```
/*
//-----------“2小时从C到C++快速入门（2018）”-------
//youtube：https://www.youtube.com/watch?v=NQjiS9XLm0k&list=PLBijWKRKPQMIf5VxAa16muCQLnVeofvAn
//bilibili:https://www.bilibili.com/video/BV1kW411Y76d-----------

//-------------------源代码文件---------------
//--------------请关注：------------
//youtube：hwdong
//博客：https://hwdong.net或https://hwdong-net.github.io
//腾讯课堂：http://hwdong.ke.qq.com
//B站：hw-dong


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;

   cout << "a contains : " << a << endl;
   cout << "b contains: " << b << endl;

   a = a * 2 + b;

   double c = a + b * a; //“即用即定义”，且可用表达式初始化

   cout << "c contains: " << c << endl;
}


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
#include <iostream>
using namespace std;

int main ()
{
   double a;

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

   {
      int a = 1; // "int a"隐藏了外部作用域的“double a"
      a = a * 10 + 4;
      cout << "Local number: " << a << endl;
   }

   cout << "You typed: " << a << endl; //main作用域的“double a"

   return 0;
}

7. for循环语句可以定义局部变量。
```

```

#include <iostream>
using namespace std;

int main (){
   int i = 0;
   for (int i = 0; i < 4; i++)
   {
      cout << i << endl;
   }

   cout << "i contains: " << i << endl;

   for (i = 0; i < 4; i++)
   {
      for (int i = 0; i < 4; i++)        // we're between
      {                                  // previous for's hooks
         cout << i<< " ";
      }
      cout << endl;
   }
   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;
}

int main(){
    int a = 3, b = 4;

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

    return 0;
}

/*
x,y得到2个int型变量的指针,x,y本身没有修改
修改的是x,y 指向的那2个int型变量的内容
*/
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;
}

int main() {
    int a = 3, b = 4;

    swap(&a, &b);  // &a赋值给x,&b赋值给y,
                   //x,y分别是int*指针，指向a,b
                   //*x, *y就是a和b
    cout << a << ", " << b << endl;         // Displays 100, 4.

    return 0;
}
```

```
265
266
267 //x,y是实参的引用
268 void swap(int &x, int &y) {
269     cout << "swap函数内交换前：" << x << " " << y << endl;
270     int t = x; x = y; y = t;
271     cout << "swap函数内交换后：" << x << " " << y << endl;
272 }
273
274 int main(){
275     int a = 3, b = 4;
276
277     swap(a, b); //x,y将分别是a,b的引用，即x就是a,y就是b
278     cout << a << ", " << b << endl;        // Displays 100, 4.
279
280     return 0;
281 }
282
283 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，
284 如果不希望因此无意中修改实参，可以用const修改符。如
285 #include <iostream>
286 using namespace std;
287
288 void change (double &x, const double &y,double z){
289    x = 100;
290    y = 200;  //错！ y不可修改，是const double &
291    z = 300;
292 }
293
294 int main (){
295    double a,b,c;//内在类型变量未提供初始化式，默认初始化为0
296
297    change(a, b, c);
298    cout << a << ", " << b << ", " << c << endl;        // Displays 100, 4.
299
300    return 0;
301 }
302
303 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"，
304 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，
305 提高程序执行效率
306 #include <iostream>
307 #include <cmath>
308 using namespace std;
309
310 inline double distance(double a, double b) {
311     return sqrt(a * a + b * b);
312 }
313
314 int main() {
315     double k = 6, m = 9;
316     // 下面2行将产生同样的代码：
317     cout << distance(k, m) << endl;
```

```
318     cout << sqrt(k * k + m * m) << endl;
319
320     return 0;
321 }
322
323
324
325 11. 通过 try-catch处理异常情况
326 正常代码放在try块，catch中捕获try块抛出的异常
327
328 #include <iostream>
329 #include <cmath>
330 using namespace std;
331
332 int main (){
333    int a, b;
334
335    cout << "Type a number: ";
336    cin >> a;
337    cout << endl;
338
339    try {
340       if (a > 100) throw 100;
341       if (a < 10)  throw 10;
342       throw "hello";
343    }
344    catch (int result) {
345       cout << "Result is: " << result << endl;
346       b = result + 1;
347    }
348    catch (char * s) {
349        cout << "haha " << s << endl;
350    }
351
352    cout << "b contains: " << b << endl;
353
354    cout << endl;
355
356    // another example of exception use:
357
358    char zero[] = "zero";
359    char pair[] = "pair";
360    char notprime[] = "not prime";
361    char prime[] = "prime";
362
363    try {
364       if (a == 0) throw zero;
365       if ((a / 2) * 2 == a) throw pair;
366       for (int i = 3; i <= sqrt (a); i++){
367          if ((a / i) * i == a) throw notprime;
368       }
369       throw prime;
370    }
```

```
    catch (char *conclusion) {
        cout << "异常结果是： " << conclusion << endl;
    }
    catch (...) {
        cout << "其他异常情况都在这里捕获 " << endl;
    }

    cout << endl;

    return 0;
}



12. 默认形参： 函数的形参可带有默认值。必须一律在最右边

#include <iostream>
using namespace std;

double test(double a, double b = 7) {
    return a - b;
}

int main() {
    cout << test(14, 5) << endl;
    cout << test(14) << endl;

    return 0;
}

/*错： 默认参数一律靠右*/
double test(double a, double b = 7, int c) {
    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;
}

```

```
424
425 //错：编译器无法区分int add (int a, int b),void  add (int a, int b)
426 void  add(int a, int b) {
427     return a - b;
428 }
429
430
431 int main() {
432     double   m = 7, n = 4;
433     int      k = 5, p = 3;
434
435     cout << add(m, n) << " , " << add(k, p) << endl;
436
437     return 0;
438 }
439
440
441 14.运算符重载
442
443 #include <iostream>
444 using namespace std;
445
446 struct Vector2{
447    double x;
448    double y;
449 };
450
451 Vector2 operator * (double a, Vector2 b){
452     Vector2 r;
453
454    r.x = a * b.x;
455    r.y = a * b.y;
456
457    return r;
458 }
459
460 Vector2 operator+ (Vector2 a, Vector2 b) {
461     Vector2 r;
462
463     r.x = a.x + b.x;
464     r.y = a.y + b.y;
465
466     return r;
467 }
468
469 int main (){
470    Vector2 k, m;              // C++定义的struct类型前不需要再加关键字struct:  ↵
       "struct vector"
471
472    k.x =  2;                  //用成员访问运算符.访问成员
473    k.y = -1;
474
475    m = 3.1415927 * k;         // Magic!
```

```
476
477     cout << "(" << m.x << ", " << m.y << ")" << endl;
478
479     Vector2 n = m + k;
480     cout << "(" << n.x << ", " << n.y << ")" << endl;
481     return 0;
482 }
483
484
485
486
487 #include <iostream>
488 using namespace std;
489
490 struct Vector2 {
491     double x;
492     double y;
493 };
494
495 ostream& operator << (ostream& o, Vector2 a){
496     o << "(" << a.x << ", " << a.y << ")";
497     return o;
498 }
499
500 int main (){
501     Vector2 a;
502
503     a.x = 35;
504     a.y = 23;
505     cout << a << endl; // operator <<(cout,a);
506     return 0;
507 }
508
509
510
511
512
513 15. 模板template函数：厌倦了对每种类型求最小值
514
515 #include <iostream>
516 using namespace std;
517 int minValue(int a, int b) {//return a<b?a:b
518     if (a < b) return a;
519     else return b;
520 }
521 double minValue(double a, double b) {//return a<b?a:b
522     if (a < b) return a;
523     else return b;
524 }
525
526 int main() {
527     int i = 3, j = 4;
528     cout << "min of " << i << " and " << j << " is " << minValue(i, j) <<
```

```
          endl;
529       double x = 3.5, y = 10;
530       cout << "min of " << x << " and " << y << " is " << minValue(x, y) <<
            endl;
531
532   }
533
534   //可以转化成：  模板函数
535   #include <iostream>
536   using namespace std;
537
538   //可以对任何能比较大小(<)的类型使用该模板让编译器
539   //自动生成一个针对该数据类型的具体函数
540   template<class TT>
541   TT minValue(TT a, TT b) {//return a<b?a:b
542       if (a < b) return a;
543       else return b;
544   }
545
546   int main() {
547       int i = 3, j = 4;
548       cout << "min of " << i << " and " << j << " is " << minValue(i, j) <<
            endl;
549       double x = 3.5, y = 10;
550       cout << "min of " << x << " and " << y << " is " << minValue(x, y) <<
            endl;
551
552       //但是,不同类型的怎么办？
553       cout << "min of " << i << " and " << y << " is " << minValue(i, y) <<
            endl;
554   }
555
556
557
558   //可以对任何能比较大小(<)的类型使用该模板让编译器
559   //自动生成一个针对该数据类型的具体函数
560   #include <iostream>
561   using namespace std;
562
563   template<class T1, class T2>
564   T1 minValue(T1 a, T2 b) {//return a<b?a:b
565       if (a < b) return a;
566       else return (T2)b; //强制转化为T1类型
567   }
568
569   int main() {
570       int i = 3, j = 4;
571       cout << "min of " << i << " and " << j << " is " << minValue(i, j) <<
            endl;
572       double x = 3.5, y = 10;
573       cout << "min of " << x << " and " << y << " is " << minValue(x, y) <<
            endl;
574
```

```
575     //但是,不同类型的怎么办？
576     cout << "min of " << i << " and " << y << " is " << minValue(i, y) <<
          endl;
577 }
578
579
580
581
582 //堆存储区
583 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,
584 可以对类对象调用初始化构造函数或销毁析构函数
585
586 #define _CRT_SECURE_NO_WARNINGS    //windows
587 #include <iostream>
588 #include <cstring>
589 using namespace std;
590 int main() {
591     double d = 3.14;           // 变量d是一块存放double值的内存块
592     double *dp;                // 指针变量dp：保存double类型的地址的变量
593                                // dp的值得类型是double *
594                                // dp是存放double *类型值 的内存块
595
596     dp = &d;                  //取地址运算符&用于获得一个变量的地址,
597                               // 将double变量d的地址(指针)保存到double*指针变量
                      dp中
598                               // dp和&d的类型都是double *
599
600     *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)
601                              //*dp就是dp指向的那个d
602     cout << "*dp= " << *dp << "  d=:" << d << endl;
603
604     cout << "Type a number: ";
605     cin >> *dp;                    //输出dp指向的double内存块的值
606     cout << "*dp= " << *dp << "  d=:" << d << endl;
607
608     dp = new double;           // new 分配正好容纳double值的内存块（如4或8个字
          节）
609                                // 并返回这个内存块的地址，而且地址的类型是
                      double *
610                                //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
611                                // 但目前这个内存块的值是未知的
612
613                                // 注意：
614                                // 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
615                                //而d,dp等局部变量是这个程序自身的静态存储空间
616                                // 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
617
618
```

```
619     *dp = 45.3;                 //*dp指向的double内存块的值变成45.3
620
621     cout << "Type a number: ";
622     cin >> *dp;                     //输出dp指向的double内存块的值
623     cout << "*dp= " << *dp << endl;
624
625     *dp = *dp + 5;                  //修改dp指向的double内存块的值45.3+5
626
627     cout << "*dp= " << *dp << endl;
628
629     delete dp;                          // delete 释放dp指向的动态分配的double
        内存块
630
631
632     dp = new double[5];                 //new 分配了可以存放15个double值的内存
        块,
633                                         //返回这块连续内存的起始地址，而且指针类
                型是
634                                         //double *，  实际是第一个double元素的
                地址
635                                         // 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
636
637     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
638     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
639
640     cout << "d[0]=: " << dp[0] << "   d[1]=: " << dp[1] << endl;
641
642     delete[] dp;                        // 释放dp指向的多个double元素占据的内存
        块,
643                                         // 对每个double元素调用析构函数以释放资
                源
644                                         // 缺少[]，只释放第一个double元素的内存
                块，这叫“内存泄漏”
645
646
647     int n = 8;
648
649     dp = new double[n];                 // new 可以分配随机大小的double元素,
650                                         // 而静态数组则必须是编译期固定大小，即
                大小为常量
651                                         // 如 double arr[20];
652                                         //通过下标访问每个元素
653     for (int i = 0; i < n; i++) {
654         dp[i] = i;
655     }   //通过指针访问每个元素
656
657     double *p = dp;
658     for (int i = 0; i < n; i++) {
659         cout << *(p + i) << endl; //p[i]或dp[i]
660     }
```

```
661     cout << endl;
662
663     for (double *p = dp, *q = dp + n; p < q; p++) {
664         cout << *p << endl;
665     }
666     cout << endl;
667
668     delete[] dp;
669
670     char *s;
671     s = new char[100];
672
673     '\0'
674     strcpy(s, "Hello!"); //将字符串常量拷贝到s指向的字符数组内存块中
675
676     cout << s << endl;
677
678     delete[] s;  //用完以后，记得释放内存块，否则会“内存泄漏”！
679
680     return 0;
681 }
682
683
684
685
686
687 17．类：是在C的struct类型上，增加了“成员函数”。
688 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，
689 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。
690
691
692 #include <iostream>
693 using namespace std;
694
695 struct Date {
696     int d, m, y;
697     void init(int dd, int mm, int yy) {
698         d = dd; m = mm; y = yy;
699     }
700     void print() {
701         cout << y << "-" << m << "-" << d << endl;
702     }
703 };
704
705 int main (){
706     Date day;
707     day.print();           //通过类Date对象day调用类Date的print方法
708     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
709     day.print();           //通过类Date对象day调用类Date的print方法
710
711    return 0;
712 }
713
```

```
714
715
716 // 成员函数 返回“自引用” （*this）
717 #include <iostream>
718 using namespace std;
719
720 struct Date {
721     int d, m, y;
722     void init(int dd, int mm, int yy) {
723         d = dd; m = mm; y = yy;
724     }
725     void print() {
726         cout << y << "-" << m << "-" << d << endl;
727     }
728     Date& add(int dd) {
729         d = d + dd;
730         return *this;     //this是指向调用这个函数的类型对象指针，
731                           // *this就是调用这个函数的那个对象
732                           //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
733                           //通过返回自引用，可以连续调用这个函数
734                           // day.add(3);
735                           // day.add(3).add(7);
736     }
737 };
738
739 int main() {
740     Date day;
741     day.print();           //通过类Date对象day调用类Date的print方法
742     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
743     day.print();           //通过类Date对象day调用类Date的print方法
744     day.add(3);
745     day.add(5).add(7);
746     day.print();
747
748     return 0;
749 }
750
751 //成员函数重载“运算符函数”
752 #include <iostream>
753 using namespace std;
754
755 struct Date {
756     int d, m, y;
757     void init(int dd, int mm, int yy) {
758         d = dd; m = mm; y = yy;
759     }
760     void print() {
761         cout << y << "-" << m << "-" << d << endl;
762     }
763     Date& operator+=(int dd) {
764         d = d + dd;
765         return *this;     //this是指向调用这个函数的类型对象指针，
766                           // *this就是调用这个函数的那个对象
```

```
767                             //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
768                             //通过返回自引用，可以连续调用这个函数
769                             // day.add(3);
770                             // day.add(3).add(7);
771     }
772 };
773
774 int main() {
775     Date day;
776     day.print();           //通过类Date对象day调用类Date的print方法
777     day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
778     day.print();           //通过类Date对象day调用类Date的print方法
779     day += 3;             // day.add(3);
780     (day += 5) += 7;        //day.add(5).add(7);
781     day.print();
782
783     return 0;
784 }
785
786 18. 构造函数和析构函数
787
788 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，
789 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
790
791 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
792
793 #include <iostream>
794 using namespace std;
795
796 struct Date {
797     int d, m, y;
798
799     Date(int dd, int mm, int yy) {
800         d = dd; m = mm; y = yy;
801         cout << "构造函数" << endl;
802     }
803     void print() {
804         cout << y << "-" << m << "-" << d << endl;
805     }
806     ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
807         //目前不需要做任何释放工作，因为构造函数没申请资源
808         cout << "析构函数" << endl;
809     }
810 };
811
812 int main(){
813     Date day;     //错：会自动调用构造函数，但没提供3个参数
814     Date(4, 6, 1999);     //会自动调用构造函数Date(int dd, int mm, int yy)
815 //  day.init(4, 6, 1999); //通过类Date对象day调用类Date的init方法
816     day.print();           //通过类Date对象day调用类Date的print方法
817
```

```
818     return 0;
819 }
820
821 执行上述代码，看看构造函数和析构函数执行了吗？
822
823 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）？
824
825 Date day;
826 Date day1（2）;
827 Date day2(23, 10);
828 Date day3(2,3,1999);
829
830 当然可以的
831 struct Date {
832     int d, m, y;
833     Date() {
834         d = m = 1; y = 2000;
835         cout << "构造函数" << endl;
836     }
837     Date(int dd) {
838         d = dd;  m = 1; y = 2000;
839         cout << "构造函数" << endl;
840     }
841     Date(int dd, int mm) {
842         d = dd;  m = mm;  y = 2000;
843         cout << "构造函数" << endl;
844     }
845     Date(int dd, int mm, int yy) {
846         d = dd; m = mm; y = yy;
847         cout << "构造函数" << endl;
848     }
849     void print() {
850         cout << y << "-" << m << "-" << d << endl;
851     }
852     ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
853              //目前不需要做任何释放工作，因为构造函数没申请资源
854         cout << "析构函数" << endl;
855     }
856 };
857
858 为什么不用默认参数呢？
859 #include <iostream>
860 using namespace std;
861
862 using namespace std;
863 struct Date {
864     int d, m, y;
865     Date(int dd = 1, int mm = 1, int yy = 1999) {
866         d = dd; m = mm; y = yy;
867         cout << "构造函数" << endl;
868     }
869     void print() {
870         cout << y << "-" << m << "-" << d << endl;
```

```
871     }
872     ~Date() {//析构函数名是~和类名，且不带参数，没有返回类型
873              //目前不需要做任何释放工作，因为构造函数没申请资源
874         cout << "析构函数" << endl;
875     }
876 };
877 
878 
879 int main(){
880     Date day;
881     Date day1(2);
882     Date day2(23, 10);
883     Date day3(2, 3, 1999);
884 
885     day.print();
886     day1.print();
887     day2.print();
888     day3.print();
889     return 0;
890 }
891 
892 
893 //析构函数示例
894 #define _CRT_SECURE_NO_WARNINGS //windows系统
895 #include <iostream>
896 #include <cstring>
897 using namespace std;
898 
899 struct student {
900     char *name;
901     int age;
902 
903     student(char *n = "no name", int a = 0) {
904         name = new char[100];                  // 比malloc好!
905         strcpy(name, n);
906         age = a;
907         cout << "构造函数，申请了100个char元素的动态空间" << endl;
908     }
909 
910     virtual ~student(){                                  // 析构函数
911         delete name;                              // 不能用free!
912         cout << "析构函数，释放了100个char元素的动态空间" << endl;
913     }
914 };
915 
916 int main() {
917     cout << "Hello!" << endl << endl;
918 
919     student a;
920     cout << a.name << ", age " << a.age << endl << endl;
921 
922     student b("John");
923     cout << b.name << ", age " << b.age << endl << endl;
```

```
924
925     b.age = 21;
926     cout << b.name << ", age " << b.age << endl << endl;
927
928     student c("Miki", 45);
929     cout << c.name << ", age " << c.age << endl << endl;
930
931     cout << "Bye!" << endl << endl;
932
933     return 0;
934 }
935
936
937 19. 访问控制、类接口
938 将关键字struct换成class
939
940 #include <iostream>
941 #include <cstring>
942 using namespace std;
943
944 class student {
945     char *name;
946     int age;
947
948     student(char *n = "no name", int a = 0) {
949         name = new char[100];                 // 比malloc好!
950         strcpy(name, n);
951         age = a;
952         cout << "构造函数，申请了100个char元素的动态空间" << endl;
953     }
954
955     virtual ~student() {                              // 析构函数
956         delete name;                             // 不能用free!
957         cout << "析构函数，释放了100个char元素的动态空间" << endl;
958     }
959 };
960
961 int main() {
962     cout << "Hello!" << endl << endl;
963
964     student a; //编译出错:无法访问 private 成员(在“student”类中声明)
965     cout << a.name << ", age " << a.age << endl << endl; //编译出错
966
967     student b("John"); //编译出错
968     cout << b.name << ", age " << b.age << endl << endl;//编译出错
969
970     b.age = 21;  //编译出错
971     cout << b.name << ", age " << b.age << endl << endl;//编译出错
972
973     return 0;
974 }
975
976  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
```

```
977 #include <iostream>
978 #include <cstring>
979 using namespace std;
980
981  class student {
982 //默认私有的，等价于 private:
983      char *name;
984      int age;
985  public: //公开的
986      student(char *n = "no name", int a = 0) {
987          name = new char[100];                 // 比malloc好!
988          strcpy(name, n);
989          age = a;
990          cout << "构造函数，申请了100个char元素的动态空间" << endl;
991      }
992
993      virtual ~student() {                              // 析构函数
994
995          delete name;                              // 不能用free!
996          cout << "析构函数，释放了100个char元素的动态空间" << endl;
997      }
998  };
999
1000  int main() {
1001      cout << "Hello!" << endl << endl;
1002
1003      student a;  //OK
1004      cout << a.name << ", age " << a.age << endl ; //编译出错：无法访问
          private 成员(在“student”类中声明)
1005
1006      student b("John");
1007      cout << b.name << ", age " << b.age << endl ;//编译出错
1008
1009      b.age = 21;
1010      cout << b.name << ", age " << b.age << endl 1;//编译出错
1011      return 0;
1012  }
1013
1014  a.name，a.age仍然不能访问，  如何进一步修改呢？
1015
1016 #include <iostream>
1017 #include <cstring>
1018 using namespace std;
1019
1020  class student {
1021      //默认私有的，等价于 private:
1022      char *name;
1023      int age;
1024  public: //公开的
1025      char *get_name() { return name; }
1026      int get_age() { return age; }
1027      void set_age(int ag) { age = ag; }
1028      student(char *n = "no name", int a = 0) {
```

```
1029            name = new char[100];                   // 比malloc好!
1030            strcpy(name, n);
1031            age = a;
1032            cout << "构造函数，申请了100个char元素的动态空间" << endl;
1033        }
1034
1035        virtual ~student() {                                    // 析构函数
1036            delete name;                            // 不能用free!
1037            cout << "析构函数，释放了100个char元素的动态空间" << endl;
1038        }
1039    };
1040
1041    int main() {
1042        cout << "Hello!" << endl << endl;
1043
1044        student a;
1045        cout << a.get_name() << ", age " << a.get_age() << endl ; //编译出错
1046
1047        student b("John");
1048        cout << b.get_name() << ", age " << b.get_age() << endl l;//编译出错
1049
1050        b.set_age(21);
1051        cout << b.get_name() << ", age " << b.get_age() << endl ;//编译出错
1052
1053        return 0;
1054    }
1055
1056
1057    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，
1058        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！
1059
1060    定义一个数组类array
1061
1062    #include <iostream>
1063    #include <cstdlib>
1064    using namespace std;
1065
1066     class Array {
1067        int size;
1068        double *data;
1069     public:
1070        Array(int s) {
1071            size = s;
1072            data = new double[s];
1073        }
1074
1075        virtual ~Array() {
1076            delete[] data;
1077        }
1078
1079        double &operator [] (int i) {
1080            if (i < 0 || i >= size) {
```

```
1081                cerr << endl << "Out of bounds" << endl;
1082                throw  "Out of bounds";
1083            }
1084            else return data[i];
1085        }
1086    };
1087
1088    int main() {
1089        Array t(5);
1090
1091        t[0] = 45;                          // OK
1092        t[4] = t[0] + 6;                    // OK
1093        cout << t[4] << endl;               // OK
1094
1095        t[10] = 7;                          // error!
1096        return 0;
1097    }
1098
1099    20. 拷贝： 拷贝构造函数、赋值运算符
1100
1101       下列赋值为什么会出错？
1102          “student m(s);
1103            s = k;”
1104       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
1105       赋值运算符：一个对象赋值给另外一个对象
1106
1107    #define  _CRT_SECURE_NO_WARNINGS   //windows系统
1108    #include <iostream>
1109    #include <cstdlib>
1110    using namespace std;
1111
1112    struct student {
1113        char *name;
1114        int age;
1115        student(char *n = "no name", int a = 0) {
1116            name = new char[100];                  // 比malloc好!
1117            strcpy(name, n);
1118            age = a;
1119            cout << "构造函数，申请了100个char元素的动态空间" << endl;
1120        }
1121
1122        virtual ~student() {                                  // 析构函数
1123            delete[] name;                               // 不能用free!
1124            cout << "析构函数，释放了100个char元素的动态空间" << endl;
1125        }
1126    };
1127    int main() {
1128        student s;
1129        student k("John", 56);
1130        cout << k.name << ", age " << k.age << endl;
1131
1132        student m(s); //拷贝构造函数
1133        s = k;  //赋值运算符
```

```
1134      cout << s.name << ", age " << s.age << endl;
1135
1136      return 0;
1137  }
1138
1139  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！
1140  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。
1141
1142  #define  _CRT_SECURE_NO_WARNINGS
1143  #include <iostream>
1144  #include <cstdlib>
1145  using namespace std;
1146  struct student {
1147      char *name;
1148      int age;
1149
1150      student(char *n = "no name", int a = 0) {
1151          name = new char[100];                  // 比malloc好!
1152          strcpy(name, n);
1153          age = a;
1154          cout << "构造函数，申请了100个char元素的动态空间" << endl;
1155      }
1156      student(const student &s) {                // 拷贝构造函数 Copy constructor
1157          name = new char[100];
1158          strcpy(name, s.name);
1159          age = s.age;
1160          cout << "拷贝构造函数，保证name指向的是自己单独的内存块" << endl;
1161      }
1162      student & operator=(const student &s) {                // 拷贝构造函数 Copy constructor
1163          strcpy(name, s.name);
1164          age = s.age;
1165          cout << "拷贝构造函数，保证name指向的是自己单独的内存块" << endl;
1166          return *this; //返回 “自引用”
1167      }
1168      virtual ~student(){                                    // 析构函数
1169          delete[] name;                                 // 不能用free!
1170          cout << "析构函数，释放了100个char元素的动态空间" << endl;
1171      }
1172  };
1173  int main() {
1174      student s;
1175      student k("John", 56);
1176      cout << k.name << ", age " << k.age << endl ;
1177
1178      student m(k);
1179      s = k;
1180      cout << s.name << ", age " << s.age << endl ;
1181      return 0;
1182  }
1183
```

```
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];
    }

    virtual ~Array() {
        delete[] data;
    }

    T &operator [] (int i) {
        if (i < 0 || i >= size) {
            cerr << endl << "Out of bounds" << endl;
```

```
1236                throw "index out of range";
1237            }
1238            else return data[i];
1239        }
1240    };
1241
1242    int main() {
1243        Array<int> t(5);
1244
1245        t[0] = 45;                          // OK
1246        t[4] = t[0] + 6;                    // OK
1247        cout << t[4] << endl;               // OK
1248
1249        t[10] = 7;                          // error!
1250
1251        Array<double> a(5);
1252
1253        a[0] = 45.5;                          // OK
1254        a[4] = a[0] + 6.5;                    // OK
1255        cout << a[4] << endl;               // OK
1256
1257        a[10] = 7.5;                          // error!
1258        return 0;
1259    }
1260
1261
1262    23. typedef 类型别名
1263
1264    #include <iostream>
1265    using namespace std;
1266    typedef int INT;
1267    int main() {
1268        INT i = 3; //等价于int i = 3;
1269        cout << i << endl;
1270        return 0;
1271    }
1272
1273    24. string
1274
1275    //string对象的初始化
1276    #include <iostream>
1277    #include <string>            //typedef std::basic_string<char> string;
1278    using namespace std;
1279    typedef string String;
1280
1281    int main() {
1282        // with no arguments
1283        string s1;          //默认构造函数：没有参数或参数有默认值
1284        String s2("hello"); //普通构造函数     String就是string
1285        s1 = "Anatoliy";    //赋值运算符
1286        String s3(s1);   //拷贝构造函数 string s3  =s1;
1287
1288        cout << "s1  is: " << s1 << endl;
```

```
1289      cout << "s2  is: " << s2 << endl;
1290      cout << "s3  is: " << s2 << endl;
1291
1292      // first argumen C string
1293      // second number of characters
1294      string s4("this is a C_sting", 10);
1295      cout << "s4  is: " << s4 << endl;
1296
1297      // 1 - C++ string
1298      // 2 - start position
1299      // 3 - number of characters
1300      string s5(s4, 6, 4); // copy word from s3
1301      cout << "s5  is: " << s5 << endl;
1302
1303      // 1 - number characters
1304      // 2 - character itself
1305      string s6(15, '*');
1306      cout << "s6  is: " << s6 << endl;
1307
1308      // 1 - start iterator
1309      // 2 - end iterator
1310      string s7(s4.begin(), s4.end() - 5);
1311      cout << "s7  is: " << s7 << endl;
1312
1313      // you can instantiate string with assignment
1314      string s8 = "Anatoliy";
1315      cout << "s8  is: " << s8 << endl;
1316
1317      string s9 = s1 + "hello"+ s2;  //s1 + "hello"+ s2的结果是string类型的对象
          (变量)
1318      cout << "s9  is: " << s9 << endl;
1319      return 0;
1320  }
1321
1322  //访问其中元素、遍历
1323  #include <iostream>
1324  #include <string>
1325  using namespace std;
1326
1327  int main() {
1328      string s = "hell";
1329      string w = "worl!";
1330      s = s + w; //s +=w;
1331
1332      for (int ii = 0; ii != s.size(); ii++)
1333          cout << ii << " " << s[ii] << endl;
1334      cout << endl;
1335
1336      string::const_iterator cii;
1337      int ii = 0;
1338      for (cii = s.begin(); cii != s.end(); cii++)
1339          cout << ii++ << " " << *cii << endl;
1340  }
```

25. vector

```cpp
#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：

```cpp
#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();
};
```

```
1394
1395 Employee::Employee(string n) :name(n)//初始化成员列表
1396 {
1397     //name = n;
1398 }
1399 void Employee::print() {
1400     cout << name << endl;
1401 }
1402
1403 Manager::Manager(string n, int l) :Employee(n), level(l) {
1404 }
1405
1406 //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。
1407 Manager::Manager(string n, int l) : name(n), level(l) {
1408 }
1409
1410 int main() {
1411     Manager m("Zhang",2);
1412     Employee e("Li");
1413     m.print();
1414     e.print();
1415 }
1416
1417
1418 class Manager : public Employee
1419 {
1420     int level;
1421 public:
1422     Manager(string n, int l = 1);
1423     void print();
1424 };
1425 Manager::Manager(string n, int l) :Employee(n), level(l) {
1426 }
1427 void Manager::print() {
1428     cout << level << "\t";
1429     Employee::print();
1430 }
1431 int main() {
1432     Manager m("Zhang");
1433     Employee e("Li");
1434     m.print();
1435     e.print();
1436 }
1437
1438 27. 虚函数Virtual Functions
1439 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？
1440 int main() {
1441     Employee *p;
1442     Manager m("Zhang", 1);
1443     Employee e("Li");
1444     p = &e;
1445     p->print();
```

```
1446      p = &m;
1447      p->print();
1448  }
1449
1450  //可以将print声明为虚函数Virtual Functions
1451  class Employee{
1452      string name;
1453  public:
1454      Employee(string n);
1455      virtual void print();
1456  };
1457  class Manager : public Employee
1458  {
1459      int level;
1460  public:
1461      Manager(string n, int l = 1);
1462      void print();
1463  };
1464  Employee::Employee(string n) :name(n) {
1465  }
1466  void Employee::print() {
1467      cout << name << endl;
1468  }
1469
1470  Manager::Manager(string n, int l) :Employee(n), level(l) {
1471  }
1472  void Manager::print() {
1473      cout << level << "\t";
1474      Employee::print();
1475  }
1476  int main() {
1477      Employee *p;
1478      Manager m("Zhang", 1);
1479      Employee e("Li");
1480      p = &e;
1481      p->print();
1482      p = &m;
1483      p->print();
1484  }
1485
1486  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？
1487  难道用2个数组：
1488  Manager managers[100]; int m_num=0;
1489  Employee employees[100]; int e_num=0;
1490  //但经理也是雇员啊？
1491  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
1492  Employee* employees[100]; int e_num=0;
1493
1494  int main() {
1495      Employee* employees[100]; int e_num = 0;
1496      Employee* p;
1497      string name; int level;
1498      char cmd;
```

```
1499      while (cin >> cmd) {
1500          if (cmd == 'M' || cmd == 'm') {
1501              cout << "请输入姓名和级别" << endl;
1502              cin >> name >> level;
1503              p = new Manager(name, level);
1504              employees[e_num] = p; e_num++;
1505          }
1506          else if (cmd == 'e' || cmd == 'E') {
1507              cout << "请输入姓名" << endl;
1508              cin >> name;
1509              p = new Employee(name);
1510              employees[e_num] = p; e_num++;
1511          }
1512          else break;
1513          cout << "请输入命令" << endl;
1514      }
1515      for (int i = 0; i < e_num; i++) {
1516          employees[i]->print();
1517      }
1518  }
1519
1520
1521  当然，我们可以从一个类派生出多个不同的类，如：
1522  class Employee{
1523      //...
1524  public:
1525      virtual void print();
1526  };
1527
1528  class Manager : public Employee{
1529      // ...
1530  public:
1531      void print();
1532  };
1533
1534  class Secretary : public Employee{
1535      // ...
1536  public:
1537      void print();
1538  };
1539
1540
1541  //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)
1542
1543  class One{
1544      // class internals
1545  };
1546
1547  class Two{
1548      // class internals
1549  };
1550
1551  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>
class Animal // This Animal is an abstract base class
{
protected:
    std::string m_name;

public:
    Animal(std::string name)
        : m_name(name)
    {   }

    std::string getName() { return m_name; }
    virtual const char* speak() = 0; // note that speak is now a pure virtual function
};

int main() {
    Animal a; //错：抽象类不能实例化(不能定义抽象类的对象(变量))
}

//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”

#include <iostream>
class Cow : public Animal
{
public:
    Cow(std::string name)
        : Animal(name)
    {
    }

    // We forgot to redefine speak
};

int main(){
    Cow cow("Betsy"); //仍然错：因为Cow仍然是抽象类
    std::cout << cow.getName() << " says " << cow.speak() << '\n';
}

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
#include <iostream>
class Cow : public Animal
{
public:
```

```
1604      Cow(std::string name)
1605          : Animal(name)
1606      {
1607      }
1608
1609      virtual const char* speak() { return "Moo"; }
1610  };
1611
1612  int main()
1613  {
1614      Cow cow("Betsy");
1615      std::cout << cow.getName() << " says " << cow.speak() << '\n';
1616  }
1617
1618
1619
1620  //关注：
1621  //微博和B站：hw-dong
1622  //网易云课堂：hwdong
1623  //博客：https://a.hwdong.com
1624  //腾讯课堂：http://hwdong.ke.qq.com
1625
1626
1627
1628
```