LESSON 6

FLOW OF CONTROL

This lesson discusses a variety of flow of control constructs. The break and continue statements are used to interrupt ordinary iterative flow of control in loops. In addition, the break statement is used within a switch statement. A switch statement can select among several different cases. It can be considered a generalization of the if-else statement. Similar in function to an if-else statement, the conditional operator provides for the selection of alternative values. The lesson also discusses the nested flow of control structures and a brief discussion of the goto statement. After that, there is a discussion of nested blocks and their effect on the "scope" of variables, in conjunction with the different types of storage class attributes.

THE break AND continue STATEMENTS

To interrupt normal flow of control within a loop, you (the programmer) can use the two special statements break; and continue. The break statement, in addition to its use in loops, can also be used in a switch statement. It causes an exit from the innermost enclosing loop or switch statement.

The following example illustrates the use of a break statement. A test for a negative value is made, and if the test is true, the break statement causes the while loop to be exited. Program control jumps to the statement immediately following the loop.

```
while(scanf("%Lf", &x)  ==  1){
      if (x   <  0.0)  {
printf ("Break out of loop x is negative.\n");
      break;      /* exit loop if value is negative */
      }  /* end of if. */
printf ("%Lf\n", sqrt(x));
}  /* end of while loop */
/* break jumps to here */
```

This is a typical use of a break statement. When a special condition is met, an appropriate action is taken and the loop is exited. In this example if x is less than 0.0 the value is printed and the break is executed and the program jumps out of the while loop.

The continue statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately. The following code processes all characters except digits.

```
while ((c = getchar(   ))   !=   EOF) {
     if(isdigit(c))
        continue;


      /*Some       program statements to           */
      /*       process other characters            */




/* The continue jumps to just before the closing brace */
}/* end of while loop  */
```

In this example all characters except digits are processed. When the continue statement is executed, control jumps to just before the closing brace, causing the loop to begin execution at the top again. Notice that the continue statement ends the current iteration, whereas a break statement would end the entire loop.

A break statement can occur only inside the body of a for, while, do, or switch statement. The continue statement can occur only inside the body of a for, while, or do statement.

THE switch STATEMENT

The switch statement is a multi-way conditional statement. The switch statement general form is given by:

```
switch (expression) {

   statement

}/* end of switch */
```

Statement is typically a compound statement containing case labels, the break and optionally the default label.

Typically, a switch is composed of many cases, and the expression in parentheses following the keyword switch determines which, if any, of the cases get executed. The precise details of how a switch works are explained by looking at a specific example. The following program counts the occurrences of the letters a, b, c, and C, with the letters c and C counted together. A switch statement is used to accomplish this task.

```
/*Program to count a, b, c and C */
# include <stdio.h>
main ( )
{
    int chr, a_cnt = 0, b_cnt = 0, cC_cnt =0, other_cnt =0;
    while ((chr = getchar(   )) != EOF)
    switch( chr )
     case 'a' :              /* case argument must be a constant */
         ++a_cnt;
      break;                 /* break out of the switch statement   */

      case 'b' :
        ++b_cnt;
       break;

      case 'c':      /* notice how these two case are together */
      case 'C'
        ++cC_cnt;
      break

      default:    /*This is optional but you should include in all switch */
         ++other_cnt;
      break;
    }/* end of switch */


printf("\n%9s%5d\n%9s%5d\n%9s%5d\n%9s%5d\n%9s%5d\n\n",
        "The a count:", a_cnt, "The b count:", b_cnt,
        "The c count: " , cC_cnt, "All othesr: " , other_cnt,
        "The total count: ", a_cnt + b_cnt + cC_cnt + other_cnt);
} /* end of main */
```

PROGRAM ANALYSIS

The value of the next character in the input stream is assigned to chr.

```
while ((chr = getchar(   )) ! = EOF)
```

The loop is exited when the end-of-file sentinel EOF is detected.

This construct is a switch statement.

```
            switch (chr) {
```

The integral expression in the parentheses following the keyword switch is evaluated with the usual arithmetic conversions taking place. In this program the expression is just the variable chr. The value of chr is used to transfer control to one of the case labels or to the default label.

```
            case 'a' :          /* case argument must be a constant */
                 ++a_cnt;
              break;             /* break out of the switch statement   */
```

When the value of chr is 'a' .this case is executed, causing the value of a_cnt to be incremented by one. The break statement causes control to exit the switch statement. The while statement is then continued.

```
              case 'c':    /* notice how these two case are together */
              case 'C'
                ++cC_cnt;
              break
```

Multiple case labels allow the same actions to be taken for different values of the switch expression. Here the same action is to be taken when c has the value c or C.

If the value of the switch expression does not match one of the case labels, then control is passed to the default label, if there is one.

```
            default:   /*This is optional but you should include in all switch */
                 ++other_cnt;
              break;
```

If there is no default label, then the switch statement is exited. In this example the default label is used to cause the variable other_cnt to be incremented.

The case label is of the form:

```
            case constant integral expression :
```

In a switch statement the case labels must all be unique. Typically, the action taken after each case label ends with a break statement. If there is no break statement, then execution passes through to the next statement in the succeeding case. If no case label is selected, then control passes to the default label, if there is one. A default label is not required. If no case label is selected, and there is no default label, then the switch statement is exited. To detect errors, you should always include a default even when all the expected cases have been accounted for. The keywords case and default cannot occur outside of a switch statement.

The effect of a switch

1. Evaluate the integral expression in the parentheses following switch.
2. Execute 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, execute the default label, or, if there is no default label, terminate the switch.
3. Terminate the switch when a break statement is encountered, or terminate the switch at the end.

THE CONDITIONAL OPERATOR

The conditional operator ? : is unusual in that it is a ternary operator. It takes three expressions as operands. In a construct such as

```
expl ? exp2 : exp3
```

Here expl is evaluated first. If it is nonzero (true), then exp2 is evaluated and that is the value of the conditional expression as a whole. If expl is zero (false),then exp3 is evaluated and that is the value of the expression. The conditional expression can be used to do the work of an if-else statement. An example is next shown.

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

These statements assign to x the minimum of y or z. This also can be accomplished in one statement by writing:

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

The parentheses are not necessary because the precedence of the conditional operator is just above = operator. It is good programming practice to use parentheses. Parentheses are used to make clear the expression is being tested. The type of the conditional expression expl ? exp2 : exp3 is determined by exp2 and exp3. The usual conversion rules are applied. Note carefully that the type of the conditional expression does not depend on which of the two expressions exp2 or exp3 is evaluated. The conditional operator ? : has precedence just above the assignment operators, and it associates "right to left".

THE goto STATEMENT

The goto statement is the most primitive method of interrupting ordinary control flow. It is an unconditional branch to an arbitrary labeled statement in the function. The goto statement is considered a harmful construct in programming. It can undermine all the useful structure provided by other flow of control mechanisms (for, while, do, if, switch). Here is a program that uses a goto statement to create an infinite loop that prints integers. Keep in mind that such a program must be terminated by a control-c or a delete key on most systems.

```
main(   )
{
   int i = 0;
    loop:   printf("%d\n", i++);

    /*   some processing   */

     goto loop;
}/* end main*/
```

In this example loop is a label called loop.

```
loop: printf("%d\n", i++);
```

This is called a labeled statement. Whenever the goto statement is reached, it passes control to the labeled statement. The program has no means of exiting this loop.

A label is a unique identifier. Some examples of labeled statements:

```
spot_1:    a = b + c;
error_one:    printf ("Error one reached.\n");
bug_1: bug_2: bug_3:    printf("Bug 1, 2, 3 \n");   /* multiple labels */
```

These examples are not labeled statements.

```
333: a =b + c;   /* 333 is not an identifier */
a:  a +=  b * c;    /* a is not a unique identifier */
```

By executing a goto statement of the form goto label; control is unconditionally transferred to a labeled statement. An example would be:

```
if (d  ==  0.0)
    goto error;
else
     ratio = n / d;
error: printf("ERROR: division by zero!  \n");
```

Both the goto statement and its corresponding labeled statement must be in the body of the same function.

In general, the goto should be avoided. The goto is a primitive method of altering flow of control and, in a richly structured language, is unnecessary. Labeled statements and goto's are the hallmark of incremental patchwork program design. A programmer who modifies a program by adding goto's to additional code fragments soon makes the program incomprehensible.

One conceptual use of the goto is to give a technical explanation of the effect of a continue statement in a for loop.

SCOPE RULES

A block statement is a series of declarations followed by a series of statements all surrounded by the braces { and }. Its basic use is to group statements into an executable unit. When declarations are not present, a block statement is also called a compound statement.

The following is an example of a block statements.

```
{
   int a = 2;
    printf("%d\n", a);   /*The int 2 is printed */
    {
      int b = 3:
      printf("%d\n", b);   /*The int 3 is printed */
    }
}
```

The basic reason for blocks is to allow memory for variables to be created in the function when needed. This is actually not good programming practice because all variables are not declared at the beginning of the function. However, if memory is a scarce resource, then block exit will release the storage allocated locally to the block, allowing the memory to be used for some other purpose.

Also, blocks associate names in their neighborhood of use, making the code more readable. Functions can be viewed as named blocks with parameters and return statements allowed.

The basic rule of scoping is that identifiers are accessible only within the block in which they are declared. They are unknown outside the boundaries of that block. This would be an easy rule to follow, except that programmers for a variety of reasons choose to use the same identifier in different declarations. They then have the question of which object the identifier refers to.

Let us give a simple example of this state of affairs.

```
{                       /*outer block a */
  int a = 2;
  printf("%d\n", a);   /*The int 2 is printed */
   {                    /*inner block a */
     int a == 3;
     printf("%d\n", a):   /*The int 3 is printed */
   }   /*   back to the outer block */
  printf("%d\n", a);   /*The int 2 is printed */
}
```

Each block introduces its own nomenclature. An outer block name is valid unless an inner block redefines it. If redefined, the outer block name is hidden, or masked, from the inner block. Inner blocks may be nested to arbitrary depths which are determined by system limitations.

STORAGE CLASS

Every variable and function in C has two attributes: Type and storage class. The four storage classes are automatic, external, static, and register, with the corresponding keywords

```
auto   extern   static   register
```

THE STORAGE CLASS auto

Variables declared inside of function bodies are automatic by default. Thus automatic is the most common of the four storage classes. Although it is usually not done, the storage class of automatic variables can be made explicit by use of the keyword auto.

The code

```
{
  char c;
   int i, j, k;
}
```

is equivalent to

```
{
  auto char c;
  auto int i, j, k;
}
```

When a block is entered, the system sets aside adequate memory for the automatically declared variables. Within that block those variables are defined, and they are considered to be "local" to the block. When the block is exited, the system no longer reserves the memory set aside for the automatic variables. Thus, the values of these variables are no longer of usable. If the block is reentered, the storage once again is appropriately allocated, but previous values are not kept or known for use. Each invocation of a function, (i.e. call to a function) sets up a new environment (reinitializes the local variables).

The STORAGE CLASS extern

All functions and all variables declared outside of function bodies have external storage class. One method of transmitting information across blocks and functions is to use external variables. When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is external. A declaration for an external variable looks just the same as a variable declaration inside a function or block. Such a variable is considered to be global to all function declared after it. When the function exits, the external variable remains in existence.

```
int a = 7;
main (   )
{
  printf("%d\n", a);   /*The int 7 is printed */
  f(   );
}/* end main */

f(   )   /* function f */
{
 printf("%d\n", a);   /*The int 7 is printed from f(   )*/
}  /* end function f  */
```

In the above program it would be wrong to code

```
extern int a = 7;
```

The keyword extern is used to tell the system to look for a variable externally, perhaps even in another file that is used to make up the program. Here is an example of this in a file called file1.c.

```
file1.c  /* this is a separate file called file1 */

int v =33; /*v an external variable is globally defined in the file1*/

main(   )
{
   double x = 1.11;
   printf("%d\n", v);
   f(x);    /* call function f

file2.c      /*This is a separate file called file2.c */
f(double x)
{
  extern int v;
  /*******************************************************************/
/*The system will look for v externally,  either in this file or       */
/* in another file that will be linked into the final executable     */
/*program.                                                             */

    printf("v from main = %d\n x passed from main = %d\n", v, x);
}/* end f */
```

The use of extern to avoid undefined variable names makes possible the separate compilation of functions in different files. The functions main(   ) and f(   ) written in file1.c and fiie2.c, respectively, can be compiled separately. The extern declaration of the variable v in fiie2.c tells the system that it will be declared externally, either in this file or some other file and resolved at link time. The executable program obtained by compiling these two functions separately will act no differently than a program obtained by compiling a single file containing both functions and the external variable defined at the beginning of the file.

External variables never disappear i.e. they stay resident in memory. Since they exist throughout the execution life of the program, they can be used to transmit values across functions. Of course, a variable may be hidden if it is redefined in an inner block. Thus information can be passed into a function in two ways: by use of external variables and by use of the parameter passing mechanism. Although there are exceptions, the use of the parameter mechanism is the preferred method to pass information into a function. This tends to improve the modularity of the code, and the possibility of undesirable side effects is reduced.

One form of "side effect" occurs when a function changes a global variable within its body rather than through its parameter list. Such a construction is error prone. Correct practice is to effect changes to global variables through the parameter and return mechanisms. However, to do this requires the use of pointers and the address operator, material which will be covered later. Adhering to this practice improves the modularity and readability of programs.

THE STORAGE CLASS register

The storage class register, used in a declaration, indicates that, if physically and semantically possible, the associated variables will be stored in high-speed memory registers. Since resources limitations and semantic constraints sometimes makes this impossible, this storage class defaults to automatic whenever the compiler cannot allocate an appropriate physical register.

Basically, the use of storage class register is an attempt to improve execution speed. When speed is of concern, the programmer may choose a few variables that are most frequently accessed and declare them to be of storage class register. Common candidates for such treatment include loop variables and functio parameters. An example is:

```
{
  register int i;
    for (i = 0; i <= LIMIT; ++i)  {



    }/* end for loop  */
} /* end block exit will free register */
```

Note that the register variable i was declared as close to its place of use as possible. This is to allow maximum availability of the physical registers, using them only when needed. Always remember that a register declaration is taken only as advice to a compiler.

THE STORAGE CLASS static

Static declarations have an important and distinct use. This is to allow a local variable to retain its previous value upon reentry into a block. This is in contrast to ordinary automatic variables which lose their value upon block exit. As an example of this use, you can use a static variable to find out how many times a function is called during execution of a program.

```
f(   )
{
        char a, b, c;
         static int cnt = 0;
         printf("cnt = %d\n"  ++cnt):
    }
```

The variable cnt in f(   ) is initialized to 0 only once since the key word static is used. Whenever the function is called, the old value is retained, then it is indexed and printed out.