## 1.简单语句：

;//空语句，常用在循环后面，语法上需要而逻辑上不需要

ival + 5;//没什么用的语句

while(cin>>s){}//空块，作用等于空语句

## 2.语句作用域：

## 3.条件语句：

switch语句如果表达式和某个case标签的值匹配成功，就从该标签后的第一条语句开始执行。case关键字和它对应的值一起被称为case标签，case标签必须是整形常量表达式。要及时跳出case，否则会执行后面的语句。

switch (表达式){

case 值1 : 语句1

break;

case 值2 : 语句2

break;

...

default : 语句n

break;

}

写在一行里：

switch(ch)

{

case ‘a’:case ‘e’:case ‘i’:case ‘o’:case ‘u’:

++cnt;

break;//不用大花括号

}

default也是一种特殊的case标签。如果没有一个case能匹配上switch表达式的值，程序将执行default标签。

C++语言规定，不允许跨过变量的初始化语句直接跳转到该变量作用域内的另一个位置。如果需要为某个case分支定义并初始化一个变量，我们应该把变量定义在块内（只有这种情况用语句块），从而确保后面所有case标签都在变量的作用域之外。当不确定到底要迭代多少次、想在循环结束后访问循环控制变量时，使用while。

## 4.迭代语句：

for循环省略condition的效果等价于在条件部分写了一个true。

### 4.1范围for语句：

这种语句用来遍历容器或其他序列的所有元素。格式为：for(declaration:expression) statement。每次迭代都会重新定义循环控制变量，并将其初始化为序列中的下一个值，之后执行statement。

expression表示的是一个序列，可以是用花括号括起来的初始值列表、容器对象、数组等，这些类型的共同特点是拥有能返回 迭代器的begin和end序列。

declaration表示的是一个标量，序列中的每个元素都能转换成该变量的类型。

### 4.2do while语句：

不管条件的值如何，我们都至少执行一次循环。因为对于do while来说先执行语句或者块，后判断条件，所以不允许在条件部分定义变量。

do statement

while(condition);

## 5.跳转语句：

### 5.1continue语句：

只有当switch语句嵌套在迭代语句内部时，才能在switch里使用continue。对于范围for循环来说，则是用序列的下一个元素初始化循环控制变量。

### 5.2goto语句：

标签标示符独立于变量或其他标示符的名字，可以和程序中其他实体的标示符使用同一个名字而不会相互干扰。goto语句和控制权转向的那条带标签的语句必须位于同一个函数之内。

和switch语句类似，goto语句也不能将程序的控制权从变量的作用域之外转移到作用域之内。

goto end;int ix = 10;//错误，goto语句绕过了一个带初始化的变量定义

end:

ix = 42;

begin:

int sz = 1;

if(sc > 0) goto begin;//合法，系统将销毁该变量，重新创建

## 6.try语句块和异常处理：

### 6.1throw表达式：

用throw表达式引发一个异常，throw表达式包含关键字throw和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。

throw runtime\_error(“Data must refer to same ISBN.”);//该异常类型是runtime\_error类型对象，抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。

### 6.2try语句块：

有抛有抓，666666

try

//执行的代码，其中可能有异常。一旦发现异常，则立即跳到catch执行。

{

if (b == 0) throw runtime\_error("divisor is 0");

cout << static\_cast<double>(a) / b << endl;

cout << "Input two integers: ";

}

catch (runtime\_error err) //异常声明，对应的是异常类型

{

cout << err.what() << "\n"<< "Try again.\nInput two integers: ";

}

finally(){

//不管什么情况都会执行，包括try catch 里面用了return

}

对于try语句的嵌套，寻找处理代码的过程与函数调用链刚好相反。当异常被抛出时，首先搜索抛出该异常的函数。如果没找到匹配的catch子句，终止该函数，并在调用该函数的函数中继续寻找。如果还是没有找到匹配的catch子句，这个新的函数也被终止，继续搜索调用它的函数。

如果最终没有找到匹配的catch子句，程序转到名为terminate的标准库函数，该函数的行为与系统有关，会导致程序非正常退出。对于那些没有try语句定义的异常，也按照类似的方式处理。

编写异常安全的代码非常困难。异常中断了程序的正常流程，一场发生时，调用者请求的一部分计算可能已经完成了，另一部分则尚未完成。那些在异常发生期间正确执行了“清理”工作的程序被称作异常安全的代码。一些程序异常发生时能简单地终止程序，这类不需要担心异常安全问题。但是对于那些要处理异常并继续执行的程序，就要加倍注意了。

### 6.3标准异常：

一组类用于报告标准库函数遇到的问题。只定义了几种预算，包括创建或拷贝异常类型的对象，以及为异常类型的对象赋值。我们只能以默认初始化的方式初始化exception、bad\_alloc、bad\_cast对象，不允许为这些对象赋初值；其他的异常类型的行为恰好相反：应该使用string对象或者C风格字符串初始化这些类型的对象。

异常类型只定义了一个名为what的成员函数，该函数返回值是const \*char。

①excepion头文件：之报告异常的发生

②stdexcept头文件：定义了几种常用的异常类

③new头文件：定义了bad\_alloc异常类型

④type\_info头文件：定义了bad\_cast异常类型

|  |  |
| --- | --- |
| exception | 最常见的问题 |
| runtime\_error | 只有在运行时才能检测出的问题 |
| range\_error | 运行时错误：生成的结果超出了有意义的值域范围 |
| overflow\_error | 运行时错误：计算上溢 |
| underflow\_errer | 运行时错误：计算下溢 |
| logic\_error | 程序逻辑错误 |
| domain\_error | 逻辑错误：参数对应的结果值不存在 |
| invalid\_error | 逻辑错误：无效参数 |
| length\_error | 逻辑错误：试图创建一个超出该类型最大长度的对象 |
| out\_of\_range | 逻辑错误：使用一个超出有效值范围的值 |