**作业**

**贪心：钱币找零问题**

#include<iostream>

using namespace std;

/\*

钱币找零问题：

假设1元、2元、5元、10元、20元、50元、100元的纸币分别有c0, c1, c2, c3, c4, c5, c6张。现在要用这些钱来支付K元，至少要用多少张纸币？

\*/

/\*

思路：

要求是使用最少的纸币数去支付，所以，应从最大额度的纸币开始讨论，当当前纸币达到最大用量，剩余的用比当前纸币额度少一级的去讨论，直到用到1元纸币，或者其中讨论到某额度纸币时刚好支付完。

因为最大额度是100元纸币，所以先将K除以100得到的整数部分n1就是100元钱币数量，再将K减去n1\*100，然后继续讨论剩余的数量，以此类推，就可以得到结果。

\*/

void f(int K)

//K为所需支付的钱

{

int n, k = K, c0, c1, c2, c3, c4, c5, c6;

cout << endl; //换行(能让视觉好一些)

c6 = k / 100; //计算100元所需张数

cout << "100元纸币需要：" << c6 << endl;

k = k - c6 \* 100; //减去用100元支付过的额度

c5 = k / 50; //计算50元所需张数

cout << "50元纸币需要：" << c5 << endl;

k = k - c5 \* 50; //减去用50元支付过的额度

c4 = k / 20; //计算20元所需张数

cout << "20元纸币需要：" << c4 << endl;

k = k - c4 \* 20; //减去用20元支付过的额度

c3 = k / 10; //计算10元所需张数

cout << "10元纸币需要：" << c3 << endl;

k = k - c3 \* 10; //减去用10元支付过的额度

c2 = k / 5; //计算5元所需张数

cout << "5元纸币需要：" << c2 << endl;

k = k - c2 \* 5; //减去用5元支付过的额度

c1 = k / 2; //计算2元所需张数

cout << "2元纸币需要：" << c1<< endl;

k = k - c1 \* 2; //减去用2元支付过的额度

c0 = k / 1; //计算1元所需张数

cout << "1元纸币需要：" << c0 << endl<<endl;

n = c0 + c1 + c2 + c3 + c4 + c5 + c6;

cout << "所需钱币总张数：" << n << endl<<endl;

}

int main()

{

int K;

while (1)

{

cout << "请输入要支付的金额(整数)：";

cin >> K;

f(K);

}

return 0;

}

/\*

对该题的理解:

贪心算法：局部最优解

贪心算法的主要思想就是通过分解来解决当前问题，并得到当前问题的最优解。

对于上述问题，要支付K元，需要各种纸币数，但每种纸币数的数量是不定的，例如：我要支付10元，可以选择一张10元，或者两张5元，或者一张5元、两张2元、一张1元，等等。这就有很多种解。而条件中有一条：要求使用最少纸币数。这就给我们提供了一种选择：要使纸币数最少，就要尽量选择大额度的纸币。这样，就能一步一步得到我们最需要的到了解。

贪心算法，就是通过一个条件得到一种选择方法，再通过这种选择方法，一步一步去选择，每选择一次，将选出来后剩下的继续选择，最终得到需要的答案。如上述题，通过条件，我们得到的选择方法就是：尽量选大额度的纸币。

\*/

/\*

对于函数的理解：

我自己认为，函数就像我们做件事，一个函数对应一件事，而每条语句就是要完成事情的每一个步骤，因为对代码运行的顺序是从上到下的，而语句的摆放顺序就是我们要做一件事，首先该干什么，然后干什么，之后干什么。举个最简单的例子：将大象放进冰箱。将大象放进冰箱就是要做的事，即函数。首先打开冰箱，即第一条语句。然后放大象进去，即第二条语句。关上冰箱，即第三条语句。顺序完成这三条语句，则函数也就完成，我们的事也就完成。

学长说：每一条语句都是一个函数。

就如同将大象放进冰箱这一个函数的第一条语句，打开冰箱，我们要如何打开冰箱呢？我们可以把打开冰箱看成一件事，即函数，然后为了完成这件事，我们要进行一系列有顺序的步骤，如，将手放到冰箱的握把上，用力拉冰箱，最终打开冰箱。那我们要如何把手放到握把上呢？要怎样用力呢？这样，每一条语句都是一个函数，就不难理解了。

\*/



/\*

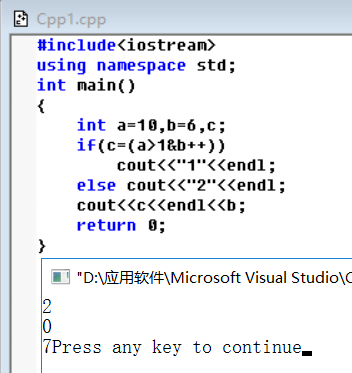
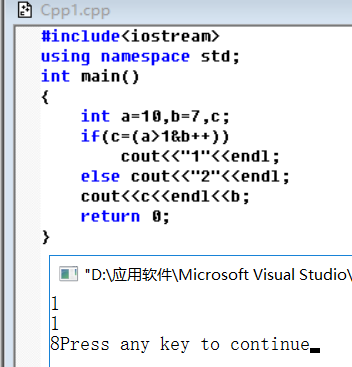
&和&&的区别：

我查百度解释是：

&和&&都可以做逻辑与算符，即and。必须两边的运算结果都为true时，该式子的运算结果才为true，否则为false。而用&作为逻辑运算符时，左右两边不管结果如何，都要进行计算。其中&还可以做位与运算符。

而我自己理解：

出现一个&时是位与运算。

图（1）图（2）

我查了bool类型在计算机里的存储方式，有一个解释让我认同，true存储的值是1，false存储的值是0，而对于非0的值看成ture，0看成false。

如上图，我只改变了b的值，但结果就不一样了。

我自己的解释：

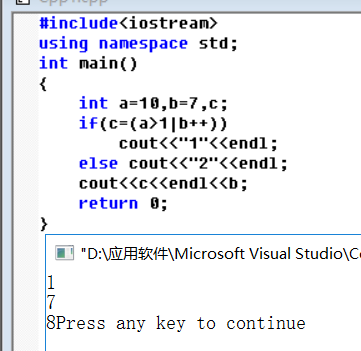
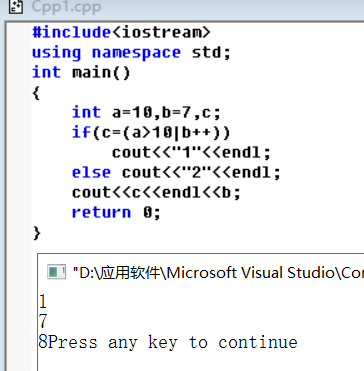
如图（1），首先关系运算符>和自增运算符++的优先级高于位运算符&，所以先执行a>1和b++，因为a=10，b=6，所以，a>1的结果是true即1，而对于b++,先执行a>1&b，再b加一。而a>1&b => 1&b,因为1的二进制数是1，而6的二进制数是110，将二进制数1和110进行位运算得到的是0，即false，所以得到如图（1）结果。

对于图（2），因为我将b改成7，而7的二进制数是111，将1和111进行位与运算得到1，即true，所以会有如图（2）结果。

\*/

/\*

|和||的区别：

图（1）图（2）

对于|，和我对&的理解一样。

如图（1），因为a>1为真，即ture，即1，1的二进制数为1，7的二进制数为111，所以，对二进制数1和二进制数111进行位或运算得到二进制数111，即7，即true。所以有如图（１）结果。

而图（２），因为a>10为假，即false，即0，0的二进制数为0，7的二进制数为111，对二进制数0和二进制数111进行位或运算得到二进制数111，即7，即ture，所以有如图（2）结果。

\*/

/\*

引用和指针的区别：

物理上，指针是消耗一个单位的存储空间去存储另外一个变量的物理地址，若要间接访问该变量，需要通过寻找指针变量所存储的地址来访问该变量。而引用，是给该变量取一个新的名字，就像小名一样，叫大名（该变量的原名）或者小名（该变量的引用名）都是叫它，即都是访问该变量。

个人认为，如果只是单纯传参等等，不需要改变指向，用引用更加简单、便捷，而且不会浪费存储空间。而若像链表之类的要总改变指向，那用指针最好。

但从本质来讲，引用也是指针的一种。引用是C++对C指针的一种简化，能在不接触更深层次的指针应用时，使用引用，就能完成很多需要指针来完成的相对简单的工作。（毕竟指针刚接触时会很难懂，也容易出错）

\*/