  }
1087
1088 20. 拷贝： 拷贝构造函数、赋值运算符
1089
1090    下列赋值为什么会出错?
1091       “student m(s);
1092        s = k;”
1093    拷贝构造函数：定义一个类对象时用同类型的另外对象初始化
1094    赋值运算符：一个对象赋值给另外一个对象
1095
1096 #define  _CRT_SECURE_NO_WARNINGS  //windows系统
1097 #include <iostream>
1098 #include <cstdlib>
1099 using namespace std;
1100
1101 struct student {
1102     char *name;
1103     int age;
1104     student(char *n = "no name", int a = 0) {
1105         name = new char[100];                 // 比malloc好!
1106         strcpy(name, n);
```

```
1107          age = a;
1108          cout << "构造函数，申请了100个char元素的动态空间" << endl;
1109      }
1110
1111      virtual ~student() {                              // 析构函数
1112          delete[] name;                              // 不能用free!
1113          cout << "析构函数，释放了100个char元素的动态空间" << endl;
1114      }
1115  };
1116  int main() {
1117      student s;
1118      student k("John", 56);
1119      cout << k.name << ", age " << k.age << endl;
1120
1121      student m(s); //拷贝构造函数
1122      s = k;  //赋值运算符
1123      cout << s.name << ", age " << s.age << endl;
1124
1125      return 0;
1126  }
1127
1128  默认的“拷贝构造函数”是“硬拷贝”或“逐成员拷贝”，name指针同一块动态字符数组，当多次释放同一块内存就不错了！
1129  指应该增加“拷贝构造函数”，保证各自有单独的动态数组空间。
1130
1131  #define  _CRT_SECURE_NO_WARNINGS
1132  #include <iostream>
1133  #include <cstdlib>
1134  using namespace std;
1135  struct student {
1136      char *name;
1137      int age;
1138
1139      student(char *n = "no name", int a = 0) {
1140          name = new char[100];                 // 比malloc好!
1141          strcpy(name, n);
1142          age = a;
1143          cout << "构造函数，申请了100个char元素的动态空间" << endl;
1144      }
1145      student(const student &s) {               // 拷贝构造函数 Copy constructor
1146          name = new char[100];
1147          strcpy(name, s.name);
1148          age = s.age;
1149          cout << "拷贝构造函数，保证name指向的是自己单独的内存块" << endl;
1150      }
1151      student & operator=(const student &s) {               // 拷贝构造函数 Copy constructor
1152          strcpy(name, s.name);
1153          age = s.age;
1154          cout << "拷贝构造函数，保证name指向的是自己单独的内存块" << endl;
1155          return *this; //返回 “自引用”
1156      }
1157      virtual ~student(){                                 // 析构函数
1158          delete[] name;                              // 不能用free!
1159          cout << "析构函数，释放了100个char元素的动态空间" << endl;
1160      }
```

```
};
int main() {
    student s;
    student k("John", 56);
    cout << k.name << ", age " << k.age << endl ;

    student m(k);
    s = k;
    cout << s.name << ", age " << s.age << endl ;
    return 0;
}
```

21. 类体外定义方法（成员函数），必须在类定义中声明，类体外要有类作用域，否则就是全局外部函数了！

```
#include <iostream>
using namespace std;
class Date {
    int d, m, y;
public:
    void print();
    Date(int dd = 1, int mm = 1, int yy = 1999) {
        d = dd; m = mm; y = yy;
        cout << "构造函数" << endl;
    }
    virtual ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
            //目前不需要做任何释放工作，因为构造函数没申请资源
        cout << "析构函数" << endl;
    }
};

void Date::print() {
    cout << y << "-" << m << "-" << d << endl;
}

int main() {
    Date day;
    day.print();
}
```

22. 类模板：我们可以将一个类变成“类模板”或“模板类”，正如一个模板函数一样。

```
//将原来的所有double换成模板类型T，并加上模板头 template<class T>

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

template<class T>
class Array {
    T size;
    T *data;
public:
    Array(int s) {
        size = s;
        data = new T[s];
```

```
1216      }
1217
1218      virtual ~Array() {
1219          delete[] data;
1220      }
1221
1222      T &operator [] (int i) {
1223          if (i < 0 || i >= size) {
1224              cerr << endl << "Out of bounds" << endl;
1225              throw "index out of range";
1226          }
1227          else return data[i];
1228      }
1229  };
1230
1231  int main() {
1232      Array<int> t(5);
1233
1234      t[0] = 45;                          // OK
1235      t[4] = t[0] + 6;                    // OK
1236      cout << t[4] << endl;               // OK
1237
1238      t[10] = 7;                          // error!
1239
1240      Array<double> a(5);
1241
1242      a[0] = 45.5;                          // OK
1243      a[4] = a[0] + 6.5;                    // OK
1244      cout << a[4] << endl;               // OK
1245
1246      a[10] = 7.5;                          // error!
1247      return 0;
1248  }
1249
1250
1251  23. typedef 类型别名
1252
1253  #include <iostream>
1254  using namespace std;
1255  typedef int INT;
1256  int main() {
1257      INT i = 3; //等价于int i = 3;
1258      cout << i << endl;
1259      return 0;
1260  }
1261
1262  24. string
1263
1264  //string对象的初始化
1265  #include <iostream>
1266  #include <string>           //typedef std::basic_string<char> string;
1267  using namespace std;
1268  typedef string String;
1269
1270  int main() {
1271      // with no arguments
```

```
1272      string s1;          //默认构造函数：没有参数或参数有默认值
1273      String s2("hello"); //普通构造函数    String就是string
1274      s1 = "Anatoliy";    //赋值运算符
1275      String s3(s1);   //拷贝构造函数 string s3  =s1;
1276
1277      cout << "s1  is: " << s1 << endl;
1278      cout << "s2  is: " << s2 << endl;
1279      cout << "s3  is: " << s2 << endl;
1280
1281      // first argumen C string
1282      // second number of characters
1283      string s4("this is a C_sting", 10);
1284      cout << "s4  is: " << s4 << endl;
1285
1286      // 1 - C++ string
1287      // 2 - start position
1288      // 3 - number of characters
1289      string s5(s4, 6, 4); // copy word from s3
1290      cout << "s5  is: " << s5 << endl;
1291
1292      // 1 - number characters
1293      // 2 - character itself
1294      string s6(15, '*');
1295      cout << "s6  is: " << s6 << endl;
1296
1297      // 1 - start iterator
1298      // 2 - end iterator
1299      string s7(s4.begin(), s4.end() - 5);
1300      cout << "s7  is: " << s7 << endl;
1301
1302      // you can instantiate string with assignment
1303      string s8 = "Anatoliy";
1304      cout << "s8  is: " << s8 << endl;
1305
1306      string s9 = s1 + "hello"+ s2;  //s1 + "hello"+ s2的结果是string类型的对象(变量)
1307      cout << "s9  is: " << s9 << endl;
1308      return 0;
1309  }
1310
1311  //访问其中元素、遍历
1312  #include <iostream>
1313  #include <string>
1314  using namespace std;
1315
1316  int main() {
1317      string s = "hell";
1318      string w = "worl!";
1319      s = s + w; //s +=w;
1320
1321      for (int ii = 0; ii != s.size(); ii++)
1322          cout << ii << " " << s[ii] << endl;
1323      cout << endl;
1324
1325      string::const_iterator cii;
1326      int ii = 0;
1327      for (cii = s.begin(); cii != s.end(); cii++)
```

```
        cout << ii++ << " " << *cii << endl;
}

25. vector

#include <vector>
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
using std::vector;
int main() {
    vector<double> student_marks;

    int num_students;
    cout << "Number of students: " << endl;
    cin >> num_students;

    student_marks.resize(num_students);

    for (vector<double>::size_type i = 0; i < num_students; i++) {
        cout << "Enter marks for student #" << i + 1
            << ": " << endl;
        cin >> student_marks[i];
    }

    cout << endl;
    for (vector<double>::iterator it = student_marks.begin();
        it != student_marks.end(); it++) {
        cout << *it << endl;
    }
    return 0;
}


26. Inheritance继承(Derivation派生)： 一个派生类(derived class)
从1个或多个父类(parent class) / 基类(base class)继承，即继承父类的属性和行为，
但也有自己的特有属性和行为。如：

#include <iostream>
#include <string>
using namespace std;
class Employee{
    string name;
public:
    Employee(string n);
    void print();
};

class Manager: public Employee{
    int level;
public:
    Manager(string n, int l = 1);
    //void print();
};

```

```
1384 Employee::Employee(string n) :name(n)//初始化成员列表
1385 {
1386     //name = n;
1387 }
1388 void Employee::print() {
1389     cout << name << endl;
1390 }
1391
1392 Manager::Manager(string n, int l) :Employee(n), level(l) {
1393 }
1394
1395 //派生类的构造函数只能描述它自己的成员和其直接基类的初始式，不能去初始化基类的成员。
1396 Manager::Manager(string n, int l) : name(n), level(l) {
1397 }
1398
1399 int main() {
1400     Manager m("Zhang",2);
1401     Employee e("Li");
1402     m.print();
1403     e.print();
1404 }
1405
1406
1407 class Manager : public Employee
1408 {
1409     int level;
1410 public:
1411     Manager(string n, int l = 1);
1412     void print();
1413 };
1414 Manager::Manager(string n, int l) :Employee(n), level(l) {
1415 }
1416 void Manager::print() {
1417     cout << level << "\t";
1418     Employee::print();
1419 }
1420 int main() {
1421     Manager m("Zhang");
1422     Employee e("Li");
1423     m.print();
1424     e.print();
1425 }
1426
1427 27. 虚函数Virtual Functions
1428 派生类的指针可以自动转化为基类指针，用一个指向基类的指针分别指向基类对象和派生类对象，并2次调用print()函数输出，结果如何？
1429 int main() {
1430     Employee *p;
1431     Manager m("Zhang", 1);
1432     Employee e("Li");
1433     p = &e;
1434     p->print();
1435     p = &m;
1436     p->print();
1437 }
1438
```

```
1439 //可以将print声明为虚函数Virtual Functions
1440 class Employee{
1441     string name;
1442 public:
1443     Employee(string n);
1444     virtual void print();
1445 };
1446 class Manager : public Employee
1447 {
1448     int level;
1449 public:
1450     Manager(string n, int l = 1);
1451     void print();
1452 };
1453 Employee::Employee(string n) :name(n) {
1454 }
1455 void Employee::print() {
1456     cout << name << endl;
1457 }
1458
1459 Manager::Manager(string n, int l) :Employee(n), level(l) {
1460 }
1461 void Manager::print() {
1462     cout << level << "\t";
1463     Employee::print();
1464 }
1465 int main() {
1466     Employee *p;
1467     Manager m("Zhang", 1);
1468     Employee e("Li");
1469     p = &e;
1470     p->print();
1471     p = &m;
1472     p->print();
1473 }
1474
1475 假如一个公司的雇员(包括经理)要保存在一个数组如vector中，怎么办?
1476 难道用2个数组：
1477 Manager managers[100]; int m_num=0;
1478 Employee employees[100]; int e_num=0;
1479 //但经理也是雇员啊?
1480 实际上：派生类的指针可以自动转化为基类指针。可以将所有雇员保存在一个
1481 Employee* employees[100]; int e_num=0;
1482
1483 int main() {
1484     Employee* employees[100]; int e_num = 0;
1485     Employee* p;
1486     string name; int level;
1487     char cmd;
1488     while (cin >> cmd) {
1489         if (cmd == 'M' || cmd == 'm') {
1490             cout << "请输入姓名和级别" << endl;
1491             cin >> name >> level;
1492             p = new Manager(name, level);
1493             employees[e_num] = p; e_num++;
1494         }
```

```
        else if (cmd == 'e' || cmd == 'E') {
            cout << "请输入姓名" << endl;
            cin >> name;
            p = new Employee(name);
            employees[e_num] = p; e_num++;
        }
        else break;
        cout << "请输入命令" << endl;
    }
    for (int i = 0; i < e_num; i++) {
        employees[i]->print();
    }
}


当然，我们可以从一个类派生出多个不同的类，如：
class Employee{
    //...
public:
    virtual void print();
};

class Manager : public Employee{
    // ...
public:
    void print();
};

class Secretary : public Employee{
    // ...
public:
    void print();
};


//我们也可以从多个不同的类派生出一个类来：多重派生(Multiple inheritance)

class One{
    // class internals
};

class Two{
    // class internals
};

class MultipleInheritance : public One, public Two
{
    // class internals
};


28. 纯虚函数（pure virtual function ）和抽象类(abstract base class)

函数体=0的虚函数称为“纯虚函数”。包含纯虚函数的类称为“抽象类”

#include <string>
```

```
1551 class Animal // This Animal is an abstract base class
1552 {
1553 protected:
1554     std::string m_name;
1555
1556 public:
1557     Animal(std::string name)
1558         : m_name(name)
1559     {   }
1560
1561     std::string getName() { return m_name; }
1562     virtual const char* speak() = 0; // note that speak is now a pure virtual
        function
1563 };
1564
1565 int main() {
1566     Animal a; //错：抽象类不能实例化(不能定义抽象类的对象(变量))
1567 }
1568
1569 //从抽象类派生的类型如果没有继承实现所有的纯虚函数，则仍然是“抽象类”
1570
1571 #include <iostream>
1572 class Cow : public Animal
1573 {
1574 public:
1575     Cow(std::string name)
1576         : Animal(name)
1577     {
1578     }
1579
1580     // We forgot to redefine speak
1581 };
1582
1583 int main(){
1584     Cow cow("Betsy"); //仍然错：因为Cow仍然是抽象类
1585     std::cout << cow.getName() << " says " << cow.speak() << '\n';
1586 }
1587
1588 像下面这样实现所有纯虚函数就没问题了，Cow不是一个抽象类
1589 #include <iostream>
1590 class Cow : public Animal
1591 {
1592 public:
1593     Cow(std::string name)
1594         : Animal(name)
1595     {
1596     }
1597
1598     virtual const char* speak() { return "Moo"; }
1599 };
1600
1601 int main()
1602 {
1603     Cow cow("Betsy");
1604     std::cout << cow.getName() << " says " << cow.speak() << '\n';
1605 }
```

```
1606
1607
1608
1609  //关注：
1610  //微博和B站：hw-dong
1611  //网易云课堂：hwdong
1612  //博客：https://a.hwdong.com
1613  //腾讯课堂：http://hwdong.ke.qq.com
1614
1615
1616
1617
```