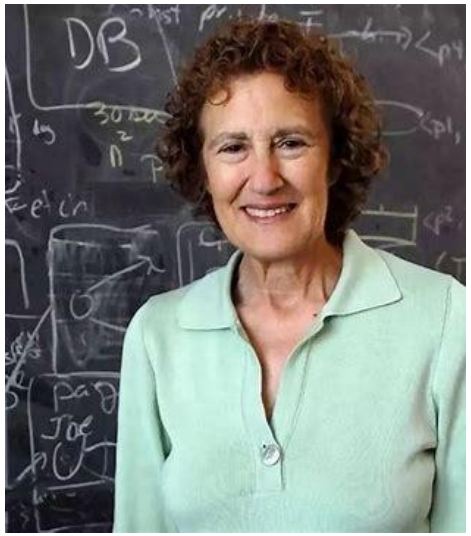
第9讲
作用域规则和编译预处理命令

周水庚

2026年10月29日/11月5日

Barbara Liskov: 第2位获图灵奖的女计算机科学家

- **Barbara Liskov (November 7, 1939 -)**
  - **1961年获加州大学伯克利分校数学学士学位**
  - **1968年获斯坦福大学计算机科学博士学位（美国第一个计算机科学女博士，导师是John McCarthy）**
  - **1968年-1972年在Mitre公司工作；此后进入 MIT，曾任副系主任和副教务长**
    - **主要贡献：CLM语言、Argus语言、数据抽象、分布计算**
    - **指导20名博士生。其儿子也在MIT获得计算机博士学位**
  - **1989年入选美国工程院院士**
  - **2004年获IEEE冯·诺伊曼奖**
  - **2009年获ACM图灵奖：**



"For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

一门三“图灵”

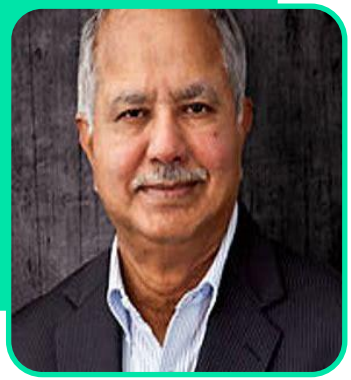
人工智能

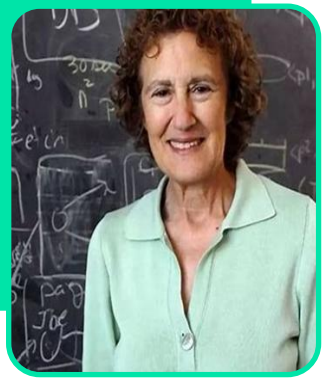


John McCarthy
(1971)

人工智能

Raj Reddy
(1994)



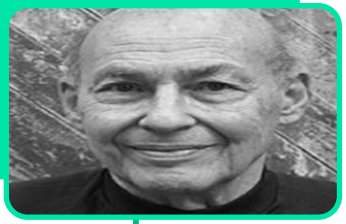


编程语言

Barbara Liskov
(2008)

一门五“图灵”

人工智能



Marvin Minsky
(1969)

密码与安全



Manuel Blum
(1995)

密码与安全



Leonard Max Adleman
(2002)

密码与安全



Silvio Micali
(2012)

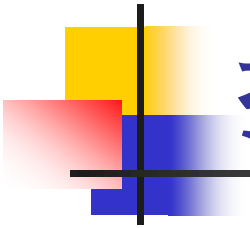


Shafi Goldwasser
(2012)

密码与安全

提要

- 局部变量和全局变量
- 存储类
- 编译预处理命令简介



提要

- **局部变量和全局变量**
- 存储类
- 编译预处理命令简介

变量分类

- 变量可以在程序中三个地方定义：函数内部、函数的形参定义中和所有的函数外部
- 从变量作用的空间范围来看，变量可以分为：全局变量和局部变量
- 根据所定义位置的不同，变量可分为局部变量、形式参量和全局变量
- 从变量作用的时间范围来分，可以有静态存储变量和动态存储变量之分

程序对象作用域

- 一个C程序通常由多个函数组成，可分放在若干源程序文件中，且允许各源程序文件分别单独编译
- 程序用标识符命名程序对象，能直接用标识符引用它代表的程序对象的程序代码范围称为该对象的作用域
- 本讲以变量为例讨论程序对象的作用域和存储类，有关结论可类似地适用于函数、类型、语句等其它程序对象

全局变量（Global Variable）

- 全局变量也称为外部变量，是在函数的外部定义的。它的作用域从变量定义处开始，到本程序文件的末尾
- 在其它地方（它的定义之前或别的源程序文件中）要用变量名引用该变量，需在引用前对它作外部说明
- 全局变量存放在静态存储区，在程序开始执行时给全局变量分配存储区，程序执行完毕释放
- 局部变量能和全局变量重名，但是局部变量会屏蔽全局变量
  - 函数内引用这个变量时，会用到同名的局部变量，而不会用到全局变量

局部变量（Local Variable）

- 局部变量在函数内部或复合语句内定义
- 局部变量的作用域是定义它的函数体或复合语句
  - 在定义它的函数或复合语句内能用变量名引用它
  - 在定义它的函数体或复合语句外不能使用它
- 局部变量只有在程序执行到定义它的函数或者复合语句模块时才被赋予内存空间，一旦退出该函数或者复合语句模块，该变量消失

局部变量及其作用域

```
void fullch(char ch, char *chpt)
{   *chpt = ch;
   /* 这里可以使用 fullch()，形参 ch，chpt */
}
void makestr(char ch, int x, char *str)
{
  char s = ch;
  /* 这里可以使用 fullch()，makestr(), 形参 ch, x，str, 局部变量 s */
  if (x == 0)   *str = '\0';
  else { makestr(ch+1, x-1, str+1); fullch(s, str);}
 }
```

局部变量及其作用域(续)

（接前页）

```
#include <stdio.h>
int main()
{  char s[120];     int m = 26;
   /*这里可使用fullch()，makestr()，局部变量s，m，以及stdio,h中定义的程序
    对象*/
   makestr('A', m, s);    printf("STR = %s\n", s);
 }
```

- 复合语句内定义的变量只在它的复合语句内有效

```
   {    /* 这里实现交换整型变量 x 和 y 的值 */
        int temp; /* temp只在本复合语句内有效 */
        temp = x;      x    = y;      y    = temp;
   }
```

全局变量及其作用域

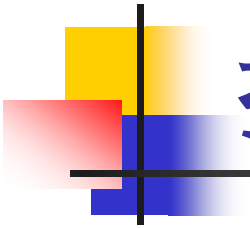
```
float scale = 1.0 ;    /* 带初值的全局变量定义 */
extern int k;          /* 对k作外部说明 */
int f1( int x)
{  int i;     /* 局部变量 */
      … /*因全局部量k有外部说明，这里也可使用k，但不能直接使用t*/
}
int k;         /* 全局变量 */
int t;         /* 全局变量 */
float f2(float a)
{ int i,j;
      …
}
void main()
{ int n; float m;
      …
}
```

scale 的自动有效范围

k 和t的自动有效范围

全局变量及其作用域(续)

- **全局变量存在性**
  - 全局变量在程序执行期间一直存在。全局变量可在定义之前的函数和别的源程序文件中使用，但需在使用之前对全局变量作外部说明
- **全局变量的作用**
  - 在函数间提供一个直接**数据传递通道**
  - 在某些应用中，若干函数共享一组全局变量，函数的执行效果记录在全局变量中；在另一些应用中，通过全局变量来**减少函数调用时实参与形参之间的数据传递**
- **不要过多使用全局变量**
  - 因程序中的所有函数都能使用全局变量，某个函数改变全局变量的值可能会影响其他函数的执行，产生函数调用的副作用
  - 全局变量在程序执行期间一直占用内存空间



提要

- 局部变量和全局变量
- 存储类
  - 自动 (auto)
  - 静态 (static)
  - 外部 (extern)
  - 寄存器 (register)
- 编译预处理命令简介

变量的可见性与存在性

- **变量具有可见性和存在性两种基本属性**
  - **变量的可见性是指用变量名对变量的可访问性。在程序的某个范围内的所有语句可以用变量的名访问变量，这个范围就是变量的作用域**
  - **变量的存在性是指在程序执行的过程中，变量在某个时段或某种情况下在内存中存在，占有着存储空间，它的值被保留着**
  - **可见性与存在性在某些场合是一致的，但在有些场合则不一致：可见必存在，存在未必可见。**
    - **存在这样的情况，变量虽然存在，但程序的当前位置不可访问它**

存储类(Storage Class)

- C语言中的存储类型用于描述变量的存储位置、生命周期（存在性）、作用域和初始值

- C语言中变量的存储类共有四类
  - 自动 (auto)
  - 静态 (static)
  - 外部 (extern)
  - 寄存器 (register)

自动变量

- **在函数或复合语句内定义，在类型说明符之前冠或不冠以关键字auto**
  - 如 auto int x = 2; double y; 定义x和y是自动的
- **自动变量主要特点是临时性**
  - **在函数体内定义的自动变量，只在该函数被调用时，系统临时为它们分配存储单元，函数执行结束时，系统就回收它们的存储单元。自动变量作用域是定义它的函数或复合语句，其它函数或复合语句可用同样的标识符定义其它程序对象**

自动变量(续)

- **在函数内部，自动变量的可见性和存在性是一致的**
  - 自动变量随函数被调用或进入复合语句而存在，函数返回或离开复合语句而消失。一次函数调用到下一次函数调用之间或相继两次进入复合语句之间，自动的变量之值是不保留的。
  - 在每次进入函数或复合语句时，第一次对自动变量的引用是为它设定初值
  - 特别指出：**函数的形参也按自动变量处理**

静态变量

- 静态变量可以是局部的，也可以是全局的。在变量定义时，在类型说明符之前冠以关键字static即可
  - 如static int z = 2;定义z是整型静态的变量
- 静态变量的特点
  - 静态变量在程序编译时预分配，并在程序执行前就被确定存储单元
  - 静态的局部变量的作用域与自动变量一样，局限于定义它的函数或复合语句
  - 但是它又与自动变量不同，静态的局部变量在程序执行过程中，始终存在，但在它的作用域之外不可直接用其名对它访问
  - 静态的局部变量在其作用域内提供了专有的、永久性的存储。函数体内定义的静态局部变量能保存函数前一次调用后的值，供下一次调用时继续使用
- 静态局部变量的存在性和可见性可能不一致

静态局部变量与自动变量的区别（例子）

```
#include <stdio.h>
p( int x )    /* 函数 p() 的定义 */
{  auto   int y = 1;  /*定义自动的变量y  "auto"可不写*/
   static int z = 2;   /*定义静态的局部变量z*/
   y++;
   z++;
   return x+y+z;
}
int main()
{ printf("%d\t", p(3));
  printf("%d\t", p(3));
  printf("%d\n", p(3));
  return 0;
}
```

程序运行将输出

8　　9　　10

静态局部变量与自动变量的区别

- 定义静态局部变量时，如同时指定初值，该初值是在程序开始执行前就被设定的。以后每次调用该函数时不再重新设定初值，而是保留上次函数调用结束时的值
- 自动变量置初值是在每次函数被调用时进行。函数每次被调用，它就被重新创建，并重新设定初值
- 在定义静态局部变量时，如未指定初值，则系统自动给静态的局部变量赋一个二进位信息全为0的初值
- 为便于程序移植、阅读和修改，建议程序明确给出静态局部变量的初值
- 对于自动变量，如定义时未给定初值，它的初值是不确定的。函数在引用定义时未给定初值的自动变量时，必须先为它设定值

静态全局变量

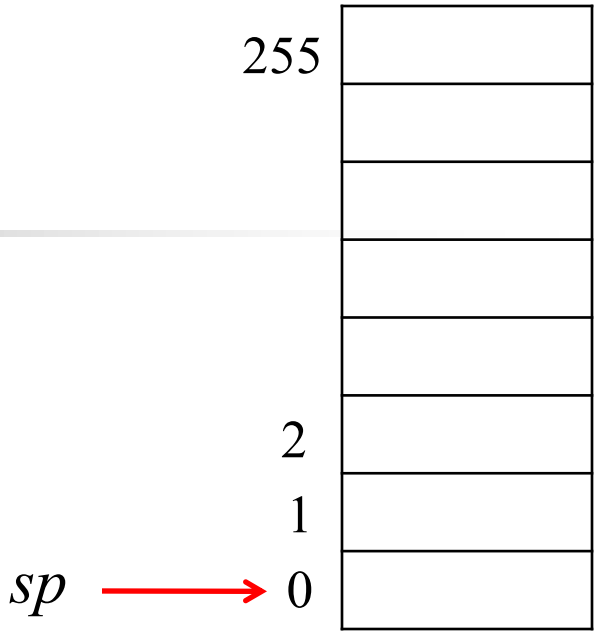
- 静态全局变量对于同一源程序文件中的函数是可见的。但与一般的全局变量不同，它不能被任何其它源程序文件中的函数可见

- 静态全局变量让同一源程序文件中的函数共享数据，但不让其它源程序文件中的函数使用它

- 静态全局变量的存在性与可见性也可能不一致

静态全局变量示例

```
static int intStack[256]; /*静态全局变量*/
static int sp = 0;        /*静态全局变量*/
int pop(int *ipt)
{  if (sp == 0) return 0;
   *ipt = intStack[--sp];    return 1;
 }
int push(int e)
{ if (sp == 256) return 0;
  intStack[sp++] = e;   return 1;
 }
void initStack()
{ sp = 0;
}
```



1）函数pop()、push()和initStack()共享数组intStack[]和整型变量sp，又不让位于其它源程序文件中的函数使用

2）sp指向当前未存放数据位置

静态函数

- 在C程序中，函数是全局的，即函数能让整个程序中的其它函数使用
- 也可以把函数定义成静态的，从而使该函数不能让别的源程序文件中的函数使用
- 把函数定义成静态的，只要在函数定义的类型说明符之前加上“static”

静态的意义

- “静态”包含两方面的意义
  - 从程序对象在程序执行期间的存在性来看，静态表示该程序对象“永久”存在
  - 从程序对象可访问或可调用（即可见性）来看，静态表示该程序对象的专用特性，亦即对可见性进行限制
- 静态局部变量只有定义它的函数中可访问；静态全局变量和静态函数只有在定义它的源程序文件中的函数中可访问或可调用
- 静态全局变量和静态函数为把数据对象及数据对象上的操作隐藏起来。这正是结构化程序设计方法和程序设计语言所追求的
- 需要指出的是，C语言用静态来实现这个要求并不是最好的

外部变量

- 外部变量是指该变量是一个在函数之外某处定义的全局变量。在变量说明的类型说明符之前冠以关键字extern
  - 如extern int x, a[ ];说明整型变量x和数组a是外部的
  - 变量x和a被说明为外部后，在该说明之后，就可访问全局整型变量x和数组a的元素
- 一个C程序可以由一个或多个源程序文件组成，而全局变量可直接被引用的范围是从它的定义处开始到定义它的源程序文件的末尾。如果在全局变量定义之前的函数中要引用该全局变量，需在访问之前对它作外部说明。同样地，如果在定义全局变量源程序文件之外的其它源程序文件中要访问该全局变量，也需在访问之前对它作外部说明

外部变量示例1

```
extern int k; /* 说明整型变量 k 是外部的 */
float f2( );  /* 对f2()的说明，或写成外部说明 extern float f2(); */
int f1(int x)
{
 /* 因前面已有对 k和f2 的外部说明， 故这里可使用 k和f2 */
       ⋮
 }
 int k;          /* 全局变量k的定义 */
 float f2(float a)
 {
       /* 因本函数位于变量k定义之后，可使用k */
       ⋮
 }
```

外部变量示例2

```
extern int k; /*对全局变量k的外部说明*/
int f1();     /*对函数f1()的说明*/
float f2();   /*对函数f2()的说明*/
int f3()
{  /*因前面的外部说明，故这里可引用变量 k、函数f1()和f2()*/
        ⋮
}
```

在一个源程序文件中，调用另一个源程序文件中定义的函数时，也要对被调用的函数作外部说明。但由于函数总是全局的，对函数的外部说明常常不用关键词extern

寄存器存储类

- 只有int型、char型及指针类型的自动变量和形参才可以是寄存器存储类的。全局变量或其它复杂数据类型的变量都不可以是寄存器存储类
- 定义变量是寄存器存储类，只要在变量定义时，在类型说明符之前冠以关键字“register”即可
  - 如 register int i; register char ch;
  - 参见如下形式的函数定义

```
f(i, j)
register int i;  /* 形参 i 是寄存器的 */
int j;           /* 形参j不是寄存器的 */
{  register int k;  /* 自动的变量 k 是寄存器的 */
   ...
 }
```

寄存器存储类(续)

- 将一个自动变量的存储类指定为寄存器，是提醒编译程序，这个变量在程序中使用得十分频繁。在为该变量分配存储时，有可能的话，尽可能为它分配寄存器，因为访问寄存器要比访问内存储单元快

- 将一个形参的存储类指定为寄存器，可能是因为要访问某些特殊设备的驱动程序，这些设备的驱动程序要求以寄存器为参数与系统传递信息

寄存器存储类(续)

- 含有寄存器存储类的形参或自动变量的函数，当它被调用时，该函数将会用一些寄存器存放形参或变量的值，函数调用结束释放它占用的寄存器
- 因计算机寄存器数目有限，只能有很少几个寄存器存储类的形参或变量
- 因实现C语言的系统环境不同，不同系统在实现C语言的各种设施的方法也有差异
  - 某些C系统把寄存器存储类自动变量全部作为一般自动变量处理，为它们分配存储单元，而并不真正为它们分配寄存器
- 需特别指出：寄存器存储类的变量不能施取地址运算(用运算符&)和不能是静态的

存储类对比

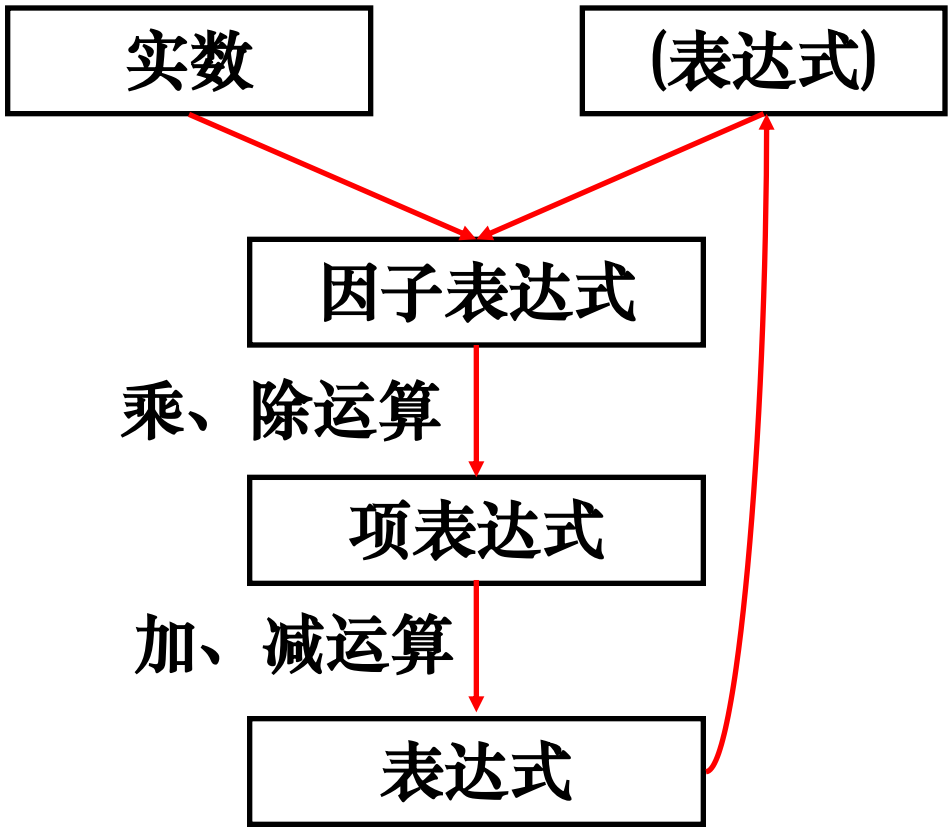
存储类型	作用域	生命周期	存储位置	默认初值
`auto`	代码块内	代码块执行期间	栈	随机值
`static`（局部）	代码块内	整个程序	静态区	0
`static`（全局/函数）	本文件内	整个程序	静态区	0
`extern`	全局（跨文件）	整个程序	静态区	0（定义处）
`register`	代码块内	代码块执行期间	寄存器（建议）	随机值

代码块：复合语句；本文件：当前源程序文件

存储类程序设计实例

- 下面程序实例说明局部变量、全局变量和静态全局变量的作用和用法
  - 编制一个能完成加(+)、减(-)、乘(*)、除(/)四则运算数值表达式的计算程序
  - 有很多方法实现这个程序，这里采用递归下降法
    - 一种自顶向下的语法分析方法
  - 具体来说，运算数值表达式可分成四级
    - 实数是表达式最简单的形式
    - 接着是因子表达式，因子表达式（factor）有两种可能形式：或是实数，或是带园括号的表达式
    - 对若干因子表达式组成的连续乘除运算，构成项表达式(term)
    - 由若干项表达式组成的连续加减运算构成表达式(expression)

四则运算数值表达式



存储类程序设计实例(续)

- **编制一个能完成加(+)、减(-)、乘(*)、除(/)四则运算的数值表达式的计算程序(续)**
  - 数值表达式的基本词法单位有以下几种：实数、四则运算符、圆括号、换行符、结束符、其它非法符号
  - 数值表达式的计算程序由几个函数组成
    - 单词分析函数getToken()从输入字符列，分析出一个独立单词，并得到其内部标记
      - 如单词是**实数**，函数将返回数值运算分量的内部标记
      - 如单词是**运算符**或是**其它字符**，返回对应字符的内部标记

存储类程序设计实例(续)

- 完成四则运算的数值表达式的计算程序(续)
  - 把完成因子表达式计算、项表达式计算和表达式计算分别编写成独立的函数
  - 将程序中有关常量和函数说明写在头文件trans.h中。头文件trans.h主要定义有关常量，各单词的内部标记和分散在程序文件中的函数说明等

```
/*头文件trans.h*/
#define NUM    1
#define ADD    2
#define SUB    3
#define MUL    4
#define DIV    5
```

```
#define NL     6
#define LP     7
#define RP     8
#define END   -1
#define ERR    0
```

```
extern int token;
int getToken(char *, double *);
int getch(void);
void ungetch(int);
void initBuf(void);
double expr(void);
double term(void);
double factor(void);
```

存储类程序设计实例(续)

- 完成四则运算的数值表达式的计算程序(续)：主函数调用表达式计算函数实现表达式的计算，代码如下

```
/* 设主函数文件名为 expmain.c */
 #include  <stdio.h>
 #include  "trans.h"
 int main()
{  double result; char ans;
    while (1){  initBuf();  printf("Enter numeric expression!\n");
      result=expr(); printf("The expression's result is %.6f\n", result);
      printf("Continue?(y/n)\n");  scanf(" %c", &ans);
      if (ans != 'y' && ans != 'Y') break;
      while(getchar() != '\n'); /* 掠过回答字符后的回车符 */
    }
}
```

存储类程序设计实例(续)

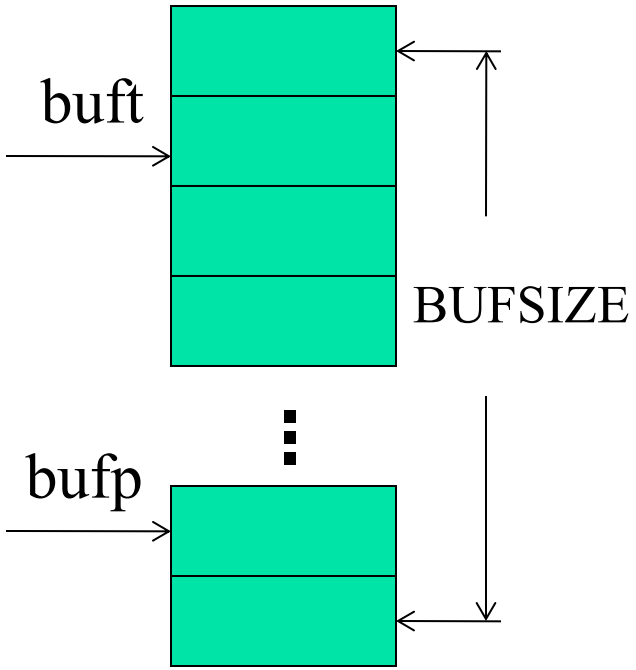
```
/* 输入单词函数的源程序文件，设文件名为 gettoken.c */
#include  <ctype.h>
#include  "trans.h"
int symble[][2] = {'+', ADD, '-', SUB, '*', MUL, '/', DIV,
                   '(', LP, ')', RP, '\n', NL, -1, EOF, 0, ERR};
int token;
int getToken(char *chp, double *dp)  /* 返回输入类型 */
{ int k, c;  double num, rad;
   while ((c = getch()) == ' ' || c == '\t') ;
    if ((c < '0' || c > '9') && c != '.') {  *chp = c; / * 不是数!!! */
        for(k = 0; symble[k][0]; k++)
            if (c == symble[k][0]) break;
        return symble[k][1]; /* 返回一个符号 */
    }
```

存储类程序设计实例(续)

```
    num  = 0.0;
    while(c >= '0' && c <= '9') { /* 计算整数部分 */
            num = num *10.0 + c - '0';    c = getch();
     }
    if (c == '.') {  /* 计算小数部分 */
       rad = 0.1;
       while ((c = getch()) >= '0' && c <= '9') {
          num += (c - '0')*rad;     rad /= 10.0;
        }
     }
    ungetch(c); /* 把最后一个字符存起来 */
    *dp = num;
    return NUM; /* 返回一个具体的参与运算的数 */
}
```

存储类程序设计实例(续)

```
/* 输入字符函数和返还输入字符函数的源程序文件。设文件名为
getch.c */
    #define BUFSIZE  10
    static int buf[BUFSIZE];
    static int bufp, buft;
/* bufp表示buf中下一个要取的数的位置 */
/* buft表示buf中下一个要存储数据的位置 */
    int getch()
    {  int c;
       if (bufp == buft) return getchar();
       c = buf[bufp];
       bufp = (bufp + 1) % BUFSIZE;
       return c;
    }
```



存储类程序设计实例(续)

```
void ungetch(int c)      /* 返还字符函数 */
 {  int t = (buft+1) % BUFSIZE;
    if (t == bufp) /* 检查存了该数后，buf是否满？ */
          printf("ungetch's buffer is FULL!\n");
    else {buf[buft] = c; buft = t; }
 }

void initBuf(void)
 {
      bufp = buft = 0;
 }
```

存储类程序设计实例(续)

以下是定义函数expr()、term()和factor()的源程序文件。设文件名为 exp.c

```
#include <math.h>
#define Epsilon  1.0e-5
#include "trans.h"
/* 表达式计算函数调用项计算函数完成连续的加减计算 */
double expr(void)
{  double ev, tv;     char op, ch;    ev = term();
   while ((op = getToken(&ch, &tv))== ADD || op == SUB) {
       tv = term();
       if (op == ADD) ev += tv;
       else ev -= tv;
    }
   ungetch(ch);      return ev;
}
```

存储类程序设计实例(续)

```
/* 项计算函数调用因子计算函数完成连续的乘除计算 */
double term()
{  double tv, fv, non;       char op, ch;     tv = factor();
   while ((op = getToken(&ch, &non)) == MUL || op == DIV) {
        fv = factor();
        if (op == MUL)  tv *= fv;
        else if (fabs(fv) > Epsilon) tv /= fv;
             else {  printf("Error! Divided by zero.\n");
                     return 1.0;
              }
    }
   ungetch(ch);    return tv;
}
```

存储类程序设计实例(续)

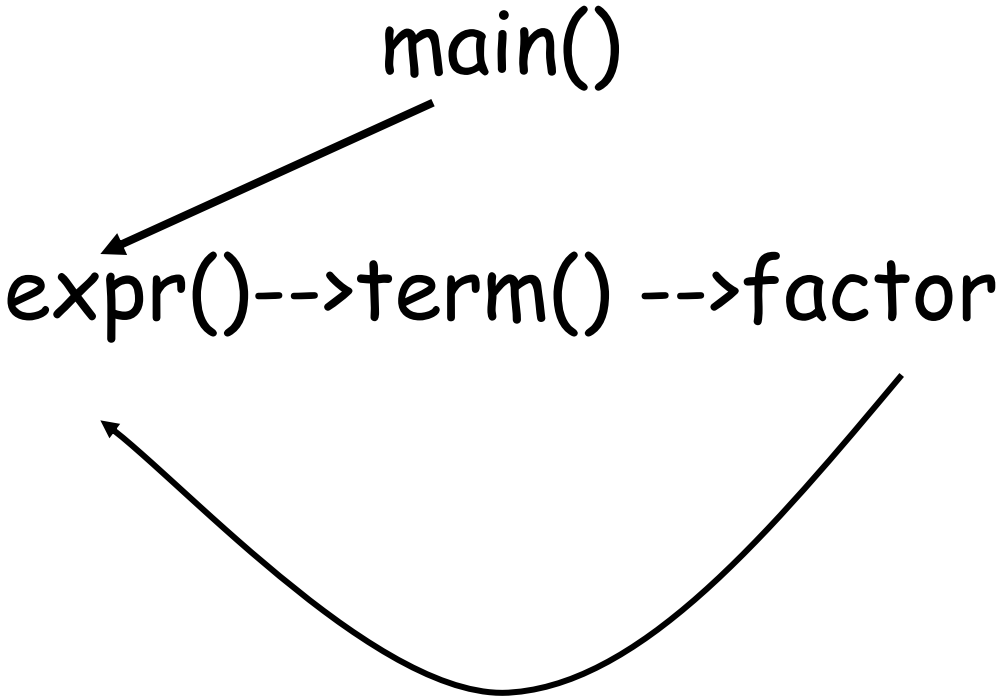
```
/*因子计算函数或获得一个实数值，或递归调用表达式计算函数获得因子的值 */
double factor(void)
{   char ch;    double res, non;     token = getToken(&ch, &res);
    if(token == NUM)   return res;
    if (token = LP) {  res = expr();
            if (getToken(&ch, &non) == RP)   return res;
            ungetch(ch);   printf("Error! no right parenthesis\n");
            return 1.0;
       }
    ungetch(ch);   printf("Error! Expect factor.\n");
    return 1.0;
}
```

存储类程序设计实例(续)

- 要将上述源程序文件构成一个程序，在TURBO C环境，可预设以下项目文件(设文件名为exp.prj)，并编译该文件即可

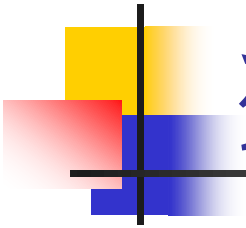
getch.c
gettoken.c
exp.c
expmain.c

这是一个递归程序！



变量定义与变量说明

- 在C中，变量定义和变量说明的作用是不同的
  - 变量定义：如int a[100]; 出现在任何函数之外，则它定义了全局变量，在程序执行之前就要为它分配存储单元。同时，该定义还作为该源程序文件余下部分的变量说明，使源程序文件余下部分可直接引用它
  - 变量说明：只是告诉编译系统，某标识符所代表的意义，不要求为说明提及的变量分配存储单元。如变量说明 extern int a[]; 只是对源程序文件余下部分说明标识符a是数组，它的元素是int型的，并不要求为它分配存储单元。因此在数组说明中，数组元素个数不必指定

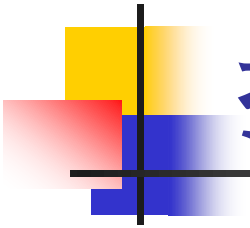


变量分类

- 从存储类别
  - 自动变量
  - 静态变量
  - 外变量
  - 寄存器变量
- 从定义位置
  - 局部变量
  - 全局变量

变量的可见性与存在性

- 局部变量：存在即局部可见
- 寄存器变量：存在即局部可见
- 全局变量：存在即全局可见
- 外部变量：同全局变量
- 静态局部变量
  - 长期存在，局部短期可见
- 静态全局变量
  - 长期存在，局部短期可见



提要

- 局部变量和全局变量
- 存储类
- 编译预处理命令简介
  - 宏定义、文件包含、条件编译和行控制

编译预处理命令简介

- 编译预处理是指在对源程序作正常编译之前(包括词法和语法分析、代码生成、优化等)，先对源程序中一些特殊的预处理命令作解释，产生一个新的源程序；然后再对新的源程序进行常规的编译
- C语言提供的预处理命令（preprocessor directive）主要有
  - 宏定义、文件包含、条件编译和行控制
- 在源程序中，为区别预处理命令和一般的C代码，所有预处理命令行都以字符"#"和预处理命令开头

为什么要编译预处理命令?

- 目的在于在正式编译之前，对源代码进行纯文本层面的“预加工”，以**弥补C语言本身在代码组织、跨平台适配和编译期配置能力上的不足**
  - 实现代码复用与模块化（文件包含）
  - 提供符号常量与轻量级代码抽象（宏定义）
    - 可读性与可维护性（用宏定义常量）；性能考量（用宏定义取代函数）
  - 支持条件编译（跨平台与配置管理）
    - 跨平台移植，调试与发布版本切换
  - 弥补语言本身的“编译期”能力缺失

编译预处理命令：宏定义

- 宏定义两种形式
  - 不带形式参数宏定义
  - 带形式参数宏定义
- 不带形式参数的宏定义
  - 一般形式为：　#define　标识符　字符序列
  - #define之后标识符称为宏定义名(简称宏名)，要求宏名与字符序列之间用空格符分隔
  - 这种宏定义要求编译预处理程序将源程序中随后出现的所有宏名 (注释与字符串常量中的除外)均用字符序列替换
    - 前面经常使用的定义常量是宏定义的最简单应用

编译预处理命令：宏定义(续)

- **不带形式参数的宏定义例子**
  - 如有 `#define TRUE 1`
    `#define FALSE 0`
  - 则在它们被定义的源程序文件中，后面程序正文中凡出现TRUE的地方将用1替代之，FALSE的地方用0替代之
  - 以下是经常使用的不带参数宏定义的例子

```
#define PI 3.1415926
#define NL printf("\n")
#define EOF (-1)
```

不带形式参数的宏定义

- 不带形式参数的宏定义的格式说明及辖区
  - 在宏定义的#之前可以有若干空格、制表符，但不允许有其它字符
  - 宏定义在源程序中单独另起一行，换行符是宏定义的结束标志。如果一个宏定义太长，一行不够时，可采用续行的方法。续行是在键入回车符之前先键入符号‘\’。 注意回车要紧接在符号‘\’之后，中间不能插入其它符号
  - 宏定义的有效范围称为宏定义名的辖域，辖域从宏定义的定义结束处开始到其所在的源程序文件末尾。宏定义名的辖域不受程序结构的影响。可以用预处理命令#undef终止宏定义名的辖域

不带形式参数的宏定义

- **不带形式参数的宏定义的辖区示例**
  - 如
```
#define Macro 5.4
main()
{  ⋮
 }
#undef Macro
f()
 ⋮
```
  - 在以上正文中，由于预处理命令#undef Macro，使Macro的辖域在#undef行处终止。因此，从函数f()开始，Macro已不再有定义

不带形式参数的宏定义

- **不带形式参数的宏定义的重复定义**
  - 在新的宏定义中，可以使用前面已定义的宏名
  - 如
```
#define R     2.5
#define PI    3.1415926
#define Circle  2*PI*R
#define Area   PI*R*R
```
    - 程序中Circle展为2*3.1415926*2.5, Area为3.1415926*2.5*2.5
  - **如有必要，同一宏名可被重复定义。被重复定义后， 宏名原先的意义被新的意义所代替**
  - 如
```
#define PI 3.14
    ⋮
#define PI 3.1415926
```
则随后的宏名 PI 就代表 3.1415926

不带形式参数的宏定义

- **不带形式参数的宏定义的应用**
  - **多用于定义常量**
  - 用宏表示常量值，便于程序前后统一，不易出错；便于修改，可提高程序可读性和可移植性
  - 给数组元素个数一个宏定义，用宏名定义数组元素个数能部分弥补数组元素个数固定的不足
  - 注意，**预处理程序在处理宏定义时，只作字符序列的替换工作，不作任何语法的检查。**如果宏定义不当，错误要到预处理之后的编译阶段才能发现

不带形式参数的宏定义

- **不带形式参数的宏定义的换行**
  - 宏定义以换行结束，不需要分号等符号作分隔符
  - 如有以下宏定义 `#define PI 3.1415926;`
  - 原希望用PI求园的周长的语句 `circle = 2*PI*r;`
  - 经宏展开后，变成 `circle = 2*3.1415926; *r;`
    这就不能达到希望的要求

编译预处理命令：宏定义(续)

- **带形式参数的宏定义**
  - 带形式参数的宏定义进一步扩充了不带形式参数宏定义的能力，在字符序列替换同时还能进行参数替换
  - 带形式参数宏定义的一般形式为

    #define 标识符(形式参数表) 字符序列
  - 形式参数表中的形式参数之间用逗号分隔，字符序列中应包含形式参数表中的形式参数
  - 在定义带形式参数的宏时，标识符（宏名）与左园括号之间不允许有空白符，应紧接在一起，否则变成了不带形式参数的宏定义

带形式参数的宏定义

- **带形式参数的宏定义示例**
  - 宏定义： #define MAX(A, B) ((A) > (B) ? (A) : (B))
  - 代码 ： y = MAX(p+q, u+v)
  - 将被替换成 y =((p+q) > (u+v) ? (p+q) : (u+v))
  - **程序中宏调用是被替换展开的，即分别用宏调用中实参数字符序列替换宏定义字符序列中对应所有出现的形式参数**(如用p+q替代所有形式参数A，用u+v替代所有形式参数B)，**而宏定义字符序列中不是形式参数的其它字符则保留。**这样形成的字符序列，即为宏调用的展开替换结果
  - **宏调用提供实在参数个数须与宏定义中形式参数个数相同**

带形式参数的宏定义

- **带形式参数的宏调用**与**函数调用**的**区别**
  - **函数调用**时，实参表达式分别独立求值在前，执行函数体在后
  - **宏调用**是实在参数字符序列替换形式参数，替换后实在参数字符序列就与相邻的字符自然连接，实在参数的独立性就不一定依旧存在
  - 如宏定义#define SQR(x) x*x希望实现表达式的平方计算。对于宏调用p=SQR(y)能得到希望的宏展开p=y*y。但对于宏调用q=SQR(u+v)得到的宏展开是q=u+v*u+v。显然，后者的展开结果不是程序设计者所希望的

带形式参数的宏定义

- 宏调用与函数调用的区别
  - 为保持实在参数替换后的独立性，应在宏定义中给形式参数加上括号
    - 如：SQR宏定义改写成 #define SQR(x) (x)*(x)
  - 为了保证整个宏调用的独立性，有时还需将整个宏定义表达式加上括号
    - 如：SQR宏定义改写成：#define SQR(x) ((x)*(x))

带形式参数的宏定义

- 宏调用与函数调用的区别
  - 函数调用在程序运行时实行，而宏展开是在编译的预处理阶段进行
  - 函数调用占用程序运行时间，宏调用只占编译时间
  - 函数调用对实参有类型要求，而宏调用实在参数与宏定义形式参数之间没有类型的概念，只有字符序列的对应关系
  - 函数调用可返回一个值，宏调用获得希望的C代码

带形式参数的宏定义示例

如宏定义：

```
#define EXCHANGE(x, y) t = x; x = y; y = t;
#define PI 3.1415926
#define CIRCLE(R,L,S,V) L = 2*PI*(R);\
        S = PI*(R)*(R);\
        V = 4.0/3.0*PI*(R)*(R)*(R); /* R两边的园括号不能少 */
```

宏调用EXCHANGE(a, b)的宏展开为

```
t = a; a = b; b = t;
```

而宏调用CIRCLE(r+2.0, clen, area, volume)的宏展开为

```
clen = 2*3.1415926*(r+2.0);
area = 3.1415926*(r+2.0)*(r+2.0);
volume=4.0/3.0*3.1415926*(r+2.0)*(r+2.0)*(r+2.0);
```

宏定义示例

- 对于简短的表达式计算函数，为了提高程序的执行效率，可将函数定义改写成宏定义。所以合理使用宏定义，可以使程序更简洁
- 下面的例子能说明这样的应用。程序先将使用的输出格式定义成宏，然后调用宏实现数据的输出

```
#define PR printf
#define NL "\n"
#define F "%6.3f"
#define F1 F NL
#define F2 F F NL
#define F3 F F F NL
```

宏定义示例

```
main ()
{ float x, y, z;
  x = 1.2;  y = 2.3;  z = 3.4;
  PR (F1, x);  PR (F2, x, y);  PR (F3, x, y, z);
 }
```

运行该程序，输出结果：

```
1.200
1.200  2.300
1.200  2.300  3.400
```

程序用PR代表printf，以NL代表换行，F代表浮点型输出格式。又用以上三个宏定义定义输出一个浮点数的格式F1，输出两个浮点数的格式F2和输出三个浮点数的格式F3

宏定义的字符序列

- **在宏定义的字符序列中有用双引号括起来的字符串，并且字符串中有与形式参数名相同的内容，则在宏调用展开时，整个字符串被完整复制，不会用实在参数的内容替代形式参数**
  - 如：#define PRINT(v) printf ("v = %d\n", v)
  - 宏调用PRINT(i)和PRINT(j)分别被展开成
    printf("v = %d\n", i) 和 printf("v = %d\n", j)
  - 而不会展成printf("i = %d\n", i) 和 printf("j = %d\n", j)

宏定义的字符序列(续)

- 预处理程序将宏定义中的字符序列看作由语言基本单位组成的序列(也称单词序列)，即由标识符、字符串、常量、定义符等构成的序列，以此来识别出字符序列中的形式参数
- 对预处理程序来说，用双引号括起来的字符序列是字符串，它已是不能再细分的基本词法单位，不再考虑在字符串之内是否还有形式参数出现等情况
- 为了实现象前述例子中后者所示那样的替换要求，ANSI C采用变相的方法，增加了两个特殊的预处理字符“#”和“##”

宏定义的字符序列(续)

- 在宏定义的字符序列中，若一个形式参数标识符前有#号，则该形式参数与#号一同被相应实在参数字符列替换，并在实在参数字符列的前和后都加上双引号，所以字符#能将实在参数字符列构成字符串
  - 如有宏定义#define STR(pt,str) pt = #str
  - 代码STR(s, a string)被替换成 s = "a string"
  - 而宏定义 #define PRINT(v) printf (#v" = %d\n", v)
    使代码 PRINT(i) 和 代码PRINT(j) 将分别被替换成
    printf("i = %d\n", i) 和 printf("j = %d\n", j)

宏定义的字符序列(续)

- 在宏定义的字符序列中，若一个形式参数前有双字符##，则在进行宏替换时，先用实在参数字符列替换形式参数，再把得到的字符列中的双字符##都删去，并将其前后两个相邻单词合成一个单词
  - 如 #define DEBUG(s,t) printf("y"#s"=%d,y"#t"=%d",y##s,y##t)
  - 代码“DEBUG(1,2)”先被替换成

    printf("y" "1" "=%d,y" "2" "=%d", y##1, y##2)
  - 由于两个紧接的字符串常量会自动连接成一个字符串常量，并将所得字符序列中的##删去，上述替换最终结果为

    printf("y1=%d,y2=%d", y1, y2)

宏定义的进一步说明-1

- **宏可以嵌套定义。为处理可能有嵌套的宏定义，预处理程序对替换后得到的串要重新扫描**
  - 例如

```
#define X Y/10
X+c
#define Y 8+z*X
X+c
```

  - 对上述两个X+c，前一个被替换成Y/10+c；后一个将被替换成8+z*X/10+c

宏定义的进一步说明-1(续)

- 宏定义本身不会被别的宏定义替换成新的宏定义，如宏定义“#define Y 8+z*X”不被替换成“#define Y 8+z*Y/10”
- 宏定义的预处理过程采用多遍处理的方法实现
  - 如所得字符序列还有宏名，且该宏名在此位置是有效的，这个宏名还未被处理过，则将对该宏名作替换处理
  - 若新产生的字符序列又出现已被替换过的宏名、或新产生的宏名在该处是无效的，则不再对该宏名作替换处理
  - 如有 #define Z (Z)，代码 Z+Y 被替换成 (Z)+Y。对替换产生的Z是已替换过的宏名，不再对它进行再替换。如前面的后一个X+c，第一遍处理后，它被替换成Y/10+c。第二遍处理时，因Y的宏定义对它有效，再被替换成8+z*X/10+c。以后因宏X已被处理过，对新产生的X就不再替换

宏定义的进一步说明-2

- **带形式参数宏定义的形式参数表是一个用逗号分隔的标识符序列**
- **而宏调用时，除实在参数个数须与形式参数个数相同外，实在参数可以是任意字符序列，可以包含配对的括号，及配对的括号中可以包含逗号**
  - 如 `#define A(x) u x + v x`
  - 代码 `A((a, b))` 被替换成 `u(a, b) + v (a, b)`

文件包含预处理命令

- 文件包含预处理命令实现将指定文件的内容作为当前源程序的一部分
- 文件包含预处理命令(简称文件包含命令)一般形式为

  #include “文件名”　或　#include <文件名>

文件包含预处理命令示例

如有两个不同的文件：

文件format.h的内容是

```
#define  PR  printf
#define  NL  "\n"
#define  F   "%6.3F"
#define  F1  F  NL
#define  F2  F  F  NL
#define  F3  F  F  F  NL
```

文件example.c的内容是

```
#include <stdio.h>
#include "format.h"
void main( )
{ float x, y, z;
   x = 1.2; y = 2.3; z = 3.4;
   PR(F1, x); PR(F2, x, y); PR(F3,x, y, z);
 }
```

文件example.c中有文件包含命令#include “format.h“，使文件format.h的全部内容被复制并插在此文件包含命令处，形成新的文件。在接着往下进行的编译过程中，就以上述预处理后的新文件作为编译的源文件

文件包含预处理命令(续)

- 文件包含命令为组装大程序和程序文件复用提供了一种手段
- 在编写程序时，习惯将公共的常量定义、数据类型定义和全局变量的外部说明构成一个源文件。 称这类没有执行代码的文件为头文件，并以“.h”为文件名的后缀
- 其他程序文件凡要用到头文件中定义或说明的程序对象时，就用文件包含命令使它成为自己的一部分
- 这样编程的好处是各程序文件使用统一的数据结构和常量，能保证程序的一致性，也便于修改程序
- 头文件如同标准零件一样被其他程序文件使用，减少了重复定义的工作量

文件包含预处理命令(续)

- 头文件常被别的文件所包含，如头文件有修改，则所有包含它的文件都得重新编译
- 一个文件包含命令只能指定一个被包含文件。如要包含多个文件，就得用多个文件包含命令
- 在有多个文件包含命令的情况下，文件包含命令的出现顺序也是有关系的
  - 如果文件file1.c包含文件 head1.h和head2.h，且head2.h要引用文件head1.h中定义或说明的程序对象，则包含 head1.h 的文件包含命令必须出现在包含head2.h的文件包含命令之前

文件包含预处理命令(续)

- 一个被包含文件也可有文件包含命令，包含别的其他文件
- 在某些情况下，因多个文件都要包含一连串相关的文件，可编写一个全由文件包含命令组成的头文件包含这组相关文件
- 这样，一条包含这个文件的文件包含命令就能实现包含一组文件的作用

两种包含命令的区别

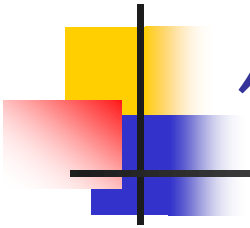
- 有两种包含命令，如

  #include <stdio.h> 和 #include "format.h"

  - 对于用双引号括住文件名的文件包含命令，预处理程序首先在当前文件所在的文件目录中寻找被包含文件。如找不到，再按预先指定的寻找顺序检索其他目录
  - 对于用尖括号括住文件名的文件包含命令，预处理程序不检查当前目录，直接按系统指定的寻找顺序检索文件目录
  - 一般地，用户自己编写的被包含文件多用双引号方式的文件包含命令；系统提供的被包含文件用尖括号方式的文件包含命令

第6章 小结

- 局部变量和全局变量
- 存储类
  - 自动的(auto)、静态的(static)
  - 外部的(extern)、寄存器的(register)
- 变量定义
- 编译预处理命令简介
  - 宏定义、文件包含
  - 条件编译和行控制



作业

- 习题五
  - 第1、2、3、5、10、14题