C语言

第6讲：预处理指令和模块化编程

赵岩

哈尔滨工业大学软件学院

August 20, 2011

目录

预编译指令

- 编译器在开始正式编译之前处理的指令，叫预编译指令
  - 编译器编译的是经过预编译处理后的代码
  - 预编译过程不进行C语法检查

#define

- #define　宏名字　替换文本
- 在#define之后，所有独立出现“宏名字”的地方（除了字符串内）都被“替换文本”替换
- “替换文本”中可以有空格
- 宏可以有参数
  - #define max(A,B) ((A) > (B) ? (A) : (B))
- 定义宏的时候注意替换发生后产生的非预想结果
  - 用括号可以避免一些非预想结果

#define的缺点

- 它只是简单的替换，不进行任何语法检查

Demo 1: A example of define

```
1  #define PI 3.1415;
2  #define max(A,B) ((A) > (B) ? (A) : (B))
3  void main()
4  {
5      int i = 4;
6      printf("%f\n",PI); /* compile error*/
7
8      max(i++,3);
9      /*((i++)>3?(i++):3); */
10     printf("%d\n",i);
11 }
```

#define的替换方法

- const 常量
- 模板内联函数

Demo 2: Better method

```
1  const double PI 3.1415;
2
3  template<class T>
4  inline const T& max(const T& a, const T& b)
5  { return a > b ? a : b; }
```

#include指令

- 用#include指定文件的内容替换#include所在的行
- 也是简单的替换，养成在头文件后面加上分号的习惯。
- 用<>或者“ ”括上文件名
  - <>表示在编译器的include目录内查找文件
  - “ ”表示在当前目录下查找文件
- 文件名中可以带有路径

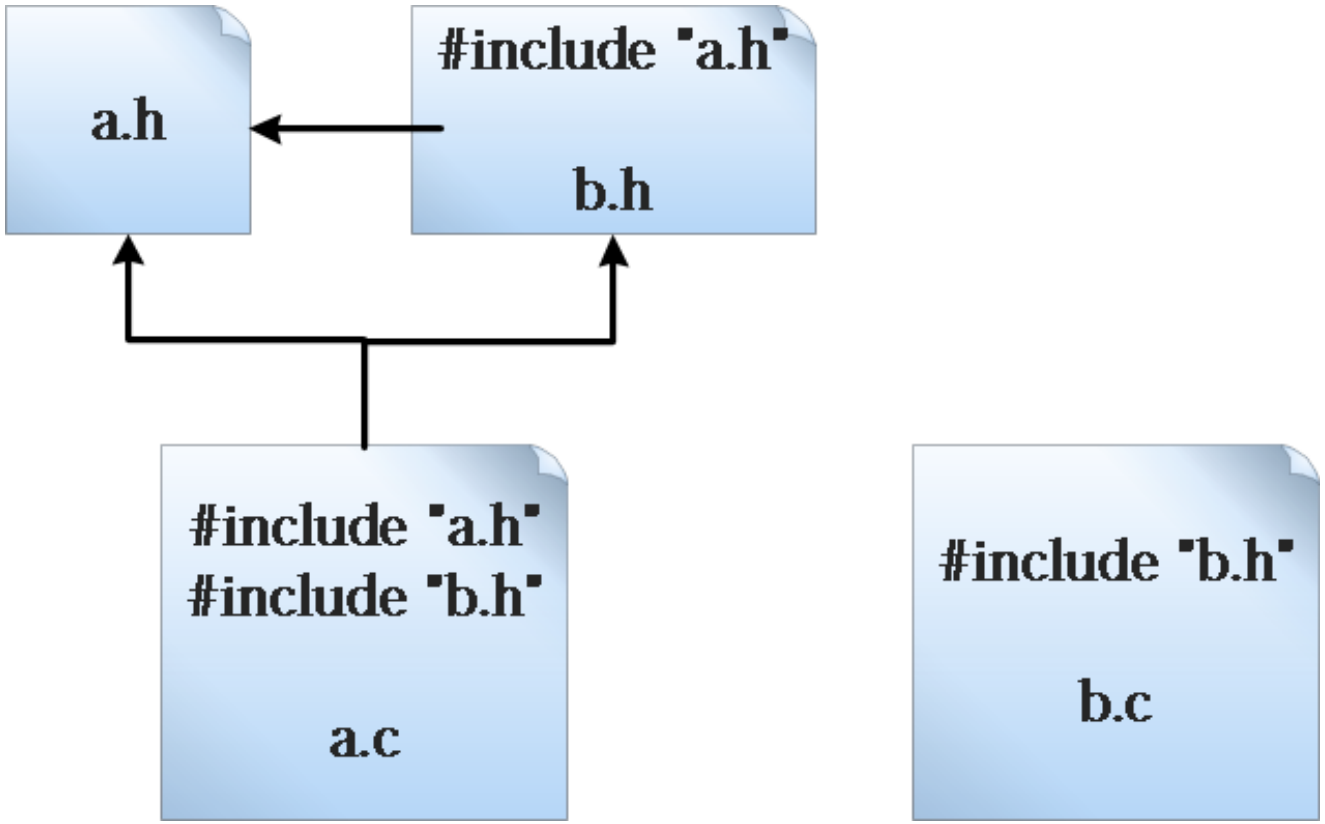
#ifndef

- #undef，从现在开始取消#define的定义
- #undef MAXLINE
- #if，#else，#elif，#endif
- #ifdef，#ifndef
- 这些预编译指令通常用来处理多文件工程和程序多版本的问题。
- （程序多版本一般是不同平台的版本，不同用户等级的版本，不同开发阶段的版本等）

跨平台

```
1 #ifdef linux
2     #include <ext/hash_map>
3     #include <ext/hash_set>
4 #else
5     #include <hash_map>
6     #include <hash_set>
7 #endif
```

避免重复包含



使用预编译指令的目的

- 增强程序可读性
  - 但是调试时宏可能会带来很多问题
- 精简源代码，提取变化
  - 这一点更多时候用函数的效果更好，但宏也有其不可替代的优势
- 不编译无用代码，精炼目标代码

基本知识之编译器和链接器

- C语言是每个C文件单独编译的
  - 每次只编译一个C文件，看不到其它的C文件
- 编译器
  - 编译器确认每个标识符的类型
  - 我要**认识**它
- 链接器
  - 链接过程在内存中定位这个标识符
  - 我要**唯一地**找到它

基本知识之声明和定义

- 声明就是让编译器能够认识它，
- 而定义就是让链接器能够找到它。

```
1 int i; /* define */
2 extern int i /* declare */
```

```
1 foo(); /* declare */
2 foo(){ /* define */
3 ....
4 };
```

问题

foo(){}; 是声明还是一个定义？

例子1

- a.c文件

```
1  int  i=5; /* a.c */
2  main(){
3        printf("%d",foo());
4  }
```

- b.c文件

```
1  foo(){  /*b.c*/
2        i*=2;
3  };
```

例子2

- a.c文件

```
1  int  i=5; /* a.c */
2  main(){
3        printf("%d",foo());
4  }
```

- b.c文件

```
1  int  i; /*b.c*/
2  foo(){
3        i*=2;
4  };
```

例子3

- a.c文件

```
1  int  i=5; /* a.c */
2  foo();
3  main(){
4        printf("%d",foo());
5  }
```

- b.c文件

```
1  int  i; /*b.c*/
2  foo(){
3        i*=2;
4  };
```

正确例子

- a.c文件

```
1  int i=5; /* a.c */
2  foo();
3  main(){
4        printf("%d",foo());
5  }
```

- b.c文件

```
1  extern int i; /*b.c*/
2  foo(){
3        i*=2;
4  };
```

不仅正确，而且优雅！

- a.c文件

```
1 #include "b.h" /* a.c */
2 int i=5;
3 main(){
4       printf("%d",foo());
5 }
```

- b.c文件

```
1 #include "b.h" /*b.c*/
2 foo(){
3       i*=2;
4 };
```

不仅正确，而且优雅！

- b.h文件

```
1  #ifndef _B_H  /* b.h */
2  #define _B_H
3  foo();
4  extern int i;
5  #endif
```

工程-project

- 工程通常包含：
  - 源文件(xxx.c)：一系列相关函数和全局变量的**定义**
  - 头文件(xxx.h)：函数和全局变量的**声明**，如：函数声明、外部变量声明、宏定义、类型定义...
- 可以将模块编译为.o、.a、.obj或.lib文件，同.h文件一起供别人使用，从而保护了源代码
- 使用模块的过程
  - 建立一个工程（project）
  - 把各模块都加入到工程中
  - #include模块的头文件
  - 开始使用此模块

文件中的变量

- 文件的信息隐藏
  - 用static定义的函数和变量只在此模块（文件）中有效（建议采用）
- 允许被其它文件使用的全局变量
  - 在源文件中定义，不加static修饰
  - 在头文件中进行声明，加extern修饰

多文件工程实例

- 整个项目包含两个源文件（编译单元）
  - 计算圆面积
  - 主函数

多文件工程实例源码-main.c

```
1  #include"Area.h"
2  void main()
3  {
4      double r = 3;
5      double area = Circle_Area(r);
6      area = Circle_Area(r);
7      printf("area of circle is %f\n",area);
8      printf("PI is %f\n", PI);
9      GetTimes();
10     printf("function was called %d times"
11                                 ,global_times);
12 }
```

多文件工程实例源码-Area.c

```
1  #include "Area.h"
2  const double PI = 3.1415926; /*全局常量*/
3  int global_times = 0; /*全局变量*/
4  static int times = 0; /*静态变量*/
5
6  double Circle_Area(double r){
7      times++;
8      return PI*r*r;
9
10 }
11 void GetTimes(){
12     global_times = times;
13 }
```

多文件工程实例源码-Area.h

```
1 #ifndef _AREA_H
2 #define _AREA_H
3
4 extern const double PI;
5 extern int global_times;
6 double Circle_Area(double r);
7 void GetTimes();
8
9 #endif
```

总结

谢谢大家，欢迎提问！