L1、L2 绪论与编程环境

L1、L2 绪论与编程环境

By 曹菡雯（计 03）、赵晨阳（软 01）、罗华坤（化 93）、李晨宇（材 01）

readme

这一部分主要是 L1 与 L2 的笔记扩展，由小组四名同学共同完成。在课件的基础上，我们尽力做到了对于部分 PPT 有深入的解读，同时联系前后课程的内容与课程作业，对于课件中的一些操作也有一定的扩展。

5 月 2 日更新：将原文档中所有代码重新进行了编写，有助于阅读。
5 月 3 日更新：修改了 2.2.3.4 节的笔误

如果阅读时间不够充足，建议阅读课堂的扩展部分。
2.2 预编译指令
2.3 标识符
2.4.1 更详细的编译与链接过程

一、绪论

1.1 面向对象程序的可靠性

oop 三性（在 L4-课程笔记 1.0 节中有更详尽的解读）
简单性：使程序短而容易管理
清晰性：好的可读性，保证程序容易理解，无论是对人还是对机器
普遍性: 程序在很广泛的情形下都能工作得很好，也容易做修改以适应新出现的情况

1.2 对象的性质

对象是对现实世界中实际事物的一种抽象描述，它可以是有形的实体，也可以是无形的概念，作为构成世界的一个独立单位，它具有自己的静态特征和动态特征。
静态特征：可以用某种数据来描述的特征
动态特征：对象所表现的行为或所具有的功能
对象由一组属性和对这组属性进行操作的一组服务构成，是属性和服务的结合体。

二、源程序的结构、编译、链接

2.1 源程序的结构

```
#include <iostream>
```

```
using namespace std;//头文件与编译指令
int add(int a, int b)
{return a + b;}//辅助函数定义

int main() //主函数定义
{
    cout << add(3, 4);
    return 0;
}
```

2.2 预编译指令

2.2.1 定义

上文代码中提及的“头文件与编译指令”中的编译指令实际上是预编译指令的意思，而预编译与预处理是同一概念。

C/C++编译系统编译程序的过程为预编译、编译、链接。预处理器在程序源文件被编译之前根据预处理指令对程序源文件进行处理。

而预处理器指令以#号开头标识，并不是一句指令，末尾不包含分号。预处理命令不是 C/C++语言本身的组成部分，不能直接对它们进行编译和链接。C/C++语言的一个重要功能是可以使用预处理指令和具有预处理的功能。二者提供的预处理功能主要有文件包含、宏替换、条件编译等。

2.2.2 文件包含

预处理指令#include 将被包含的文件代码，直接复制到当前文件，一般被用于包含头文件（实际也能包含任意代码），有两种形式：#include <xxx.h>，#include "xxx.h"。尖括号形式表示被包含的文件在系统目录中。如果被包含的文件不一定在系统目录中，应该用双引号形式。（注意到是应该用，不是最好用）

在双引号形式中可以指出文件路径和文件名。如果在双引号中没有给出绝对路径，则默认为用户当前目录中的文件，此时系统首先在用户当前目录中寻找要包含的文件，若找不到再在系统目录中查找。对于用户自己编写的头文件，应用双引号形式。对于系统提供的头文件，既可以用尖括号形式，也可以用双引号形式，都能找到被包含的文件，但使用用尖括号形式更直截了当，效率更高。

2.2.3 宏替换

2.2.3.1 意义

#define 是 C++语言中的一个预编译指令，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。在程序被编译前，先将宏名用被定义的字符串替换，这称为宏替换，替换后才进行编译，宏替换是简单的替换。

2.2.3.2 宏定义

宏定义包括无参数宏定义和带参数宏定义两类。宏名和宏参数所代表的代码序列可以是任何意义的内容，如类型、常量、变量、操作符、表达式、语句、函数、代码块等。但是宏名和宏参数必须是合法的标识符，其所代表的内容及意义在宏展开前后必须一直是独立且保持不变的，不能分开解释和执行。

2.2.3.3 无参数宏

用一个用户指定的称为宏名的标识符来代表一个代码序列，这种定义的一般形式为#define 标识符 代码序列。其中#define 之后的标识符称为宏定义名（简称宏名），在宏定义#define 之前可以有若干个空格、制表符，但不允许有其它字符，宏名与代码序列之间用空格符分隔。

例如：

```
#define PI 3.1415926535
```

在 C++中，这种替换一般被 const 取代，进而能保证类型的正确性。

```
const double PI = 3.1415926535
```

2.2.3.3 带参数宏

带参数宏定义进一步扩充了无参数宏定义的能力，这时的宏展开既进行宏名的替换又进行宏参数的替换。带参数的宏定义的一般形式为#define 标识符(参数表) 代码序列，其中参数表中的参数之间用逗号分隔，在代码序列中必须要包含参数表中的的参数。在定义带参数的宏时，宏名与左圆括号之间不允许有空白符，应紧接在一起，否则变成了无参数的宏定义。带参数宏调用提供的实在参数个数必须与宏定义中的形式参数个数相同。

例如：

```
#define <宏名>(<参数表>) <字符串>
#define sqr(x) ( (x) * (x) )
sqr(3+2)等价于( (3+2) * (3+2) ) = 25
```

在 C++中，这种替换一般被内联函数取代，进而能保证类型的正确性。（在 L3）

```
inline double sqr(double x)
{return x * x;}
```

2.2.3.4 宏作用域

宏定义的有效范围称为宏名的作用域，宏名的作用域从宏定义的结束处开始到其所在的源代码文件末尾。宏名的作用域不受分程序结构的影响。如果需要终止宏名的作用域，可以用预处理指令#undef 加上宏名。

2.2.3.5 宏展开

预处理器在处理宏定义时，会对宏进行展开（即宏替换）。宏替换首先将源文件中在宏定义随后所有出现的宏名均用其所代表的代码序列替换之，如果是带参数宏则接着将代码序列中的宏形参名替换为宏实参名。

宏替换只作代码字符序列的替换工作，不作任何语法的检查，也不作任何的中间计算，一切其它操作都要在替换完后才能进行。如果宏定义不当，错误要到预处理之后的编译阶段才能发现。

源代码中的宏名和宏定义代码序列中的宏形参名必须是标识符才会被替换，即只替换标识符，不替换别的东西，像注释、字符串常量以及标识符内出现的宏名或宏形参名则不会被替换。例如：

```
#define NAME vrmozart
//NAME、/*NAME*/、"NAME"、my_NAME_blog 中的宏名 NAME 都不会被替换。
```

关于标识符在 2.4 中阐述。

2.2.3.6 宏的独立性

④ 宏的独立性

前文提及，宏名和宏形参名所代表的内容及意义在宏展开前后必须一直是独立且保持不变的，不能分开解释和执行。然而，在宏调用时，预处理器用宏定义的代码序列替换宏名，用宏实参名替换宏形参名。替换后，宏定义的代码序列就与源文件中相邻的代码自然连接，宏实参名也与代码序列中相邻的代码自然连接，宏定义的代码序列和宏实参名的独立性就不一定依旧存在。例如：

```
#define SQR(x) x*x，希望实现表达式的平方计算。
```

对于宏调用 p＝SQR(y)，能得到希望的宏展开 p=y*y。但对于宏调用 q＝SQR(u＋v)，得到的宏展开是 q＝u＋v*u＋v。显然，后者的展开结果不是程序设计者所希望的。为能保持宏实参名替换后的独立性，应在宏定义中给形式参数加上括号。进一步，为了保证宏名调用的独立性，作为算式的宏定义代码序列也应加括号。SQR 宏定义改写成#define SQR(x) ((x)*(x))才是正确的宏定义。即便如此，如此的宏定义依然很危险。

2.2.3.7 宏调用与函数调用

函数调用在程序运行时实行，而宏展开是在编译的预处理阶段进行；函数调用占用程序运行时间，宏调用只占编译时间；函数调用对实参有类型要求，而宏调用实在参数与宏定义形式参数之间没有类型的概念，只有字符序列的对应关系；函数调用可返回一个值，宏调用获得希望的代码序列。

2.2.3.8 其他性质

宏名一般用大写字母，以便与变量名区别。如有必要，宏名可被重复定义，被重复定义后，宏名原先的意义被新意义所代替。

宏定义代码序列中可以引用已经定义的宏名，即宏定义可以嵌套。

2.2.4 条件编译指令

2.2.4.1 定义

一般情况下，在进行编译时对源程序中的每一行都要编译，但是有时希望程序中某一部分内容只在满足一定条件时才进行编译，如果不满足这个条件，就不编译这部分内容，这就是条件编译。条件编译主要是进行编译时进行有选择的挑选，注释掉一些指定的代码，以达到多个版本控制、防止对文件重复包含。#if,#ifndef,#ifdef,#else,#elif,#endif 是比较常见条件编译预处理指令，可根据表达

式的值或某个特定宏是否被定义来确定编译条件。

2.2.4.2 指令含义

#if 表达式非零就对代码进行编译。

#ifdef 如果宏被定义就进行编译。

#ifndef 如果宏未被定义就进行编译。

#else 作为其它预处理的剩余选项进行编译。

#elif 这是一种#else 和#if 的组合选项。

#endif 结束编译块的控制。

2.2.4.3 常用形式一

#if_#endif 形式：（注意#if_#endif 没有大写字母）

```
#if 常数表达式 或 #ifdef 宏名 或 #ifndef 宏名
程序段
#endif
```

如果常数表达式为真或者该宏名已定义或者该宏名未定义，则编译后面的程序段；否则就不编译，跳过这段程序。

类似这样的语句可以用来避免头文件重复编译：（也可以用 pragma once 替代）

```
#ifndef FUNC_H
#define FUNC_H
```

2.2.4.4 常用形式二

#if_#else_#endif 形式：

```
#if 常量表达式 或 #ifdef 宏名 或 #ifndef 宏名
    程序段 1
  #else
    程序段 2
  #endif
```

如果常数表达式为真或者该宏名已定义或者该宏名未定义，则编译后面的程序段 1；否则编译后面的程序段 2。

2.2.4.5 常用形式三

#if_#elif_#endif 形式：

```
#if 常量表达式 1
    程序段 1
  #elif 常量表达式 2
    程序段 2
    .....
  #elif 常量表达式 n
     程序段 n
  #endif
```

注意这种形式#elif 需要搭配 if 使用，不可以用于#ifdef 和#ifndef 中。后二者仅可搭配#else 使用。

2.2.4.6 预处理器表达式

预处理器表达式包括的操作符主要涉及到单个数的操作（+、-、~、<<、>>）、多个数的运算（*、/、%、+、-、&、^、|）、关系比较（<、<=、>、>=、==、!=）、宏定义判断（defined）、逻辑操作（!、&&、||），其优先级和行为方式与 C++表达式操作符相同。预处理器表达式在编译器预处理器上执行，在编译前进行。

例如：#ifndef 与#if !defined 意义相同，#ifdef 与#if defined 意义相同。

2.2.5 其他预处理指令

除了上面讨论的常用预处理指令外，还有三个不太常见的预处理指令：#line、#error、#pragma，下面分别介绍。

2.2.5.1 #line

#line 指令用于重新设定当前由__FILE__和__LINE__宏指定的源文件名字和行号。#line 一般形式为#line number "filename"，其中行号 number 为任何正整数，文件名 filename 可选。#line 主要用于调试及其它特殊应用，注意在#line 后面指定的行号数字是表示从下一行开始的行号。

2.2.5.2 #error

#error 指令使预处理器发出一条错误消息，然后停止执行预处理。一般形式为#error info，如#error MFC requires C++ compilation。

2.2.5.3 #pragma

#pragma 指令可能是最复杂的预处理指令（也最方便），它的作用是设定编译器的状态或指示编译器完成一些特定的动作。#一般形式为#pragma para，其中 para 为参数，下面介绍一些常用的参数。

#pragma once，只要在头文件的最开始加入这条指令就能够保证头文件被编译一次。

#pragma message("info")，在编译信息输出窗口中输出相应的信息，例如#pragma message("Hello")。

#pragma warning，设置编译器处理编译警告信息的方式，例如#pragma warning(disable:4507 34;once : 4385;error:164)等价于#pragma warning(disable:4507 34)（不显示 4507 和 34 号警告信息）、#pragma warning(once:4385)（4385 号警告信息仅报告一次）、#pragma warning(error:164)（把 164 号警告信息作为一个错误）。

#pragma comment(…)，设置一个注释记录到对象文件或者可执行文件中。常用 lib 注释类型，用来将一个库文件链接到目标文件中，一般形式为#pragma

comment(lib,"*.lib")，其作用与在项目属性链接器“附加依赖项”中输入库文件的效果相同。

2.3 标识符

2.3.1 定义

标识符即为编程的时候使用的“名字”，给类、接口、方法、变量、常量名等起名字的字符序列。

2.3.2 组成

英文大小写字母、数字、下划线（ _ ）和美元符号（ $ ） (可以使用汉字或其他合法字符命名，但是不推荐)。

2.3.3 定义规则

不能以数字开头。不能是关键字。严格区分大小写。可以是汉字或其他合法字符命名，但不推荐。

2.3.4 命名规范（非强制）

类和接口：首个字母大写，如果有多个单词，每个单词首字母大写：HelloWorld、Student

变量和方法: 首字母小写，如果有多个单词，从第二个单词开始首字母大写：getName、studyJava

常量名（自定义常量）：所有字母都大写，多个单词用下划线隔开（ _) :MAX_VALUE

2.4 编译与链接

2.4.1 过程

预处理: 预处理器在程序源文件被编译之前根据预处理指令对程序源文件进行处理，C/C++主要的预处理功能有文件包含、宏替换、条件编译等。

编译: 编译阶段是检查语法，生成汇编。第一遍执行语法分析和静态类型检查，将源代码解析为语法分析树的结构。第二遍由代码生成器遍历语法分析树，把树的每个节点转换为汇编语言或机器代码，生成目标模块(.o 或.obj 文件）

汇编: 汇编代码转换机器码。非底层的程序员不需要考虑这一阶段，编译器也不会出错。汇编与 c/c++开发者无关，但是我们可以利用反汇编来调试代码，学习汇编语言依然是必备的。

链接: 把一组目标模块链接为可执行程序，使得操作系统可以执行它。处理目标模块中的函数或变量引用，必要时搜索库文件处理所有的引用。（见例

2.5.2）

2.4.2 编译指令

g++ -c：只编译不链接。

g++ -o ex1.out ex1.o：链接程序。

g++ ex5_main.cpp func.cpp -o：直接编译（g++帮我们省略了一些步骤）。

2.4.3 链接

将各个目标文件中的各段代码进行地址定位，生成与特定平台相关的可执行文件。外部函数的声明（一般声明在头文件中）只是令程序顺利通过编译，此时并不需要搜索到外部函数的实现（或定义）。在链接过程中，外部函数的实现（或定义）才会被寻找和添加进程序，一旦没有找到函数实现，就无法成功链接。

2.5 头文件

2.5.1 意义

有时辅助函数(如全局函数)会在多个源文件中被使用。将辅助函数编入头文件中，从而避免反复编写同一段声明，也能够统一辅助函数的声明，避免错误。

2.5.2 例子

```
// func.h
int ADD(int a, int b);
```

```
// func.cpp
#include "func.h"
int ADD(int a, int b)
{ return a + b; }
```

```
// ex5_main.cpp
#include <iostream>
#include <cstdlib>   // atoi()
#include "func.h"    // ADD()
int main(int argc, char** argv) {
    ……
    std::cout << ADD(a, b) << std::endl;
    return 0;
}
```

```
g++ func.cpp ex5_main.cpp -o main.out
```

在我们的例子中
去掉红色语句(include)是
否仍然能够成功编译及链接

A 可以

B 不可以

提交

程序在连接时，搜索了编译命令里的所有文件，本题中即为两个 cpp。因为 main.cpp 里面写了 include“func.h”，所以能找到 func.h。如果把 main.cpp 里的 include 也去掉，就无法编译了。

2.6 函数的声明与定义

2.6.1 概念

函数声明：int ADD(int a, int b); int ADD(int, int); //变量名可省略，例如后缀运算符重载中的哑元。

函数定义（即实现）：int ADD(int a, int b) {return a + b;}

同一个函数可以有多次声明，但只能有一次实现，多次实现会导致链接错误。（multiple definition）注意这和重载的区别，重载是同名函数参数不同。

2.6.2 变量的声明与定义

关于变量的声明与定义更加详细的讨论在 L5-创建与销毁二 1.3.0 节，此处不再赘述。

2.6.3 extern 关键字

关于 extern 修饰符更加详细的讨论在 L5-创建与销毁二 1.2.5 节，此处不再赘述。

2.6.4 结论

对于头文件，尽量只申明函数而不实现函数。尽量只声明全局变量而不定义全局变量。