Chapter Five

Functions

1

Functions

- Functions can be either *library* functions or *user-defined* functions
- A function may *call* other functions to help it perform one of its subtasks
- Functions allow us to *reuse* pieces of code easily
- Library functions are the *most commonly used* functions

2

Library Functions

- stdio.h
  - printf, scanf
- stdlib.h
  - abs
- math.h
  - sqrt, pow, sin, fabs

3

Function Declarations

- All functions *must* be *declared before* they are *used*
- A *function declaration* specifies the *prototype* of a function

  result-type func-name (*argument-specifiers*) ;
- *func-name* is the *name* of the function
- *argument-specifiers* specifies the *type* of each argument to the function and a (optional) descriptive *name* for the argument
- *result-type* specifies the *type* of the value returned by the function

4

Examples

- stdio.h
  - void printf(char * format, …);
  - int scanf(char * format, …);
- stdlib.h
  - int abs(int n);
- math.h
  - double sqrt(double x);
  - double pow(double x);
  - double sin(double theta);
  - double fabs(double x);

5

User-defined Functions

- Programmers can define their *own* functions if the required functions are not provided in the library
- A piece of code can be considered to be written as a function if it will be *used multiple times* in the program or it will be *used by other programs* (i.e., build your own library)

6

An Example

Compute the combination function

$$C(n, k) = n! / (k! \times (n - k)!)$$

7

Function Definitions

- A *function definition* specifies the *implementation details* of a function
- A function definition has this form:

result-type func-name (*parameter-specifiers*)
{
    declarations
    statements
}

Head

Body

8

An Example

```
int fact(int m)
{
   int  product, i;                 } declarations

   product = 1;
   for (i = 1; i <= m; i++) {       } statements
      product *= i;
   }
   return product;
}
```

9

Return Statements

- Most functions will evaluate to a value. This value is passed back to the calling function via *return statements*
- A *return statement* has the form
  return expression;
  the value of *expression* is passed back as the value evaluated by the function

10

Return Statements

- If a function does not evaluate to a value, its *result-type* is *void*
- If a function's result-type is *void*, its return statement has the form
  return;
  and this statement can be omitted
- If a function's result-type is *int*, its result-type can be omitted

11

Examples

```
void printf(char * format, …);

int fact(int m);
fact(int m);
```

12

An Example

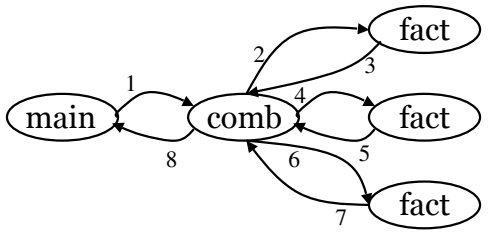
```
int fact(int m);

int comb(int n, int k) {
   return fact(n) / (fact(k) * fact(n – k));
}

main( ) {
   int n, k;
   scanf("%d %d", &n, &k);
   printf("C(%d, %d) = %d\n", n, k, comb(n, k));
}
```

13

Flow of Control



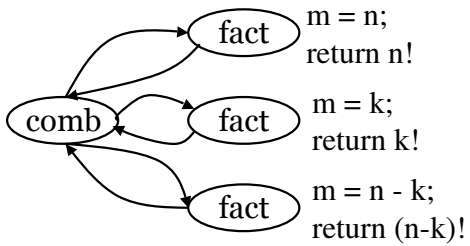
14

Parameter Passing

- The values of each *argument* are *evaluated*
- The values of *arguments* are *assigned* to the *parameters* in order. If necessary, automatic type conversion are performed

15

An Example



16

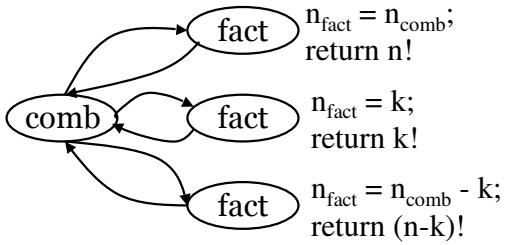
Local Variables

- Variables declared within a function are called *local variables* or *automatic variables*
- The *scope* of a local variable declaration is restricted *within the function body*; therefore, different functions may use local variables with the same name

```
int fact(int n);
int comb(int n, int k)
```

17

An Example



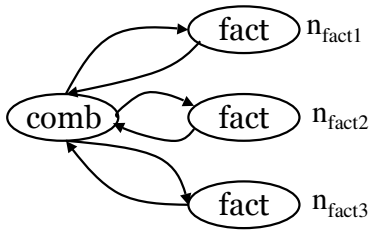
18

Local Variables

- Each *local variable* in a function *comes to existence* only when the function is *called*, and *disappears* when the function is *exited*
- The *allocation* and *deallocation* of memory for local variables are performed *automatically*
- Local variables must *be explicitly set* upon each entry

19

An Example



20

Predicate Functions

- A function is called a *predicate* function if it returns a value of Boolean type

```
int  isEven(int n)
{
   return (n % 2 == 0);
}
```

21

Symbolic Constants

- *Symbolic constants* facilitate the understanding and maintenance of programs

```
#define  symbol  value

#define  pi  3.14159
```

22

Defining New Type Names

- The keyword typedef introduces a *new type name* for an old type

```
typedef  old-type  new-type;

typedef  int  bool;
```

23

An Example

```
typedef  int  bool;
#define  TRUE  1
#define  FALSE  0

bool  isLeapYear(int year)
{
   return ( (year % 4 == 0)  &&
            (year % 100 != 0) ||
            (year % 400 == 0) );
}
```

24

Macros

- Functions incur *overheads* for passing arguments, managing memory for local variables, returning values, and so on
- *Macros* allow us to use *text substitution* to avoid such overheads for *short* functions

```
#define  name  value
```

25

An Example

```
#define  isEven(n)  ((n) % 2 == 0)

if  (isEven(i)) { … }

if  (((i) % 2 == 0)) { … }
```

26

Macros

- Avoid arguments that have *side effects* when the corresponding parameter occurs *multiple times* in the body

```
#define  isLeapYear(year) \
            ( ((year) % 4 == 0)  && \
              ((year) % 100 != 0) || \
              ((year) % 400 == 0) )
```

27

Stepwise Refinement

- Functions enable us to divide a large programming problem into smaller pieces that are *individually easy to understand*
- *Stepwise refinement* (or *top-down design*) allows us to start with the main function, and then refine step-by-step its subtasks by dividing it into gradually smaller functions

28

An Example

Print calendar

December 2001

Su	Mo	Tu	We	Th	Fr	Sa
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

29

An Example

```
main( )
{
   int  year;

   giveInstructions( );
   year = getYearFromUser( );
   printCalendar(year);
}
```

30

An Example

```
void giveInstructions(void)
{
   printf(“This program displays a calendar for a full\n”);
   printf(“year. The year must not be before 1990.\n”);
}
```

31

An Example

```
int getYearFromUser(void)
{
   int  year;

   while (TRUE)  {
      printf(“Which year? ”);
      scanf(“%d”, &year);
      if (year >= 1900) return year;
      printf(“The year must be at least 1900.\n”);
   }
}
```

32

An Example

```
void printCalendar(int year)
{
   int  month;

   for (month = 1; month <= 12; month++) {
      printCalendarMonth(month, year);
      printf("\n");
   }
}
```

33

An Example

```
void printCalendarMonth(int month, int year) {
   int  weekday,  nDays,  day;
   printf("    %s  %d\n", monthName(month), year);
   printf(" Su Mo Tu We Th  Fr  Sa\n");
   nDays = monthDays(month, year);
   weekday = firstDayOfMonth(month, year);
   indentFirstLine(weekday);
   for (day = 1; day <= nDays; day++) {
      printf(" %2d", day);
      if (weekday == Saturday) printf("\n");
      weekday = (weekday + 1) % 7;
   }
   if (weekday != Sunday) printf("\n");
}
```

34

An Example

```
#define  Sunday      0
#define  Monday      1
#define  Tuesday     2
#define  Wednesday   3
#define  Thursday    4
#define  Friday      5
#define  Saturday    6
```

35

An Example

```
char *monthName(int month)
{
   switch(month) {
      case  1: return "January";
      case  2: return "Febryary";
      case  3: return "March";
         …
      case  11: return "November";
      case  12: return "December";
      default: return "Illegal month";
   }
}
```

36

An Example

```
int monthDays(int month, int year)
{
   switch (month) {
      case 2:
         if (isLeapYear(year)) return 29;
         return 28;
      case 4: case 6: case 9: case 11:
         return 30;
      default:
         return 31;
   }
}
```

37

An Example

```
int firstDayOfMonth (int month, int year)  {
   int  weekday,  i;
   weekday = Monday;
   for (i = 1900; i < year; i++) {
      weekday = (weekday + 365) % 7;
      if (isLeapYear(i)) weekday = (weekday + 1) % 7;
   }
   for (i = 1; i < month; i++) {
      weekday = (weekday + monthDays(i, year)) % 7;
   }
   return weekday;
}
```

38

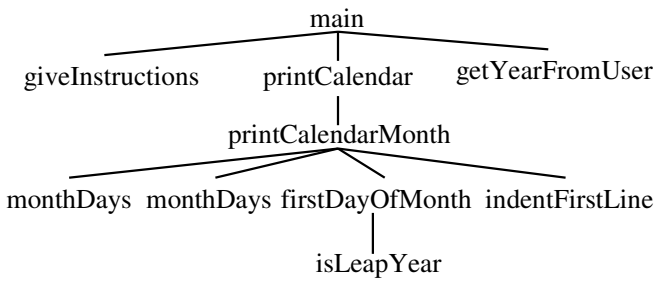
An Example

```
void indentFirstLine(int weekday)
{
   int  i;

   for (i = 0; i < weekday; i++) {
      printf("    ");
   }
}
```

39

An Example



40