Module03-07
C++ 语言基础：源文件与程序

语言基础 - 源文件与程序

- 源文件与程序
  - C++ 源文件的后缀名
  - C++ 程序的编译
  - 头文件
  - 连接
  - 关于程序

源文件与程序 - C++ 源文件后缀名

- 实现文件
  - 一般是： .c 、 .cpp 、 .cc 、 .C 、 cxx 、 ...
- 头文件
  - 一般是： .h 、 .hpp 、 .hxx 、 或无后缀名（如标准库头文件）、 ...
- 一些说明
  - 各种编译器对于文件的后缀名要求不尽相同，但通常，.h 、 .hpp 、 .cpp 、 .cc 后缀名，多数编译器都支持。

源文件与程序 -C++ 程序的编译

- 编译的主要阶段：
  - 预处理 (Preprocess)
    - 将要包含的头文件内容读入实现文件（如 .cpp 文件等）
    - 将所有的宏（ macro ）展开
    - 经过上述过程的处理，得到一个符合编译器要求的编译单元
  - 编译 (Compile)
    - 将预处理产生的编译单元编译成目标代码（ Object File ）
    - 实际上，视编译器（ C++ 实现）的不同，编译过程也可以分成若干步骤，如先产生汇编代码，再产生目标代码等
  - 连接 (Linkage)
    - 将编译阶段产生的多个目标文件组合在一起
    - 如果需要引用库：对于引用静态库而言，在连接阶段从库中加入所需的代码；对于引用共享库而言，只是产生连接符号（在运行期加入所需的代码）

源文件与程序 - 头文件

- 关于头文件
  - 为达到接口 (interfaces) 与实现 (implementations) 分离的目的，一般在头文件中定义接口
  - 从编译和连接方面而言，为保证在不同的编译单元中定义的一致性，可以采用将头文件通过 #include 指令包含到各实现文件（如 .cpp 文件等）的方式

源文件与程序 - 头文件

- 两种不同的 #include 形式
  - `#include <to_be_include>`
    - 一般用于包含 C++ 实现定义的头文件搜索路径中的头文件，如标准库、系统库、一些第三方库的头文件（对于 Linux GCC 而言，路径一般为：/usr/include ）
  - `#include "to_be_include"`
    - 一般用于包含当前目录下的头文件、或在编译选项中 -I 选项所指定的头文件路径中文件

源文件与程序 - 头文件

- 两种 #include 指令的区别：
  - #include <...> 方式只在指定的头文件搜索路径中查找头文件，如编译器定义的头文件路径、编译时 -I 选项指定的头文件路径
  - #include "..." 方式默认最先在当前目录下查找，然后在指定的头文件搜索路径中查找文件，如编译器定义的头文件路径、编译时 -I 选项指定的头文件路径

源文件与程序 - 头文件

- 头文件中放什么
  - 一般而言，建议在头文件中只放如下内容：

具名的名字空间	`namespace Dummy { /*...* / }`
类型的定义	`class E { /* ... */ };`
模板声明	`template<typename T> class A;`
模板定义	`template<typename T> class A { /* ... */ };`
函数声明	`void mySort(int a[], const int& len);`
内联函数定义	`inline bool isOdd(const int& n) { return n % 2; }`
数据声明	`extern int size;`
常量定义	`const int BUF_SIZE = 512;`
枚举	`enmu Color { RED, GREEN, BLUE };`
名字声明	`class Matrix;`
包含指令	`#include <map>`
宏定义	`#define VER 12`
条件编译指令	`#infdef __cplusplus`
注释	`/* ... */  // ...`

源文件与程序 - 头文件

- 头文件中不放什么
  - 头文件中绝不应该有：

常规的函数定义	`char get(char* p) { return *p++; }`
数据定义	`int a;`
容器对象定义	`double da[28];  vector<Point> points(120);`
无名名字空间	`namespace { /* ... */ }`
导出的模板	`export template<typename T> f(T t) { /* ... */ }`

源文件与程序 - 头文件

- 标准库头文件
  - C++ 标准库的头文件没有后缀名，如 <iostream>
  - 标准库中以 c 开头的头文件是共享 C 标准库的部分，如 <cmath>

源文件与程序 - 头文件

- 包含保护 (include guards)
  - 如果一个头文件被多个实现文件包含，其中定义的名字就会重复，出现不符合单一定义原则的现象
  - 可以使用包含保护的解决方案：

```
// Array.h
#ifndef ARRAY_H_
#define ARRAY_H_

class Array {
public:
    Array();
    virtual ~Array();
};

#endif /* ARRAY_H_ */
```

源文件与程序 - 连接

- 重提 " 单一定义原则 (ODR)"
  - 在一个程序中，一个类、枚举、模板等只能定义唯一的一次！
  - 同一个编译单元中不能对同一个类、枚举、模板等定义两次
  - 不同的编译单元中除符合下列条件外，也不允许对同一个类、枚举、模板等定义两次或多次：
    1. 它们出现在不同的编译单元
    2. 它们的定义按一个一个单词对应的相同
    3. 并且这些对应的单词代表的意义在各个编译单元中完全一致

源文件与程序 - 连接

- 重提 " 单一定义原则 (ODR)" （续）

```
// a.cpp
enum E1 { OK, CANCEL };
struct C1 { int a; int b; };
void f1(int*);

// b.cpp
enum E1 { OK, CANCEL };
struct C1 { int a; int b; };
void f1(int* p);
```

OK

```
// a.cpp
typedef int T;
enum E1 { OK, CANCEL };
struct C1 { T a; T b; };

// b.cpp
typedef short T;
enum E1 { OK, CANCEL };
struct C1 { T a; T b; };
```

偷梁换柱？

```
// a.cpp
enum E1 { OK, CANCEL };
struct C1 { int a; int b; };
void f1(int*);

// b.cpp
enum E1 { OK, CANCEL };
struct C1 { int a; int c; };
void f1(int* p);
```

b 和 c ？

```
// a.cpp
enum E1 { OK, CANCEL };
struct C1 { int a; int b; };

enum E1 { OK, CANCEL };
struct C1 { int a; int b; };
```

直接重复！

源代码与程序 - 连接

- 内部连接
  - 如果一个名字只能在其定义所在的编译单元内部使用，那么该名字是具有*内部连接*的，如：
    - inline 函数
    - const 对象
    - typedef
    - static
  - 关于内部链接的一些建议：
    - 最好将 const 和 inline 仅仅放在头文件中
    - 最好不要在函数、类定义以外使用 static 对象

源代码与程序 - 连接

- 外部连接（ extern ）
  - 如果一个名字可以在其定义所在的编译单元外使用，那么该名字是具有*外部连接*的，如：

```
// a.cpp
int i = 9;
double p = 0.618;

// b.cpp
int main() {
    extern int i;
    extern double p;
    cout << p << endl;  // OK, p in a.cpp
    cout << i << endl;  // OK, i in a.cpp
}
```

源代码与程序 - 连接

- 与非 C++ 代码连接
  - extern "C" 声明

```
extern "C" const char* md5(char*);
```

    - 上述声明表示函数 md5 将是与其它语言的代码（如 C 语言）的连接，而下面的 extern 声明，表示从其它的 C++ 编译单元（如 .cpp 文件）连接

```
extern const char* md5(char*);
```

    - extern "C" 中的 C 并不是指只能从 C 语言连接代码

源代码与程序 - 连接

- 与非 C++ 代码连接
  - extern "C" 语句块
    - 同单个 extern "C" 声明，只是将一系列的声明语句放在一起而已

```
extern "C" {
    const char* md5(char*);
    char* cat(const char*, const char*);
}
```

源代码与程序 - 程序

- 关于程序
  - 由连接器组合到一起的一组分开编译的编译单元
  - 在一个程序中，所有的函数、对象、类型必须只有唯一的定义
  - 一个程序中必须有且只能有一个 main() 函数

源代码与程序 - 程序

- 非局部对象的初始化
  - 原则上讲，在所有函数之外定义的对象（全局的、名字空间的、类的 static 成员变量），应该在 main() 函数调用前完成初始化
  - 如果没有显式的初始式，上述情况中的基本类型和枚举类型的变量的默认初始值是类型对应的 0 值

源代码与程序 - 程序

- 程序终止
  - 下面途径都可以终止程序：
    - 从 main() 函数返回（自然且推荐的方式）
    - 调用 exit() 函数
    - 调用 abort() 函数
    - 抛出的异常未被捕获
    - 其它各式各样的运行期错误
  - 关于 exit() 和 abort()
    - 调用 exit() 退出，仅保证已经构造好的静态对象正确析构，而调用它的函数中的局部对象的析构不会被执行（而 抛出异常且捕获它，则能保证局部对象能被正确析构）
    - 调用 abort() 则完全不管所有对象的析构，直接终止程序

源代码与程序 - 程序

- 程序终止
  - atexit() 函数 - 程序退出前做些事情

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

// atexit() 函数只接受无返回值、无参数的函数指针
void dummy() {
    cout << "Just a futile DEMO\n" << endl;
}

// b.cpp
int main() {
    cout << "---- begin ----\n";
    atexit(dummy);
    cout << "----- end -----\n";
}
```

源文件与程序 - Bjarne's Advices

- Bjarne's Advices
  - 使用头文件来表示接口和强调逻辑结构
  - 用 #include 指令将头文件包含到实现有关功能的源文件中
  - 避免在头文件中定义非 inline 函数
  - 只在全局或名字空间中使用 #include 指令
  - 只用 #include 包含完整的定义（不是在每个编译单元中做完整的定义）
  - 在头文件中使用 include guards
  - 用 #include 将 c 的头文件包含到名字空间中，以避免全局名字