Flow of control: switch statement, conditional operator

The `switch` statement:

The `switch` statement is a multi-way conditional statement that generalizes the `if-else if-else` ladder. A typical form of `switch` statement is

```
switch (expr) {
    case const_val1:
        // code to be executed if expr == const_val1;
         break; // optional
    case const_val2:
        // code to be executed if expr == const_val2;
        break; // optional
    // ... any number of case statements
    default: // optional
        // code to be executed if no case matches;
}       // end of switch statement
```

The `expr` *is evaluated once and must result in an integer value.* Each `case` is followed by a unique, constant integral value and a colon. If the switch `expr` matches a case value, execution begins at the statements following that `case`. The `break` statement is used to terminate the switch block and exit the control flow. Without `break`, execution will fall through to the statements below. The `default` is an optional case that is executed if none of the case values match the switch `expr`. It is similar to the `else` block in an `if-else if-else` ladder and it is usually placed at the end. A switch block is also terminated by "falling of the end".

Sometimes, a `if-else if-else` ladder can be converted to `switch` construct easily. This is specially true when the ladder consists entirely of equality checks against a single variable using constant values. For example, consider the program given in Listing 19 in Lecture 10. This we write using `switch` construct as follows.

```
#include<stdio.h>
int main()
{
char op;
int a,b,res;
printf("Enter arithmetic operator:");
scanf("%c",&op);
printf("Enter intgers a and b:");
scanf("%d%d",&a,&b);
switch(op){
```

```
 case '+': res=a+b;
    break;
 case '-': res=a-b;
    break;
 case '*': res=a*b;
    break;
 case '/': if(b==0)
    {
     printf("Divisor must be nonzero\n");
     return 0;
    }
    res=a/b;
    break;
 case '%': if(b==0)
    {
     printf("b must be nonzero in a%%b\n");
     return 0;
    }
    res=a%b;
    break;
 default: printf("Unknown operator\n");
    return 0;
}
printf("%d %c %d = %d\n",a,op,b,res);
return 0;
}
```

Listing 22: C program "carithms.c"

The characters '+', '-', '*', etc. have unique integer values, e.g., 43 for '+', 45 for '-' etc. Thus, `case '+'` is equivalent to `case 43`. Now, input character `op` has a specific integer value. Hence, it either matches with one of the `case` statement or not. In case it matches with a case statement, it executes the corresponding arithmetic and then exits the `switch` block upon encountering the `break` statement. Otherwise, execution transfers to `default` case and the program stops upon encountering the `return 0` expression.

Next we show another program where `break` expression is not necessary in all `case` statements. Here, we read a lowercase alphabet and determines whether it is a vowel or a consonant with appropriate message.

```
#include<stdio.h>
int main()
{
char alph;
printf("Enter a lowercase alphabet:");
scanf("%c",&alph);
if(alph<'a' || alph>'z')
{
 printf("Input is not a lowercase alphabet\n");
 return 0;
}

switch(alph){
 case 'a':
 case 'e':
 case 'i':
 case 'o':
 case 'u': printf("Input alphabet %c is a vowel\n",alph);
    break;

 default:  printf("Input alphabet %c is a consonant\n",alph);
}
return 0;
}
```

Listing 23: C program "vowcon.c"

The following are some expected input and output.

```
Enter a lowercase alphabet:9
Input is not a lowercase alphabet
Enter a lowercase alphabet:A
Input is not a lowercase alphabet
Enter a lowercase alphabet:i
Input alphabet i is a vowel
Enter a lowercase alphabet:s
Input alphabet s is a consonant
```

Note that the first four `case` statements are blank and do not have `break` statement. Hence, for input `i`, the expression `switch(alph)` transfers execution to `case 'i':`. In the absence of `break` statement in `case 'i'` and `case 'o'`, it transits to the statements corresponding to

`case 'u':` and prints `Input alphabet i is a vowel`. Next, it encounters the `break` statement and exits the `switch` block. Since `default` case is at the end, the execution of `switch` block ends here and `break` statement is not needed here.

The conditional operator:

The conditional operator `?:` is unusual in the sense that it is a ternary operator. It takes as operands three expressions. A construct for the conditional operator is of the form
`res=expr1?expr2:expr3;`
Here `expr1` is evaluated first. If it is true (nonzero), then `expr2` is evaluated, and that is assigned to variable `res`. If `expr1` is false (zero), then `expr3` is evaluated, and that is assigned to the variable `res`. This is similar to `if-else` construct. To see this, consider the following code segment:

```
double x=3,y=4,max;
if(x>y)
{
  max=x;
}
else
{
  max=y;
}
```

The above code segment can be replaced with the following:

```
double x=3,y=4,max;
max=x>y?x:y;
```

The conditional operator `?:` has precedence just above the assignment operators, and it associates from right to left. We extend the operator precedence and associativity table discussed in Lecture 10 (see last page). We can use the conditional operator in `#define` directive to find the maximum or minimum of variables. For example, consider the following code:

```
#include<stdio.h>
#define MAX(X,Y) (X)>(Y)?(X):(Y)
#define MIN(X,Y) (X)<(Y)?(X):(Y)
int main()
{
 double x,y,z;
 printf("Enter x,y,z:");
 scanf("%lf%lf%lf",&x,&y,&z);
```

```
 printf("Max of %.2f and %.2f is %.2f\n",x,y,MAX(x,y));
 printf("Min of %.2f and %.2f is %.2f\n",x,y,MIN(x,y));
 printf("Max of %.2f, %.2f and %.2f is %.2f\n",x,y,z,MAX(MAX(x,y),z));
 printf("Min of %.2f, %.2f and %.2f is %.2f\n",x,y,z,MIN(x,MIN(y,z)));
return 0;
}
```

Listing 24: C program "cmxmn.c"

The following is expected input and output.

```
Enter x,y,z:-2 1 5
Max of -2.00 and 1.00 is 1.00
Min of -2.00 and 1.00 is -2.00
Max of -2.00, 1.00 and 5.00 is 5.00
Min of -2.00, 1.00 and 5.00 is -2.00
```

Note that if `expr2` and `expr3` are of different types in
`res=expr1?expr2:expr3;`
then the usual conversion rules are applied. For example, consider the following code:

```
int i=3,j=4;
double a=8.0;
printf("%.2f\n",j%5==0?2*i:a+2); // 10.00 is printed
printf("%.2f\n",j%5?2*a:i+2);    // 16.00 is printed
```

Since the conditional operator `?:` has priority just above the assignent operator, the expression `j%5==0?2*i:a+2` is equivalent to `((j%5)==0)?(2*i):(a+2)`. Since `expr3=a+2` is double, `2*i` is also converted to double. Since `j%5==0` is false, the resulting value is `10.00`. Similar conversion happens in the second expression, which is equivalent to `(j%5)?(2*a):(i+2)`. Since `j%5` is nonzero, i.e., true, the resulting value is `16.00`.

Category	Operator	Associativity
Postfix operators	`( )` (parenthesis), `++` (postfix increment), `--` (postfix decrement),	left to right
Unary operators	`+` (unary plus), `-` (unary minus), `++` (prefix increment), `--` (prefix decrement), sizeof, (type) (type casting), `!` (logical not)	right to left
Multiplicative operators	`*` (multiplication), `/` (division), `%` (modulus)	left to right
Additive operators	`+` (addition), `-` (subtraction)	left to right
Relational operators	`<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)	left to right
Equality operators	`==` (equal to), `!=` (not equal to)	left to right
Logical and	`&&`	left to right
Logical or	`||`	left to right
Conditional (ternary) operator	`?:`	right to left
Assignment operators	`=`, `+=`, `-=`, `*=`, `/=`, `%=`	right to left

Return to Lecture Topics