Assignment Operators

CMSC 104, Spring 2014

Christopher S. Marron

(thanks to John Park for slides)

Assignment Operators

Topics

- Increment and Decrement Operators
- Assignment Operators
- Debugging Tips
- The char type and getchar() function

Increment and Decrement Operators

- The **increment operator** ++
- The **decrement operator** --
- Precedence: lower than (), but higher than * / and %
- Associativity: right to left
- Increment and decrement operators can only be applied to variables, not to constants or expressions

Increment Operator

- If we want to add one to a variable, we can say:

  count = count + 1 ;

- Programs often contain statements that increment variables, so to save on typing, C provides these shortcuts:

  count++ ;   OR   ++count ;

  Both do the same thing. They change the value of count by adding one to it.

Postincrement Operator

- The position of the ++ determines when the value is incremented. If the ++ is after the variable, then the incrementing is done last (a **postincrement**).

```
int amount, count ;
count = 3 ;
amount = 2 * count++ ;
```

- amount gets the value of 2 * 3, which is 6, and then 1 gets added to count.
- So, after executing the last line, amount is 6 and count is 4.

Preincrement Operator

- If the ++ is before the variable, then the incrementing is done first (a **preincrement**).

```
int amount, count ;
count = 3 ;
amount = 2 * ++count ;
```

- 1 gets added to count first, then amount gets the value of 2 * 4, which is 8.
- So, after executing the last line, amount is 8 and count is 4.

Code Example Using ++

```
#include <stdio.h>
int main ( )
{
    int i = 1 ;
     /* count from 1 to 10 */
    while ( i < 11 )
    {
        printf (“%d  ”, i) ;
        i++ ;                        /* same as ++i */
    }
    return 0 ;
}
```

Decrement Operator

- If we want to subtract one from a variable, we can say:

```
count = count - 1 ;
```

- Programs often contain statements that decrement variables, so to save on typing, C provides these shortcuts:

```
count-- ;     OR      --count ;
```

Both do the same thing. They change the value of count by subtracting one from it.

Postdecrement Operator

- The position of the -- determines when the value is decremented. If the -- is after the variable, then the decrementing is done last (a **postdecrement**).

```
int amount, count ;
count = 3 ;
amount = 2 * count-- ;
```

- amount gets the value of 2 * 3, which is 6, and then 1 gets subtracted from count.
- So, after executing the last line, amount is 6 and count is 2.

Predecrement Operator

- If the -- is before the variable, then the decrementing is done first (a **predecrement**).

```
int amount, count ;
count = 3 ;
amount = 2 * --count ;
```

- 1 gets subtracted from count first, then amount gets the value of 2 * 2, which is 4.
- So, after executing the last line, amount is 4 and count is 2.

A Hand Trace **Example**

int answer, value = 4 ;

Code	Value	Answer
	4	garbage
value = value + 1 ;		
value++ ;		
++value ;		
answer = 2 * value++ ;		
answer = ++value / 2 ;		
value-- ;		
--value ;		
answer = --value * 2 ;		
answer = value-- / 3 ;		

Practice

Given

```
    int a = 1, b = 2, c = 3 ;
```

What is the value of this expression?

```
            ++a * b - c--
```

What are the new values of a, b, and c?

More Practice

Given

```
int a = 1, b = 2, c = 3, d = 4 ;
```

What is the value of this expression?

```
++b / c + a * d++
```

What are the new values of a, b, c, and d?

Assignment Operators

=   +=   -=   *=   /=   %=

Statement	Equivalent Statement
a = a + 2 ;	a += 2 ;
a = a - 3 ;	a -= 3 ;
a = a * 2 ;	a *= 2 ;
a = a / 4 ;	a /= 4 ;
a = a % 2 ;	a %= 2 ;
b = b + (c + 2) ;	b += c + 2 ;
d = d * (e - 5) ;	d *= e - 5 ;

Practice with Assignment Operators

int i = 1, j = 2, k = 3, m = 4 ;

Expression	Value
i += j + k	
j *= k = m + 5	
k -= m /= j * 2	

Code Example Using /= and ++
Counting the Digits in an Integer

```
#include <stdio.h>
int main ( )
{
    int num, temp, digits = 0 ;
    temp = num = 4327 ;
   while ( temp > 0  )
    {
       printf (“%d\n”, temp) ;
       temp /= 10 ;
       digits++ ;
    }
    printf (“There are %d digits in %d.\n”, digits, num) ;
    return 0 ;
}
```

Debugging Tips

- Trace your code by hand (a **hand trace**), keeping track of the value of each variable.
- Insert temporary printf() statements so you can see what your program is doing.
  - Confirm that the correct value(s) has been read in.
  - Check the results of arithmetic computations immediately after they are performed.

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.