Chapter 3 Notes – **Structured Program**

C How to Program, 6th Edition

Deitel & Deitel

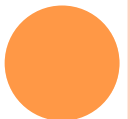
SWE110 Lecture Note
Instructor: Thien An Nguyen (An)

(Slides credit: Thien An Nguyen (An))

Objectives

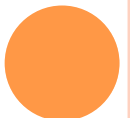
- Algorithm
- Pseudo code
- Control statements: if … else, nested if, while loop
- Examples with loops
- Require reading: chapter 3

Organize your thoughts:
- Graphic
- Pseudo-code
- Flowcharts



3.2 - Algorithm

- An algorithm is a series of `actions` in a `specific order` for solving a problem.
- For an algorithm to work well, it must be logically put together:
  - Get up, Take a shower, Get dressed, Eat breakfast, Go to work
- An alternated version as below could work well too:
  - Get up, Take a shower, Get dressed, Go to work, Eat breakfast
- However, another alternate is not as good:
  - Get up, Get dressed, Eat breakfast, Take a shower, Go to work
- Thus, the `order of the actions` is very important



3.3 – Pseudo-code

- Pseudo-code is a developed algorithm with greater details
- It is not a programming code
- There is no syntax to follow
- When writing pseudo-code, programmer can freely mix up their native speaking language with programming expressions
- Example:

```
int ivar1,                    // a variable to hold the 1st number
    ivar2;                    // a variable to hold the 2nd number

Prompt user for 2 int numbers
Read the numbers into &ivar1, &ivar2

If the 1st number is == the 2nd number
   print "The numbers are equal."
. . .
```

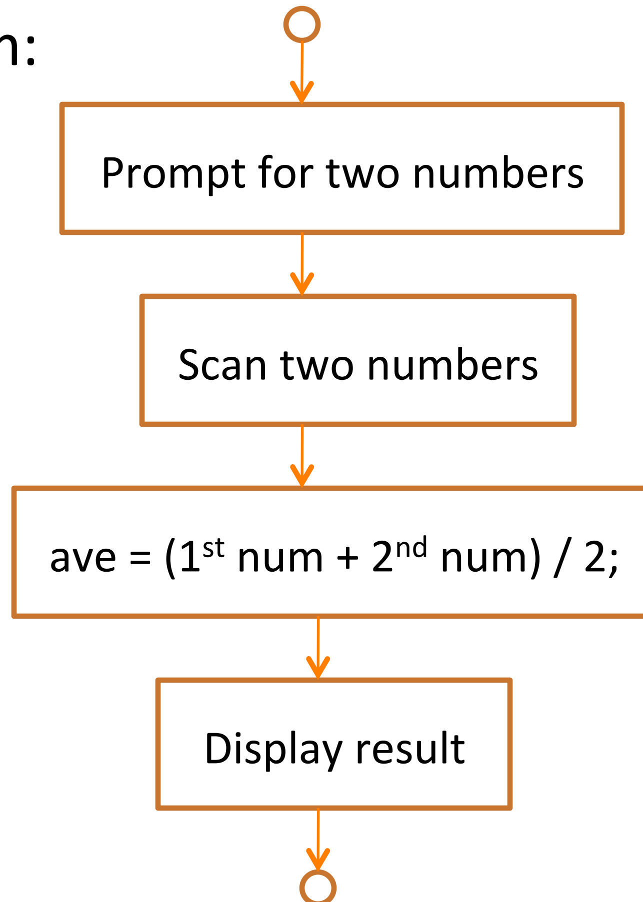
3.3 – Pseudo-code cont.

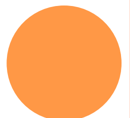
- Below is the transformation from pseudo-code to a C program:

```
//*********************************************************************
//  IfTest.c
//
//  Perform if checks on user input.
//*********************************************************************
#include <stdio.h>

//------------------------------------------------------------------
//  prompt user two integers and print the comparison result
//------------------------------------------------------------------
int main (void)
{
   int ivar1, ivar2;              // declarations

   printf ("Enter two integers: ");// prompt user to enter 2 integers
   scanf ("%d%d", &ivar1, &ivar2);// read the integers

   if (ivar1 == ivar2)            // equality expression
     printf ("The numbers are equal.\n");
. . .
}
```

3.4 – Flow Charts

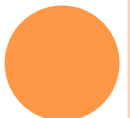
- A flow chart is a visual representation of an algorithm or part of it
- Below is a flow chart for the average of two `ints` algorithm:





3.4 - Control Structures

- Statements in a program are executed in the order they are written.
- To control the flow of program execution, `control flow statements` can be used to skip, branch, or loop around
  - There are three types of selection statements:
    - The `if`
    - The `if … else`
    - The `switch`(chapter 4)
  - The loop statements:
    - The `while`
    - The `do … while` (chap. 4)
    - The `for`(chap. 4)

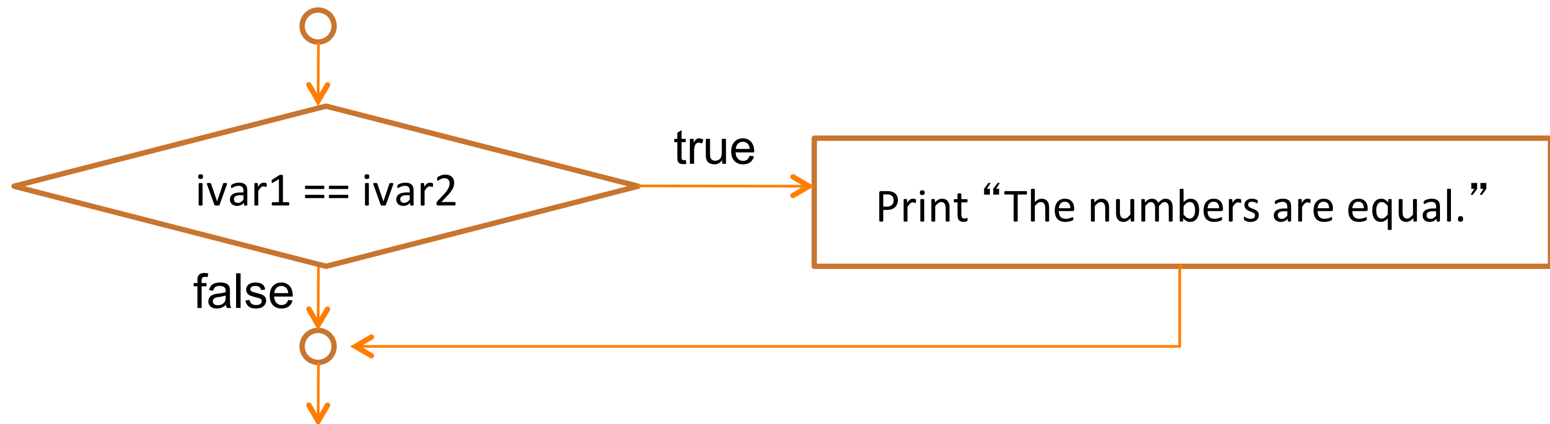


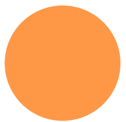
3.5 – The `if` Statement

- In chap. 2

```
. . .
if (ivar1 == ivar2)                // equality expression
  printf ("The numbers are equal.\n");
. . .
```

- Here is a flow chart for it:





3.6 - The `if...else` statement

- The following code fragments are similar:

```
if (ivar1 == ivar2)
  printf ("The numbers are equal.\n");

if (ivar1 != ivar2)
  printf ("The numbers are not equal.\n");
```

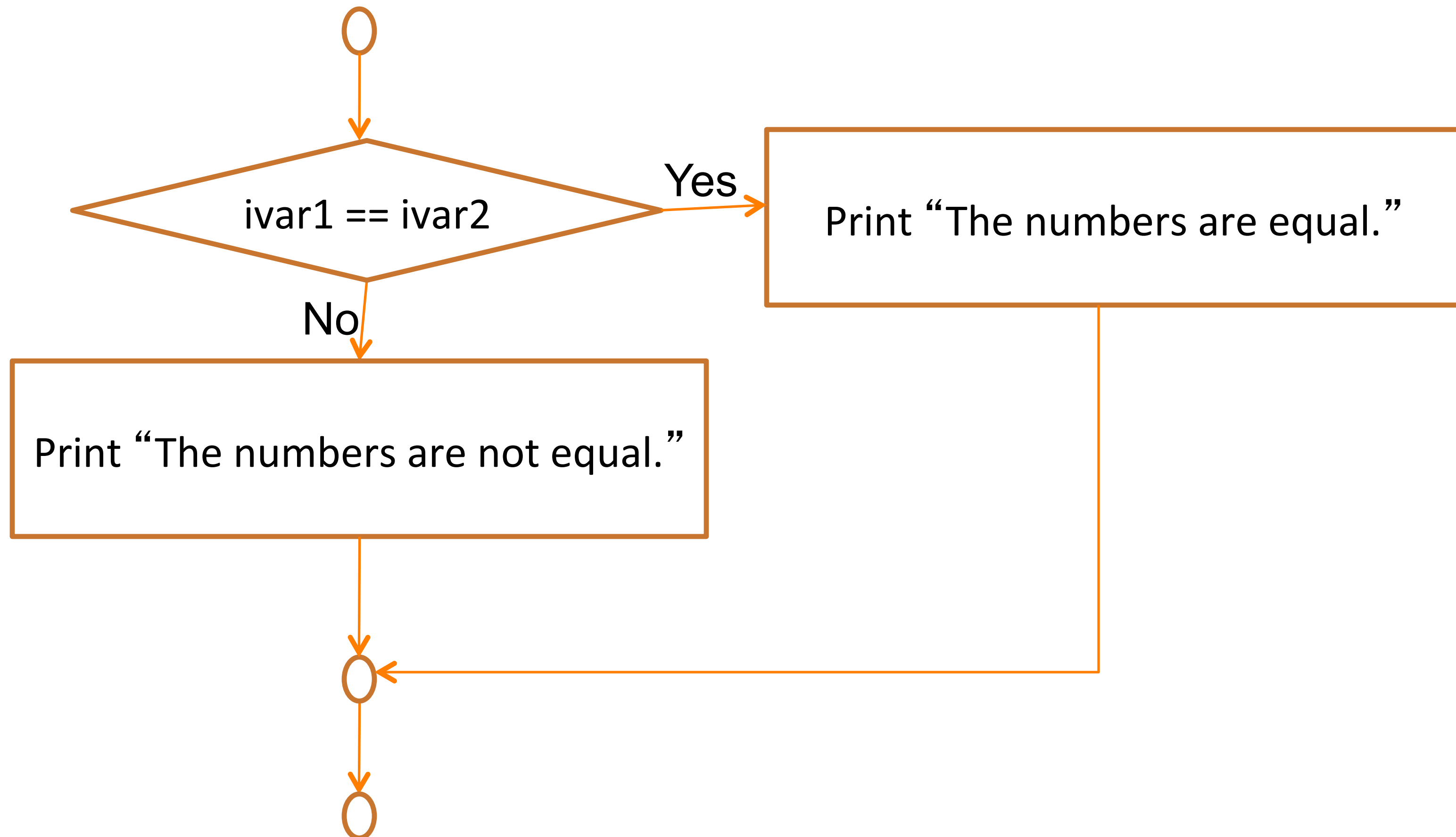
```
if (ivar1 == ivar2)
  printf ("The numbers are equal.\n");
else
  printf ("The numbers are not equal.\n");
```

```
if (ivar1 != ivar2)
  printf ("The numbers are not equal.\n");
else
  printf ("The numbers are equal.\n");
```

Remember: Block of statements

3.6 – The `if`…`else` Statement cont.

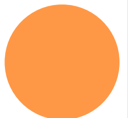
- The flow chart for the previous slide if … else statement:



3.6 – Nested `if…else` Statement cont.

- Nested `if` statements are `if` statements inside if statements
- Nested `if…else` statements are `if…else` statements inside `if…else` statements
- An else clause is matched to the last if clause (no matter what the indentation implies)

```
if (ivar1 < ivar2)          // indentation goes wrong
   if (ivar1 > 0)
      printf ("The 1st number is smaller and positive.\n");
else
   printf ("The 1st number is smaller and not positive.\n");
else
   printf ("The 1st number is greater or equal to the 2nd.\n");
```



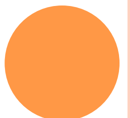
3.6 – Nested `if`...`else` Statement cont.

- An *else* clause is matched to the last *if* clause (no matter what the indentation implies)

```
if (ivar1 < ivar2)          // indentation goes wrong
   if (ivar1 > 0)
      printf ("The 1st number is smaller and positive.\n");
else
   printf ("The 1st number is smaller and not positive.\n");
else
   printf ("The 1st number is greater or equal to the 2nd.\n");
```

- The above code fragment should look like this:

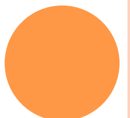
```
if (ivar1 < ivar2)          // indentation fixed
   if (ivar1 > 0)
      printf ("The 1st number is smaller and positive.\n");
   else
      printf ("The 1st number is smaller and not positive.
\n");
else
   printf ("The 1st number is greater or equal to the 2nd.\n");
```



3.6 – Nested `if…else` Statement cont.

- To prevent un-expected result, braces and indentation should be combined:

```
if (ivar1 < ivar2){        // good practice code
   if (ivar1 > 0){
      printf ("The 1st number is smaller and positive.\n");
   } else{
      printf ("The 1st number is smaller and not positive.\n");
   }
} else{
   printf ("The 1st number is greater or equal to the 2nd.\n");
}
```



3.6 – Nested `if`…`else` Statement cont.

- A programmer wrote:

```
if (ivar1 != ivar2)        // bad practice leads to un-expected error
  if (ivar1 < ivar2)
    if (ivar1 > 0)
      printf ("The 1st number is smaller and positive.\n");
    if (ivar1 == 0)
      printf ("The 1st number is smaller and 0.\n");
    else
      printf ("The 1st number is smaller and negative.\n");
```

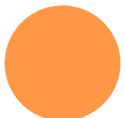
- The compiler interprets as:

```
if (ivar1 != ivar2)        // bad practice leads to un-expected error
  if (ivar1 < ivar2)
    if (ivar1 > 0)
      printf ("The 1st number is smaller and positive.\n");

if (ivar1 == 0)
  printf ("The 1st number is smaller and 0.\n");
else
  printf ("The 1st number is smaller and negative.\n");
```

3.6 – Nested `if…else` Statement cont.

- The programmer really meant:

```
if (ivar1 != ivar2){        // good practice and clear
  if (ivar1 < ivar2){
    if (ivar1 > 0){
      printf ("The 1st number is smaller and positive.\n");
    } else if (ivar1 == 0){
      printf ("The 1st number is smaller and 0.\n");
    } else{
      printf ("The 1st number is smaller and negative.\n");
    }
  }
}
```

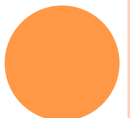


3.6 – Nested `if…else` Statement cont.

- Another **confusing** example:

```
// find the min number
int num1, num2, num3, min;
. . .
if (num1 < num2)
if (num1 < num3)
  min = num1;
else
  min = num3;
else
if (num2 < num3)
  min = num2;
else
  min = num3;
```

- Question: What is wrong with the code? Is the algorithm correct?

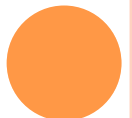


3.6 – Nested `if`…`else` Statement cont.

- First, let' s see what the compiler would interpret.

```
// find the min number
int num1, num2, num3, min;
. . .
if (num1 < num2){
  if (num1 < num3)    // if num1 < num2 and num1 < num3
    min = num1;       // num1 is the min
  else                // otherwise, num3 < num1 < num2
    min = num3;       // num3 is the min
}else
  if (num2 < num3)    // if num1 > num2 and num2 < num3
    min = num2;       // num2 is the min
  else                // otherwise, num1 > num2 > num3
    min = num3;       // num3 is the min
}
```

- Answer: The code is hard to read. However, the algorithm is correct.

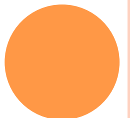


3.7 – The while Loop Statement

```
while (expression)
    statement
```

- Keep execute the statement while the expression is true
- The most important parts are the `loop condition expression` and `the counter`.

```
int MAX_VALUE = 5;

int count = 0;                    // this is the loop counter
while (count < MAX_VALUE){        // the while loop expression
  printf ("%d ", count);
  count = count + 1;              // update the counter
}
```



3.8 – The `while` Loop Statement

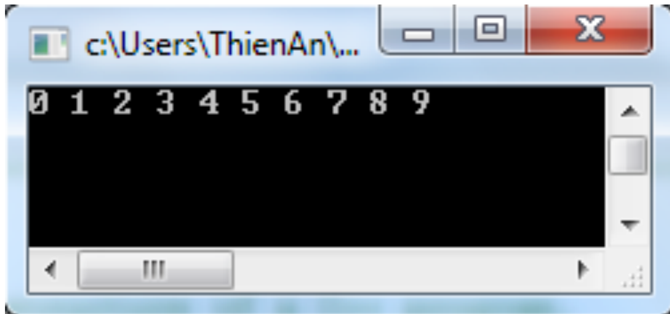
```
int MAX_VALUE = 10;

int count = 0;
while (count < MAX_VALUE){
  printf (“%d ”, count);
  count = count + 1; // update cou
}

// after exited the loop,
// count is 10
```

count	MAX_VALUE	count < MAX_VALUE
0	10	true
1	10	true
2	10	true
3	10	true
4	10	true
5	10	true
6	10	true
7	10	true
8	10	true
9	10	true
10	10	false

3.8 – The while Loop Statement

```
int MAX_VALUE = 10;

int count = 0;
while (count < MAX_VALUE){
  printf (“%d ”, count);
  count = count + 1; // update cou
}

// after exited the loop,
// count is 10
```

count	MAX_VALUE	count < MAX_VALUE
0	10	true
1	10	true
2	10	true
3	10	true
4	10	true
5	10	true
6	10	true
7	10	true
8	10	true
9	10	true
10	10	false



3.8 – The `while` Loop Statement

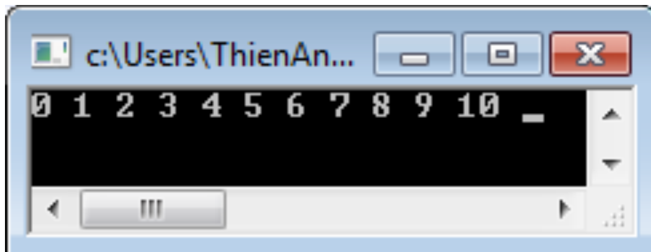
```
int MAX_VALUE = 10;

int count = 0;
while (count <= MAX_VALUE){
  printf (“%d ”, count);
  count = count + 1; // update cou
}

// after exited the loop,
// count is 11
```

count	MAX_VALUE	count < MAX_VALUE
0	10	true
1	10	true
2	10	true
3	10	true
4	10	true
5	10	true
6	10	true
7	10	true
8	10	true
9	10	true
10	10	true
11	10	false

3.8 – The `while` Loop Statement

```
int MAX_VALUE = 10;

int count = 0;
while (count <= MAX_VALUE){
  printf ("%d ", count);
  count = count + 1; // update cou
}

// after exited the loop,
// count is 11
```

count	MAX_VALUE	count < MAX_VALUE
0	10	true
1	10	true
2	10	true
3	10	true
4	10	true
5	10	true
6	10	true
7	10	true
8	10	true
9	10	true
10	10	true
11	10	false



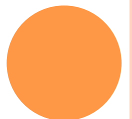
3.8 – The while Loop Statement

- Question: What if the counter was not updated?

```
int MAX_VALUE = 10;

int count = 0;
while (coun < MAX_VALUE){
  printf ("%d ", count);
  count = count + 1; // update c
}
```

count	MAX_VALUE	count < MAX_VALUE
0	10	true
0	10	true
0	10	true
0	10	true
0	10	true
0	10	true
…	10	always true

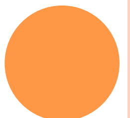
- Answer: Infinite loop… ! Program hang



3.8 – Counter Control Repetition Examples

- In lab1, we use the program Average.c to compute the average of 2 numbers by adding them up and divide the total by 2.
- The pseudo-code algorithm for it is as follows:

```
Declare variables to store 2 integers an the result

Prompt for the 1st integer,
Read it into a variable.

Prompt for the 2nd integer,
Read it into a variable.

Compute the average: ave = (var1 + var2) / 2
Display the result

Exit program
```



The 1st Average.c Program

```
//****************************************************************
//  Average.c
//
//  Compute the average of 2 numbers.
//****************************************************************
#include <stdio.h>

int main (void)
{
   int ivar1, ivar2, ave;

   printf ("Enter integer 1: ");    // prompt for an integer
   printf ("%d", &ivar1);            // read the integer into ivar1

   printf ("Enter integer 2: ");   // prompt for an ingeger
   printf ("%d", &ivar2);           // read it into ivar2

   ave = (ivar1 + ivar2) / 2;      // compute the average
   printf ("The average is: %d", ave); // display the result

   return 0;
}
```

The 1st Average.c Program

```
//********************************************************************
//  Average.c
//
//  Compute the average of
//************************
#include <stdio.h>

int main (void)
{
   int ivar1, ivar2, ave;

   printf (“Enter integer 1
   printf (“%d“, &ivar1);

   printf (“Enter integer 2: ");  // prompt for an ingeger
   printf (“%d“, &ivar2);          // read it into ivar2

   ave = (ivar1 + ivar2) / 2;      // compute the average
   printf (“\nThe average is: %d”, ave); // display the result

   return 0;
}
```

Sample Run:

```
Enter integer 1: 10
Enter integer 2: 6

The average is: 8
```

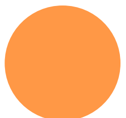
3.8 – Counter Control Repetition cont.

- Now, we' re going to modify Average.c to use the while loop to compute the average of 10 numbers.
- The pseudo-code algorithm is therefore changed to reflect the new design:

```
Declare 1 variable to hold the input from the user
Declare a counter to control the loop
Declare a variable to hold the total, and another to hold the result

While counter is less than or equal to 10
   prompt for an integer.
   Read it into a variable.
   Accumulate it into the total

Compute the average: ave = total / 10
Display the result

Exit program
```

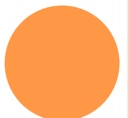
The 2nd Average.c Program

```
//******************************************************************
//   Average.c
//
//   Compute the average of 10 numbers.
//******************************************************************
#include <stdio.h>

int main (void)
{
   int ivar,                          // store the user input
       count,                         // the control counter
       total,                         // the accumulated result
       ave;                           // the average result

. . .
```



The 2nd Average.c Program

```
   // initialize the variables
   count = 1;                        // count starts from 1, not 0
   total = 0;                        // set total to 0

   // loop 10 times
   while (count <= 10){
      printf (“Enter an integer: “);    // prompt for an integer
      printf (“%d“, &ivar);             // read the integer into ivar

      total = total + ivar;             // update total
      count = count + 1;                // update count
   }

   // compute the average
   ave = total/ 10;                  // use 10. don't use count
   printf (“The average is: %d”, ave); // display the result

   return 0;
}
```

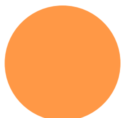


The 2nd Average.c Program

```
    // initialize the variables
    count = 1;                      // count starts from 1, not 0
    total = 0;

    // loop 10 times
    while (count <= 10){
       printf ("Enter an int
       printf ("%d", &ivar);

       total = total + ivar;
       count = count + 1;
    }

    // compute the average
    ave = total/ 10;
    printf ("The average is:

    return 0;
}
```

Sample Run:

```
Enter an integer: 10
Enter an integer: 6
Enter an integer: 10
Enter an integer: 6
Enter an integer: 10
Enter an integer: 6
Enter an integer: 10
Enter an integer: 6
Enter an integer: 10
Enter an integer: 6

The average is: 8
```



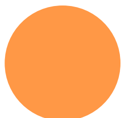
3.8 – Counter Control Repetition cont.

- With the modification, Average.c computes the average of 10 numbers. What if we don't know in advance how many numbers to be averaged?
- The pseudo-code algorithm is therefore changed one more time to reflect the new design:

```
Declare 1 variable to hold the input from the user
Declare a variable to hold the total, and another to hold the result
prompt for an integer.
Read it into a variable.
Accumulate it into the total

While number-entered is not -1
   prompt for an integer.
   Read it into a variable.
   Accumulate into the total

Compute the average: ave = total / 10
Display the result

Exit program
```

The 3rd Average.c Program

```
//**********************************************************************
//   Average.c
//
//   Compute the average of un-known numbers.
//   Assumes the numbers are non-negatives.
//**********************************************************************
#include <stdio.h>

int main (void)
{
   int ivar,                              // store the user input
       count,                             // count the number of inputs
       total,                             // the accumulated result
       ave;                               // the average result

. . .
```



The 3rd Average.c Program

```
    // initialize the variables
    total = 0;                          // reset total to 0
    count = 0;                          // reset count to 0
    printf ("Enter an integer: ");  // prompt for an integer
    printf ("%d", &ivar);             // read the integer into ivar

    // loop until -1 entered
    while (ivar != -1){
        total = total + ivar;         // update total
        count = count + 1;            // update count

        printf ("Enter an integer: ");      // prompt for the next integer
        printf ("%d", &ivar);                 // read the integer into ivar
    }

    // compute the average
    if (count >= 1){
        ave = total / count;                     // compute the average
        printf ("The average is: %d", ave); // display the result
    } else{
        printf("No numbers to compute the average!");
    }

    return 0;
}
```

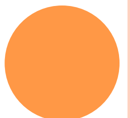
3.11 – Assignment Operators

- There are several assignment operators (see Figure 3.11 text):

```
ivar = 0;

ivar += 7;                // same as ivar = ivar + 7. ivar = 7

ivar -= 3;                // same as ivar = ivar - 3. ivar = 4

ivar *= 3;                // same as ivar = ivar * 3. ivar = 12

ivar /= 3;                // same as ivar = ivar / 3. ivar = 4

ivar %= 3;                // same as ivar = ivar % 3. ivar = 1
```

- Syntax:

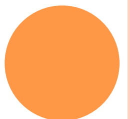
  Variable `operator=` expression;



Quick Check

- What is the result for the following statements?

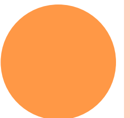
```
int ivar1 = 3;
int  num = 4;
ivar1 += (num / 2);   // ivar1 = ?

ivar1 = 3;
ivar1 += num / 2;     // ivar1 = ?
```



Quick Answers

- What are the result for the following statements?

```
int ivar1 = 3;
int  num = 4;
ivar1 += (num / 2);   ivar1 = 5

ivar1 = 3;
ivar1 += num / 2;     ivar1 = 5. Operator/ has
                             higher precedence
```



3.12 – Increment and Decrement Operators

- They are: ++ and --

```
count++;          // increment by 1 or count = count + 1
--count;          // decrement by 1 or count = count - 1
```

- They can be used as prefix or postfix:

```
++count;          // prefix increment
--count;          // prefix decrement

count++;          // postfix increment
count--;          // postfix decrement
```

- They are often seen as follows:

```
++count;                    //

if (count++)                // uses as in control expression
   printf (“%d”, ++count);// uses in a statement
```

3.12 – Increment and Decrement Operators

- With prefix increment or decrement, the operator is evaluated BEFORE it is used in the expression.

- With postfix increment or decrement, the operator is evaluated AFTER it is used in the expression.

```
count = 0;                          // reset count
++count;                            // count = 1. No difference count++
printf(“count is %d: ”, count++);  // count is: 1
printf (“count is %d: ”, count);   // count is: 2

count = 0;                          // reset count
count--;                            // count = -1. No difference --count
printf(“count is %d: ”, --count);  // count is: -2
printf (“count is %d: ”, count);   // count is: -2
```

3.12 – Increment and Decrement Operators

- Suppose count = 0. What would be the result?

```
count = 0;                         // reset count
++count;                           // count = 1
printf(“count is %d: ”, count++);  // count is: 1
printf (“count is %d: ”, count);   // count is: 2

count = 0;                         // reset count
count--;                           // count = -1
printf(“count is %d: ”, --count);  // count is: -2
printf (“count is %d: ”, count);   // count is: -2
```