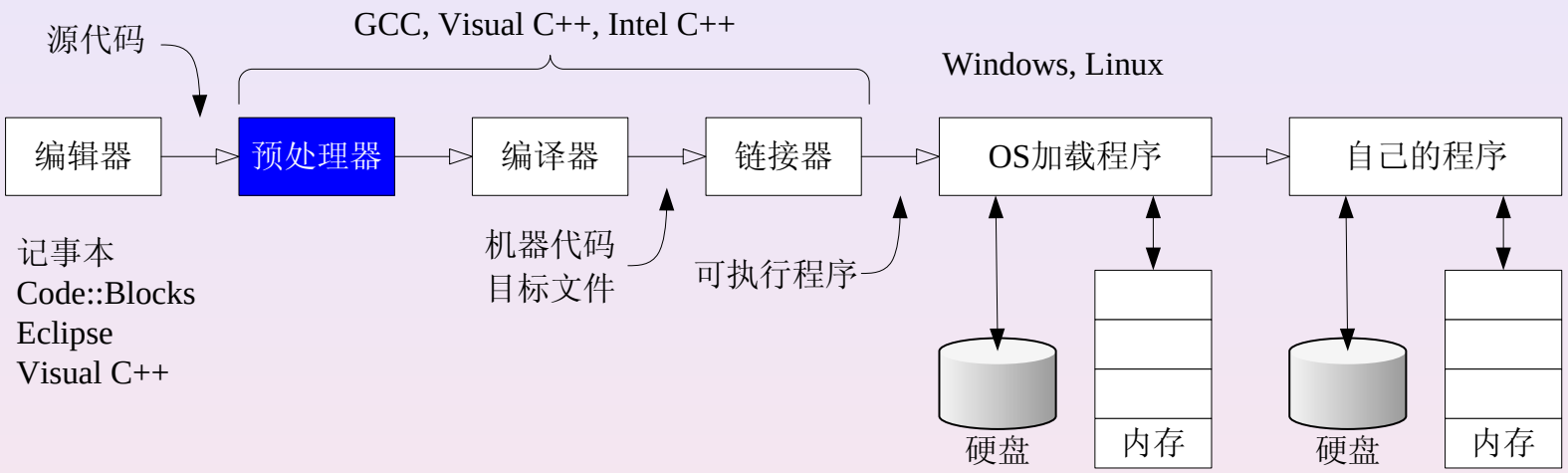
Part VII

编译预处理

主要内容

编译预处理



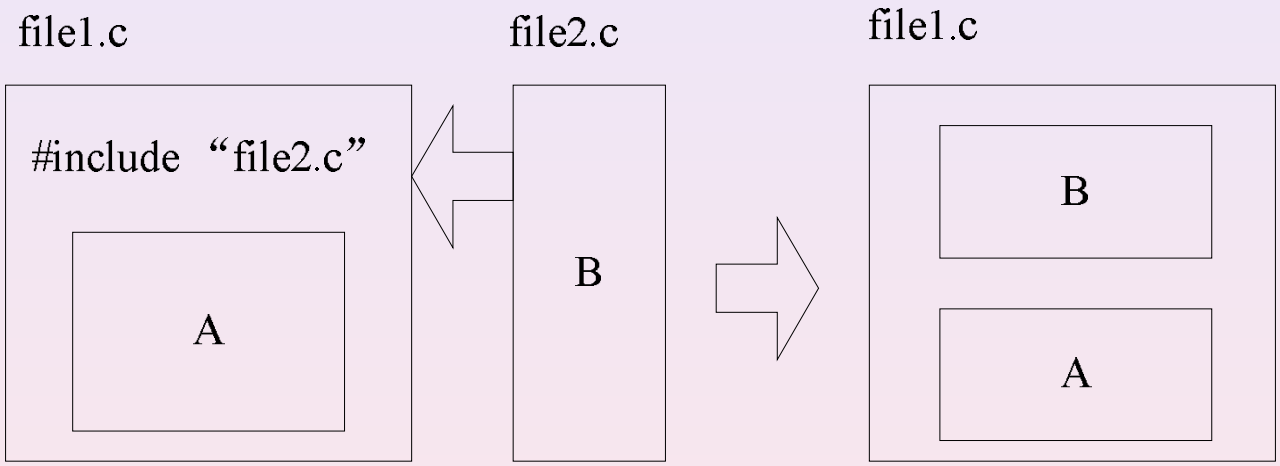
- 在源代码中以#开始的那些行，称为编译预处理命令。
- 根据源代码中的编译预处理指令，对源代码首先进行“预先”处理，然后将处理过的源代码送给编译器。
- GCC编译器中可使用以下命令获得编译预处理后的源代码：

  gcc –E –C –o output_file source_file

主要内容

文件包含处理

#include用来将另外一个文件的全部内容包含进来。



- 一个**#include**命令只能包含一个文件。
- 文件包含可以嵌套，一个被包含文件可以包含另一个文件。
- **#include**可包含任何类型的文件，但主要用来包含头文件。

两种include格式的区别

命令格式有两种：

```
#include ”文件名”
#include <文件名>
```

- **#include** ”file .h” 系统首先在引用被包含文件的源文件所在目录查找file.h，如果没有找到，则再按照系统指定的标准方式检索其它目录。
- **#include** <file.h> 不检查源文件目录而直接按照系统标准方式检索文件目录。
- 通常，包含系统提供的库文件采用尖括号形式，而包含用户自定义文件采用双引号形式。

主要内容

语法和语义

宏定义：用指定的标识符（名字）来代表一串文字。

```
#define 宏名 替代文字
```

每当在源代码中看到宏，就使用宏定义的内容替换宏名。

```
1  #define PI  3.14
2
3  int  main(void)
4  {
5      float  r  = 5.0,  area;
6      area = PI * r * r;
7      printf("area = %f\n", area);
8      return 0;
9  }
```

宏名一般建议用大写字母，以和变量名区别。

宏替换

宏替换仅仅是做简单的字符串置换，不做语法检查；语法检查是编译器的工作。

- 要特别注意宏定义行末的分号。
- 宏定义时，可以引用已定义的宏名。
- 注意预处理时，只是替换，不发生实际的运算。
- 字符串内的字符，即便是和宏名相同，也不替换。

```
1  #define ARR_SIZE 10;
2  int  arr[ARR_SIZE];  // 编译预处理后⟹int arr[10;]; 编译器报错。
3
4  #define PRINT_HELLO printf("hello\n");
5  PRINT_HELLO;        // 编译预处理后⟹printf("hello\n");; 没有语法错误。
```

宏替换

宏替换仅仅是做简单的字符串置换，不做语法检查；语法检查是编译器的工作。

- 要特别注意宏定义行末的分号。
- 宏定义时，可以引用已定义的宏名。
- 注意预处理时，只是替换，不发生实际的运算。
- 字符串内的字符，即便是和宏名相同，也不替换。

```
1 #define R      3.0
2 #define PI     3.14
3 #define AREA   PI * R * R
```

宏替换

宏替换仅仅是做简单的字符串置换，不做语法检查；语法检查是编译器的工作。

- 要特别注意宏定义行末的分号。
- 宏定义时，可以引用已定义的宏名。
- 注意预处理时，只是替换，不发生实际的运算。
- 字符串内的字符，即便是和宏名相同，也不替换。

```
1  #define X   5
2  #define Y   X+1
3  #define Z   Y*X/2
4
5  int  a;
6  a = Y;                  //  编译预处理后⟹a = 5+1;
7  printf("%d\n",Z); //  编译预处理后⟹printf("%d\n",5+1*5/2);可能不是你想要的
```

宏替换

宏替换仅仅是做简单的字符串置换，不做语法检查；语法检查是编译器的工作。

- 要特别注意宏定义行末的分号。
- 宏定义时，可以引用已定义的宏名。
- 注意预处理时，只是替换，不发生实际的运算。
- 字符串内的字符，即便是和宏名相同，也不替换。

```
1  #define PI      3.14
2
3  printf("PI = %f", PI);  // 编译预处理后⟹printf("PI = %f", 3.14);
```

宏定义的优点

- 使用宏，有时可减少程序中重复书写的工作量。
- 可读性更强：使用宏名替代常量值，有意义的名字比数字更容易读懂。
- 可维护性更好：使用宏名替代常量值，当需要改变常量时，只需修改宏定义即可。

```
1  int  arr[10];
2
3  for(i=0; i<10; i++)
4  {
5        ......
6  }
```

```
1  #define ARR_SIZE 10
2
3  int  arr[ARR_SIZE];
4
5  for(i=0; i<ARR_SIZE; i++)
6  {
7        ......
8  }
```

宏定义的终止

宏定义的有效范围是从定义开始到源文件结束，如果要提前结束，可以用宏命令**#undef** macro_name来控制宏定义的作用范围。

```
1  #define FACTOR          9.8
2
3  int  main(void)
4  {
5        .......
6  }
7
8  #undef FACTOR
9  // 宏FACTOR的作用范围到此为止，可以重新对其定义
10
11 #define FACTOR          10
12
13 int  func(void)
14 {
15       ......
16 }
```

主要内容

带参数的宏定义

定义形式：

```
#define 宏名(参数表)  字符串
```

```
1  #define AREA(r)     3.14 * r * r
2
3  area = AREA(3);                    //编译预处理⟹area = 3.14 * 3 * 3;
```

```
1  #define MULTI(a,b)  a * b
2
3  result = MULTI(2,5);               //编译预处理⟹result = 2 * 5;
```

带参宏的使用极容易出错，请务必小心使用。

空格的使用

宏名和参数列表之间不能有空格，否则按无参的宏定义方式进行替换。

```
1  #define S(r)     PI * r * r                    ⟸这是一个带参数的宏
2  area = S(a);                                   编译预处理⟹area = PI * a * a;
3
4  #define S (r)    PI * r * r                    ⟸这是一个无参数的宏
5  area = S(a);                                   编译预处理⟹area = (r) PI * r * r(a);
```

括号的使用

要特别注意括号的使用。

```
1  #define MULTI(a,b)  a * b
2
3  result = MULTI(2,5);                    编译预处理⟹result = 2 * 5;✔
4
5  result = MULTI(2 + 1, 5);               编译预处理⟹result = 2 + 1 * 5;✖
```

括号的使用

要特别注意括号的使用。

```
1 #define MULTI(a,b)  a * b
2
3 result = MULTI(2,5);                    编译预处理⟹result = 2 * 5;✔
4
5 result = MULTI(2 + 1, 5);               编译预处理⟹result = 2 + 1 * 5;✖
```

```
1 #define MULTI(a,b)  (a) * (b)
2 result = MULTI(2 + 1, 5);               编译预处理⟹result = (2 + 1) * (5);✔
```

括号的使用

要特别注意括号的使用。

```
1  #define MULTI(a,b)  a * b
2
3  result = MULTI(2,5);                    编译预处理⟹result = 2 * 5;✔
4
5  result = MULTI(2 + 1, 5);               编译预处理⟹result = 2 + 1 * 5;✖
```

```
1  #define MULTI(a,b)  (a) * (b)
2  result = MULTI(2 + 1, 5);               编译预处理⟹result = (2 + 1) * (5);✔
```

```
1  #define DOUBLE(x)   (x) + (x)
2
3  a = 5;
4  printf("%d\n",10 * DOUBLE(a));          编译预处理⟹printf("%d\n",10 * (a) + (a));✖
```

括号的使用

要特别注意括号的使用。

```
1  #define MULTI(a,b)  a * b
2
3  result = MULTI(2,5);                    编译预处理⟹result = 2 * 5;✔
4
5  result = MULTI(2 + 1, 5);               编译预处理⟹result = 2 + 1 * 5;✖
```

```
1  #define MULTI(a,b)  (a) * (b)
2  result = MULTI(2 + 1, 5);               编译预处理⟹result = (2 + 1) * (5);✔
```

```
1  #define DOUBLE(x)   (x) + (x)
2
3  a = 5;
4  printf("%d\n",10 * DOUBLE(a));          编译预处理⟹printf("%d\n",10 * (a) + (a));✖
```

```
1  #define DOUBLE(x)   ((x) + (x))
2  printf("%d\n",10 * DOUBLE(5));          编译预处理⟹printf("%d\n",10 * ((5) + (5)));✔
```

宏v.s.函数

带参宏和函数有时可以完成相同的功能。

```
1  #define MAX(x,y)    (x > y ? x : y)
2
3  int  max(int x,int  y)
4  {
5      return x > y ? x : y;
6  }
```

带参宏极容易出错，务必小心使用。 双击打开

```
1  int  a = 3;
2  int  b = 3;
3  int  c = MAX(++a, b); //非预期的结果
4
5  float  m = 3.8;
6  float  n = 4.1;
7  float  z = MAX(m, n); //不检查参数类型
```

```
1  int  a = 3;
2  int  b = 3;
3  int  c = max(++a, b); //可以预测结果
4
5  float  m = 3.8;
6  float  n = 4.1;
7  float  z = max(m, n); //参数类型匹配检查
```

宏v.s.函数

宏

- 本质上是代码替换
- 顺序执行，代码长度可能会大幅增加。
- 运行期间没有额外开销。
- 类型无关，但未必安全。
- 表达式的书写需要合理使用括号，极易出错。

函数

- 本质上是运行逻辑
- 跳转执行，代码长度不会大幅增加。
- 存在调用/返回的额外开销。
- 严格类型检查，是安全的。
- 实参表达式求值，复制给形参，结果容易预测。

主要内容

条件编译

源文件中的某部分代码只在满足一定条件时才进行编译。

条件编译形式1：

```
1  #ifdef 标识符
2  程序段1
3  #else
4  程序段2
5  #endif
6
7  #ifdef 标识符
8  程序段1
9  #endif
```

- 含义：如果标识符已经被定义(通常由#define定义)，则对程序段1进行编译，否则编译程序段2。
- 比较示例代码分别使用以下两个命令的预处理结果。

```
gcc -E -C -DMY_DEBUG -o output.c main.c
gcc -E -C -o output.c main.c
```

条件编译

源文件中的某部分代码只在满足一定条件时才进行编译。

条件编译形式2：

```
1  #ifndef 标识符
2  程序段1
3  #else
4  程序段2
5  #endif
6
7  #ifndef 标识符
8  程序段1
9  #endif
```

含义：如果标识符没有被定义，则编译程序段1，否则则编译程序段2。

条件编译

源文件中的某部分代码只在满足一定条件时才进行编译。

条件编译形式3：

```
1  #if 常量表达式
2  程序段1
3  #else
4  程序段2
5  #endif
6
7  #if 常量表达式
8  程序段1
9  #endif
```

含义：如果常量表达式的值为真（非0）则编译程序段1，否则编译程序段2。

课本例7.17： 双击打开

小技巧:利用#if 0和#endif注释源代码

```
#if 0
printf(”This is the debug info\n”);
#endif
```