
CIS 115

Computer Programming with C

Lecture 13, 03/29/2007

Li Shen
Computer and Information Science
UMass Dartmouth

Notes

- Office hour change this week
 - Tuesday (03/27): 11am-1pm
 - Thursday (03/29): 11am-1pm
 - Friday (03/30): no office hour
 - Homework
 - HW4 with complete description in the portal
 - Midterm evaluation
-

Flow of Control

- Relational Operators and Expressions
 - Equality Operators and Expressions
 - Logical Operators and Expressions
 - The Compound, Expression and Empty Statements
 - The `if` and the `if-else` statements
 - The `while` statement
 - The `for` statement
 - The Comma Operator
 - The `do` Statement
 - The `goto`, `break` and `continue` Statements
 - The `switch` Statement
 - The Conditional Operator
-

The Comma Operator

- The comma operator is occasionally useful in `for` statements. Of all the operators in C, it has the lowest priority.
 - The `,` operator is a binary operator with expressions as operands; it associates from left to right:
$$\textit{comma_expression} ::= \textit{expr} , \textit{expr}$$
 - In a comma expression of the form
$$\textit{expr1} , \textit{expr2}$$
`expr1` is evaluated first, and then `expr2`.
 - The comma expression returns the value and type of its right operand (e.g., `a = 0 , b = 1` will return `1` type `int` assuming that `b` is `int`)
-

The `do` Statement

- The `do` statement can be considered a variant of the `while` statement (or Pascal `repeat-until`), where instead of making its test at the top of the loop, it makes it at the bottom
do_statement ::= do statement while (expr) ;
 - First, statement is executed.
 - If the value of `expr` is nonzero (true), then control passes back to the beginning of the `do` statement and the process repeats itself.
 - When `expr` is zero (false) then control passes back to next statement after `do`.
-

The `goto` Statement

```
goto error;
.....
error: {
    printf("An error has occurred -bye!\n");
    exit(1);
}

while (scanf("%lf", &x) == 1){
    if (x<0.0)
        goto negative_alert;
    printf("%f %f\n", sqrt(x), sqrt(2*x));
}

negative_alert: printf("Negative value encountered!\n");
```

The break Statement

- The syntax of a break statement is
`break_statement ::= break ;`
- The break statement causes an exit from the innermost enclosing loop (while, for, do) or switch statement
- *Example:* a test for a negative argument is made, and if the test is true, then a break statement is used to pass control to the statement immediately following the loop

```
while (1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;
    /* no square root if number is negative, exit loop */
    printf("%f\n", sqrt(x));
}
/* break jumps to here */
```

The continue Statement

- The continue statement syntax
`continue_statement ::= continue ;`
- The continue statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately.
- The continue statement may occur only inside for, while and do loops
- *Example:* processing all characters in the for loop, except digits

```
for (i = 0; i < TOTAL; ++i) {
    c = getchar();
    if (c >= '0' && c <= '9')
        continue;
    ... /* processing other characters */
}
/* continue transfers control to here to begin next iteration */
```

The switch Statement

- The `switch` statement provides a multiway conditional branch, which can be understood as a generalization of the `if-else` statement.
- It is useful when dealing with a large number of special cases.
- The syntax for a switch statement is the following:

```

switch_statement ::= switch ( integral_expression )
                    { case_statement | default_statement | switch_block }1
case_statement ::= { case constant_integral_expr : }1+ statement
default_statement ::= default : statement
switch_block ::= { { declaration_list }opt { case_group }0+ { default_group }opt }
case_group ::= { case constant_integral_expr : }1+ { statement }1+
default_group ::= default : { statement }1+

```

The switch Statement

- The execution of a `switch` statement proceeds as follows
 - Evaluate the `switch` integral expression
 - Go to the `case` label having a constant value that matches the value of the expression found in step 1, or, if a match is not found, go to the default label, or, if there is no default label, terminate the switch.
 - Terminate the `switch` when a `break` statement is encountered, or terminate the switch by "falling off the end" (i.e., checking all possible cases without `break`).
- The constant integral expressions following the `case` labels must all be unique

The switch Statement

- Typically, the last statement before the next case or default label is a `break` statement.
 - If there is no `break` statement, the execution "fall through" to the next statement in the succeeding case.
 - There may be at most one `default` label in a switch. Typically, it occurs last, although it can occur anywhere.
 - The keywords `case` and `default` cannot occur outside of a `switch`.
-

The switch Statement

- *Example* of a switch statement:

```
switch (c) {
case 'a':
    ++a_cnt;
    break;
case 'b':
case 'B':
    ++b_cnt;
    break;
default:
    ++other_cnt;
}
```

The Conditional Operator

- The conditional operator `?:` is a ternary operator, i.e., it takes as operands 3 expressions:

conditional_expression ::= expr ? expr : expr

- In a construct such as

`expr1 ? expr2 : expr3`

- `expr1` is evaluated first.
 - If it is nonzero (true), then `expr2` is evaluated, and that is the value of the conditional expression as a whole.
 - If `expr1` is zero (false), then `expr3` is evaluated, and this is the value of the conditional expression as a whole.
-

The Conditional Operator

- A conditional expression can be used to do the work of an if-else statement, i.e.,

`if (y < z) x = y; else x = z;`

can be written using a conditional expression:

`x = (y < z) ? y : z;`

- The conditional operator `?:` has precedence just above the assignment operators, and it associates from right to left
-

The Conditional Operator

- Note that the type of the conditional expression is determined by both `expr2` or `expr3`.
- If they are of different types, then the usual conversion rules are applied, e.g.,
 - if `i = 1` and `j = 2` are of type `int`,
 - and `double x = 7.07`,
 - then `j % 3 ? i + 4 : x` will return value of type `double 5.0` because `x` is `double`

The Conditional Operator

Declarations and initializations			
<pre>Char a = 'a', b = 'b'; // a has decimal value 97 int i = 1, j = 2; double x = 7.07;</pre>			
Expression	Equivalent expression	Type	Value
<code>i == j ? a - 1 : b + 1</code>	<code>(i == j) ? (a - 1) : (b + 1)</code>	<code>int</code>	99
<code>j % 3 == 0 ? i + 4 : x</code>	<code>((j % 3) == 0) ? (i + 4) : x</code>	<code>double</code>	7.07
<code>j % 3 ? i + 4 : x</code>	<code>(j % 3) ? (i + 4) : x</code>	<code>double</code>	5.0

Flow of Control

- Relational Operators and Expressions
 - Equality Operators and Expressions
 - Logical Operators and Expressions
 - The Compound, Expression and Empty Statements
 - The `if` and the `if-else` statements
 - The `while` statement
 - The `for` statement
 - The Comma Operator
 - The `do` Statement
 - The `goto`, `break` and `continue` Statements
 - The `switch` Statement
 - The Conditional Operator
-

Functions

Outline

- Function definition
 - The `return` statement
 - Function prototypes
 - Creating a table of powers
 - Function declarations from the compiler's viewpoint
 - An alternate style for function definition order
 - Function invocation and call-by-value
 - Developing a large program
 - Using assertions
 - Scope rules
 - Storage classes
 - Static external variables
 - Default initialization
 - Recursion
 - The towers of Hanoi
-

Functions

- Functions are the most general structuring concept in C.
 - They should be used to implement "top-down" problem solving
 - Namely, breaking up a problem into smaller and smaller sub problems until each piece is readily expressed in code
-

Function Definition

- The function definition describes what a function does and has the following BNF syntax

```
type function_name( parameter_list ) /* header */  
{ declarations statements }          /* body */
```

- Everything before the first brace comprises the *header*, and everything between the braces comprises the *body* of the function definition

```
long power(int m, int n)  
{  
    int    i;  
    long   product = 1;  
  
    for (i = 1; i <= n; ++i)  
        product *= m;  
    return product;  
}
```

Function Definition

- A function definition starts with the *type* of the function.
 - If no value is returned, then the *type* is void (and becomes an equivalent to what in some other languages is called a *procedure*).
 - If the *type* is something other than void, then the value returned by the function will be converted, if necessary, to this *type*.
 - If a function definition does not specify the function type, then it is `int` by default (this includes the situation if we skip *type* in `main()` function definition).
 - The *parameter_list* follows *function_name*, and is a comma separated list of *formal parameter* declarations that act as placeholders for values that are passed when the function is invoked
 - The function body is a *compound statement*, and it too may contain declarations of local variables/data structures
-

The `return` Statement

- The `return` statement terminates the execution of a function and passes control back to the calling environment.
- It may or may not include an expression

```
return_statement ::= return { expression }opt ;
```

- If the `return` statement contains an expression, then the value of the expression is passed back to the calling environment as well
- The expression being returned can be enclosed in parentheses, but this is not required, e.g.,

```
return;  
return ++a;  
return (a * b);
```

Function Prototypes

- Functions should be declared before they are used.
 - ANSI C provides for a function declaration syntax called the *function prototype*
 - The function prototype is the special case of the *function declaration*, it tells the compiler
 - the number and type of arguments that are to be passed to the function and
 - the type of the value that is to be returned by the function
-

Function Prototypes

- It has the following BNF syntax

```
type function_name( parameter_type_list ) ;
```

- The *parameter_type_list* is typically a comma-separated list of types.
- Identifiers are optional; they do not affect the prototype
- For example, the function prototypes are equivalent

```
void f(char c, int i);  
void f(char, int);
```

Creating a Table of Powers

```
#include <stdio.h>  
  
#define N 7  
  
long power(int, int);  
void prn_heading(void);  
void prn_tbl_of_powers(int);  
  
int main(void)  
{  
    prn_heading();  
    prn_tbl_of_powers(N);  
    return 0;  
}  
  
void prn_heading(void)  
{  
    printf("\n:::::: A TABLE OF POWERS :::::\n\n");  
}
```

[tbl_of_power.c](#)

Creating a Table of Powers

```
void prn_tbl_of_powers(int n)
{
    int i, j;

    for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j)
            if (j == 1)
                printf("%ld", power(i, j));
            else
                printf("%9ld", power(i, j));
            putchar('\n');
        }
    }

long power(int m, int n)
{
    int i;
    long product = 1;

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

Outline

- Function definition
 - The `return` statement
 - Function prototypes
 - Creating a table of powers
 - Function declarations from the compiler's viewpoint
 - An alternate style for function definition order
 - Function invocation and call-by-value
 - Developing a large program
 - Using assertions
 - Scope rules
 - Storage classes
 - Static external variables
 - Default initialization
 - Recursion
 - The towers of Hanoi
-

After Class

- Read
 - Kelley Chapters 4 and 5
 - Work on HW4
-