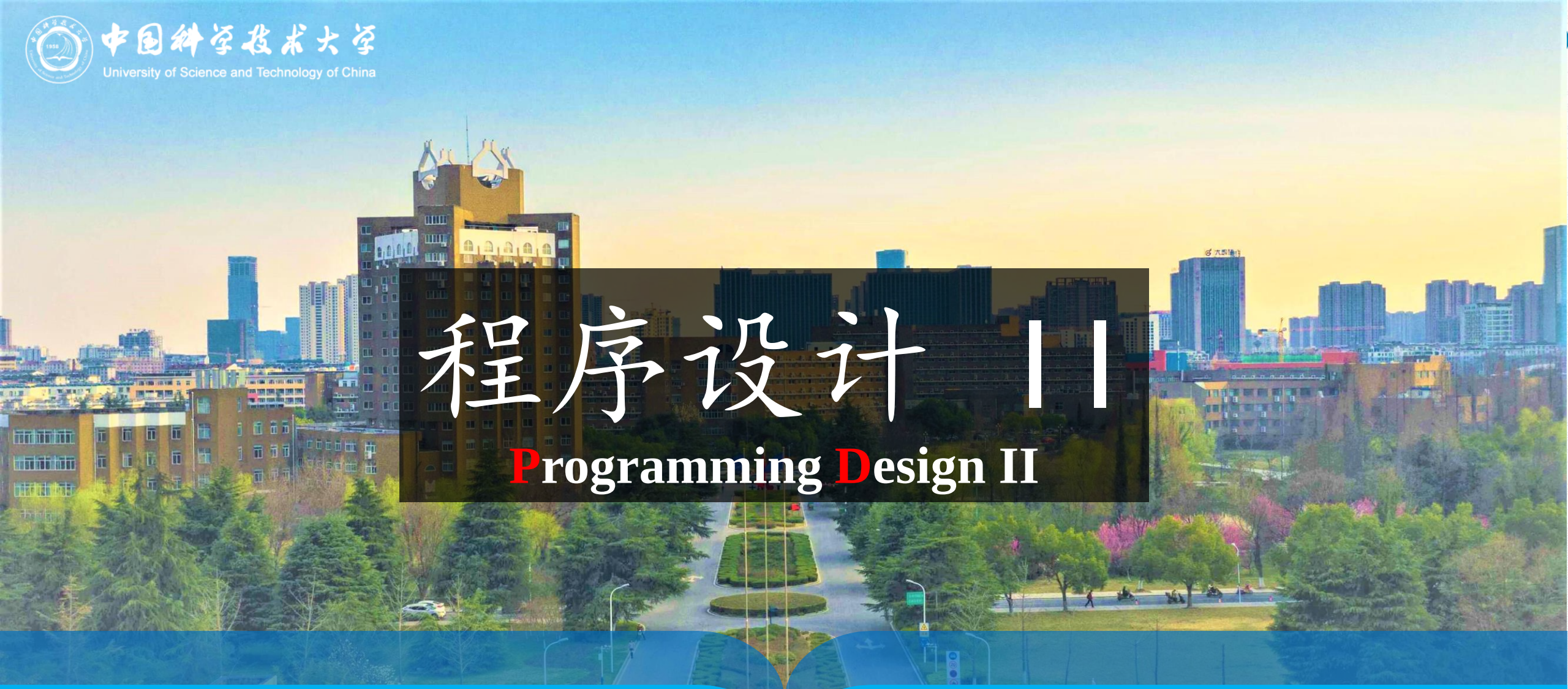

中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II

C语言回顾

主讲：吴锋

目录
CONTENTS

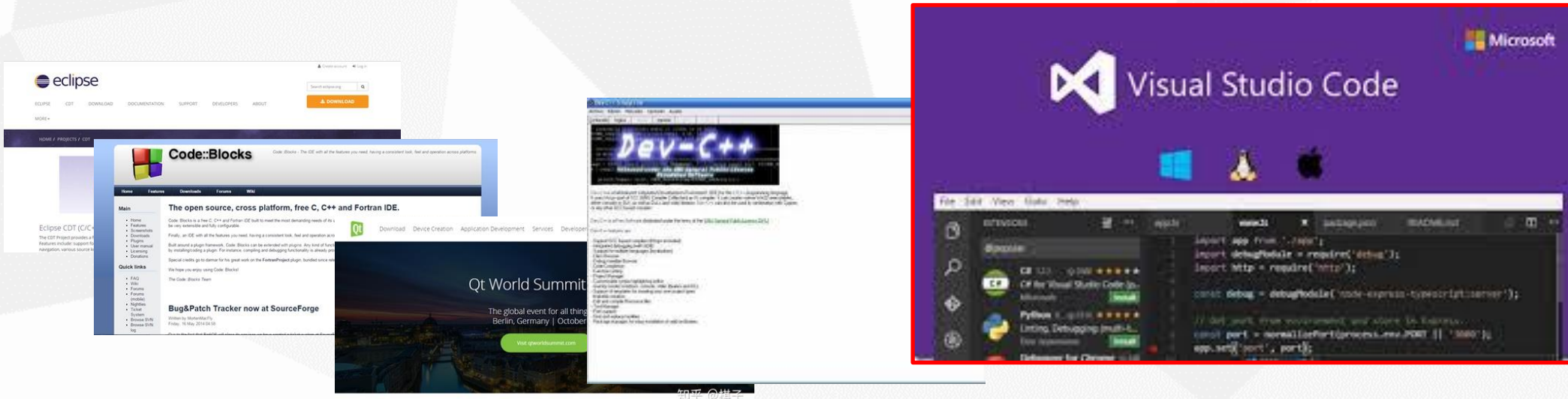
◎ 集成开发环境

- IDE (Integrated Development Environment)
  - 将源程序编译、编译、链接、执行、调试等功能集成在一个软件包中

工欲善其事，必先利其器

——《论语 • 卫灵公》

- 总有一款适合你
  - “深受C/C++程序员欢迎的11款IDE开发工具”
  - “C/C 开发者必不可少的15款编译器 IDE”



集成开发环境

- 应当注意：
  - 不同的IDE平台，对相同程序的编译执行结果可能不同
  - 不同的编译器优化选项，也可能会导致不同的运行结果
  - C对于某些运算的规定是模糊的，允许编译器的设计者自行决定

```
int i=1;
printf("%d, %d, %d, %d\n", i++, i++, i=2, i);
printf("%d\n", i);
```

- 课堂使用的集成开发环境：
  - VLAB 实验中心：https://vlab.ustc.edu.cn/vscode/

编译过程

源代码（.c, .cpp, .h）

预处理

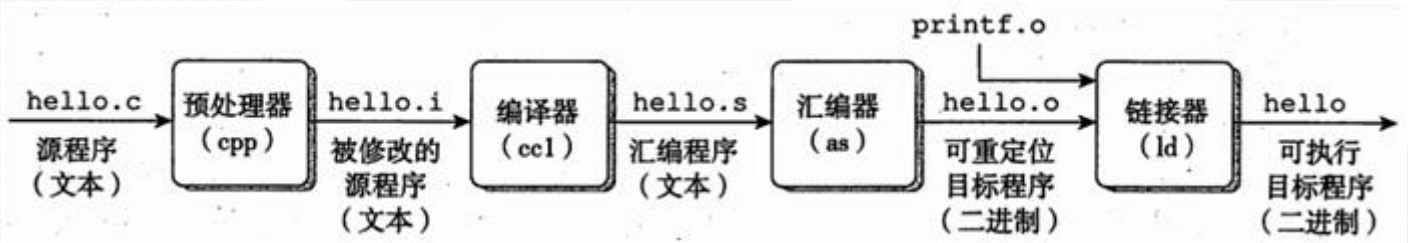
头文件嵌入、宏展开的源代码（.i）

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

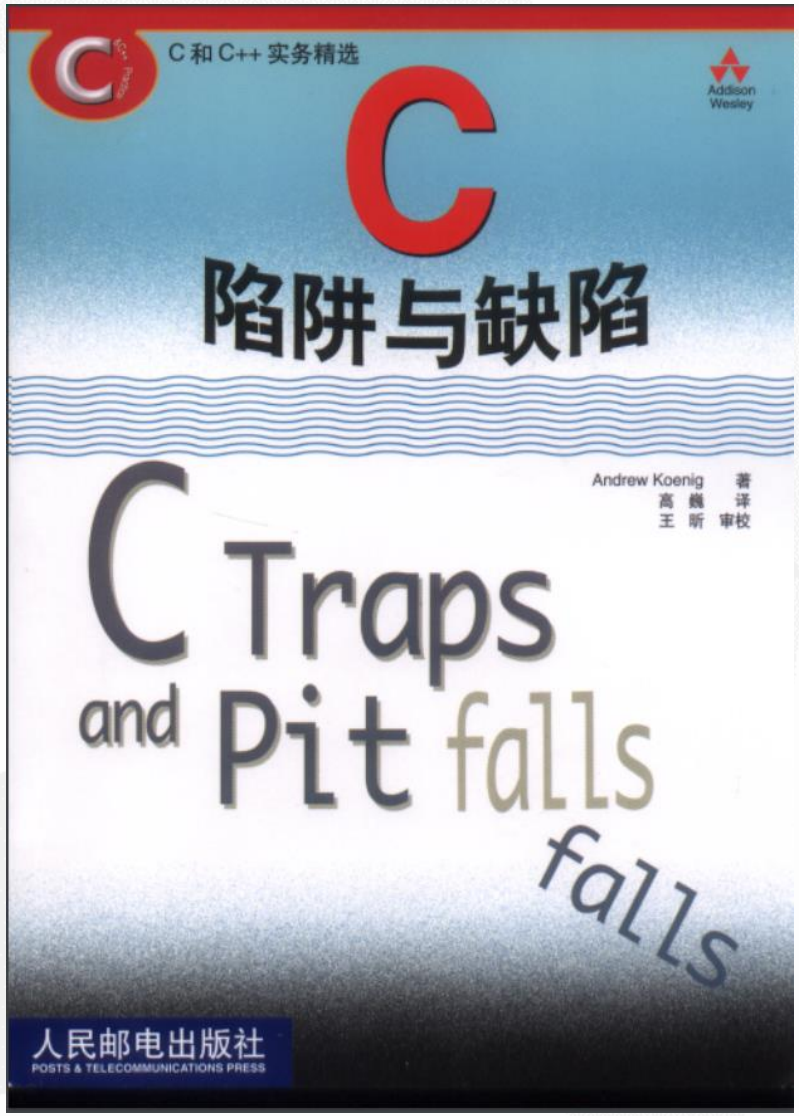
```
justify: justify.o word.o line.o
        gcc -o justfy justify.o word.o line.o
justify.o: justfy.c word.h line.h
        gcc -c justify.c
word.o: word.c word.h
        gcc -c word.c
line.o: line.c line.c
        gcc -c line.c
```

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

C语言的陷阱与缺陷



中文版序

我动笔写作《C 缺陷与陷阱》时，可没想到 14 年后这本书仍然在印行！它之所以历久不衰，我想，可能是书中道出了 C 语言编程中一些重要的经验教训。就是到今天，这些教训也还没有广为人知。

C 语言中那些容易导致人犯错误的特性，往往也正是编程老手们为之吸引的特性。因此，大多数程序员在成长为 C 编程高手的道路上，犯过的错误真是惊人地相似！只要 C 语言还能继续感召新的程序员投身其中，这些错误就还会一犯再犯。

大家通常读到的程序设计书籍中，那些作者总是认为，要成为一个优秀的程序员，最重要的无非是学习一种特定程序语言、函数库或者操作系统的细节，而且多多益善。当然，这种观念不无道理，但也有偏颇之处。其实，掌握细节并不难，一本索引丰富完备的参考书就已经足矣；最多，可能还需要一位稍有经验的同事不时从旁点拨，指明方向。难的是那些我们已经了解的东西，如何“运用之妙，存乎一心”。

学习哪些是不应该做的，倒不失为一条领悟运用之道的路子。程序设计语言，就比如说 C 吧，其中那些让精于编程者觉得称心应手之处，也格外容易误用；而经验丰富的老手，甚至可以如有“先见之明”般，指出他们误用的方式。研究一种语言中程序员容易犯错之处，不但可以“前车之覆，后车之鉴”，还能使我们更谙熟这种语言的深层运作机制。

知悉本书中文版即出，将面对更为广大的中国读者，我尤为欣喜。如果您正在读这本书，我真挚地希望，它能对您有所裨益，为您释疑解惑，让您体会编程之乐。

Andrew Koenig

美国新泽西州吉列

2002 年 10 月

前言

对于经验丰富的行家而言，得心应手的工具在初学时的困难程度往往要超过那些容易上手的工具。刚刚接触飞机驾驶的学员，初航时总是谨小慎微，只敢沿着海岸线来回飞行，等他们稍有经验就会明白这样的飞行其实是一件多么轻松的事。初学骑自行车的新手，可能觉得后轮两侧的辅助轮很有帮助，但一旦熟练过后，就会发现它们很是碍手碍脚。

这种情况对程序设计语言也是一样。任何一种程序设计语言，总存在一些语言特性，很可能会给还没有完全熟悉它们的人带来麻烦。令人吃惊的是，这些特性虽然因程序设计语言的不同而异，但对于特定的一种语言，几乎每个程序员都是在同样的一些特性上犯过错误、吃过苦头！因此，作者也就萌生了将这些程序员易犯错误的特性加以收集、整理的最初念头。

我第一次尝试收集这类问题是在 1977 年。当时，在华盛顿特区举行的一次 SHARE（IBM 大型机用户组）会议上，我作了一次题为“PL/I 中的问题与‘陷阱’”的发言。作此发言时，我刚从哥伦比亚大学调至 AT&T 的贝尔实验室，在哥伦比亚大学我们主要的开发语言是 PL/I，而贝尔实验室中主要的开发语言却是 C。在贝尔实验室工作的 10 年间，我积累了丰富的经验，深谙 C 程序员（也包括我本人）在开发时如果一知半解将会遇到多少麻烦。

C语言的陷阱与缺陷

- = 不同于 ==，&和|不同于&&和||
  - if (x = 0) 或 while (x = 1)
- 词法分析中的“贪心法”
  - a---b 与 a-- - b相同，与 a - -- b不同
  - n-->0的含义是n-- > 0，而不是 n- -> 0
  - y = x/*p 会报错，正确应为 y = x / *p 或 y = x/(*p)
- 整形和字符型的进制
  - 10 是10进制整数，010是8进制整数（值为8），0x10是16进制（值为16）
  - '\100'是8进制字符（值为64），'\x10'是16进制字符（值为16）

示例代码一

- 问：以下代码的输出是什么？

```
#include <stdio.h>
void main()
{
    if (sizeof(int) > -1)
        printf("True\n");
    else
        printf("False\n");
}
```

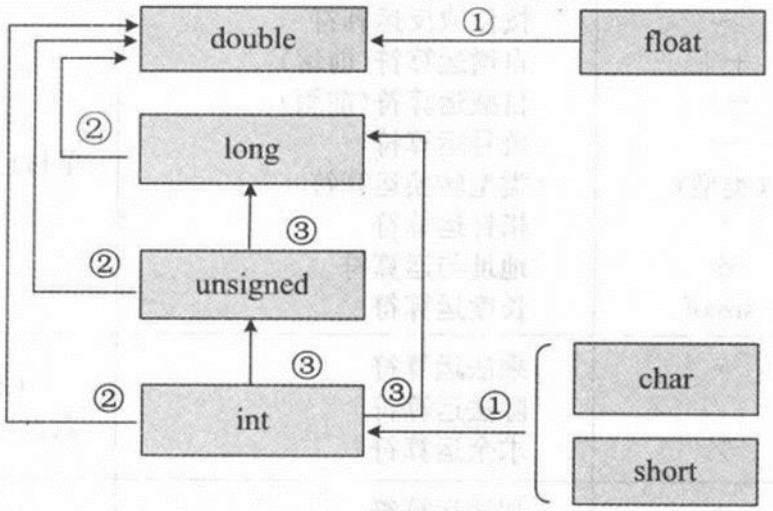


图 3.6　数据类型间的转换

隐式类型转换

示例代码二

- 问：以下代码的输出是什么？

```
#include <stdio.h>
void main()
{
    float f = 0.1;
    if (f == 0.1)
        printf("True\n");
    else
        printf("False\n");
}
```

```
#define EPS 1e-6

if (fabs(f – 0.1) < EPS) {
        … …
}
```

浮点类型的表示精度

示例代码三

- 问：以下代码的输出是什么？

```
#include <stdio.h>
void main()
{
    int a, b = 1, c = 1;

    a = sizeof(c = ++b + 1);
    printf("%d %d %d\n", a, b, c);
}
```

示例代码四

- 问：以下代码的输出是什么？

```
#include <stdio.h>
void main()
{
    int a = 1, b = 3, c = 2;
    if (a > b)
        if (b > c)
            printf("True\n");
    else
        printf("False\n");
}
```

```
#include <stdio.h>
#define N 255
void main()
{
    char c;
    for (c = 0; c <= N; n++)
        printf("%c ", c);

    printf("\n");
}
```

示例代码五

- 问：以下代码的输出是什么？

```
#include <stdio.h>
void main()
{
    char *p = 0;

    *p = "A";
    printf("%c", *p);
}
```

```
#include <stdio.h>
void main()
{
    char *p = 0;

    p = 'A';
    printf("%c", *p);
}
```

```
#include <stdio.h>
void main()
{
    char *p = 0;

    p = "A";
    printf("%c", *p);
}
```

◎ 类型声明解析

```
long **c[7];
/* c is array of 7 pointers to pointer to long */

char * const *(*c)();
/* c is pointer to function returning pointer to const pointer to char */

const char * (*c(int, void (*f)(int)))(int);
/* c is function (with parameters of int and pointer to function with parameter of int returning
void) returning pointer to function (with parameter of int) returning pointer to const char */

char * (*c[10])(int **p);
/* c is array of 10 pointers to function (with parameters of pointer to pointer to int) returning
pointer to char */

char *(*(**c[][8])())[];
/* c is array of array of 8 pointers to pointer to function returning pointer to array of pointer
to char */
```

类型声明解析

- 基本解析规则
  1. 从变量名开始
  2. 从左到右解析，回到左边当：
     - 遇到右括号
     - 遇到函数的两对括号
     - 遇到声明的末尾
  3. 结束于某个基本类型
- 基本类型：
  - * 指针（pointer to ...）
  - [] 数组（array of ...）
  - () 函数 （function returning ...）
- const 和 volatile 修饰符
  - 与左边的* 指针相结合
  - 若左边没有指针则与右边的基本类型相结合
- 例如：
  - const char *p; 代表p是一个指针，指向常量字符类型 (p is pointer to const char)
  - char * const p; 代表p是常量指针，指向的是字符类型 (p is const pointer to char)
  - const char * const p;代表p是常量指针，指向的是常量字符类型 (p is const pointer to const char)

运算符优先级

- 基本规则：
  - 单目 > 双目 > 多目
  - 算术 > 关系 > 逻辑 > 赋值
- 一些说明：
  - *p++ 解析为 *(p++)
  - a = b = c 解析为 a = (b = c)
  - while(c = getc(in) != EOF) 会错误的解析为 while(c = (getc(in) != EOF))，正确的写法为 while((c = getc(in)) != EOF)
- 求值顺序（与优先级不是一回事）
  - i = ++i + i++; // 未定义行为
  - f(++i, ++i); // 未定义行为
  - a[i] = i++; // 未定义行为

优先级	运算符	描述	结合性
1	++ -- () [] . ->	后缀自增与自减 函数调用 数组下标 结构体成员访问 结构体成员通过指针访问	从左到右
2	++ -- + - ! ~ (*type*) * & sizeof	前缀自增与自减 一元加与减 逻辑非与逐位非 类型转换 指针取值 取址 取大小	从右到左
3	* / %	乘法、除法及余数	从左到右
4	+ -	加法及减法	
5	<< >>	逐位左移及右移	
6	< <= > >=	分别为 < 与 ≤ 的关系运算符 分别为 > 与 ≥ 的关系运算符	
7	== !=	分别为 = 与 ≠ 关系	
8	&	逐位与	
9	^	逐位异或（排除或）	
10	|	逐位或（包含或）	
11	&&	逻辑与	
12	||	逻辑或	
13	?:	三元条件	从右到左
14	= += -= *= /= %= <<= >>= &= ^= |=	简单赋值 以和及差赋值 以积、商及余数赋值 以逐位左移及右移赋值 以逐位与、异或及或赋值	
15	,	逗号	从左到右

计算表达式解析

```
int a, b, c;
double x = 2.6, y = 2.0, s = 3.0, t = 1.0;

a=5;                    // 表达式的值为5
a=b=c=1;                // a=(b=(c=1))
a=(b=4)+(c=3);          // b=4, c=3, a=7
a+=a*=b+2;              // a=a+(a=a*(b+2)); 84
x+=5.0;                 // x=x+5.0; 7.6
y*=s+t;                 // y=y*(s+t); 8.0
```

```
int i=3, j, a, b=1, c=2;

j = ++i;                //正确 i=i+1, j=i, i=4, j=4
j = i++;                //正确 j=i, i=i+1, i=5, j=4
j = -i++;               //正确 -(i++), i=6, j=-5
j = i++*2;              //正确 (i++)*2, i=7, j=12
a = (b+c)++;            //错误，不是左值!
a = 34++;               //错误，不是左值!
j = ++i++;              //错误，不合语法!
j = a+++b+++c++;        //正确j=a++ + b++ + c++
j = a++++b;             //错误 j=a ++ ++ b
j = a+++ +b;            //正确 j=a ++ + +b
```

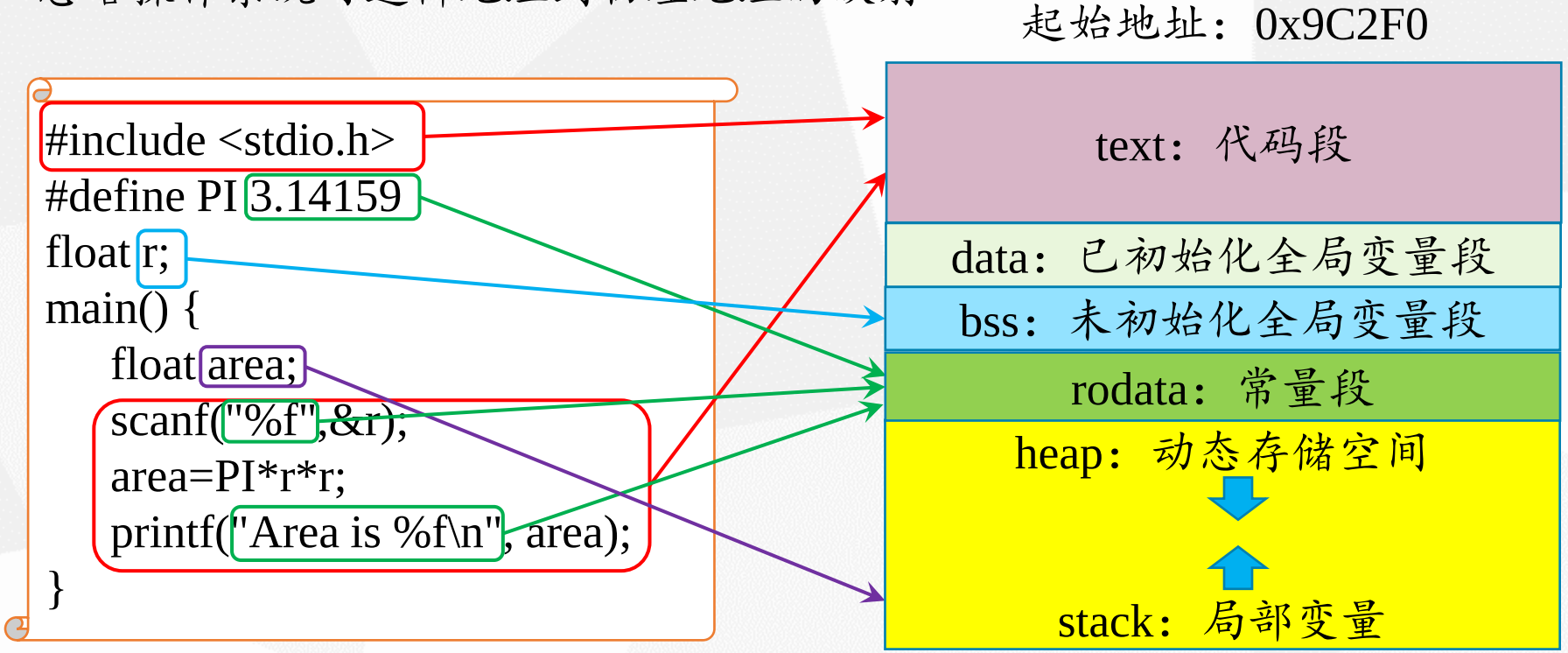
◎ 逻辑表达式解析

```
int a=3, b=4, c=5, k;

a<=b          //值为1
a+b>c-b       //等价于(a+b)>(c-b)，值为1
k=3<=c        //等价于k=(3<=c)，值为1
a==3<=c       //等价于(a)==(3<=c)，值为0
a<b<c         //等价于(a<b)<c，值为1
a==a==a       //等价于(a==a)==a，值为0
(i>=j)+(i==j) //根据i，j关系，得0，1，2
```

```
int a=3, b=3, y=2000;
char c='A';
double x=0.0, z=7.2, f;

!(a-b)                    //值为1
c&&(a-b)||(f=3.2)         //等价于(c&&(a-b))||(f=3.2)，值为1
!x*!!z                    //等价于(!x)*(!(!z))，值为1
y%4==0&&y%100||!(y%100)&&!(y%400)  //值为1
(y%100!=0)&&!(y%4) || (y%100==0)&&!(y%400)
```

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



◎ 指针的概念

- 地址、指针的概念
  - 地址：内存单元的编号
  - 指针：指向变量的地址，如&i、&j、&k、&h、&ptr
  - 指针变量：存放其它变量地址的变量。如图中ptr指向变量i的地址i

```
int i=3, j=6, k=9, h;
int *ptr=&i;
*ptr = 4;
```

上述定义中的*表示ptr是一个指针变量

通过指针变量ptr可以引用变量i的内容

关键字：static

```
#include <stdio.h>

static int x; // 静态全局变量

static void print_num() { // 静态函数
    static int y = 0; // 静态局部变量
    printf("%d\n", y++);
}

void main() {
    print_num();
    print_num();
    print_num();
} // 函数的输出是什么?
```

1. 变量 x 的生存周期和作用域是什么？

2. 变量 y 的生存周期和作用域是什么？

3. 函数 print_num 的作用域是什么？

从函数中返回数组

- 下列代码是否有问题？如果有问题话，存在什么问题？

```
char *ret_str() {
    char str[] = "Hello";
    return str;
}
```

```
void main() {
    char s[10];
    s = ret_str();
}
```

```
char *ret_int() {
    int arr[] = {1, 2, 3};
    return arr;
}
```

```
void main() {
    int a[10];
    a = ret_int();
}
```

从函数中返回数组

- 使用字符串常量进行返回
  - 字符串常量存储在只读存储区
  - 生存周期是程序级的，可以被函数返回
  - 返回的只能是字符串常量，不能被修改

```
const char *ret_str() {
    char *const_str = "Hello";
    return const_str;
}
```

```
void main() {
    char s[10];
    strcpy(s, ret_str());
}
```

从函数中返回数组

- 使用全局数组变量进行返回
  - 全局数组存储在全局存储区
  - 生存周期是程序级的，可以被返回
  - 能被所有函数访问和修改，破坏函数的模块化
  - 过大的全局数组会造成内存浪费，过小则容易数组越界

```
char global_str[1024];
char *ret_str() {
    ... ...
    return global_str;
}
```

```
void main() {
    char s[10];
    strcpy(s, ret_str());
}
```

从函数中返回数组

- 使用局部静态数组变量进行返回
  - 局部静态数组存储在静态存储区
  - 生存周期是程序级的，可以被返回
  - 只能函数内部访问，不破坏函数的模块化
  - 每次调用都可能会对数组值进行修改，破坏函数的独立性

```
char *ret_str() {
    static char static_str[1024];
    ... ...
    return static_str;
}
```

```
void main() {
    char s[10];
    strcpy(s, ret_str());
}
```

◎ 从函数中返回数组

- 使用动态内存分配的数组进行返回
  - 动态内存分配的数组存储在堆区
  - 生存周期是程序级的，可以被返回
  - 能够保持函数的模块化设计和调用的独立性
  - 但动态分配的内存需要被正确的释放，否则容易造成内存泄露

```
char *ret_str() {
    char *dynamic_str = (char *)
        malloc(128*sizeof(char));
    ... ...
    return dynamic_str;
}
```

```
void main() {
    char s[10];
    char *p = ret_str();
    strcpy(s, p);
    free(p);
}
```

从函数中返回数组

- 使用函数的指针参数进行返回
  - 由函数的调用者负责准备实参数组的存储空间
  - 在函数内，通过指针形参向实参数组填充内容
  - 大多数的标准库函数采取这种方式返回数组值
  - 函数的返回值通常用来返回函数的执行状态（是否发生错误）

```
int ret_str(char *str) {
    ... ...
    return status_code;
}
```

```
void main() {
    char s[10];
    ret_str(s);
}
```

从函数中返回数组

- 使用结构体中的数组成员进行返回
  - 不需要事先分配好内存
  - 函数返回通过结构体的内存拷贝完成
  - 如果数组较大时，函数返回的开销也比较大

```
struct Str { char str[1024]; };
struct Str ret_str() {
    struct Str struct_str;
    ... ...
    return struct_str;
}
```

```
void main() {
    char s[10];
    strcpy(s, ret_str().str);
}
```

良好的编码风格

- 良好的编码风格能够提高代码的可读性，提高编码效率，减少Bug的数量。
- 良好的编码风格能够提高代码的可维护性，有利于团队合作，以及持续的改进。
- 不同的编码风格没有优劣之分，重要的是保持代码风格的一致性，符合团队规范。

◎ 标识符的命名

- 匈牙利命名法为C程序标识符的命名定义了一种非常标准化的方式。这种命名方式是以两条规则为基础的：
  - 变量的名字以一个或者多个小写字母前缀开头，前缀能够体现变量数据类型、作用域等信息。
  - 在标识符内，前缀以后就是一个或者多个第一个字母大写的单词，这些单词清楚地指出了该标识符的作用。

标识符的命名

- 变量命名加前缀表面变量的类型

n	int
c	char
f	浮点数 (float)
b	取值只为真和假的整型变量 如 bValid
p	指针
a	数组
fp	文件指针 FILE *

标识符的命名

- 变量名中单词开头字母大写或以‘_’隔开，其他字母小写
  - 如：bValid 或 b_valid
  - 但是常用的意义明显的变量，如 i,j,k, 坐标 x,y等不必遵循 1),2)
- 常量和宏都是大写，单词间用 ‘_’分隔

```
#define      MAX_WIDTH   5
#define      ABS(x)      ((x)>=0?(x):-(x))
```

标识符命名基本原则

- 标识符号应能提供足够信息，最好是可以发音的。
- 为全局变量取长的描述信息多的名字，为局部变量取短名字。
- 名字太长时可以适当采用单词的缩写。但要注意，缩写方式要一致。要缩写就全都缩写。
  - 如：单词Number, 如果在某个变量里缩写成了: int nDoorNum；那么最好包含 Number单词的变量都缩写成 Num。
- 注意使用单词的复数形式。
  - 如：int nTotalStudents, nStudents ;容易让人理解成代表学生数目，而 nStudent 含义就不十分明显。
- 一般变量和结构名用名词，函数名用动词或动宾词组。

标识符命名基本原则

- 对于返回值为真或假的函数，加“Is”前缀如：

```
int         IsCanceled();
int         isalpha(); // C语言标准库函数
BOOL        IsButtonPushed();
```

- 对于获取某个数值的函数，加“Get”前缀

```
char * GetFileName();
```

- 对于设置某个数值的函数，加“Set”前缀

```
void SetMaxVolume();
```

程序书写格式

- 正确使用缩进：
  - 首先，一定要有缩进，否则代码的层次不明显。
  - 缩进应为4个空格较好。需要缩进时一律按Tab键，或一律按空格键，不要有时用Tab键缩进，有时用空格键缩进。
  - 一般开发环境都能设置一个Tab键相当于多少个空格，此时就都用Tab键。

程序书写格式

- 行宽与折行：
  - 一行不要太长，不能超过显示区域。以免阅读不便。
  - 太长则应折行。折行最好发生在运算符前面，不要发生在运算符后面如：

```
if (Condition1() && Condition2()
        && Condition3()) {
}
```

程序书写格式

- 注意 ‘{’,‘}’位置不可随意，要统一。
  - 如果写了：

```
if  (condition1()) {
        DoSomething();
}
```

  - 别处就不要写：

```
if (condition2())
{
        DoSomething() ;
}
```

程序书写格式

- 变量和运算符之间最好加1个空格

```
int  nAge = 5;
nAge = 4;
if (nAge >= 4)
        printf( “%d”,nAge);

for (i = 0; i < 100; i++);
```

一些编程习惯

- 尽量不要用立即数，而用const 定义成常量或定义成宏，以便以后修改。

```
const  int MAX_STUDENTS  = 20
struct SStudent aStudents [MAX_STUDENTS];
```

比

```
struct SStudent aStudents [20];
```

好

```
#define TOTAL_ELEMENTS  100
for(  i = 0; i < TOTAL_ELEMENTS; i ++)   { }
```

一些编程习惯

- 使用sizeof()运算符，不直接使用变量所占字节数的数值，以便适应不同的系统

```
int nAge;
for( j = 0; j < 100; j++ )
        fwrite( fpFile,& nAge, 1, sizeof(int));
```

比

```
for( j = 0; j < 100; j++ )
        fwrite( fpFile,& nAge, 1, 4);
```

好

一些编程习惯

- 稍复杂的表达式中要积极使用括号，以免优先级理解上的混乱

```
n = k +++ j;  //不好
n = ( k ++ ) + j; //好一点
```

- 不很容易理解的表达式应分几行写：

```
n = ( k ++ ) + j; 应该写成：
n = k + j;
k ++;
```

一些编程习惯

- 不提倡在表达式中使用?:形式，而用if .. else语句替代。

```
xp =  2 * k < ( n-m) ? c[k+1] : d[k--];

if( 2*k < (n-m))
        xp = c[k+1];
else
        xp = d[k--];
```

一些编程习惯

- 嵌套的if else 语句要多使用 { }

```
if( Condition1() )
        if( Condition2() )
                DoSomething();
        else
                NoCondition2();
```

不够好，应该：

```
if( Condition1() ) {
    if( condition2() )
        DoSomething();
    else
        NoCondition2();
}
```

一些编程习惯

- 应避免 if else 多重嵌套，而用并列完成。

```
if( Condition1() ) {
    if ( Condition2() ) {
            if( Condition3() ) {

            Condition123();
            }else  {

            NoCondition3();
            }
    }else {
            NoCondition2();
    }
}else {
    NoCondiction1();
}
```

替换为：

```
if( ! condition1 ) {
    NoCondition1();
}else if( ! condition2 ) {
    NoCondition2();
}else if( ! condition3) {
    NoCondition3();
}else {
    Condition123();
}
```

一些编程习惯

- 写出来的代码应该容易读出声

比如

```
if( !( n > m ) && !( s > t))
```

就不如

```
if( ( m <= n ) && ( t <= s ))
```

```
if( !( c == 'y' || c == 'z'))
```

不如

```
if( c!= 'y' && c!= 'z');
```

谷歌代码规范

- 阅读谷歌代码规范：https://zh-google-styleguide.readthedocs.io/en/latest/google-cpp-styleguide/ ，并在编程实践中严格执行，养成良好的编码风格和习惯。
  - 7. 命名约定
  - 8. 注释
  - 9. 格式

常用的库函数

C语言在线手册：https://en.cppreference.com/w/c

- 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

C语言在线手册：https://en.cppreference.com/w/c

- 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
```

输入输出函数

- 非格式化输入输出：
  - getchar / putchar： 输入 / 输出一个字符
  - gets (gets_s) / puts： 输入 / 输出一个字符串
- 格式化输入输出：
  - scanf / printf：从键盘输入 / 输出到终端
  - sscanf / sprintf：从字符串输入 / 输出到字符串
  - fscanf / fprintf：从文件输入 / 输出到文件
- 所有输入输出都必须包含头文件 stdio.h

```
#include <stdio.h>
```

◎ 非格式化输入输出

- 输入输出一个字符
  - int getchar(void); // 成功时返回一个字符，失败时返回 EOF
  - int putchar(int ch); // 成功时返回输出的字符，失败时返回 EOF
- 输入输出一个字符串
  - char *gets(char *str); / char *gets_s(char *str, rsize_t n);
    - 当遇到换行'\n'或文件末尾时，从终端读入字符串到str （不包含'\n'）
    - 成功时返回str指针，失败时返回NULL指针。
  - int puts(const char *str);
    - 输出字符串str （并附带换行 '\n'），字符末尾的 '\0' 不会输出
    - 成功时返回一个非负值（与编译器相关），失败时返回 EOF

◎ 非格式化输入输出

- 输入输出一个字符
  - int getchar(void); 等价于 int fgetc(stdin);
  - int putchar(int ch); 等价于 int fputc(int ch, stdout);
- 输入输出一个字符串
  - char *gets(char *str); 对字符串数字越界没有检测，容易受到溢出攻击，因此不安全。通常建议使用 char *gets_s(char *str, rsize_t n); 或者 char *fgets(char *str, int count, stdin);
  - int puts(const char *str); 会在输出字符串的末尾自动添加换行'\n'，而int fputs(const char *str, stdout); 不会输出换行。
  - puts / fputs 的返回值有些编译器是输出字符长，有些是最后一个字符，甚至有些是任意非负值。

格式化输入输出

int **scanf**(const char *format , ...);

- 参数可变的函数
  - 第一个参数是格式字符串，后面的参数是变量的地址，函数作用是按照第一个参数指定的格式，将数据读入后面的变量
- 返回值
  - >0： 成功读入的数据项个数；
  - 0 ： 没有项被赋值；
  - EOF：第一个尝试输入的字符是EOF(结束)

◎ 格式化输入输出

int **printf**(const char *format , ...);

- 参数可变的函数
  - 第一个参数是格式字符串，后面的参数是待输出的变量，函数作用是按照第一个参数指定的格式，将后面的变量在屏幕上输出
- 返回值
  - 成功打印的字符数；
  - 返回负值为出错

格式控制符号

%d 读入或输出int变量

%c 读入或输出char变量

%f 读入或输出float变量

%s 读入或输出char * 变量

%lf 读入或输出double 变量

%e 以科学计数法格式输出数值

%x 以十六进制读入或输出 int 变量

%p 输出指针地址值

%.5lf 输出浮点数，精确到小数点后5位

◎ 格式化输入输出（例）

```
int n = scanf("%d%c%s%lf%f", &a, &b, c, &d, &e); // 输入是变量的地址，除数组、指针外一般变量要用取地址符
printf("%d %c %s %lf %e %f %d",a, b, c, d, e, e, n);
```

Input:

123a teststring 8.9 9.2

Output:

123 a teststring 8.900000 9.200000e+000 9.200000 5

Input:

123ateststring 8.9 9.2

Output:

123 a teststring 8.900000 9.200000e+000 9.200000 5

Input:

123 a teststring 8.9 9.2

Output:

123 a 0.000000 0.000000e+000 0.000000 3

正确的输入语句：

```
int n = scanf("%d %c%s%lf%f", &a, &b, c, &d, &e);
```

◎ 格式化输入输出

- **scanf** 常见错误
  - 输入项必须是变量的地址。
    - 如：char str[10]; scanf("%c %s", &str[0], str);
  - 若输入格式串中加入了格式符以外的其它字符，则输入时必须同样输入，否则结果不确定；
  - 若无其它字符，则可用空格、回车或制表符作为间隔标记。
  - 使用“%c”时，输入的字符不能加间隔标记。
  - 输入数据遇到空格、回车或制表符以及其它各种非法输入时，认为该项数据输入结束。注意，这些输入并没有被跳过去。
  - 输入数据比表列更多时，会被自动用于下一次输入。

格式化输入输出

- **printf** 常见错误
  - 输出项表列需要与格式控制字符一一对应
  - 类型不匹配时，不会进行隐式类型转换，会解析失败，输出0
  - 当格式符个数少于输出项时，多余的输出项不予输出。
  - 当格式符个数多于输出项时，结果为不定值。

◎ 格式化输入输出

int **sscanf**(const char *buffer, const char * format[, address, ...]);

- 和scanf的区别：从字符串buffer里读取数据

int **sprintf**(char *buffer, const char *format[, argument, ...]);

- 和printf的区别：往字符串buffer里输出数据

int **fscanf**(FILE *fp, const char * format[, address, ...]);

- 和scanf的区别：从文件fp里读取数据

int **fprintf**(FILE *fp, const char *format[, argument, ...]);

- 和printf的区别：往文件fp里输出数据

代码版本控制 *

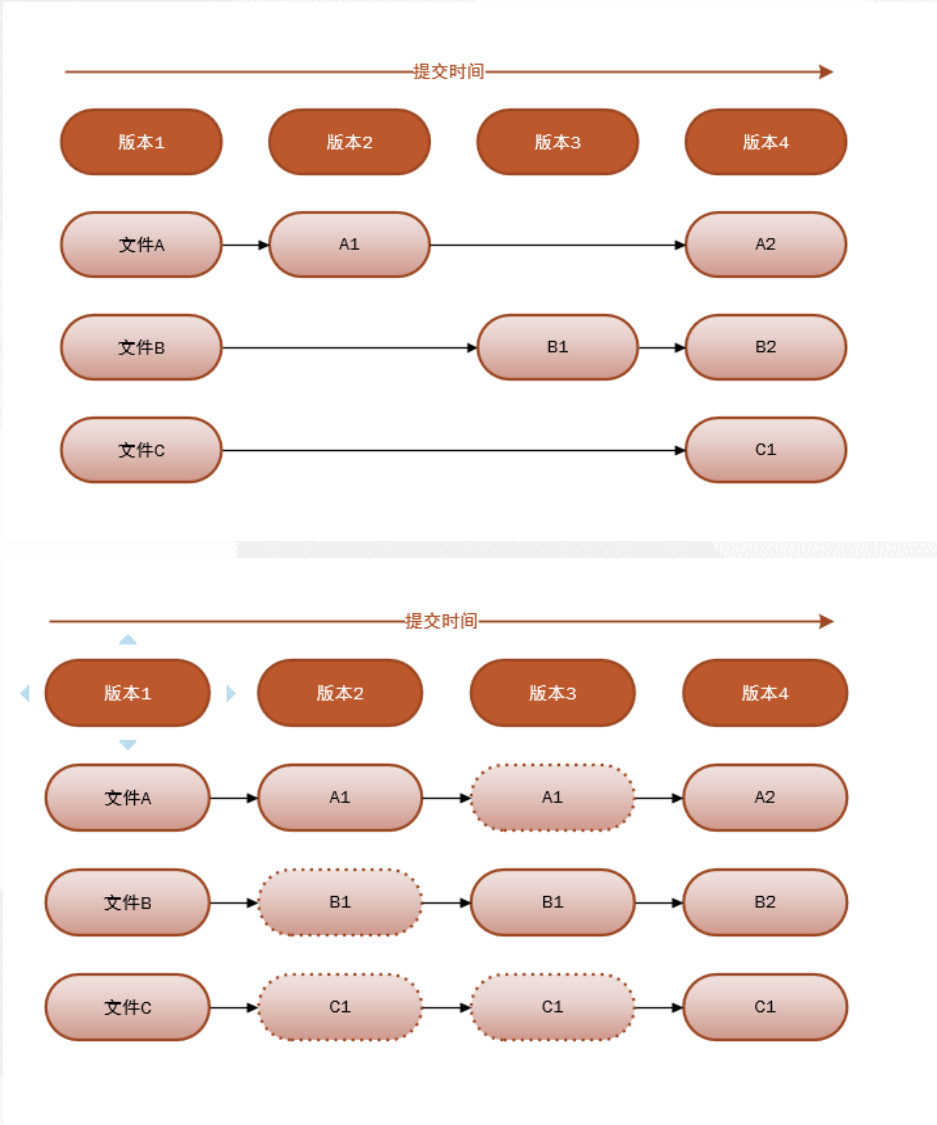
- 版本控制（Version Control System，VCS）是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。
- 版本控制可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态。
- 版本控制可以比较文件的变化细节，查出最后是谁修改了哪个地方，从而找出导致怪异问题出现的原因。
- 在团队开发中使用版本控制系统的好处
  - 作为数据备份，防止重要数据意外丢失
  - 避免版本管理混乱，可以同时维护多个版本
  - 提高代码质量，记录代码修改的历史信息
  - 明确分工责任，提高协同、多人开发时的效率

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 *

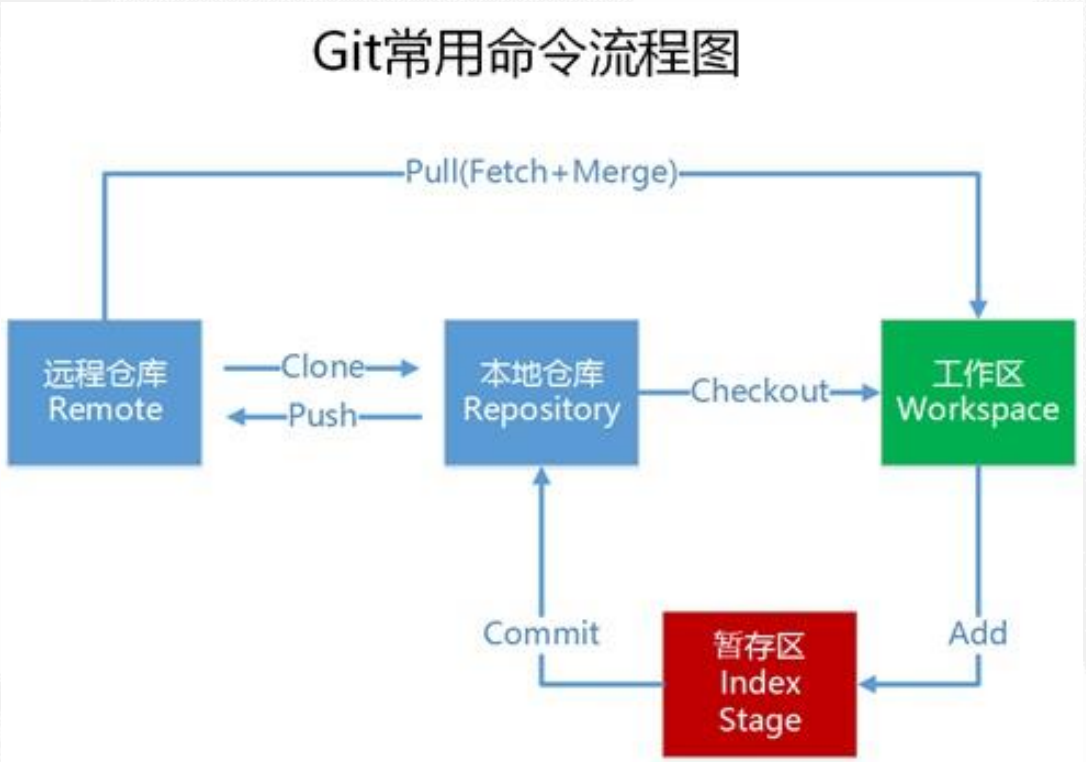
- Git 基本命令
  - 从远程仓库将项目clone到本地；

    $ 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