在main函数之前运行

2021年7月26日 星期一 22:33

In the following example, the static objects `sd1` and `sd2` are created and initialized before entry to `main`. After this program terminates using the `return` statement, first `sd2` is destroyed and then `sd1`. The destructor for the `ShowData` class closes the files associated with these static objects.

C++ Copy

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
public:
   // Constructor opens a file.
   ShowData( const char *szDev ) {
   errno_t err;
      err = fopen_s(&OutputDev, szDev, "w" );
   }

   // Destructor closes the file.
   ~ShowData() { fclose( OutputDev ); }

   // Disp function shows a string on the output device.
   void Disp( char *szData ) {
      fputs( szData, OutputDev );
   }
private:
   FILE *OutputDev;
};

//  Define a static object of type ShowData. The output device
//   selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

//  Define another static object of type ShowData. The output
//   is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
   sd1.Disp( "hello to default device\n" );
   sd2.Disp( "hello to file hello.dat\n" );
}
```

左值和右值

2021年7月26日 星期一 22:42

1. 定义

lvalue	左值	有可访问的内存地址	变量名，const变量，数组元素，返回左值引用的函数调用，类成员
prvalue	纯右值	没有可访问的内存地址	字面值常量，返回非引用类型的函数调用，在表达式求值期间创建但是只能被编译器访问的临时对象
xvalue	将亡值	有地址但不再会被访问，可以初始化右值引用	返回右值引用的函数调用，数组下标

rvalue = prvalue + xvalue

NOTE：字面值中的字符串常量是左值

```
                (const char [4])"abc"
std::cout << &("abc") << std::endl;
```

前自增是左值表达式

后自增是右值表达式

```
std::cout << &(++a) << std::endl;

                expression must be an lvalue or a function designator C/C++(158)
                View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
std::cout << &(a++) << std::endl;
```

2. 非const左值引用不能被赋予右值

```
          initial value of reference to non-const must be an lvalue C/C++(461)
          View Problem (Alt+F8)   Quick Fix... (Ctrl+.)
int &a = 2; // 非const左值引用不能被赋予右值
const int &a = 2;
```

3. 左值引用

可以视为对象的另一个名称，必须被初始化且不能修改

4. 右值引用，移动语义move，完美转发forward

move语义可以实现将资源从一个对象转移到另一个对象

move语义之所以有效是因为它允许资源从临时对象转移，这个临时对象不再会被其他地方引用

举个move优势的例子：vector插入元素，如果超出容量了，需要重新分配内存空间，然后将原有元素复制过来，销毁原来的元素，再插入新元素，使用move之后则能够直接移动原来的元素

完美转发：减少了对重载函数的需求，并有助于避免转发问题

转发问题：

```
struct W{
    W(int &, int &) {}
};

struct X{
    X(const int &, int &) {}
};

struct Y{
    Y(int &, const int &) {}
};

struct Z{
    Z(const int &, const int &) {}
};

template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2){
   return new T(a1, a2);
}

int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);

    Z *pz = factory<Z>(2, 2); //需要左值引用作为参数，但是这里传递了右值
}
```

上述一个模板函数不能完全适配四个类，可以使用std::forward解决而不需要再定义新的factory重载版本

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
   return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

extern

2021年7月27日 星期二 21:21

1. 指定符号具有外部链接

2. 根据上下文有四种含义

a. 非const全局变量（或函数）声明，它指示这个变量或函数在另一个翻译单元中定义，extern此时必须用于除定义变量的文件之外的所有文件

```C++
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
extern int i;  // declaration only. same as i in FileA

//fileC.cpp
extern int i;  // declaration only. same as i in FileA

//fileD.cpp
int i = 43; // LNK2005! 'i' already has a definition.
extern int i = 43; // same error (extern is ignored on definitions)
```

b. 修饰const变量，它指示这个变量具有外部链接，extern必须应用于所有文件中的所有声明

```C++
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i;  // declaration only. same as i in FileA
```

c. extern "C"指示函数在其他地方定义并且使用C语言的调用规则

d. 在模板声明中，extern指示模板已在其他地方实例化

union

2021年7月27日 星期二 22:29

1. Union

union内所有成员共享同一块内存空间，也就是说某个时刻union只能持有成员列表中的一个对象，分配空间时按照最大的成员分配

函数重载

2021年7月27日 星期二 22:44

1. 根据参数类型和数量为同名函数提供不同的语义

2. 重定义而非重载的

max(int, int)

max(int &, int &)

max(const int, const int)

max(volatile int, volatile int)

是重定义而非重载

但是**使用const或vlolatile修饰的引用可以用于函数重载，被视为与其基类型引用不同**，即

max(int &, int &)

max(const int &, const int &)

是重载