计算机程序设计

Computer Programming

编写大型程序

主讲：吴锋

目录
CONTENTS

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，文件可包含预编译指令
- 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

编译过程

源代码（.c, .cpp, .h）

预处理

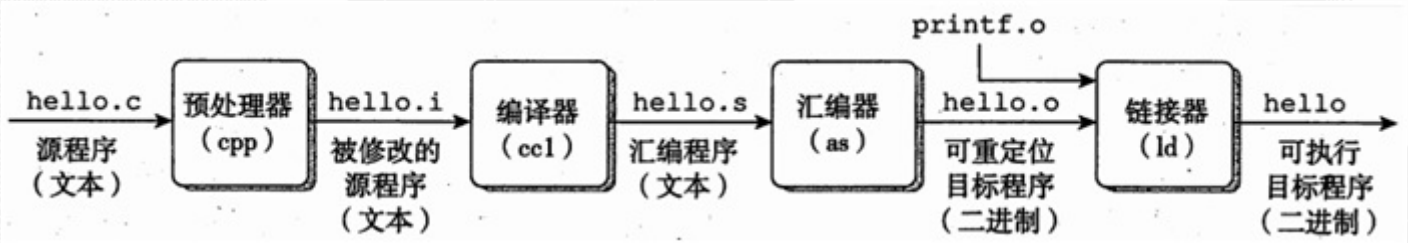
头文件嵌入、宏展开的源代码（.i）

编译、汇编

目标代码（.o .obj）

链接　　函数、变量等进行交叉映射

可执行文件（.exe）



GCC编译系统

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 justify.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
- 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）
- 指令在第一个换行符处结束，除非用“\”明确指出要换行延续

- 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
- 为区别源程序中的C代码行与预处理命令行，所有预处理命令行都以#开头
- 预处理命令遇到换行符就会结束。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。例如：

```
#define N 20
void screw_mtrx1(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)
```

思考：int a=5, b = 10; c = min(a++, b++); c 的值是多少？

  - 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

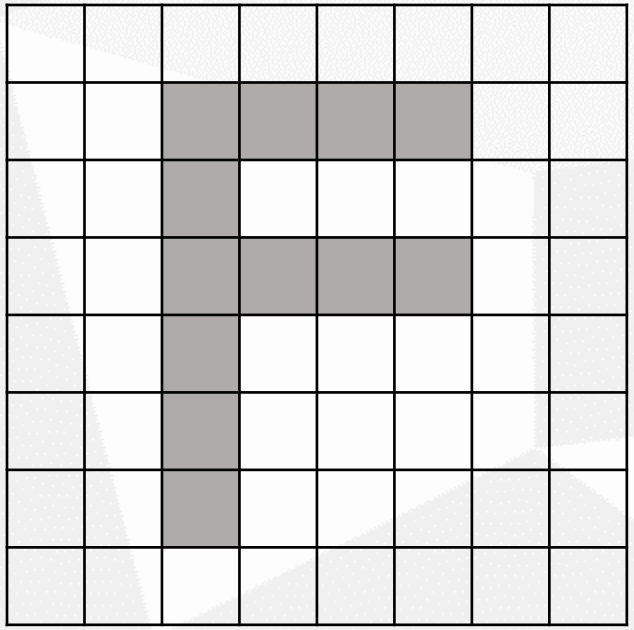
- 定义交换两个整数的宏：SWAP(a, b)
  - 方法一：#define SWAP(a, b) a = a+b; b = a–b; a = a–b;
    - SWAP(x, y); // OK
    - if (x < 0) SWAP(x, y); // ERROR
      - 替换的结果：if (x< 0) x= x+y; y=x–y; x = x–y;
  - 方法二：#define SWAP(a, b) { a = a+b; b = a–b; a = a–b; }
    - if (x < 0) SWAP(x, y); // OK
      - 替换的结果：if (x< 0) { ... };
    - if (x < 0) SWAP(x, y); else SWAP(x, z); // ERROR
      - 替换的结果：if (x< 0) { ... }; else { ... };
  - 方法三：#define SWAP(a, b) do { a = a+b; b = a–b; a = a–b; } while(0)

◎ 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
};
```

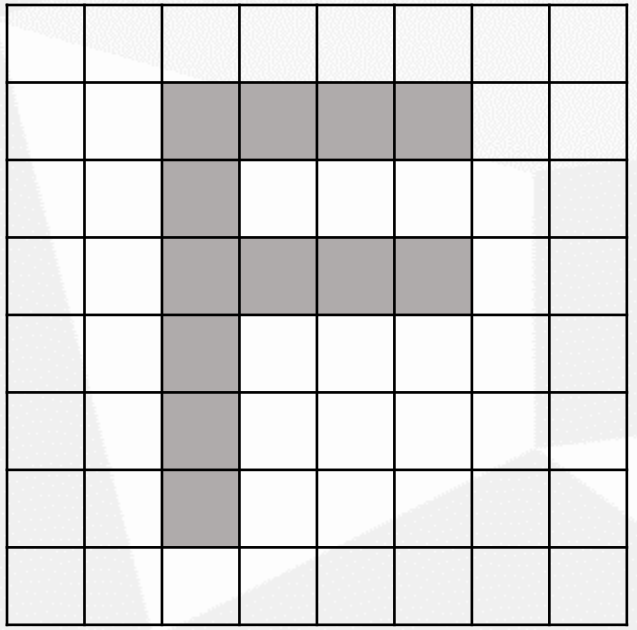
  - 很不直观



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 _ _ _ _ _ _ _ _
};
#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

```
#ifdef 标识符
    程序段1
#else
    程序段2
#endif
```

◦ 例：#define DEBUG　//常用于调试

```
#ifdef DEBUG
    printf(……);
#endif
```

◎ 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

- 可编写子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

头文件：hello.h

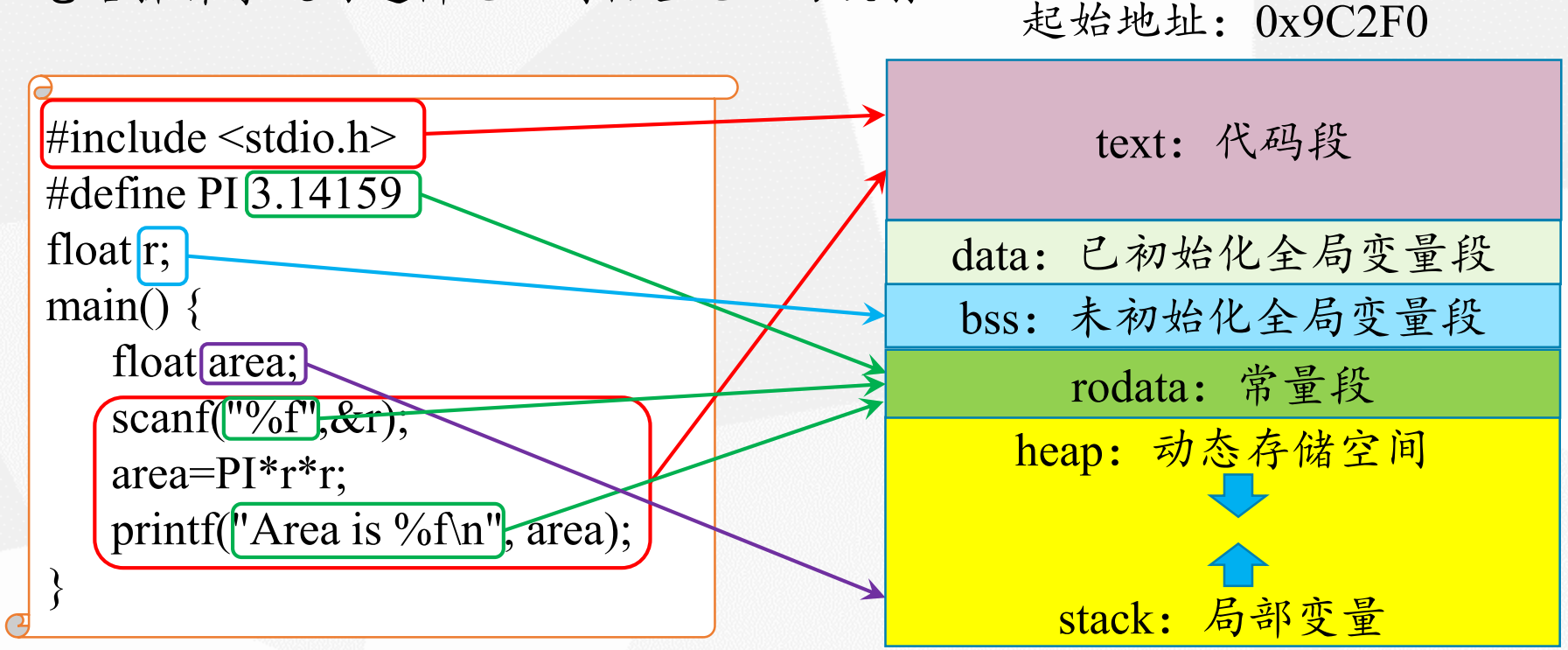
```
#ifndef _HELLO_H_
#define _HELLO_H_
// 增加上述宏定义
// 避免头文件被重复包含

// 头文件的内容
... ...

#endif
```

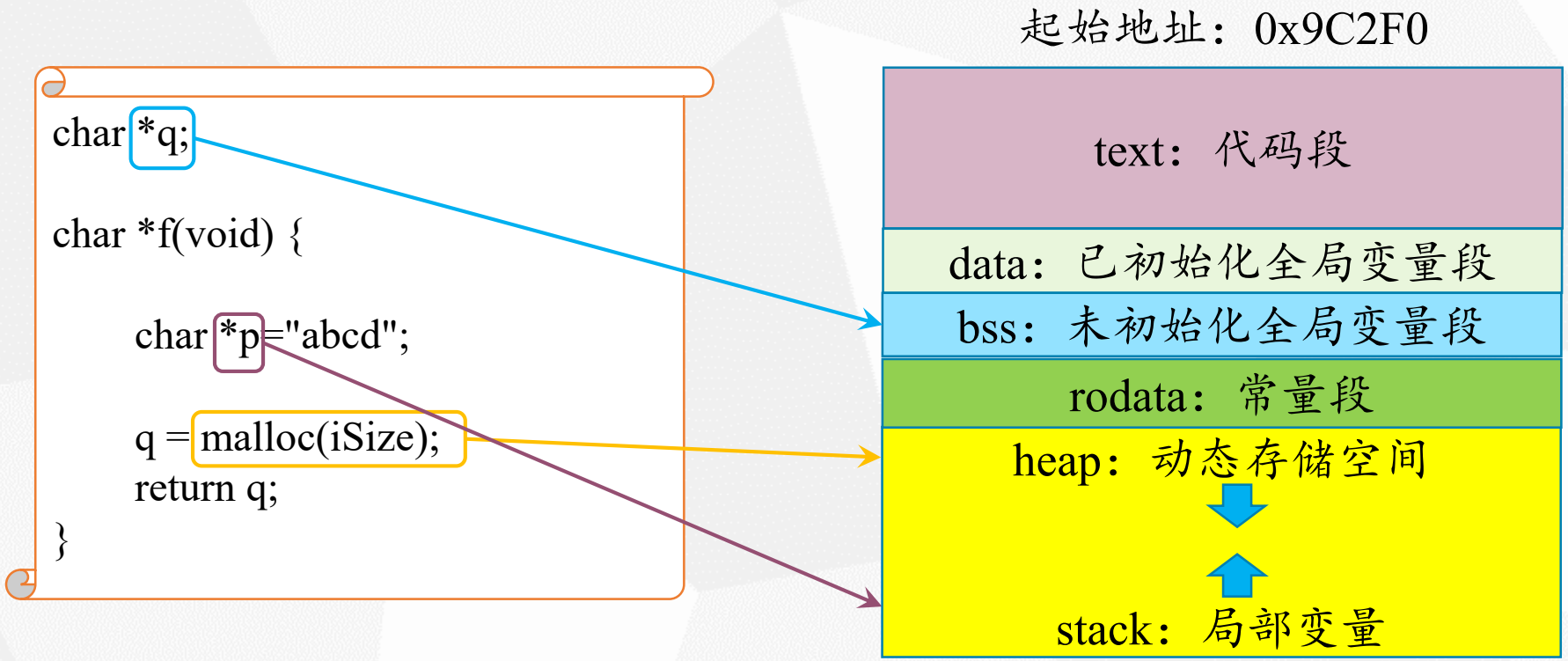
◎ 运行时环境

- 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



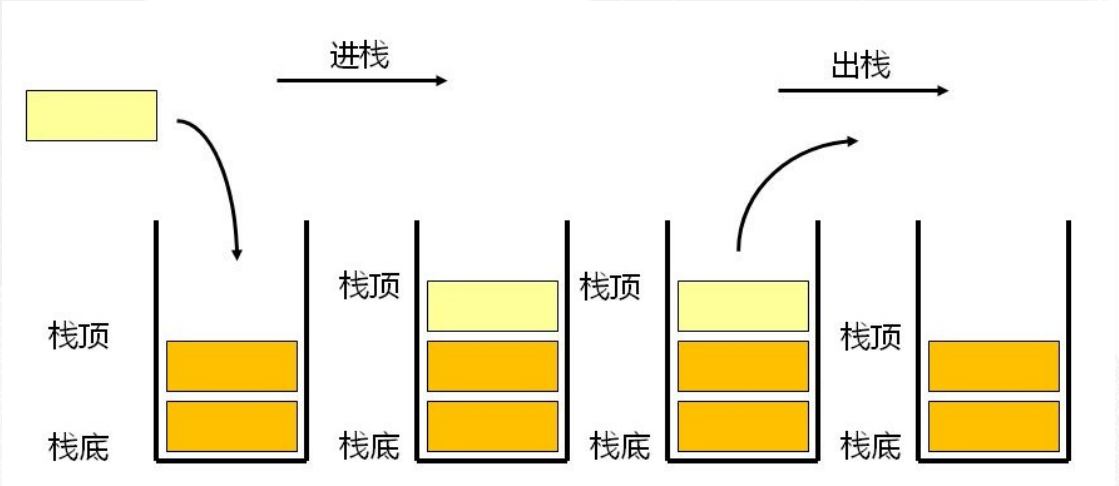
运行时环境

- 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

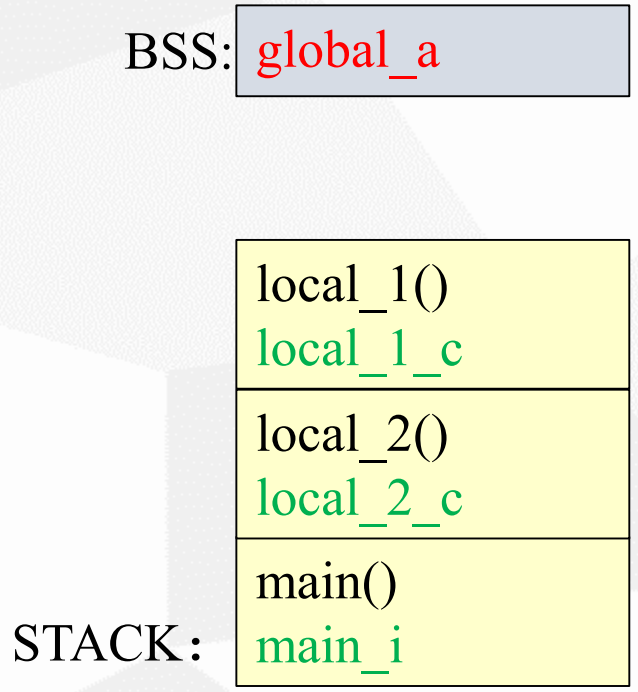
- 全局变量（global variable）
  - 在函数外部定义的变量，成为全局变量
  - 其作用域是从定义变量的位置开始到所在文件结束，可被此范围内的任何函数访问
  - 程序载入内存时即分配存储空间，之后一直占用此空间，程序结束运行时才释放
  - 全局变量的生存周期为整个程序的执行周期
  - 分配在全局变量data段或BSS(Block Started by Symbol)段
  - 全局变量可以作为函数之间交换信息的通道，但占用固定空间且破坏结构化，应慎用

- 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：

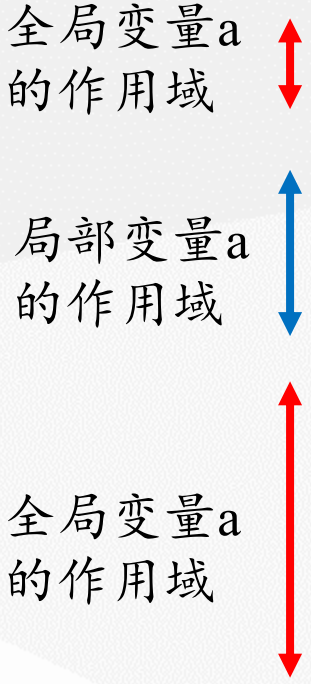
```
int global_a;
void local_1() {
    char local_1_c;
}
int local_2() {
    char local_2_c;
    local_1();
}
main() {
    int main_i;
    for (main_i=0; main_i<3; main_i++)
        local_2();
}
```



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

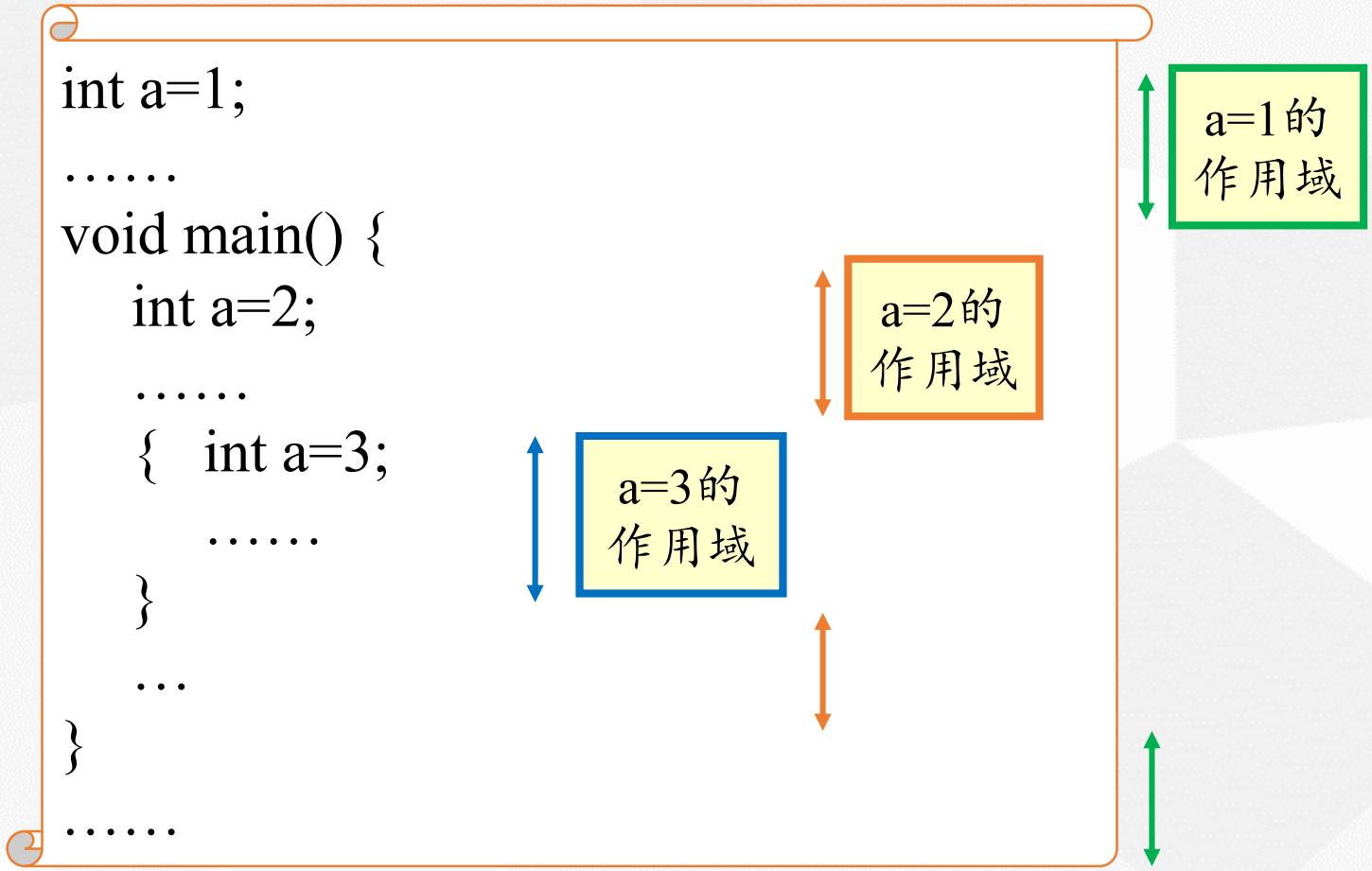



```
#include<stdio.h>
int a;
void test1() {
    int a=2;
    printf("test1_a=%d",a);
}
void main() {
    printf("global_a=%d,",a);
    test1();
}
```

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
  - 编译器尝试将对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
char * const *(*)();	next	next是...
char * const *( )();	* next	next是指向...的指针
char * const * ();	(* next)	
char * const * ;	(* next)()	next是指向函数的指针，该函数返回...
char * const ;	*(* next)()	next是指向函数的指针，该函数返回指向...的指针
char * ;	const *(* next)()	next是指向函数的指针，该函数返回指向只读的...的指针
char ;	* const *(* next)()	next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
	char * const *(* next)();	next是指向函数的指针，该函数返回指向只读的指向char的指针的指针

常用的库函数

- 输入输出（I/O）库：#include <stdio.h>
  - 如：scanf, printf, getchar, putchar, gets, puts 等
- 数学运算法库：#include <math.h>
  - 如：sin, cos, tan, log, sqrt, pow, fabs 等
- 字符串处理库：#include <string.h>
  - 如：strlen, strcmp, strcat, strcpy, strstr 等
- 函数可变参数库：#include <stdarg.h>
  - 如：va_list, va_start, va_arg, va_end 等
- 标准常用函数库：#include <stdlib.h>
  - 如：atof, atoi, malloc, calloc, realloc, free, qsort, bsearch 等

◎ 常用的库函数

- 字符处理库：#include <ctype.h>
  - 如：isdigit, isalpha, isprint, islower, isupper, tolower, toupper 等
- 整形变量的表示范围：#include <limits.h>
  - 字符类型（char）：CHAR_MIN，CHAR_MAX, UCHAR_MAX
  - 整数类型（int）：INT_MIN, INT_MAX, UINT_MAX
  - 长整数类型（long）：LONG_MIN, LONG_MAX, ULONG_MAX
- 浮点型变量的表示范围：#include <float.h>
  - 最大值：FLT_MAX, DBL_MAX, LDBL_MAX
  - 精度：FLT_EPSILON, DBL_EPSILON, LDBL_EPSILON

◎ 常用的库函数

- 伪随机（Pseudo-Random）数生成：#include <stdlib.h>
  - int rand(void)：返回 0～RAND_MAX 之间的一个随机整数
    - RAND_MAX 是一个不小于 32767 的常数
    - 例：rand() % 100，返回 0～100 之间的随机整数
    - 例：10 + rand() % 10，返回 10～20 之间的随机整数
    - 例：(double)rand()/(double)RAND_MAX，返回 0～1 之间的随机浮点数
    - 例：a + (b – a)*(double)rand()/(double)RAND_MAX，返回 a～b 之间的随机浮点数
  - void srand(unsigned int seed)：设定伪随机生成函数rand的种子
    - 随机种子确定后，在同一台机器生成的随机序列保持确定。
    - 如：srand(0)，rand()%100 的序列确定为：83 86 77 15 93 35 86 92 49 21 ... ...
    - 为保证随机序列不同，通常采用时间作为随机种子: srand((unsigned) time(NULL));

◎ 常用的库函数

- 运行时间计时：#include <time.h>
  - time() : 返回从1970/01/01 00:00:00到此刻所经过的秒数（GMT）。
    - 可移植性好，性能稳定
    - 精度较低，只能精确到秒

```
time_t start, end;
start = time(NULL); //or time(&start);
// 要计时的程序段
end = time(NULL);
printf("time=%d\n", difftime(end, start));
```

  - clock() :返回值是CPU时钟计数，要换算成秒，需要除以CLK_TCK
    - 可以精确到毫秒
    - 受硬件影响，可能不稳定

```
clock_t start, end;
start = clock();
//  要计时的程序段
end = clock();
printf("time=%f\n", (double)(end-start)/CLK_TCK);
```

◎ 常用的库函数

- 调试断言： #include <assert.h>
  - void assert(int expr)：如果表达式expr的值为假(即为0),那么它就先向stderr打印一条出错信息，然后通过条用abort来终止程序；否则什么也不做
    - 使用assert时，断言的内容（表达式）要明确，尽量一项内容写一行
    - 在调试程序时，在怀疑有问题的地方插入断言，可阻断Bug的扩散

```
int func(int x, int y,) {
    assert (x < a1 && x > b1); // 函数执行前，关于参数 x 的断言
    assert (y < a2 && y > b2); // 函数执行前，关于参数 y 的断言
    ...
    assert ( ... ); // 函数执行中，关于临时变量的断言
    ...
    assert (z < a4 && z > b4); // 函数执行后，关于返回值 z 的断言
    return z;
}
```

◎ 常用的库函数

- 调试断言： #include <assert.h>
  - void assert(int expr)：如果表达式expr的值为假(即为0),那么它就先向stderr打印一条出错信息，然后通过条用abort来终止程序；否则什么也不做
    - 使用assert时，频繁的调用会影响程序的性能，增加额外的开销
    - 在调试结束后，可以通过插入 #define NDEBUG 来禁用assert调用

```
#ifdef NDEBUG
    // 如果NDEBUG的宏在程序中被定义，则assert定义为空（即什么也不做）
    #define assert(e) ((void)0)
#else
    // 否则，如果表达式e为假，则终止程序，打印出问题的文件和行号
    #define assert(e) \
            ((void) ((e) ? ((void)0) : __assert (#e, __FILE__, __LINE__)))
#endif
```

◎ 代码版本控制 *

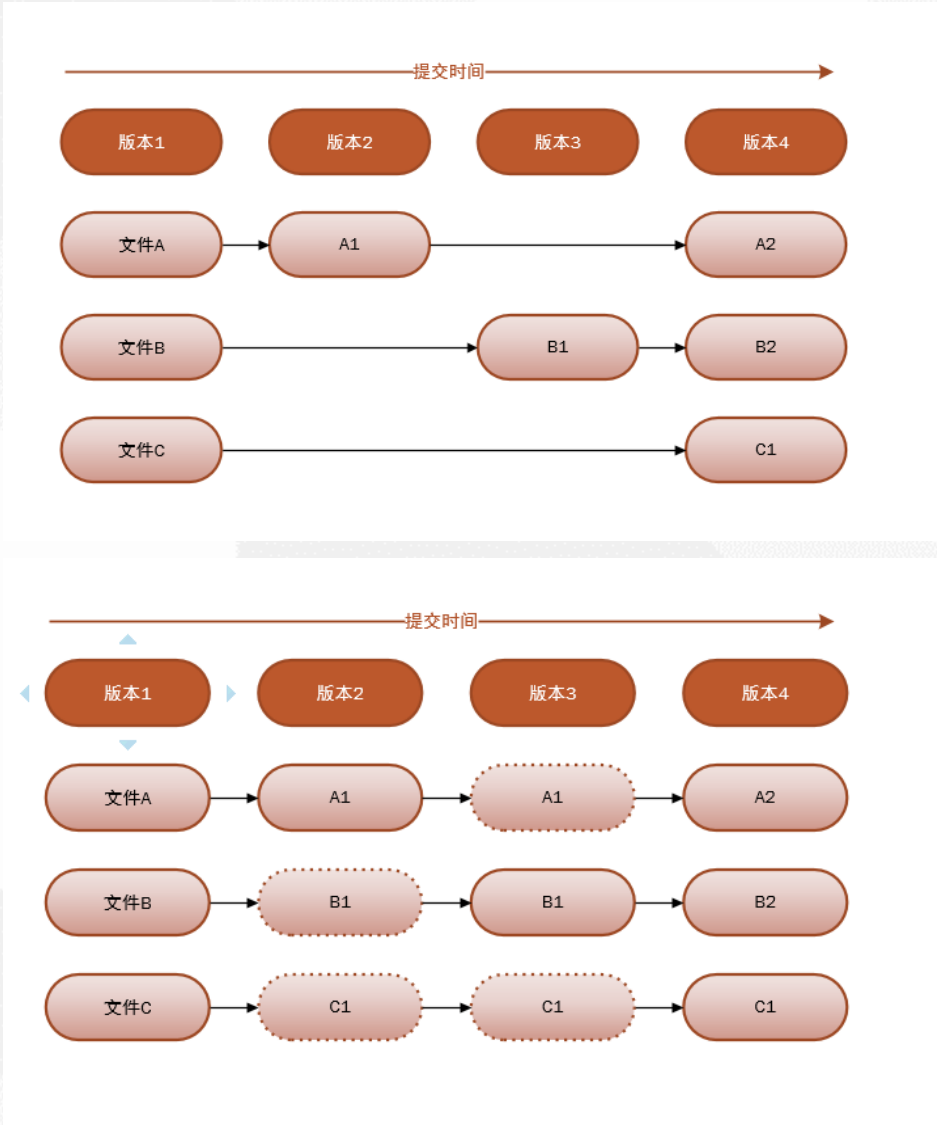
- 版本控制（Version Control System，VCS）是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。
- 版本控制可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态。
- 版本控制可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因。
- 在团队开发中使用版本控制系统的好处
  - 作为数据备份，防止重要数据意外丢失
  - 避免版本管理混乱，可以同时维护多个版本
  - 提高代码质量，记录代码修改的历史信息
  - 明确分工责任，提高协同、多人开发时的效率

◎ 代码版本控制 *

- 版本控制系统的类型
  - 本地版本控制系统（个人本地使用，无法多人协作）
    - 如：RCS的工作原理是在硬盘上保存补丁集(补丁是指文件修订前后的变化)通过应用所有的补丁，可以重新计算出各个版本的文件内容。
  - 集中化的版本控制系统（需要联网，容易出现单点故障）
    - 如：CVS、SVN等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。
  - 分布式版本控制系统（当前使用最多的版本控制系统）
    - 如：Git、Mercurial、Bazaar 等，客户端并不只提取最新版本的文件快照，而是把代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。因为每一次的克隆操作，实际上都是一次对代码仓库的完整备份。

代码版本控制 *

- Git 是 Linux 发明者 Linus 开发的一款分布式版本控制系统，是目前最为流行和软件开发着必须掌握的工具。
- Git 是一个分布式版本控制系统，保存的是文件的完整快照，而不是差异变换或者文件补丁。保存每一次变化的完整内容。
- Git 每一次提交都是对项目文件的一个完整拷贝，因此可以完全恢复到以前的任何一个提交。
- Git 每个版本只会完整拷贝发生变化的文件，对于没有变化的文件，只会保存一个指向上一个版本的文件的指针，即对一个特定版本的文件，只会保存一个副本，但可以有多个指向该文件的指针。



◎ 代码版本控制 *

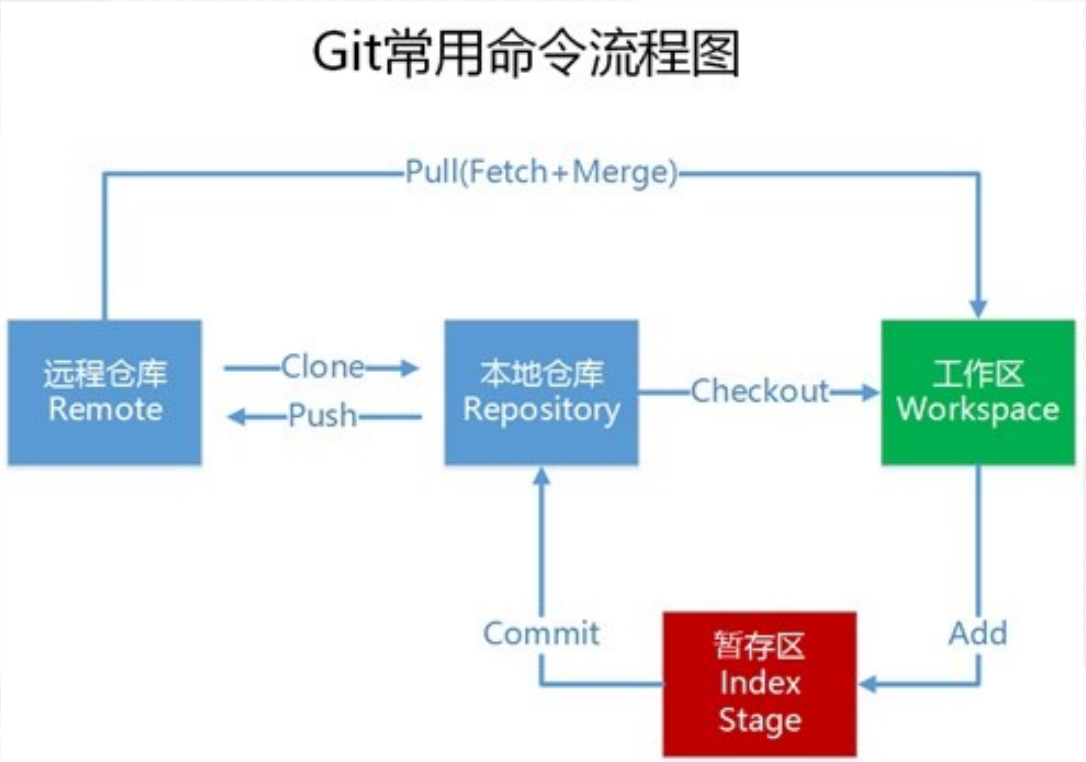
- Git 基本命令
  - 从远程仓库将项目clone到本地；

    $ git clone http://github.com/xxx.git
  - 在本地工作区进行开发：增加、删除或者修改文件；
  - 将更改的文件add到暂存区域；

    $ git add file.c
  - 将暂存区的更新commit到本地仓库；

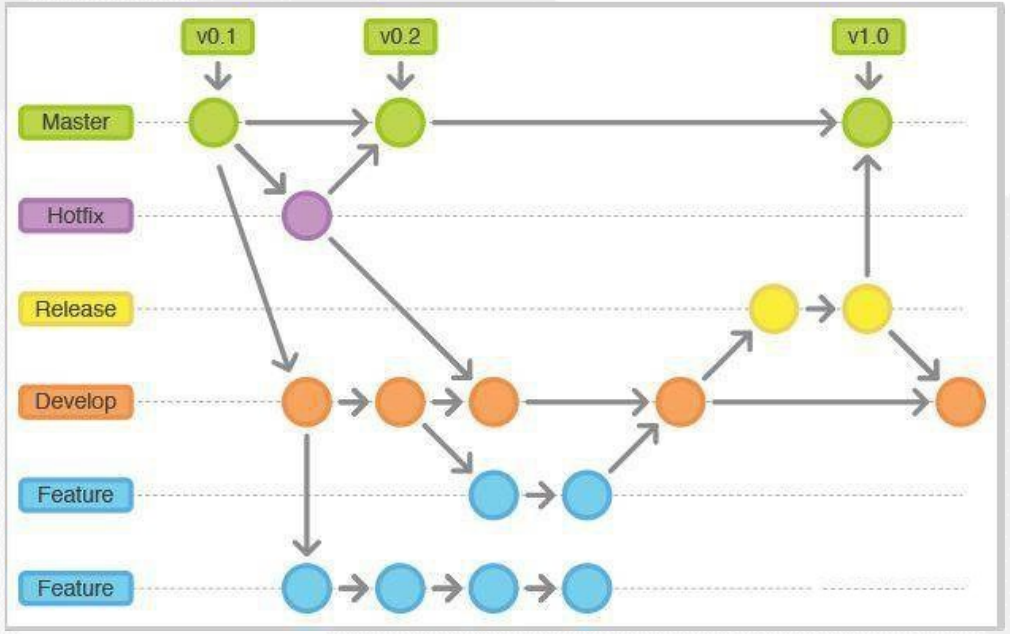
    $ git commit –a –m "Add file.c"
  - 将本地仓库push到服务器。

    $ git push



◎ 代码版本控制 *

- Git 的进阶使用
  - Git 完整命令手册地址：http://git-scm.com/docs
  - 全球最大的Git 仓库：http://github.com
  - 科大Git仓库：http://git.ustc.edu.cn



作业

1. 编写一个宏 ARRAY_SIZE(a) 来计算一维数组a中元素的个数。

2. 定义一个带参数的宏，功能是将两个参数的值互换。要求能够尽量支持多种数据类型，包括字符型、整型、浮点型、结构体、甚至字符串。

3. 定义一个宏，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。

注：写在作业本

◎ 课外练习

- 国内常用的OJ网站：
  - USTC （题数 1400+）：http://acm.ustc.edu.cn/
  - PKU（题数 4000+）：http://poj.org/
  - HDU（题数 7100+）：http://acm.hdu.edu.cn/
- C语言名题精选百则+技巧篇 （冼镜光编着）
  - 100则C语言的程序设计题，并附有解答和代码
  - 豆瓣评分 9.0，版本较为古老（2005年）
  - 科大图书馆可借