



中国科学技术大学
University of Science and Technology of China

计算机程序设计

Computer Programming



程序结构



主讲：李卫海

目录

CONTENTS

大型程序与编译过程

- 源文件
- 头文件
- 文件包含include

预处理

- 宏定义define
- 条件编译

运行时环境

名字的类型与声明

◎ 大型程序

- 常见的大型软件都是由多个源文件和库等组成
 - 源文件包括.c文件（C文件）和.h文件（头文件）
 - C文件一般包含函数定义和全局变量
 - 库由若干个源文件编译打包而成
 - 头文件包含可以在源文件、库之间共享的信息
- C文件
 - 一个程序中，有且仅能有一个C文件包含main()函数
 - 通常将完成特定功能的相关函数和变量放在同一个C文件中
 - 使程序结构清晰
 - 便于在其他程序中复用
 - 不同的C文件可以分别编译，得到目标模块



◎ 文件包含

- 为了在各个C文件之间调用函数、访问变量，可以文件包含预编译指令
- 文件包含预处理命令的一般形式：

`#include <文件名>` 或 `#include "文件名"`

- 预处理器用指定文件的内容替换该指令
- 使用 `<>` 时，预处理器直接到存放标准头文件的目录中寻找文件
- 使用 `" "` 时，预处理器首先在当前目录中查找文件，再到操作系统的path命令设置的自动搜索路径中查找，最后到标准头文件目录中查找
- 文件名本身包含路径时，则只到该路径查找
- `#include`指令的本质是替换文本，对文件名并无要求（.c、.h甚至.exe以及其他扩展名，或无扩展名都可以），使用.h只是习惯



◎ 文件包含

- 例

max.c

```
int max(int x, int y) {  
    return x>y?x:y;  
}
```

myprog.c

```
#include "max.c"  
void main() {  
    int a=3,b=4,c;  
    c=max(a,b);  
}
```

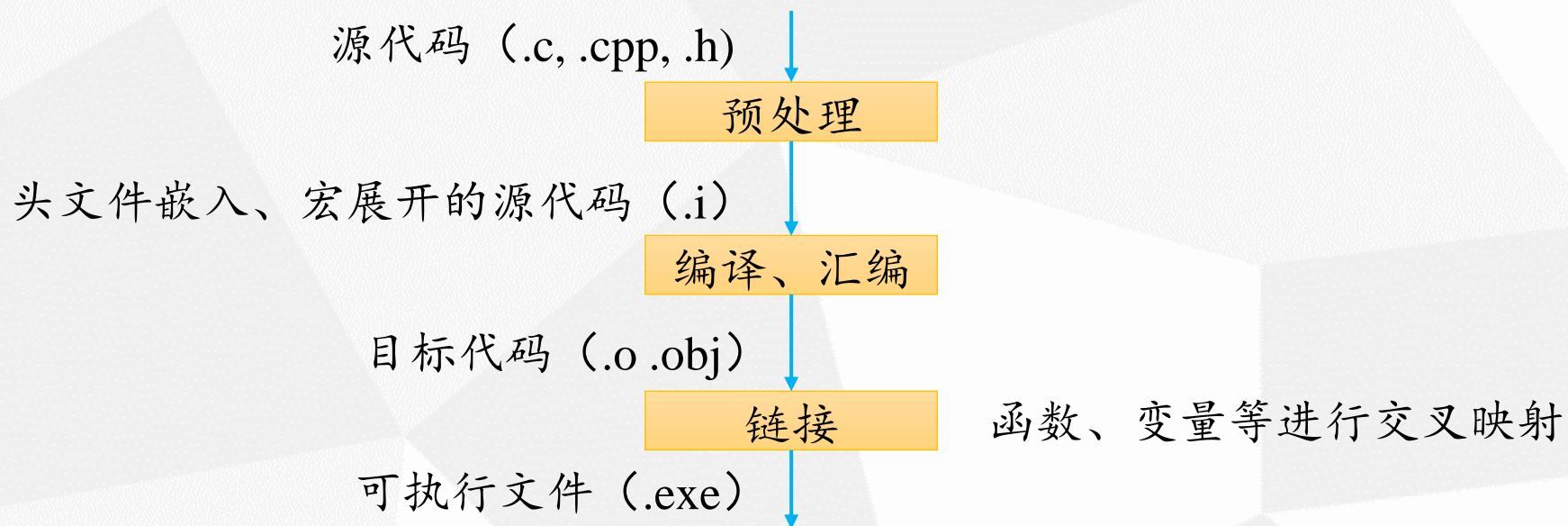
经过文件包含处理
后的新文件myprog.i

```
int max(int x, int y) {  
    return x>y?x:y;  
}  
void main() {  
    int a=3,b=4,c;  
    c=max(a,b);  
}
```

◎ 文件包含

- 文件包含指令为组装大程序或提高程序的复用度提供了一种手段
- 编写程序时，常将一些公用的常量定义、数据类型的定义和全局变量的外部声明、函数的外部声明等写在一个头文件里（常用.h作扩展名）
- 当C文件需要用到这些内容时，用#include指令将头文件包含到自己的程序中
 - 头文件中的结构类型声明会因#include出现在多个C文件中，C语言会将不同C文件中完全相同的结构类型视为名字等价
- 一个#include指令只能指定一个包含文件
- 被包含的文件内也可以出现#include指令，形成#include指令的嵌套
- 对大型程序，包含关系复杂后，可能会造成重复定义
 - 避免（直接或间接）重复包含头一个文件
 - 需要使用条件编译





◎ makefile

- makefile文件是UNIX系统发明的概念
- 描述了构成程序的文件，以及文件之间的依赖性

```
justify: justify.o word.o line.o
    gcc -o justfy justify.o word.o line.o
justify.o: justfy.c word.h line.h
    gcc -c justfy.c
word.o: word.c word.h
    gcc -c word.c
line.o: line.c line.c
    gcc -c line.c
```

描述依赖关系

待执行的命令

- 用make运行makefile时，检查每个文件的日期和时间。当所依赖的文件发生变化时，执行第二行的命令，重构目标文件或执行文件



◎ 预编译

- 预处理器一般内置于C编译器中
- C语言提供的预处理命令：文件包含、宏定义和条件编译
- 预处理是处理C源程序中所有以#开头的命令行（称为预处理命令）
- 预处理命令的语法与C语言的语法完全独立，预处理器不处理C语言语句
- 指令在第一个换行符处结束，除非用“\”明确指出要换行延续
 - 如果需要换行，可以在预处理命令行的行尾加“\”，并紧跟换行符，表示续行



◎ 宏定义

- 使用宏定义的好处

- 便于修改程序：如可以把宏体改为3.1415926即可提高PI的精度，或定义数组的大小
- 例：

```
#define N 20
```

```
void screw_mtx1(int a[][N], int m, int n) { ... }
```

```
int main() { int a[N][N]=... }
```

- 用有意义的宏名代替常量，可提高程序可读性
- 用宏名替代频繁出现的长字符序列，减少源程序的书写量
- 其它：如适合某种编程习惯



◎ 不带形参的宏

- 一般形式：`#define` 标识符 字符序列

`#define PI 3.14`

`#define STU struct student`

- 其中标识符称为宏定义名（简称宏名），字符序列称为宏体
- 宏名习惯用大写字母，以便与一般变量名区别
- 宏体仅被视为字符序列，即文本，无数据类型
- 预处理时，预处理器会将C源程序中宏定义之后出现的所有宏名都直接替换成宏体（出现在注释或字符串常量内的宏名除外），此操作称为宏展开
 - 即使宏定义在函数内，它也不仅仅是在函数内起作用，而是作用到文件末尾



◎ 不带形参的宏

- 宏体中可以引用别的宏名，如

```
#define R 18.75 //半径  
#define PI 3.1415926 //圆周率  
#define Circumference 2.0*PI*R //圆周长
```

- 同一宏名可以重复定义，如

```
#define PI 3.14  
... //此范围内使用的PI都被替换为3.14  
#define PI 3.1415926  
... //此范围内使用的PI都被替换为3.1415926
```

- 可以用`#undef` 终止宏定义的作用范围，如

```
#define N 50  
... //此范围内N被替换为50  
#undef N  
... //此范围内N无效
```



◎ 带形参的宏

- 一般形式：`#define 宏名(形参列表) 宏体`
- 左圆括号“(”必须紧随宏名之后，中间不能有空格，否则就变成不带形参的宏定义（宏体从“(”开始）
- 形参列表中可以出现多个用逗号隔开的不重名的标识符
- 宏调用的展开：分别用宏调用中的实参文本去替换宏体中对应的形参，宏体中的其它文本不变
 - 例：`#define AREA(W,H) W*H`
`AREA(a,b)` 宏展开为 `a*b`
`AREA(a+1,b+1)` 宏展开为 `a+1*b+1`

修改为：`#define AREA(W,H) (W)*(H)`
`AREA(a+1,b+1)` 宏展开为 `(a+1)*(b+1)`
- 宏展开仅是文本替换，不会进行表达式计算
- 实参可以为空，但逗号不能省。展开时替换为空串



◎ 函数与宏的区别

• 函数调用与宏调用的区别

- 函数调用是在程序运行时处理的，涉及内存单元的分配与回收以及表达式的计算等；宏调用是在预处理阶段进行的，只进行文本替换，既不分配内存单元也不计算
 - 函数调用的形参与实参都有数据类型，存在类型匹配或转换问题；宏调用仅是文本的替换，不存在类型的问题
 - 函数调用往往有返回值；宏调用没有返回值的概念
- getchar()和putchar()实际上都是定义在stdio.h中的宏，常见定义：

```
#define getchar() fgetc(stdin) //形参列表为空
```

```
#define putchar(x) fputc(x, stdout)
```



◎ 宏的例子

- 一些带参数宏定义的例子

- 例，求2个、3个、4个数值中最小值的宏定义

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

```
#define min3(x,y,z) min(min(x,y),z)
```

```
#define min4(r,s,t,u) min(min3(r,s,t),u)
```

- 例，使两个参数的值互换的宏

```
#define SWAP(type,x,y) {type temp=x; x=y; y=temp;}
```

```
double a1=2.3, a2=4.8;
```

```
SWAP(double, a1, a2);
```



◎ 宏的例子

- 例，位模式图形（二值图标）：用0或1表示该像素是“暗”或“亮”，用一串十六进制编码表示图形

- 例 unsigned char sample[] = {

0x00,

0x3c,

0x20,

0x3c,

0x20,

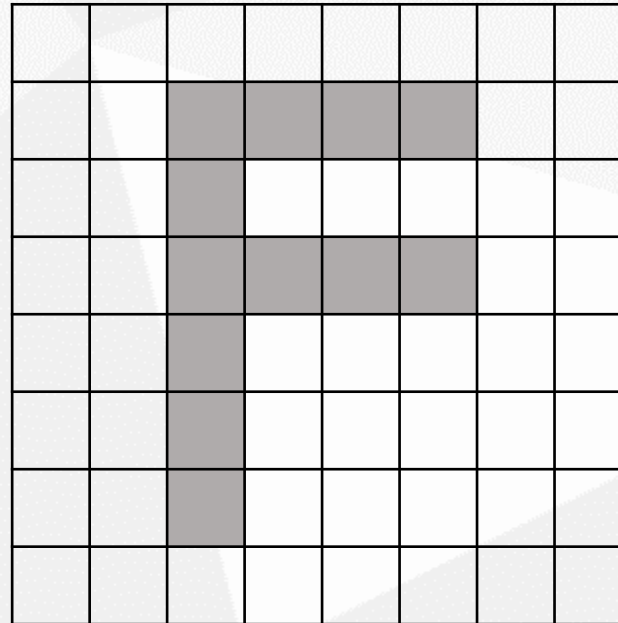
0x20,

0x20,

0x00

};

- 很不直观

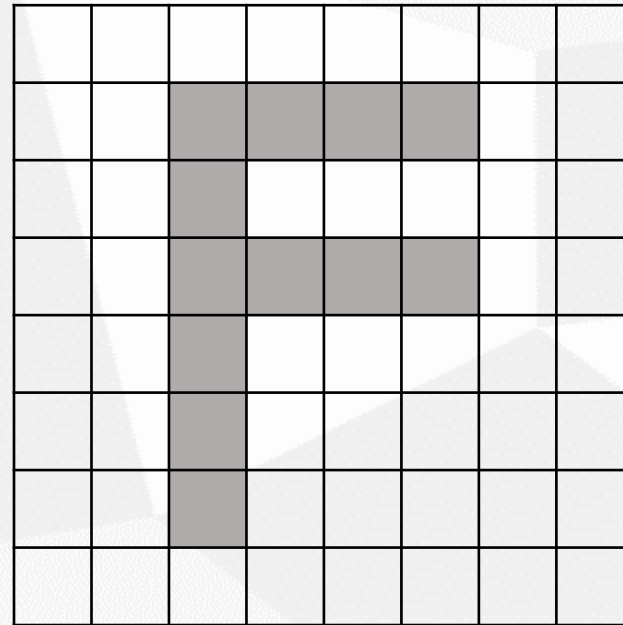


◎ 宏的例子

- 例，位模式图形（二值图标）：用0或1表示该像素是“暗”或“亮”，用一串十六进制编码表示图形

- 做如下宏定义

```
#define X )*2+1
#define _ )*2
#define s (((((((0
unsigned char sample[] = {
    s  _ _ _ _ _ _ _ _ ,
    s  _ _ X X X X _ _ ,
    s  _ _ X _ _ _ _ _ ,
    s  _ _ X X X X _ _ ,
    s  _ _ X _ _ _ _ _ ,
    s  _ _ X _ _ _ _ _ ,
    s  _ _ X _ _ _ _ _ ,
    s  _ _ X _ _ _ _ _ ,
    s  _ _ _ _ _ _ _ _ ,
};
#undef X
#undef _
#undef s
```



◎ 宏的例子

- 例，如何判断一个变量是有符号数还是无符号数？
 - 例如，ANSI C中，char型变量可以是有符号数，也可以是无符号数，由编译器设计者决定
 - 判断依据：无符号数永远不会是负的

对变量a

```
#define IsUnsigned(a) (a>=0 && ~a>=0)
```

对类型type

```
#define IsUnsigned(type) ((type)0 - 1 > 0)
```



◎ 条件编译

- 条件编译允许预处理器根据条件，选择性地传递源程序中的文本行给编译器进行处理，或者忽略
- 根据指定的标识符是否定义过

#ifdef 标识符
 程序段1
#endif

或

#ifndef 标识符
 程序段1
#else
 程序段2
#endif

- 例：

```
#define DEBUG
#ifdef DEBUG
    printf(.....);
#endif
```

//常用于调试

◎ 条件编译

- 根据指定的标识符是否 **未定义** 过

#ifndef 标识符
程序段1
#endif

或

#ifndef 标识符
程序段1
#else
程序段2
#endif

- 根据 **常量表达式** 的值是否非0

#if 常量表达式
程序段1
#endif

或

#if 常量表达式
程序段1
#else
程序段2
#endif

或

#ifdef 标识符
程序段1
#elif 常量表达式
程序段2
#else
程序段3
#endif

◎ 条件编译

- 可编写于多台机器或多种操作系统之间可移植的程序
- 可编写用不同的编译器编译的程序
- 为宏提供默认定义

```
#ifndef BUFFER_SIZE
```

```
#define BUFFER_SIZE 256
```

```
#endif
```

- 临时屏蔽包含注释/* ... */的代码

```
#if 0
```

```
    包含/* ... */的代码段
```

```
#endif
```

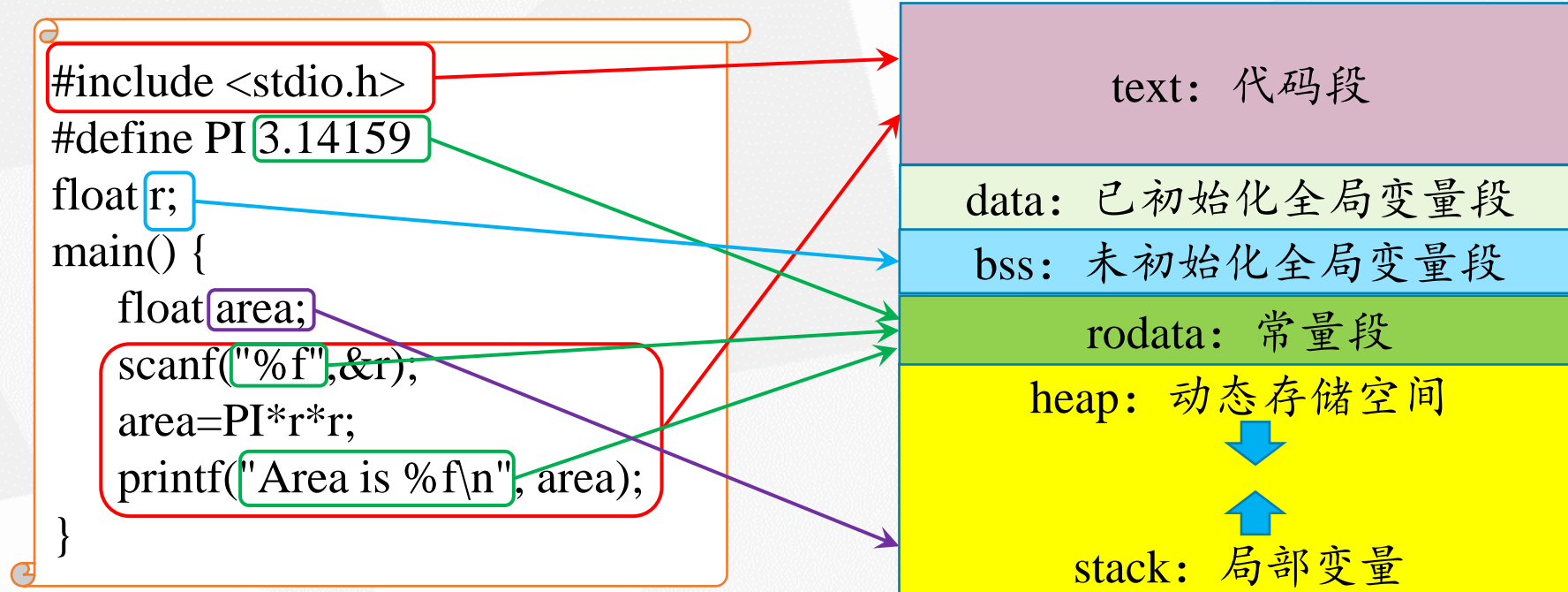
- 保护头文件以避免重复包含



◎ 运行时环境

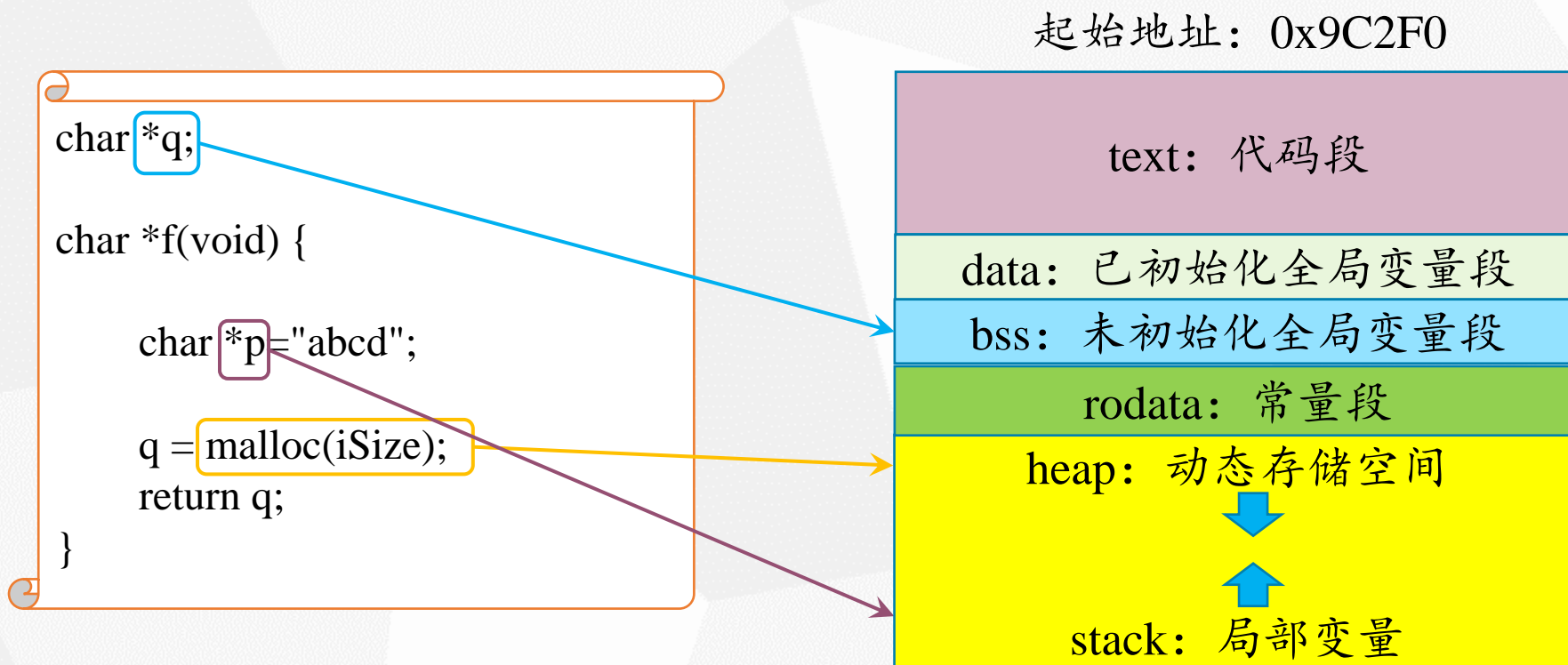
• 运行时环境简介

- 可执行程序，被操作系统从读入到从指定位置内存中运行，分为代码区和数据区
 - 可执行程序：二进制机器语言表示的代码+数据，与汇编语言相似
- 程序的各部分在内存空间中的分布，大概如下图：
- 忽略操作系统对逻辑地址到物理地址的映射



◎ 运行时环境

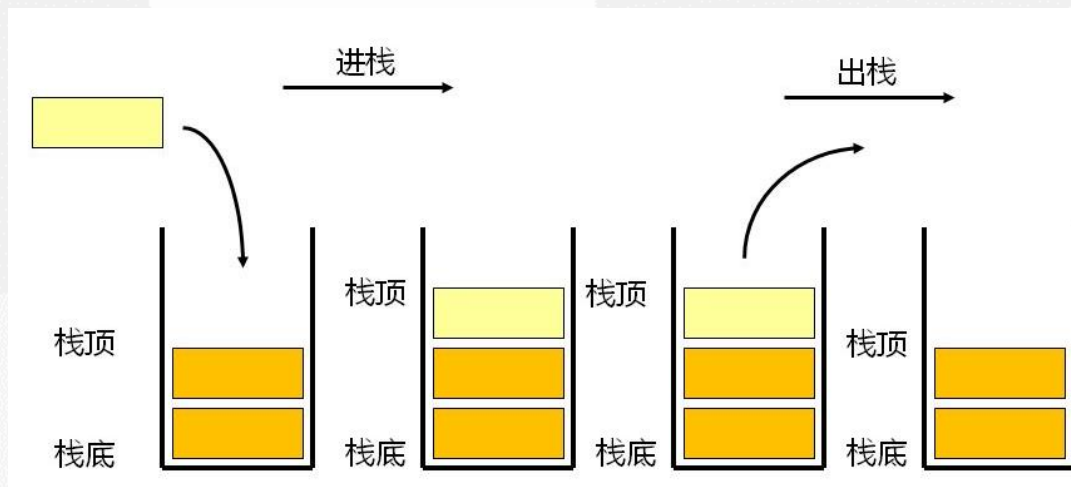
- malloc申请的存储空间分配在heap(堆)内，与stack(栈)相对
- 用于操作该空间的指针一般位于bss或stack区，偶尔也会在data区（曾初始化做它用）



◎ 运行时环境

• 函数调用过程:

- 主调函数计算实参的值
- 主调函数在栈中为被调函数**创建活动记录**，其中包括为返回值、形参和局部变量**分配存储空间**
- 主调函数将**实参值传递**/复制到活动记录中
- 程序执行的**控制流程转移**到被调函数，被调函数在需要时**读取形参值**，完成函数功能
- 被调函数将**返回值（如有的话）传递**/复制到活动记录中，将**控制流程转移**给主调函数
- 主调函数从活动记录中**读取返回值**
- 主调函数**释放活动记录**的存储空间
- 除第一步计算实参值外，其余工作均由编译器自动生成汇编代码



◎ 名字的类型

• 局部变量 (local variable)

- 在函数（包括main函数）内部定义的变量，称为局部变量
- 其作用域（源代码中静态概念，可以访问它的地方）是从定义变量的位置开始到所在函数结束
- 当函数被调用时，在栈上建立函数的活动记录，在活动记录中为该变量分配空间。当函数返回时，活动记录出栈，局部变量分配的空间被释放
- 其生存周期生存周期是函数的本次调用周期
- 分配在栈上
- 同一局部变量在不同函数调用中，对应不同存储单元，应视为不同变量
- 不同函数的局部变量可以同名
- 形式参数也可视为局部变量，只是位于活动记录中的不同位置



◎ 名字的类型

- 全局变量（global variable）

- 在函数外部定义的变量，成为全局变量
 - 其作用域是从定义变量的位置开始到所在文件结束，可被此范围内的任何函数访问
 - 程序载入内存时即分配存储空间，之后一直占用此空间，程序结束运行时才释放
 - 全局变量的生存周期为整个程序的执行周期
 - 分配在全局变量data段或BSS(Block Started by Symbol)段
 - 全局变量可以作为函数之间交换信息的通道，但占用固定空间且破坏结构化，应慎用
- 局部变量与全局变量的主要区别是作用域和生存周期



◎ 名字的类型

- 例,

```
int global_a;  
void local_1() {  
    char local_1_c;  
}  
int local_2() {  
    char local_2_c;  
    local_1();  
}  
main() {  
    int main_i;  
    for (main_i=0; main_i<3; main_i++)  
        local_2();  
}
```

BSS: `global_a`

local_1()
`local_1_c`

local_2()
`local_2_c`

main()
`main_i`

STACK:

在任一时刻，都能且仅能访问当前（栈顶）活动记录中的局部变量，和全局变量

◎ 名字的类型

- 局部变量可以和外部变量重名，但应注意作用域的变化
 - 出现重名时，访问的是局部变量

全局变量a
的作用域



局部变量a
的作用域



全局变量a
的作用域

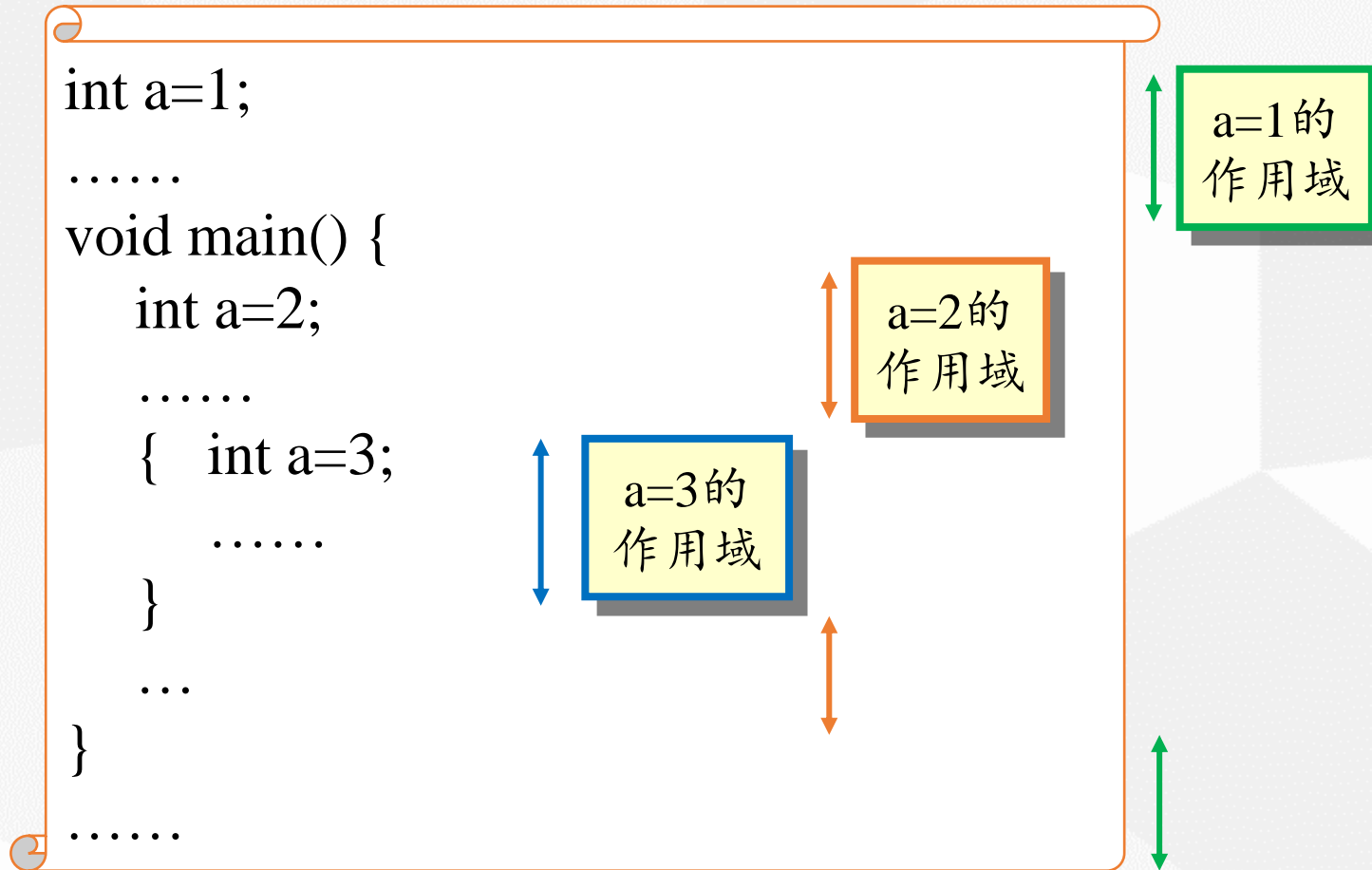


```
#include<stdio.h>
int a;
void test1() {
    int a=2;
    printf("test1_a=%d",a);
}
void main() {
    printf("global_a=%d,",a);
    test1();
}
```

输出结果：
global_a=0,test1_a=2

◎ 名字的类型

- C99允许在任意位置声明变量（C90允许在复合语句开始声明变量），局部的概念进一步拓展



◎ 名字的类型

• 静态存储与动态存储

- 静态存储指在程序运行期间为变量分配固定的存储空间（如全局变量段），无固定存储空间则为动态存储（栈上或堆上）
- 静态存储的变量简称静态变量；动态存储的变量简称动态变量
- C语言对全局变量采取静态分配策略，而对局部变量采取默认动态分配策略
- 当需要定义静态局部变量时，可使用关键字static

例如：static int a;

- 静态局部变量分配在BSS段，赋初值动作在编译时执行，且只执行一次，而不论函数被调用几次。未显示赋初值的，系统会自动用0或'\0'初始化
- 静态局部变量的作用域仍限制在定义它的函数内
 - 因此不同函数内的静态局部变量可以同名，编译器会对它们进行区分
- 静态局部变量的生存周期为整个程序的执行周期



◎ 名字的类型

- 静态局部变量有什么用？
 - 当函数退出时仍需要保留其值的时候

```
void auto_static() {  
    int var_auto=0;  
    static int var_static=0;  
    printf("var_auto is %d—",var_auto++);  
    printf("var_static is %d\n",var_static++);  
}  
main() {  
    int i;  
    for (i=0;i<5;i++) auto_static();  
    exit(0);  
}
```

程序运行结果：

var_auto is 0—var_static is 0
var_auto is 0—var_static is 1
var_auto is 0—var_static is 2
var_auto is 0—var_static is 3
var_auto is 0—var_static is 4



◎ 名字的类型

- C语言产生之初，由于编译技术的限制，采用了较为简单的语言方案，包括要求函数内不可再定义另一个函数，即**函数定义不可嵌套**
 - 因而函数没有局部函数和全局函数之分
- 但**允许嵌套调用函数**
- 若函数直接或间接地调用自身，称为**递归调用**
- 递归调用时，函数被多次（逐层）调用，直到满足某个条件后，逐层返回（回溯）



◎ 名字的类型

• 例，阶乘

```
#include<stdio.h>
long power(int n) {
    long f;
    if (n>1)
        f=power(n-1)*n;
    else
        f=1;
    printf(“%d!=%ld\n”,n,f);
    return(f);
}
void main() {
    int n;
    long y;
    printf(“Please input the number:”);
    scanf(“%d",&n);
    y=power(n);
    printf(“%d!=%ld\n”,n,y);
}
```

程序运行结果?

Please input the number:5

1!=1

2!=2

3!=6

4!=24

5!=120

5!=120

power(n=1)
f=1

power(n=2)
f=2

power(n=3)
f=6

power(n=4)
f=24

power(n=5)
f=120

main()
n=5
y=120

◎ 名字的类型

- 例，接收键盘输入的字符流，并逆序输出

```
#include<stdio.h>
void palin() {
    char next;
    next=getchar();
    if (next=='\n') {
        putchar(next);
        printf("output:\t");
    }
    else {
        palin();
        putchar(next);
    }
}
void main() {
    printf("Input:\t");
    palin();
    printf("\n");
}
```

将最后一个回车符输出

先处理后面输入的字符，
然后再输出当前字符

递归调用palin函数，将当前输入的字符
保存在栈中，直至遇到'\n'，开始逐层回
溯，将栈中字符逆序输出

◎ 名字的类型

- 当一个程序由多个源文件(.c文件)组成时，可以指定一个文件内的函数能被其它文件调用，称为**外部函数**，函数声明和定义时冠以extern
 - 例：定义的文件中 `extern double func1(double x) {...}` //声明时可以缺省extern
声明的文件中 `extern double func1(double x);`
- 也可以指定该函数只能被本文件内的其它函数调用，称为**内部函数**，函数声明和定义时冠以**static**
 - 例： `static double func2(double x) {...}`
- 内部函数的好处是
 - 在不同的文件中可以定义相同名称的函数，避免大型应用程序的开发中出现命名冲突
 - 程序员可以放心修改，不用担心影响其它文件的调用



◎ 名字的类型

- 例：

```
#include<stdio.h>
long power(int n) {
    long f;
    if (n>1)
        f=power(n-1)*n;
    else
        f=1;
    printf(“%d!=%ld\n”,n,f);
    return(f);
}
```

```
#include<stdio.h>
extern long power(int n);
void main() {
    int n;
    long y;
    printf(“Please input the number:”);
    scanf(“%d”,&n);
    y=power(n);
    printf(%d!=%ld\n”,n,y);
}
```

- 注意区分声明外部函数与文件包含的区别



◎ 名字的类型：总结

- 内部、外部，全局、局部都是所谓的**作用域**说明
 - **函数**只有**内部**、**外部**之分，static表示内部而非静态
 - **变量**也有**内部**、**外部**之分（其方法与函数一样），限制变量是否可以跨文件访问
 - **变量**还有**全局**、**局部**之分
- 静态、动态是指**存储位置**和**生存周期**的差异
 - 局部变量有**静态**、**动态**之分
 - 全局变量都是静态的



◎ 外部变量的链接

- 链接时，外部变量的地址将映射到定义该变量的文件给该变量分配的地址
- 当有多个文件都定义了重名的外部变量时，如何处理？
 - 不允许有多个已初始化的同名外部变量
 - 当有一个已初始化的外部变量，和一个以上未初始化的同名外部变量时，链接中选择已初始化的外部变量
 - 当有多个未初始化的同名外部变量时，结果不确定
- 例，

两个C文件link1.c和link2.c的内容分别如下

```
int buf[1] = { 100};
```

和

```
extern int *buf;  
main() { printf("%d\n", *buf); }
```

在X86/Linux经命令cc link1.c link2.c编译后，运行时产生如下的出错信息

Segmentation fault (core dumped)



◎ 外部变量的链接

```
int buf[1] = { 100};
```

和

```
extern int *buf;  
main() { printf("%d\n", *buf); }
```

- 两个程序独立编译时没有问题
- 链接时不检查名字的类型
 - 编译后目标代码中已没有类型信息，只有名字及其地址
 - 虽对两个目标代码对buf的类型持不同观点，但能连接成目标程序
- 链接时将两个名字映射到同一地址。第一个文件中的buf是已初始化的外部变量，使用它的地址。是静态变量，buf指向的内存单元值为100
- 运行时，取*buf的值就是取地址为100的单元的内容。该地址不在合法的可访问区域，报错



◎ 关于声明

- 格式：声明说明 声明符；
- 声明符是变量名、函数名及相关信息（指针、数组、形参...）
- 声明说明包括
 - 存储类型：auto、static、extern、register
 - 类型限定符：const、volatile、restrict（C99）
 - 类型说明符：void、int、char、.....
 - 函数说明符：inline（C99）



◎ 存储类型

- 变量的存储类型

- 自动的 `auto`

- 例, `auto int a;`
 - 局部变量缺省都是自动的

- 静态的 `static`

- 例, `static int a;`
 - 全局变量都是静态的。局部变量可以声明为静态的

- 寄存器的 `register`

- 例, `register int a;`
 - 变量优先存储在寄存器中

- 外部的 `extern`

- 例, `extern int a;`
 - 声明变量a在其它文件中, 链接时进行拼装



◎ 存储类型

- 函数的存储类型

- 外部的 `extern`

- 例, `extern int fun();`
 - 缺省值
 - 该函数定义在其它文件中, 或该函数可以被其它文件调用

- 静态的 `static`

- 例, `static int fun();`
 - 该函数只能在本文件中被调用

- 函数说明符: `inline`

- 编译器尝试将该函数的函数调用改为嵌入函数的机器指令来实现, 节省函数调用过程的时间



◎ 类型限定符

- **const**

- 声明对象是只读的，不可修改的
- 编译器会在编译时检查相关对象是否被修改，如有会报错
- 具有文档功能，提示程序员该值不会被改变

- **volatile**

- 声明对象的值是易变的
- 编译器不会优化删除对该对象的取值运算
- 常用于多线程、采集外界数据等场合

- **restrict**

- 用于修饰指针，声明对象的值只能被该指针修改，不能被别名修改（但不确保）



◎ 解释复杂声明

- 理解C语言声明的规则

- A. 声明从它的名字开始，按照优先级顺序一次解读

- B. 优先级从高到低依次是：

- B.1 声明中被括号括起来的部分

- B.2 后缀()表示这是一个函数，后缀[]表示这是一个数组

- B.3 前缀*表示这是一个“指向...的指针”

- C. 若const或volatile的后面紧跟着类型说明符，那么它作用于类型说明符；其他形况下，作用于它左边紧邻的指针星号



◎ 解释复杂声明

- 例, `char * const *(*next)();`

剩余的声明	已分析的声明	结果
<code>char * const *(*)();</code>	<code>next</code>	<code>next</code> 是...
<code>char * const *()();</code>	<code>* next</code>	<code>next</code> 是指向...的指针
<code>char * const * ();</code>	<code>(* next)</code>	
<code>char * const * ;</code>	<code>(* next)()</code>	<code>next</code> 是指向函数的指针, 该函数返回...
<code>char * const ;</code>	<code>*(* next)()</code>	<code>next</code> 是指向函数的指针, 该函数返回指向...的指针
<code>char * ;</code>	<code>const *(* next)()</code>	<code>next</code> 是指向函数的指针, 该函数返回指向只读的...的指针
<code>char ;</code>	<code>* const *(* next)()</code>	<code>next</code> 是指向函数的指针, 该函数返回指向只读的指向...的指针的指针
	<code>char * const *(* next)();</code>	<code>next</code> 是指向函数的指针, 该函数返回指向只读的指向 <code>char</code> 的指针的指针

- 软件工程的目的是控制复杂度，而不是增加复杂性。 - Dr. Pamela Zav
- 编程是“程序员努力建一个更大更傻的程序”和“世界正在尝试创造更多更傻的人”之间的一种竞赛，目前为止，后者是赢家。 - Rich Cook
- 用代码行来衡量开发进度，无异于用重量来衡量制造飞机的进度。 - Bill Gates



A22. 编写一个宏来计算一维数组a中元素的个数。

A23. 定义一个带两个参数的宏，功能是将两个参数的值互换。编写程序，在主程序中输入两个数（类型相同）作为使用宏的实参，输出已交换后的两个值。

- 努力使你的宏可以适应各种类型的数据
- 数据的值可能很大或很小，接近该类型数的上下限

A24. 定义一个宏MyLpha(c)，以判断c是大写字母还是小写字母。当c是小写字母时，宏调用取值为1，当c是大写字母时，宏调用取值为0。编程应用此宏定义，实现将输入流中的大写字母转换为小写字母，小写字母转换为的大写字母，并输出转换后的结果。

