1.6作用域与存储类型

- 作用域和存储类型是程序设计中与空间、时间相关的两个重要概念。
- C++程序中的任何变量都有自己的作用域和存储类型。
- 作用域决定变量在程序哪个区域可用。
- 存储类型说明了变量在内存中存储的方式，它决定了变量的作用域和生存期。
- 变量的作用域是指变量可以被引用的区域。变量的作用域决定了变量的可见性，说明变量在程序哪个区域可用，即程序中哪些区域的语句可以使用变量。
- 作用域有三种：局部作用域、文件作用域和全局作用域。
- 具有局部作用域的变量称为局部变量，它们声明在函数（包括main）的内部，又称为内部变量。其作用域在函数结束时结束。
- 具有全局作用域和文件作用域的变量称为全局变单它们声明在函数的外部，其作用域在程序源文件结束处结束。

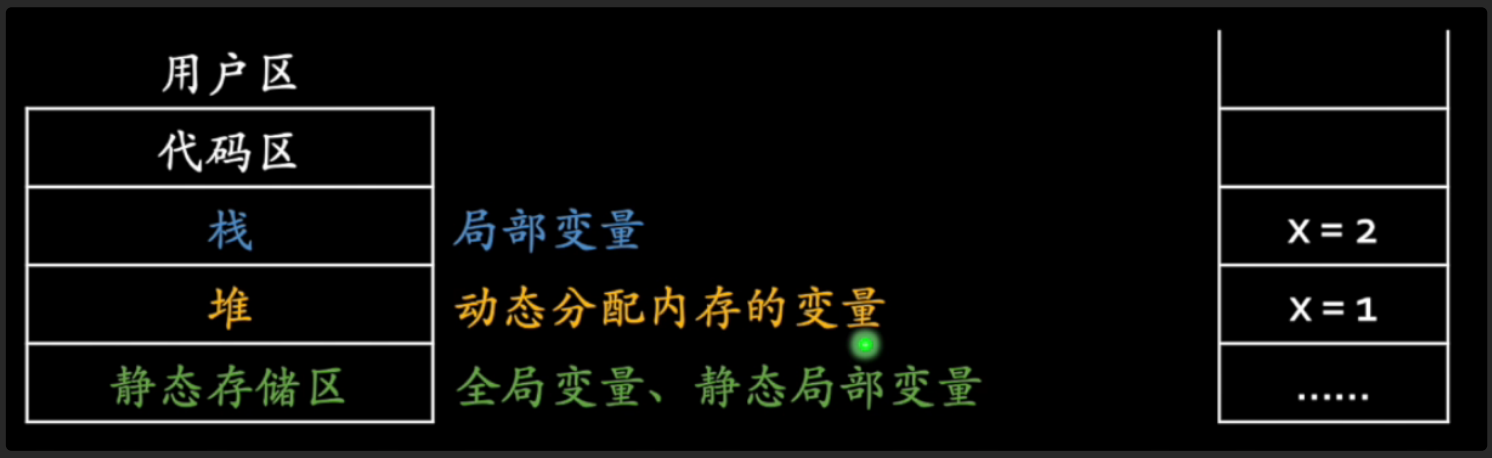
全局作用域与文件作用域的关系和区别 *

- 都属于全局变量，声明（定义）在函数的外部。
- 全局作用域范围最广，可作用于组成该程序的所有源文件。
- 当将多个独立编译的源文件链接成一个程序时，在某个文件中声明的全局作用域的变量（或函数）在其它相链接的文件中也可以使用它们，但使用前必须进行extern外部声明。
- 文件作用域的变量只在当前文件范围内可用，但不能被其它文件中的函数访问。要使变量或函数具有文件作用域，必须前加上static声明。
- 当将多个独立编译的源文件链接成一个程序时，可以利用static修饰符避免一个文件中的外部变量由于与其它文件中的变量同名而发生冲突。
- 变量的作用域是指一个范围，是变量在源程序中的一段静态区域，作用域是从代码空间的角度考虑问题。
- 变量的生存期是从时间的角度考虑问题，是指在程序执行的过程中一个变量从创建到被撤消的一段动态时间。当系统为变量分配内存空间后，变量即开始处于生存期，当变量所占用的内存空间被释放，这个变量即结束了生存期。
- 变量的生存期与作用域是密切相关的，一般变量只有在生存后才能可见（在作用域内）。
- 有些变量（如函数形参）没有生存期，但有作用域，而有时变量虽然在生存期，但却不在作用域内。

```
int x = 1; //全局变量x的作用域开始于此，结束于整个程序源文件
void eg2_27(int x)//形参x的作用域开始于此
{
    int y = 3; //局部变量y的作用域开始于此
    {
        int z = x + y;//块内局部变量乙的作用域开始于此
        //x和y在该语句块内可用
        //some code
    }//局部变量z的作用域结束
    int k;//局部变量k的作用域开始于此
    //some code
}//局部变量y、k和形参x的作用域结束
```

变量的内存分配方式

变量有3种内存分配方式：自动分配（运行时）静态分配（编译时）和动态分配（运行时）。




```
eg2_30()
{
    int x = 1;
    {
        int x(2), y(2);//变量"X=1"被屏蔽
        cout << "x =  " << x << endl;
        eg2_27(x);//变量x和y在调用函数时失去作用域
    }
}
```

变量的存储类型

- 变量的存储类型有以下四种：auto、register、extern和static。在声明变量时可以指定变量的存储类型，其一般形式为：

```
<存储类型><数据类型><变量名列表>;
```

  - auto和register用于声明内部变量，auto变量存储在栈中，register变量存储寄存器中（register建议不用）。extern用于声明外部变量，static用于声明内部变量或
  - 外部变量，extern变量和static变量是存储在静态存储区中。
  - 当声明变量时未指定存储类型，则内部变量的存储类型默认为auto类型，外部变量的存储类型默认为extern类型。

```cpp
// eg2_31.cpp
int b = 5; //等同于 extern int b = 5;
```

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

int main(int argc, const char * argv[]) {
    extern int b;
    b++;
    cout << "b = " << b << endl;
    return 0;
}
```

```
b = 6
Program ended with exit code: 0
```

建议：不用

例子：静态变量

```cpp
void fun() {
    int a = 0;
    a++;
    cout << "a = " << a << endl;
}
```

```
a = 1
a = 1
```

```cpp
void fun() {
    static int a = 0;
    a++;
    cout << "a = " << a << endl;
}
```

```
a = 1
a = 2
```

```cpp
void eg2_32() {
    for(int i=0; i<2; i++)
        fun();
}
```

```cpp
int amount = 123; //全局变量
void eg2_33()
{
    int amount = 456; //局部变量
    cout << ::amount << ',';//输出全局变量，::为作用域限定符
    cout << amount << endl;//输出局部变量

    ::amount = 789; //访问全局变量
    cout << ::amount << ',';//输出全局变量
    cout << amount << endl;//输出局部变量
}
int amount = 456; //语句块外的局部变量
{
    int amount = 456; //语句块内的局部变量
    ::amount = 789; //报错：找不到全局变量amount
}
```

命名空间

命名空间可避免全局标识符同名引起冲突是对一些成员（标识符）进行声明的一个描述性区域。命名空间中的成员主要是在空间外被使用，可以使用using语句来简化。

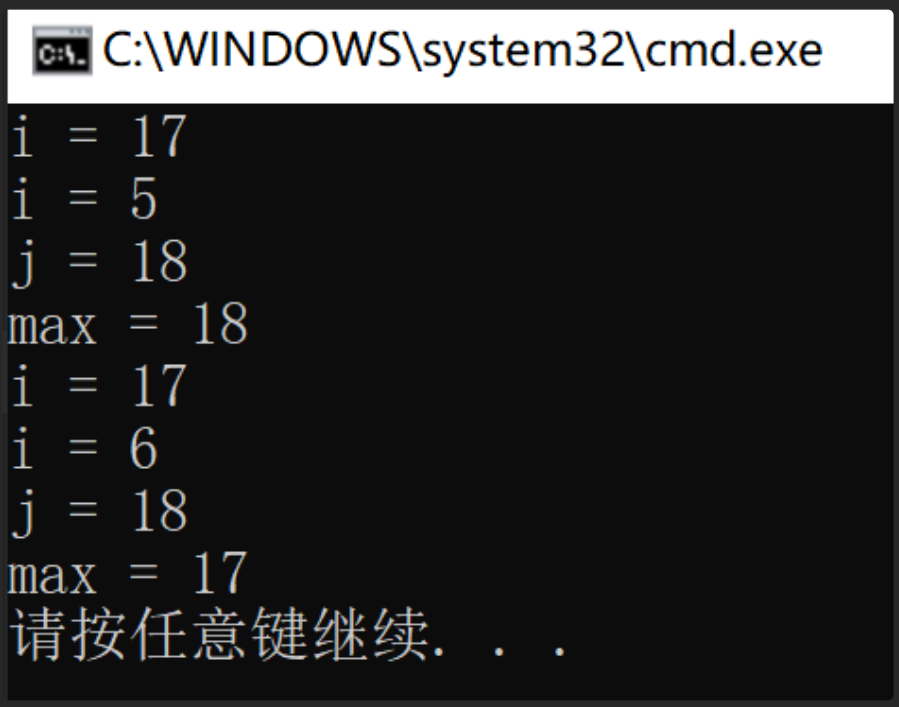
在命名空间出现之前，整个C++库是定义在唯一的全局命名空间中。

引入命名空间后，标准C++库都定义在std中。

标准C++库的头文件一般没有h后缀。

若不使用语句"usingnamespace std；"，使用标准C++库，必须加上std命名空间限定，层Bpstd：：cin,std：：cout。

```cpp
#include<bits/stdc++.h>
using namespace std;
int i = 5; //在全局命名空间中的全局变量
namespace NS
{
    int i, j; //命名空间NS中的全局变量
    int max(int x, int y) {return x > y ? x : y;}
}
void eg2_34()
{
    NS::i = 16; NS::j = 17; //为命名空间NS5中的全局变量赋值
    {// 语句块{}
        using namespace NS; //在当前语句块中开放命名空间NS
        j++;
        NS::i++;
        cout << "i = " << NS::i << endl;
        cout << "i = " << ::i << endl;
        cout << "j = " << j << endl;
        cout << "max = " << max(j, NS::i) << endl;
    }
    i++;
    cout << "i = " << NS::i << endl;
    cout << "i = " << i << endl;
    cout << "j = " << NS::j << endl;
    cout << "max = " << NS::max(i, NS::i) << endl;
}
int main()
{
    eg2_34();
    return 0;
}
```



函数的存储类型

- 与变量类似，函数也具有存储类型，包括：
  - 内部函数（static），只能被同一个源文件中的函数调用。
  - 外部函数（extern，默认值），可以被其它源文件中的函数调用。
- 调用外部函数前必须先进行外部函数声明。
- 例如：在源文件File1.cpp中定义函数YourFun()，而在源文件File2.cpp中调用函数YourFun()。

例子：函数的存储类型

```cpp
//File1.cpp
int YourFun(int x, int y) {
    return x + y;
}
```

```cpp
//File2.cpp（与File1.cpp在同一目录下）
#include <iostream>
using namespace std;

extern int YourFun(int, int);
int main(int argc, const char * argv[]) {
    int sum, a = 20, b = 30;
    sum = YourFun(a, b);
    cout << "sum = " << sum << endl;
    return 0;
}
```

建议：不用

例子：建议用法

```hpp
//File1.hpp
int YourFun(int x, int y);
```

```cpp
//File1.cpp
#include "File1.hpp"

int YourFun(int x, int y) {
    return x + y;
}
```

```cpp
//File2.cpp
#include "File1.hpp"
#include <iostream>
using namespace std;

int main(int argc, const char * argv[]) {
    int sum, a = 20, b = 30;
    sum = YourFun(a, b);
    cout << "sum = " << sum << endl;
    return 0;
}
```

动态内存分配

程序中声明的变量，其所占内存空间不需要程序员管理，编译器在编译阶段就自动将管理这些写空修间的代码加入到目标文件中。程序运行后由系统自动为变量分配内存空间，在作用域结束后自动释放内存空间。

有时，程序只能在运行时才能确定需要多少内存空间来存储数据，这时程序员就需要采用动态内存分配的方法设计程序。

动态内存分配是指在程序运行时为程序中的变量分配内存空间，它完全由应用程序自己进行内存的分配和释放。动态内存分配是在一些被称为堆的内存块中为变量分配内存空间。

内存的动态分配与释放

- C语言，动态内存分配是通过调用标准库函数mallocO和free0实现的。
- C++语言，利用new和delete运算符进行动态内存的分配和释放，该方法能够检测内存漏洞。
- 动态分配内存

```
<指针变量>=new<数据类型>;
<指针变量>=new<数据类型>[<整型表达式>];
<指针变量>=new<数据类型>(<初始值>)
```

- 内存空间的释放

```
delete<指针变量>;
delete[]指针变量>://1释放动态数组
```

例子：内存的动态分配与释放

- 编程输出Fibonacci数列

```
#include<bits/stdc++.h>
```

```cpp
using namespace std;

void eg2_36()
{
    int *f, n;
    cout << "Please input n = ";
    cin >> n;
    f = new int[n+1]; //为动态数组f分配n+1个内存单元
    if(f == nullptr || n <1 )
    {
        cout << "Error!" << endl;
        return;
    }
    f[0] = f[1] = 1;
    cout << f[0] << endl << f[1] << endl;
    for(int i = 2; i <= n; i++)
    {
        f[i] = f[i-2] + f[i-1];
        cout << f[i] << endl;
    }
    delete []f; //释放动态数组f的内存空间
}
int main()
{
    eg2_36();
    return 0;
}
```