第六章：宏定义、预处理、函数、函数库

C语言预处理

- 源码到可执行程序的过程
  - 源代码.c→**(编译)**→目标文件.o→**(链接)**→可执行程序.elf
  - 源代码.c→(预处理)→预处理过的.c文件.i→**(编译)**→汇编文件.s→**(汇编)**→目标文件.o→**(链接)**→可执行程序.elf
  - 预处理器、编译器、汇编器、链接器，再加上其它可用工具=编译工具链，gcc就是一个编译工具链
- **预处理处理了什么？**
  - **头文件包含：头文件展开原封不动的放在原地**
    - <>包含系统提供的头文件：编译器到系统指定目录查询（一般是gcc编译器存放头文件的地方，非当前目录），ubuntu在/usr/include
    - ""包含程序员自己写的：先在当前目录下寻找，找不到去系统指定目录，再找不到报错（规则允许""包含系统指定目录，但一般不这样用）
    - 自己写的头文件集中放在一个文件夹，将来使用 -I 寻找
  - **去除所有注释**
  - **简单宏定义替换（不对typedef进行处理）**
  - **条件编译宏（只保留满足规则的配置代码）**
    - 关键字：#if，#ifdef，#ifndef，#endif，#else，#elif
    - 实现功能：切换功能配置；调试宏（一般不用自己写）
- 相关gcc指令
  - -o指定要生成的输出文件，后面跟指定的名字
  - -c只编译不链接：gcc -o xx.o -c xx.c
  - -E只预处理不编译（帮助debug）：gcc -o xx.i -E xx.c

宏定义规则和本质解析

- 宏定义会递归替换：直接解析到不是宏的部分

示例

```
#define N 10
#define M N

int main(void)
{
    int a[M]={1,2,3};//生成的汇编文件M换成10
    int b[10];
}
```

- 宏定义第二部分可以带参数，称为带参宏 #define X(a,b) ((a)*(b))，第三部分可以有空格，且每个参数都应当加括号，最后再加括号

示例

```
//出错示例
#define X(a,b) (a+b)
int main(void)
{
  int x=1, y=2+4;
  int z=X(x,y);//得到z=(1*2+4)，实际上想实现的是(1*(2+4))
}

//MAX宏示例
#define MAX(a,b) (((a)>(b))?(a):(b))

//int溢出可以设成无符号数，空间会大一倍
#define NUM (32800UL)//16位机器会溢出，16位机器int为2字节
```

- 带参函数&带参宏
  - 宏定义预处理时原地展开，调用开销小；函数在编译时处理，调用开销大（需要跳转，复制传参）→函数短使用内联函数或带参宏更有优势
  - 宏定义不检查参数类型，返回值不附带类型；函数参数和返回值都有类型，调用函数编译器会做静态类型检查（使用宏定义必须注意传参类型，编译器不会检查！）
- 带参宏&内联函数
  - 内联函数前面加inline，结合了带参宏和带参函数的优点

- 编译器帮忙做静态类型检查（函数的优点），同时没有调用开销，可原地展开（带参宏的优点）
- linux内核中很多inline，一两行一般写成inline，三五行可写可不写

函数

- 函数本质
  - 函数本质：数据处理器。返回值、输出型参数输出数据，全局变量、输入型参数输入数据，函数内代码对数据进行加工
  - 程序本质：可执行代码段（函数编译而成）+数据，程序执行过程是多个函数相继运行的过程
  - 函数是人的需要，不是编译器、CPU的需要，为了实现模块化编程
- 书写规则：
  - 一个函数只做一件事、传参不要太多（ARM中不超过4个，传参太多考虑结构体）
  - 函数内部少用全局变量（尽量用传参和返回值和外部交换数据，使用全局变量毁坏了函数的模块化，移植到其它环境时需要重新编写）
- 函数使用
  - 函数三要素：定义、声明/函数原型（目的是让编译器帮忙做静态类型检查）、调用
  - 函数名表示函数在内存中的地址，函数调用实质是指针的解引用访问
  - 编译器一个一个文件、按先后顺序编译，因此要在调用之前声明，遇到函数调用时编译器去查函数声明表并做类型检查
- 递归函数（自己调用自己）
  - 递归函数在栈上运行，栈大小限制递归深度
  - 递归要求：必须有可被满足的终止条件
  - 典型应用：求阶乘，求斐波那契数列
- 函数库（事先写好的函数集合）
  - 函数库提供形式：静态链接库、动态链接库（可以用但看不到源码，以实现商业需求）
    - 静态链接库：只编译不链接形成 .o 文件，再用 ar 工具将 .o 文件归档形成 .a归档文件，又叫静态链接库文件
      - 商业公司提供 .a 和 .h 给客户使用

        - 客户通过 .h 得知函数原型，在自己的.c文件中调用，链接时编译器从 .a 中拿出被调用函数编译后的二进制代码段
    - **动态链接库：静态的改进，效率更高，现在一般使用这种**
        - 静态链接后的**文件大**，函数编译直接链接到程序里，不依赖于外部；若有多个程序调用某函数，内存中会**重复存储多份函数编译文件**，浪费内存空间
        - 动态链接得到的**文件小**，在调用库函数的地方**先做一个标记，运行时**操作系统从 .so 动态链接库中找到对应文件并**加载**到内存中，生成的可执行文件本身不包含动态链接库；若有多个程序调用某函数，内存中**存一份函数编译文件**即可
        - gcc默认动态链接，使用静态链接加上-static
- **制作静态链接库.a：创建成对的.c和.h文件，直接编译或者写个makefile**（记录编译过程）
    - **只编译不链接，生成.o文件**：gcc -c calcu.c -o calcu.o
    - **使用ar命令创建归档文件**：ar -rc libcalcu.a calcu.o（r是级联，c代表创建，.a文件名必须是lib+库名）
    - **制作完成，发布.a文件和.h文件**（先放到一个文件夹中）
    - **测试使用静态库：**包含给的头文件，编译 gcc test.c -o test -lcalcu -L.（-lcalcu，到libcalcu中找，-L告诉编译器去哪里找自己写的库文件，后面跟地址，-L. 表示到当前目录下找）
- **制作动态链接库.so（对应windows中的.dll）**
    - **只编译不链接，生成.o文件**：gcc -c calcu.c -o calcu.o -fPIC（PIC-position independent code位置无关码）
    - **使用gcc命令创建动态链接库文件**：gcc -o libcalcu.so calcu.o -shared（ -shared表示使用共享库的方式链接）
    - **制作完成，发布.文件和.h文件**（先放到一个文件夹中）

```
yuki@YukisPC:~/exercise/C_advance$ gcc -o calcu.o -c calcu.c -fPIC
yuki@YukisPC:~/exercise/C_advance$ gcc -o libcalcu.so calcu.o -shared
yuki@YukisPC:~/exercise/C_advance$ mkdir dynamic
yuki@YukisPC:~/exercise/C_advance$ mv libcalcu.so calcu.h ./dynamic
```

    - **测试使用动态库：**
        - gcc test.c -o test -lcalcu -L.（**-lcalcu：链接时搜索名为calcu的库，-L. ：从当前目录找）**
        - **直接./test执行出错**，静态链接把libcalcu.a中内容复制过来后不需要依赖外物，动态链接必须依赖于libcalcu.so这个文件（运行时加载）

        - 法1：libcalcu.so放到固定目录，一般是/usr/lib，cp libcalcu.so /usr/lib（不太好，/usr/lib是系统的共享库目录，很容易搞乱）
        - 法2：**把libcalcu.so导入到环境变量LD_LIBRARY_PATH**（操作系统会先去LD_LIBRARY_PATH中找库文件，再去/usr/lib中找）
        - export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/yuki/exercise/C_advance/dynamic
        - export LD_LIBRARY_PATH= //反向设置，撤销上一步
    - **./test执行**
- **使用库函数的注意事项**：
    - 包含相应头文件
    - 调用时注意参数返回值
    - 使用第三方库-lxxx指定链接
    - 制作动态库用-L指定动态库地址
- 有用的命令
    - ldd a.out //查看可执行文件用到了哪些库or查看动态链接库能不能被正常加载（解析可执行文件，如果其中有一个库后面标有not found，则会报错）
    - nm libcalcu.a //查看文件中包含哪些.o，文件中有哪些函数

字符串库函数（面试常考，使用或实现）

- C库中的字符串处理函数在string.h中，在ubuntu系统/usr/include中
- 常用字符串处理函数（man 3查询）

memcpy、memmove（其它数组复制）

```
用法：
memcpy:直接拷贝，假定两个数组没有重叠，有重叠结果未知
memmove:有中转拷贝，若有重叠直接覆盖，效率比memcpy低，更安全

void *memcpy(void *str1, const void *str2, size_t n)//str2指向数组的前n个字符复制到str1,
void *memmove(void *str1, const void *str2, size_t n)//str2指向数组的前n个字符复制到str1

示例：
int main(void)
```

```c
{
    const char src[50]="yuki will be better";
    char dest[50];
    //char dest[10];//C不对数组边界检查，这种方式也可正常输出
    memcpy(dest,src,strlen(src)+1);
    printf("dest is:%s\n",dest);
    return 0;
}
```

strcpy、strncpy（字符数组复制）

用法：

```c
char *strcpy(char *dest, const char *src);//src中字符串复制到dest中
char *strncpy(char *dest, const char *src, size_t n)//src中字符串前n个字符复制到dest中
```

示例：

```c
int main(void)
{
    char src[20];
    char dest[100];
    strcpy(src,"hello yuki");
    printf("initial string:%s\n",src);
    strcpy(dest,src);
    printf("copied string:%s\n",dest);
    return 0;
}
```

memset（为新申请的内存初始化）

用法：

```c
void *memset(void *str, int c, size_t n)//str指向字符串开始的前n个字符每个都替换成c
```

示例：

```c
int main(void)
{
    char a[20];
    memset(a,'z',5);
    printf("the initial string:%s\n",a);
    return 0;
}
```

strcmp、strncmp、memcmp

用法：

```c
int strcmp(const char *str1, const char *str2)
int strncmp(const char *str1, const char *str2, size_t n)//前n个字符逐个比较
```

```
int memcmp(const void *str1, const void *str2, size_t n)//前n个字节逐个比较

示例:
int main(void)
{
    char str1[20]="yuki";
    char str2[20]="doris";
    int array1[20]={2,3};
    int array2[20]={1,4};

    printf("the result of strcmp(str1,str2):%d\n",strcmp(str1,str2));
    printf("the result of strncmp(str1,str2,1):%d\n",strncmp(str1,str2,1));
    printf("the result of
memcmp(array1,array2,1):%d\n",memcmp(array1,array2,2));

    return 0;
}

结果:
the result of strcmp(str1,str2):21
the result of strncmp(str1,str2,1):21
the result of memcmp(array1,array2,1):1
```

▼ strchr、memchr

```
用法:
char *strchr(const char *str, int c)//如果在字符串str中找到字符 c，则函数返回指向该字符的指针，如果未找到该字符则返回 NULL
void *memchr(const void *str, int c, size_t n)//在str指向字符串前n个字符中搜索第一次出现无符号字符c的位置

示例:
int main(void)
{
    char str1[20]="yukislab";

    if(NULL!=strchr(str1,'u'))
        printf("you find it once\n");
    if(NULL!=memchr(str1,'u',1))
        printf("you find it again\n");

    return 0;
}

结果:
you find it once
```

```
20
```

strcat、strncat

```
用法：
char *strcat(char *dest, const char *src)//返回指向目标字符串的指针
char *strncat(char *dest, const char *src, size_t n)//追加n个字符

示例：
int main(void)
{
    char str1[20]="yukis";
    char str2[20]="lab";

    strcat(str1,str2);
    printf("final string:%s\n",str1);//yukislab

    return 0;
}
```

strtok

```
用法：
char *strtok(char *str, const char *delim)//以delim为分隔符分割字符串，返回被分解的第一个子字符串

示例：
int main(void)
{
    char str[50]="yuki_is_a_lovely_girl";
    const char*delim="_";

    printf("the first word:%s\n",strtok(str,delim));

    return 0;
}

```

其它

- 编译时错误&链接时错误（ld returned 1 exit status）
  - 编译时错误：程序写的不规范造成的
  - 链接时错误：编译正确，链接时找不到

- C链接器的工作特点：链接器默认寻找几个最常用的库，有不常用的库中的函数被调用，程序员在链接时需要明确给出要扩展查找的库的名字（使用-lxxx指示链接器到libxxx.so中去寻找）