第二次课

比较运算符

在第一节课中，我们已经学习过了算术运算符，在这节课我们学习了比较运算符，C++语言常用的比较运算符有

```
> 大于
< 小于
>= 大于等于
<= 小于等于
== 等于
!= 不等于
```

表达式的值

对于一个表达式来说，如果是一个算术式，那么我们则可以根据这个算术式来计算出结果，例如$2+3$就是一个算术式，我们可以口算出结果为5，但是计算机中还存在另外一种**逻辑式**，逻辑式的结果只有$true$，$false$，也就是真和假。

例如$2+3>4$这个逻辑式的结果就是$true$的，在计算机中，一般也可以理解为$true$就是整数1。

$4+5>3+2$这就是由两个算术式组合而成的一个逻辑式，对于这两个算术式来说，他们的值分别为9和5，对于整个逻辑式来说，$9>5$所以结果为$true$，也就为1。

条件结构

在C++语言中，条件结构只要勇于根据特定条件来执行不同的代码块。

这种结构使得程序能根据运行时的一些不同情况做出不同的决策。

if语句

```
if (条件) {
    执行条件成立的代码
}
```

```
if (a > b) {
    cout << a;
}
```

以上这个程序可以在$a>b$时输出a。程序会根据运行时的不同情况执行if内的代码。

if-else 语句

```
if (条件) {
    执行条件成立的代码
} else {
    执行条件不成立的代码
}
```

```
if (a > b) {
    cout << a;
} else {
    cout << b;
}
```

if 更关心的是条件成立应该怎么去执行，而使用$if-else$结构则是我们关心条件成立和不成立两种情况的时候，应该分别去怎么做，如果成立就执行if内的代码，如果不成立就执行else中的代码。

if-else-if语句

```
if (条件1) {
    // 条件1代码
} else if (条件2) {
    // 条件2代码
} else if (条件3){
    // 条件3代码
} else {
    // 条件4代码
}
```

$if-else-if$这个语句允许我们在一个有很多情况的条件里去摘选出一个条件，例如：桶里一共有红黄蓝绿四种颜色的球，如果是红色的球就是**条件1**，那如果对**条件1**取反面，则可能还剩下3种情况在这三种情况当中再次判断情况2成立的条件，也就是如果是黄球的话，这个时候就需要用到$if-else-if$结构。

嵌套和缩进

由于if体内的代码不再是顺着代码的顺序执行，是有条件执行的，代码的逻辑级需要在这断开，那么，我们的代码体现就需要在这个地方向前缩进一级。从体现上来看，代码向前移动了4个空格。

按下键盘上的Tab键来进行缩进

代码缩进的优点：

1. 使得代码简洁易懂、更易维护修改
2. 使得代码逻辑结构更加清晰严谨
3. 避免代码产生语法错误，如括号匹配等
4. 写出优雅美丽的格式化代码，令代码成为艺术

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
    int n, mn = INT_MAX, mx = 0, m;
    cin >> n; m = 2 * n;
    vector<int> stones(n), sum(m + 1);
    vector<vector<int> > f(m, vector<int>(m)), g(m, vector<int>(m));
    for (int i = 0; i < n; i++) {
        cin >> stones[i];
    }
    stones.insert(stones.end(), stones.begin(), stones.end() - 1);
    for (int i = 0; i < m; i++) {
        sum[i + 1] = sum[i] + stones[i];
    }
    for (int len = 2; len <= n; len++) {
        for (int i = 0; i < m - len; i++) {
            int j = i + len - 1, val = sum[j + 1] - sum[i];
            f[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + val);
                g[i][j] = max(g[i][j], g[i][k] + g[k + 1][j] + val);
            }
        }
    }

    for (int i = 0; i < n; i++) {
        mn = min(mn, f[i][i + n - 1]);
        mx = max(mx, g[i][i + n - 1]);
    }
    cout << mn << endl << mx << endl;
    system("pause");
    return 0;
```

```cpp
#include <bits/stdc++.h>
using namespace std;
int main() {
int n, mn = INT_MAX, mx = 0, m;
cin >> n; m = 2 * n;
vector<int> stones(n), sum(m + 1);
vector<vector<int> > f(m, vector<int>(m)), g(m, vector<int>(m));
for (int i = 0; i < n; i++) {
cin >> stones[i];
}
stones.insert(stones.end(), stones.begin(), stones.end() - 1);
for (int i = 0; i < m; i++) {
sum[i + 1] = sum[i] + stones[i];
}
for (int len = 2; len <= n; len++) {
for (int i = 0; i < m - len; i++) {
int j = i + len - 1, val = sum[j + 1] - sum[i];
f[i][j] = INT_MAX;
for (int k = i; k < j; k++) {
f[i][j] = min(f[i][j], f[i][k] + f[k + 1][j] + val);
g[i][j] = max(g[i][j], g[i][k] + g[k + 1][j] + val);
}
}
}
for (int i = 0; i < n; i++) {
mn = min(mn, f[i][i + n - 1]);
mx = max(mx, g[i][i + n - 1]);
}
cout << mn << endl << mx << endl;
system("pause");
return 0;
}
```

上图是一个有缩进的代码，而下图是没有缩进的代码，对于一个程序合理的缩进优势显而易见。

下图完全无法分辨代码的逻辑级，也不能确定括号的匹配。

逻辑运算符(重点)

有时，我们在判断条件的时候并不是一个条件的判断，而是需要有多个条件用逻辑符号关联在一起。例如下方的三句话：

如果有一个三角形是等腰三角形**并且**它有一个角等于90度，则这是一个等腰直角三角形

如果我带了手机**或者**10元钱，则我可以买一瓶可乐。

不是所有牛奶都叫特仑苏

我们把这三种情况，并且，或者，不是用数学语言来形容就是：**与、或、非**

对应着计算机符号:

```
与 &&
或 ||
非 !
```

如果 a + b 大于 c **并且** a * b 小于等于 d成立，则执行代码1;

```
if (a + b > c && a * b <= d) {
    //执行1
}
```

逻辑与在两边都为真时结果为真，所以有

```
0 && 0 = 0
0 && 1 = 0
1 && 0 = 0
1 && 1 = 1
```

逻辑或在两边有一个为真时结果为真，所以有

```
0 || 0 = 0
0 || 1 = 1
1 || 0 = 1
1 || 1 = 1
```

逻辑非把原本为真的取反为假，原本为假的取反为真

```
!1 = 0
!0 = 1
```

在C++语言中，有一个非常重要的概念叫做：非0即为真。

所以如果代码这么写:

```
if (2 - 3) {
    cout << "yes";
} else cout << "no";
```

程序会输出yes，因为if内的条件$2-3=-1!=0$，-1是非0的，所以结果为真，只要不是0，就认为条件是成立的。

变量的生命周期

在C++语言中，每一个变量都有它存活的生命周期。

在下方的代码中：

```cpp
#include <bits/stdc++.h>
using namespace std;
int a;
int main() {
    int b;
    cin >> a >> b;
    if (a > b) {
        int c = 5;
        cout << c;
    }
    cout << c;
    return 0;
}
```

这个程序会报错，因为**变量只能存活于创建它的那一对大小括号中。**

所以C这个变量在程序运行到了第11行的时候就已经死亡了，不被定义了，所以程序在输出c就会报错。

变量b在程序结束时死亡，唯一一个特殊的存在是变量a，因为它被定义在了全局区，全局变量贯穿整个程序。

定义在main函数外的变量被称为全局变量。

全局变量在创建的那一瞬间，就会带有初始值0，和创建在$main$函数内不同的是，全局变量的初值为0。而**创建在$main$函数以内的变量初始值是一个随机的值。**

自增减的简写形式

运算方式	表达式	等价表达式
自增1	$i++;++i;$	$i=i+1;$
自减1	$i--;--i;$	$i=i-1;$
自增m	$i+=m;$	$i=i+m;$
自减m	$i-=m;$	$i=i-m;$
自乘m	$i*=m;$	$i=i*m;$
自除m	$i/=m;$	$i=i/m;$
自模m	$i\%=m;$	$i=i\%m;$

其中，给一个变量自增自减1有两种形式，一个是前置++，一个是后置++。

在单独对一个变量进行自增自减时，前置++和后置++是没有任何区别的，但是当语句变得复杂时，前后置++就完全不同，例如：

```
int a = 5;
int b = a++;
cout << b;
```

```
int a = 5;
int b = ++a;
cout << b;
```

上方的代码输出5，下方的代码输出6，后置++相当于先使用变量a再对a自身做++，前置++相当于先对a自身做++，再使用变量a。