# Chapter 5 语句

## 5.1 简单语句

1. 表达式语句的作用是执行表达式并丢弃掉求值的结果。

```
ival + 5; // 一条没什么实际用处的表达式语句
cout << ival; // 一条有用的表达式语句
```

2. 空语句（null statement）只含有一个单独的分号

```
while(cin >> s && s != sought){
    ; // 空语句，只是读取输入流，什么也不做
}
// 使用空语句是一般需要加上注释，以表示该句有意省略
```

3. 一般来说多余的空语句是无害的，但是如果在if或者while的条件后面跟了一个额外的分号就可能完全改变了程序的初衷。

```
while(iter != svec.end()) ; // 此时循环的是一个空语句，导致循环无限下去
    ++iter; // 虽然iter有了缩进，但是它仍然不在while的作用域内
```

4. 复合语句（compound statement）是指用花括号括起来的语句和声明的序列。复合语句也被称作块（block）。一个块就是一个作用域，在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。**通常，名字在有限的区域内可见，该区域从名字定义处开始，到名字所在的（最内层）块的结尾为止。**

5. while或for的循环体必须只能包含一条语句，此时我们使用花括号括起来，就可以把语句序列转换成块。此时就可以在块中放入多条语句成为循环的语句。

```
while(val <= 10){
    sum += val;
    ++val;
}

while(val <= 10){

} // 空块，等价于空语句
```

## 5.2 语句作用域

1. 可以在if、switch、while和for语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出作用范围了。

```
while(int i = get_num())
```

```
    cout << i <<endl;
i = 0; // 错误，i在循环外部无法访问，i在块外失去作用域
```

2. 如果其他代码也要访问控制变量，则变量必须定义在语句的外部。
3. **因为控制结构定义的对象的值马上要由结构本身使用，所以这些变量必须初始化。**

## 5.3 条件语句

1. **为了避免代码混乱不清，以后修改代码时添加语句很容易找到正确的位置，最好在if或者else以及for和while的后面都跟上花括号**
2. 悬垂else（dangling else）：if分支会多于else分支，这个时候某个给定的else应该和最近的尚未匹配的if匹配，从而消除程序的二义性：

```
if(grade % 10 >= 3)
    if(grade % 10 > 7)
        lettergrade += '+';

else
    lettergrade += '-';
    // 虽然此时缩进看似乎else和第一个if匹配
    // 然而实际上是匹配的最近的那个if（第二个）
```

3. 如果想避免上述的歧义，可以使用花括号来避免这种情况。
4. 如果switch语句的表达式所有的case都没有匹配上，将直接跳转到switch结构之后的第一条语句。
5. case关键字和他对应的值一起被称为case标签（case label）。case标签必须是整型常量表达式：

```
char ch = getVal();
int ival = 42;
switch(ch){
    case 3.15: // 错误，case标签不是一个整数
    case ival: // 错误，case标签不是一个常量
}
```

6. break语句将控制权转移到switch语句外面。
7. 如果某个case分支被匹配上，那么将执行该标签后的所有分支，要想避免执行之后的所有分支，必须在下一个case之前使用break语句来中断。然而，有时候我们会故意省掉break语句，使程序能够连续执行若干个case标签。
8. **一般不要省略case之后的break，如果没写break最好加上注释说清楚程序的逻辑。**
9. 任何两个case标签的值不能相同，否则就会引发错误。另外，default也是一种特殊的case标签。如果没有任何一个case标签可以匹配上switch表达式的值，程序将执行紧跟在default标签后面的语句。

```
switch(ch){
    case 1: case 2: case 3: //故意省略break，
                            // 此时所有符合这些case的语句都会执行++count
        ++count;
        break;
    default:
        ++countNone;
        break;
}
```

10. 标签不应该孤零零的出现，它后面必须跟上一条语句或者另外一个case标签。如果switch结构以一个空的default标签作为结束，则该default标签背后必须跟上一条空语句或者空块。

```
    int b = 4;
    switch(b){
        case 3:
            cout << "b is 3" << endl;
            break;
        default: ; // default 以空语句结尾
    }
```

11. swtich的执行过程有可能会跳过某些case标签，也就导致作用域内的变量定义被直接跳过了，c++规定，如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法行为。

```
case true:
    string file_name; // 错误，控制流绕过一个隐式初始化的变量
    int ival = 0; // 错误，控制流绕过一个显式初始化的变量
    int jval; // 正确，jval没有初始化

case false:
    jval = next_num(); // 正确：给jval赋值
    if(file_name.empty()){ // file_name在作用域内，但是没有被初始化
        // code_block
    }
```

12. 上述代码形式容易产生误解或错误，所以我们应当尽量将变量的声明和定义放在花括号里（块），这样就能确保所有case标签都在变量的作用域之外。

```
case true:
{
    string file_name = get_file_name();
}
break;

case false:
    if(file_name.empty()) // 错误：file_name不在作用域内
```

## 5.4 迭代语句

1. while和for语句在执行循环体之前检查条件，do while语句先执行循环体然后再检查条件。

2. 定义在while条件部分或者while循环体内的变量每次都经历从创建到销毁的过程。

3. for语句结构：

```
for(initializer;condition;expression)
    statement
```

4. for语句头能省略掉所有init-statement、condition和expression中的任何一个。如果无须初始化，则我们可以使用一条空语句作为init-statement。

```
auto beg = v.begin();
for(;beg != v.end(); ++beg)
    // 什么也不做
```

5. 省略condition的效果相当于在条件部分写了一个true。因为条件的值永远是true，所以在循环体内必须有语句负责退出循环，否则循环就会无休止的执行下去。

6. **范围for语句中的expression必须是一个序列，比如用花括号括起来的初始值列表、数组或者vector或string等类型的对象，这些类型的共同特点是拥有能返回迭代器的begin和end成员。**

```
for(declaration: expression)
    statement
```

7. declaration定义了一个变量，序列中的每个元素都得能转换成该变量的类型，为了确保类型相容，最简单的办法是使用auto类型说明符让编译器帮助我们指定合适的类型。**如果要对序列中的元素执行写操作，循环变量必须声明成引用类型。**

8. 范围for语句的定义来源于与之等价的传统for语句,这也就是为什么不能通过for范围语句来增加vector的对象中的元素，**因为for语句中预存了end()的值，一旦添加或删减了元素就会导致end()失效**：

```
// 等价范围for循环的一般for循环语句
for(auto beg = v.begin(), end = v.end(); beg != end; ++beg){
    auto &r = *beg; // r必须是引用类型，才能对元素执行写操作
    r *= 2; // 将v中每个元素的值都翻倍
}
```

9. do-while语句和while的唯一区别是先执行循环体再检查条件。不管条件的值如何，我们都至少执行一次循环。

```
do
    statement
while(condition)
```

10. **切记，循环的条件不可以定义在do语句里面，必须在循环的外面定义,同时因为do while语句先执行do再执行while，所以也不允许在条件部分定义变量。**

```
do{
    number(foo);
}while(int foo = get_foo());
//错误。变量的声明放在了do的条件部分

do{
    int i = 0;
    cout << i << endl;
}while(i < 5) // 错误，条件变量定义在了do内部
```

## 5.5 跳转语句

1. c++提供了4种跳转语句，break、continue、goto和return。
2. break**负责终止离他最近的**while、do while、for或switch语句，并从这些语句之后的第一条语句开始继续执行。break语句只能出现在迭代语句或者switch语句内部，break语句的作用范围仅限于最近的循环或switch。
3. continue语句终止最近的循环中的当前迭代，并立即开始下一次迭代。continue只能出现在for、while和do while循环的内部。或者嵌套在此类循环里的语句或块的内部。只有当switch语句嵌套在迭代语句内部时，才能在switch里使用continue。

```
while(i < 10){
    switch(flag){
        case true:
            cout << "TRUE" << endl;
        case false:
            continue;
            //嵌套在while语句里的switch可以使用continue
    }
}
```

4. continue对于范围for循环来说会直接获取序列的下一个元素来初始化循环控制变量。
5. 不要在程序中使用goto语句，因为它使得程序既难以理解又难以修改。

## 5.6 try语句块和异常处理

1. 异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。在c++语言中，异常处理包括：
    - throw表达式（throw expression），异常检测部分使用throw表达式来表示它遇到了无法处理的问题。我们说throw引发了异常
    - try语句块，异常处理部分使用try语句块处理异常。try语句块以关键字try开始，并以一个或多个catch子句结束。try语句块中代码抛出的异常通常会被某个catch子句处理。因为catch子句处理异常，所以他们也被称作异常处理代码（exception handler）。

- 一套异常类（exception class），用于在throw表达式和相关的catch子句之间传递异常的具体信息

2. 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表达式后面通常跟一个分号，从而构成一条表达式语句。

```
if(item.isbn() != item2.isbn()){
    throw runtime_error("Data must refer to same ISBN");
}
```

3. runtime_error 是标准库异常类型的一种，定义在stdexcept头文件中。我们必须初始化runtime_error的对象，方式是给他提供一个string的对象或者一个c风格的字符串，这个字符串有一些关于异常的辅助信息。

4. try语句块当选中了某个catch子句处理异常之后，执行与之对应的块，**catch一旦完成，程序跳转到try语句块最后一个catch子句之后的那条语句继续执行。**

5. **try语句块内的变量在块外部无法访问，特别是在catch子句内也无法访问。**

6. 在嵌套的try语句里寻找对应catch的过程恰好与程序执行顺序相反。也即沿着程序的执行路径逐层回退，直到找到适当类型的catch子句为止。如果最终还是没能找到任何匹配catch子句，程序转到名为terminate的标准函数库。一般情况下执行该函数将导致程序非正常退出。（如果程序没写任何try-catch语块，遇到异常也将调用terminate函数库）

7. 标准异常：

   - exception头文件定义了最通用的异常类exception，他只报告异常的发生，不提供任何额外的信息。
   - stdexcept头文件定义了几种常用的异常类

| 类名 | 作用 |
| --- | --- |
| 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_error | 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_argument | 逻辑错误：无效参数 |
| length_error | 逻辑错误：试图创建一个超出该类最大长度的对象 |
| out_of_range | 逻辑错误：使用一个超出有效范围的值 |

   - new头文件定义了bad_alloc异常类型

- type_info头文件定义了bad_cast异常类型

8. 我们只能用默认初始化的方式初始化exception、bad_alloc和bad_cast对象，不允许为这些对象提供初始值。其他异常类型则恰好相反，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，且不允许使用默认初始化的方式。当创建此类对象时，必须提供初始值，该初始值含有错误相关的信息。

9. 异常类型之定义了一个名为what的成员函数，该函数没有任何参数。返回值是一个指向c风格的字符串的const char*。该字符串的目的是提供关于异常的一些文本信息。如果异常类型有一个字符串初始值，则what返回该字符串，而其他的类型的what函数返回的值由编译器决定。

10. 例题5.23

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

int main(int argc, const char** argv) {
    cout << "Please input two numbers:" << endl;
    int a, b;
    while(cin >> a >> b){
        try
        {
            if(b == 0) throw runtime_error("dividend cannot be 0");
            cout << "The result of dividing is "
                    << static_cast<double>(a) / b << endl;
        }
        catch(const runtime_error &e)
        {
            cerr << e.what() << '\n';
            cout << "Do you want to continue? \n0: continue \n1: finis
                    << endl;
            int choice;
            cin >> choice;
            if(choice){
                break;
            }else{
                cout << "Please input two numbers:" << endl;
            }
        }
    }

    return 0;
}
```