The C "switch" Statement

CMSC 104, Fall 2012
John Y. Park

The switch Statement

Topics

- Multiple Selection
- switch Statement
- char Data Type and getchar( )

Reading

- Section 4.7, 4.12

Multiple Selection

- So far, we have only seen **binary selection**.

```
if ( age >= 18 )
{
    printf("Vote!\n") ;
}
```

```
if ( age >= 18 )
{
    printf("Vote!\n") ;
}
else
{
    printf("Maybe next time!\n") ;
}
```

Multiple Selection (con't)

- Sometimes it is necessary to **branch** in more than two directions.
- We do this via **multiple selection**.
- The multiple selection mechanism in C is the **switch** statement.

Multiple Selection with if

```
if (day == 0 ) {
   printf ("Sunday") ;
}
if (day == 1 ) {
   printf ("Monday") ;
}
if (day == 2) {
   printf ("Tuesday") ;
}
if (day == 3) {
   printf ("Wednesday") ;
}
```

(continued)

```
if (day == 4) {
   printf ("Thursday") ;
}
if (day == 5) {
   printf ("Friday") ;
}
if (day == 6) {
   printf ("Saturday") ;
}
if ((day < 0) || (day > 6)) {
   printf("Error - invalid day.\n") ;
}
```


Multiple Selection with if-else

```
if (day == 0 ) {
   printf ("Sunday") ;
} else if (day == 1 ) {
   printf ("Monday") ;
} else if (day == 2) {
   printf ("Tuesday") ;
} else if (day == 3) {
   printf ("Wednesday") ;
} else if (day == 4) {
   printf ("Thursday") ;
} else if (day == 5) {
   printf ("Friday") ;
} else if (day == 6) {
   printf ("Saturday") ;
} else {
   printf ("Error - invalid day.\n") ;
}
```

This if-else structure is more efficient than the corresponding if structure. Why?

Are there any other functional differences?

Multiple Selection with if-else

```
if (day == 0 ) {
   printf (“Sunday”) ;
   day = 3;
}
if (day == 1 ) {
   printf (“Monday”) ;
}
if (day == 2) {
   printf (“Tuesday”) ;
}
if (day == 3) {
   printf (“Wednesday”) ;
}
if (day == 4) {
   printf (“Thursday”) ;
}
…
```

vs.

```
if (day == 0 ) {
   printf (“Sunday”) ;
   day = 3;
} else if (day == 1 ) {
   printf (“Monday”) ;
} else if (day == 2) {
   printf (“Tuesday”) ;
} else if (day == 3) {
   printf (“Wednesday”) ;
} else if (day == 4) {
   printf (“Thursday”) ;
} else if (…)
…
```


The switch Multiple-Selection Structure

```
switch ( integer expression )
{
   case constant1 :
        statement(s)
        break ;
   case constant2 :
        statement(s)
        break ;
        . . .
   default: :
        statement(s)
        break ;
}
```


switch Statement Details

- The last statement of each case in the switch should almost always be a break.
- The break causes program control to jump to the closing brace of the switch structure.
- Without the break, the code flows into the next case. This is almost never what you want.
- A switch statement will compile without a default case, but always consider using one.

Good Programming Practices

- Include a default case to catch invalid data.
- Inform the user of the type of error that has occurred (e.g., “Error - invalid day.”).
- If appropriate, display the invalid value.
- If appropriate, terminate program execution (discussed in CMSC 201).

switch Example

```
switch ( day )
{
      case 0:  printf (“Sunday\n”) ;
               break ;
      case 1:  printf (“Monday\n”) ;
               break ;
      case 2:  printf (“Tuesday\n”) ;
               break ;
      case 3:  printf (“Wednesday\n”) ;
               break ;
      case 4:  printf (“Thursday\n”) ;
               break ;
      case 5:  printf (“Friday\n”) ;
               break ;
      case 6:  printf (“Saturday\n”) ;
               break ;
      default:  printf (“Error -- invalid day.\n”) ;
               break ;
}
```

Is this structure more efficient than the equivalent nested if-else structure?

switch Example

```
switch ( day )
{
      case 1:  printf (“Monday\n”) ;
               break ;
      case 2:  printf (“Tuesday\n”) ;
               break ;
      case 3:  printf (“Wednesday\n”) ;
               break ;
      case 4:  printf (“Thursday\n”) ;
               break ;
      case 5:  printf (“Friday\n”) ;
               break ;
      case 0:
      case 6:  printf (“Weekend\n”) ;
               break ;
      default:  printf (“Error -- invalid day.\n”) ;
               break ;
}
```

Why Use a switch Statement?

- A switch statement can be more efficient than an if-else.
- A switch statement may also be easier to read.
- Also, it is easier to add new cases to a switch statement than to a nested if-else structure.

The char Data Type

- The **char** data type holds a single character.

  char ch;
- Example assignments:

  char grade, symbol;

  grade = 'B';

  symbol = '$';
- The char is held as a one-byte integer in memory. The ASCII code is what is actually stored, so we can use them as characters or integers, depending on our need.

The char Data Type (con't)

- Use

  scanf ("%c", &ch) ;

  to read a single character into the variable ch. (Note that the variable does not have to be called "ch".")
- Use

  printf("%c", ch) ;

  to display the value of a character variable.

char Example

```
#include <stdio.h>
int main ( )
{
    char ch ;

    printf (“Enter a character: “) ;
    scanf (“%c”, &ch) ;
    printf (“The value of %c is %d.\n”, ch, ch) ;
    return 0 ;
}
```

If the user entered an A, the output would be:

The value of A is 65.

The getchar ( ) Function

- The getchar( ) function is found in the **stdio** library.
- The getchar( ) function reads one character from **stdin** (the **standard input buffer**) and returns that character’s ASCII value.
- The value can be stored in either a character variable or an integer variable.

getchar ( ) Example

```
#include <stdio.h>
int main ( )
{
    char ch ;    /* int ch  would also work! */

    printf (“Enter a character: “) ;
    ch = getchar( ) ;        /*same as scanf(“%c”, &ch); */
    printf (“The value of %c is %d.\n”, ch, ch) ;
    return 0 ;
}
```

If the user entered an A, the output would be:

The value of A is 65.

Problems with Reading Characters

- When getting characters, whether using scanf( ) or getchar( ), realize that you are reading only one character.
- What will the user actually type? The character he/she wants to enter, followed by pressing ENTER.
- So, the user is actually entering two characters, his/her response and the **newline character**.
- Unless you handle this, the newline character will remain in the stdin stream causing problems the next time you want to read a character. Another call to scanf() or getchar( ) will remove it.

Improved Character Example

```
#include <stdio.h>
int main ( )
{
    char ch, newline ;

    printf (“Enter a character: “) ;
    scanf(“%c”, &ch) ;
    scanf(“%c”, &newline);
    printf (“The value of %c is %d.\n”, ch, ch) ;
    printf (“Enter another character: “) ;
    scanf(“%c”, &ch) ;
    scanf(“%c”, &newline);
    printf (“The value of %c is %d.\n”, ch, ch) ;
    return 0 ;
}
```


Additional Concerns with Garbage in stdin

- When we were reading integers using scanf( ), we didn’t seem to have problems with the newline character, even though the user was typing ENTER after the integer.
- That is because scanf( ) was looking for the next integer and ignored the newline (**whitespace**).
- If we use scanf (“%d”, &num); to get an integer, the newline is still stuck in the input stream.
- If the next item we want to get is a character, whether we use scanf( ) or getchar( ), we will get the newline.
- We have to take this into account and remove it.