# c-编译预处理

## #define

> 可以出现在代码任何地方
>
> 从本行开始之后的代码都可以使用该宏常量

> 宏表达式

```
#define MIN(a,b) ((a<b)?(a):(b))

int min(int a, int b)
{
    return ((a < b) ? a : b);
}
int main()
{
    int i = 1;
    int j = 5;
    printf("%d", MIN(++i, j));
//返回3,因为是直接替换
}
```

> 避免宏表达式中出现++

```
#define DIM(array) (sizeof(array) / sizeof(*array))

int dim(int array[])
{
    return sizeof(array) / sizeof(*array);
} //因为传进来的被看成了一个指针导致始终为1

int main()
{
    int a[] = { 1,2,3,4,5 };
    printf("%d\n",dim(a));   //1
    printf("%d\n", DIM(a));  //5
}
```

> 宏表达式预编译时被处理，编译器棒子点宏表达式的存在
>
> 宏表达式用“实参”完全代替形参，不进行任何计算
>
> 无任何调用开销
>
> 不能出现递归

> 使用范围
>
> 如果没有#undef会一直存在

宏代码块

```
#define MALLOC(type,x) (type*)malloc(sizeof(type)*x)
#define FOREVER() while(1)
#define BEGIN {
#define END }
#define FOREACH(i,m) for(i=0,i<m;i++)


int main()
{
    int array[] = { 1,2,3,4,5 };
    int x = 0;
    int* p = MALLOC(int, 5);

    FOREACH(x,5)
    BEGIN
        p[x] = array[x];
    END

    FOREVER();
    free(p);
}
```

```
__FILE__
__LINE__
__DATE__
__TIME__
__STDC__
```

内置宏

```
void log(char* s)
{
    printf("%s:%d\n", __FILE__, __LINE__, s);
}  //无法正确打印行号，都是该函数的行

#define LOG(s) printf("%s:%d\n", __FILE__, __LINE__, S) //可以正确打印
```

## 条件编译

预编译指示命令，用于控制是否编译某段代码

```
#define C 1

int main()
{
#if(C==1)
    printf("this is first printf");
#else
    printf("this is second printf");
#endif
}
```

可以避免重复包含同一个头文件

可以区分不同产品线代码

可以定义产品发布版调试版

## #error

生成编译错误信息+停止编译

#error message　　无需阔号

## #warning

生成编译警告，但不会停止编译

```
#define CONST_NAME1 "CONST_NAME1"
#define CONST_NAME2 "CONST_NAME2"

int main()
{
    #ifndef COMMAND
    #warning Compilation will be stoped ...
    #error No defined Constant Symbol COMMAND
    #endif

    printf("%s\n", COMMAND);
    printf("%s\n", CONST_NAME1);
    printf("%s\n", CONST_NAME2);

    return 0;
}
```

## #line

用于强制指定新的行号和新的编译文件名，

本质是重定义 **LINE**，**FILE**

## #pragma

> 编译器指示字，用于指示编译器完成一些特定动作
>
> 不同编译器可能用不同方式解释同一条#pragma
>
> #pragma message 在编译时输出消息到编译输出窗口

```
#include <stdio.h>

#if defined(ANDROID20)
    #pragma message("Compile Android SDK 2.0...")
    #define VERSION "Android 2.0"
#elif defined(ANDROID23)
    #pragma message("Compile Android SDK 2.3...")
    #define VERSION "Android 2.3"
#elif defined(ANDROID40)
    #pragma message("Compile Android SDK 4.0...")
    #define VERSION "Android 4.0"
#else
    #error Compile Version is not provided!
#endif

int main()
{
    printf("%s\n", VERSION);

    return 0;
}
```

## 内存对齐

> #pragma pack
>
> 如果数据未对齐，需要两次总线访问周期来访问内存，性能降低
>
> 内存读取不是连续而是分块读取
>
> struct占用内存大小：
>
> 第一个成员起始于0偏移处
>
> 偏移地址和 成员占用均需对齐
>
> 结构体总长度为其所有对其参数的整数倍
>
> 结构体默认对齐方式为其对齐参数中最大的一个

```
#pragma pack(8)

struct S1
{
    short a;
    long b;
};

struct S2
```

```
{
    char c;
    struct S1 d;    //结构体按最大的4对齐
    double e;       //按8对齐，所以从16开始
};

#pragma pack()

int main()
{
    struct S2 s2;

    printf("%d\n", sizeof(struct S1));
    printf("%d\n", sizeof(struct S2));
    printf("%d\n", (int)&(s2.d) - (int)&(s2.c));

    return 0;
}
```

## #运算符

> 用于在预编译期将宏参数转化为字符串

```
#include <stdio.h>

#define CALL(f, p) (printf("Call function %s\n", #f), f(p))
//逗号运算符确保操作数被顺序地处理：先计算左边的操作数，再计算右边的操作数。右操作数的类型和值作为整个表达式的结果。
int square(int n)
{
    return n * n;
}

int f(int x)
{
    return x;
}

int main()
{

    printf("1. %d\n", CALL(square, 4));
    printf("2. %d\n", CALL(f, 10));

    return 0;
}
```

## ##运算符

> 用于在预编译期粘连两个符号

```

```

```c
#include <stdio.h>

#define STRUCT(type) typedef struct _tag_##type type;\
struct _tag_##type

STRUCT(Student)
{
    char* name;
    int id;
};

int main()
{

    Student s1;
    Student s2;

    s1.name = "s1";
    s1.id = 0;

    s2.name = "s2";
    s2.id = 1;

    printf("%s\n", s1.name);
    printf("%d\n", s1.id);
    printf("%s\n", s2.name);
    printf("%d\n", s2.id);

    return 0;
}
```