CSE 230
Intermediate Programming in C and C++
Introduction to C

Fall 2017

Stony Brook University

Instructor: Shebuti Rayana

Overview

- A brief discussion on introductory C language concepts
  - *Variables, Expressions, Assignments*
  - *Operators*
  - *Data types*
  - *Flow of Control*

Overview

- A brief discussion on introductory C language concepts
  - *Variables, Expressions, Assignments*
  - *Operators*
  - *Data types*
  - *Flow of Control*

Variables

- **Variables** are simply names used to refer to some location in the memory
  - *A placeholder for a value*
- Before using, you need to **declare** a variables with a specific type
- All variables in C are **typed**
  - important to know the *type* of variables and the *size* of these types
- **Example: Declaring** an integer type variable "number"
  int number;
  - **Initializing** "number" with a value 10
    number = 10;
  - Declare + Initialize : int number = 10;

Variables, Expressions, Assignments - Example

Variables declaration

Assignment statements

Expression

```
/*distance of a marathon in kilometers*/
#include<stdio.h>

int main(void)
{
        int miles, yards;
        float kilometers;

        miles = 26;
        yards = 385;
        kilometers =1.609 * (miles + yards / 1760.0);
        printf("\nA marathon is %f kilometers.\n\n", kilometers);
        return 0;

}
```

Output:
A marathon is 42.185970 kilometers.

Use of #include

- #include preprocessor directive in a code causes the compiler to replace that line with the entire text of the contents of the named source file which is included
- Example: #include<stdio.h>
  - *stdio.h is a header file, which contains declaration of functions in standard i/o library*
  - *Whenever the functions printf() and scanf() are used, the header file stdio.h should be included*

Use of printf() and scanf()

- Both functions are passed a list of arguments
  - *Control string (may contain conversion specifications)*
  - *Other arguments*
- Function printf() is used for output
  - ***Usage:*** *printf(“abc”);*
    printf(“%s”,“abc”);
    printf(“%c%c%c”,’a’,’b’,’c’);
    int x = 10; printf(“%d”, x);
    float y = 10.5; printf(“%f”, y);
- Function scanf() is used for input
  - ***Usage:*** *int x; scanf(“%d”, &x);*
    char c; scanf(“%c”, &c); Here & is the address operator

Overview

- A brief discussion on introductory C language concepts
  - *Variables, Expressions, Assignments*
  - *Operators*
  - *Data types*
  - *Flow of Control*

Operators

<table>
<tr><th>Types</th><th>Operators</th></tr>
<tr><td>Arithmetic</td><td>+ - * / %</td></tr>
<tr><td>Increment/
Decrement</td><td>++
--</td></tr>
<tr><td>Assignment</td><td>= += -= *= /= %=</td></tr>
<tr><td>Relational</td><td>== < > <= >= !=</td></tr>
<tr><td>Logical</td><td>&&(AND) ||(OR) !(NOT)</td></tr>
<tr><td>Bitwise</td><td>&(AND) |(OR) ^(XOR) ~(complement)
<< (left shift) >> (right shift)</td></tr>
<tr><td>Ternary</td><td>:? (conditionalExpression ? expr1 : expr2)</td></tr>
</table>

Operator Precedence and Associativity

Operators	Associativity
() ++(postfix) --(postfix)	left to right
+(unary) –(unary) ++(prefix) --(prefix)	right to left
* / %	left to right
+ -	left to right
= += -= *= /= %=	right to left

❖ All the operators on a given line have equal precedence with respect to each other, but have higher precedence than all the operators that occur on the lines below them.

Operators: Example

- - a * b – c is equivalent to ((-a) * b) – c
- 6 / 2 * (1 + 2) = ? (1 or 9)
- int a = b = c = 0;
  a = ++c;
  b = c++;
  printf("%d %d %d\n", a, b, ++c);
  What is the output?

Example

- - a * b – c is equivalent to ((-a) * b) – c
- 6 / 2 * (1 + 2) = ? (1 or 9)
- int a = b = c = 0;
  a = ++c;
  b = c++;
  printf(“%d %d %d\n”, a, b, ++c);
  What is the output?
  Output: 1 1 3

Overview

- A brief discussion on introductory C language concepts
  - *Variables, Expressions, Assignments*
  - *Operators*
  - *Data types*
  - *Flow of Control*

Data types

Fundamental Data types in C		
char	*signed char*	*unsigned char*
short	*int*	*long*
unsigned short	*unsigned*	*unsigned long*
float	*double*	*long double*

- Enumerated type: *enum*
- Type *void*: *void* indicates that no value
- Derived Types: *pointer, array, structure, union*
- The data type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

Integral Data Types

Type	Size	Value Range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned	4 bytes	0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	--9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

*sizes are given for 64-bit UNIX machine

Floating-Point Types

Type	Storage Size	Value Range	Precision
float	4 bytes	1.2E-38 to 3.4E+38	6 decimal
double	8 bytes	1.2E-38 to 3.4E+38	15 decimal
long double	16 bytes	3.4E-49321 to 1.2E+1049321	20 decimal

*you can check the sizes of these data types using *sizeof()*

Overview

- A brief discussion on introductory C language concepts
  - *Variables, Expressions, Assignments*
  - *Operators*
  - *Data types*
  - *Control flow*

Slide Curtesy: www.tenouk.com

Control Flow

- **Program Control**

➢ Program begins execution at the `main()` function.

➢ Statements within the `main()` function are then executed from top-down style, line-by-line.

➢ However, this order is rarely encountered in real C program.

➢ The order of the execution within the `main()` body may be branched.

➢ Changing the order in which statements are executed is called program control.

➢ Accomplished by using program control flow statements.

➢ So we can control the program flows.

Control Flow

- There are three types of program controls:

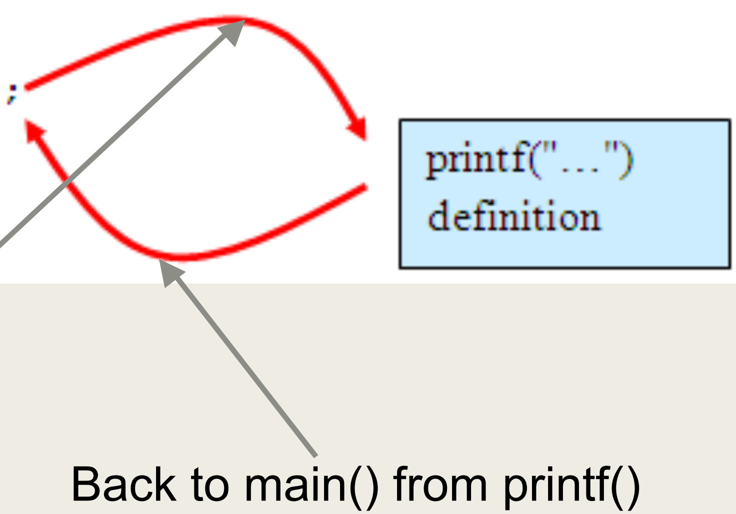
1. __Sequence__ *control structure.*
2. __Selection__ *structures such as `if`, `if-else`, nested `if`, `if-if-else`, `if-else-if` and `switch-case-break`.*
3. __Repetition__ *(loop) such as `for`, `while` and `do-while`.*

- Certain C functions and keywords also can be used to control the program flows.

Sequence

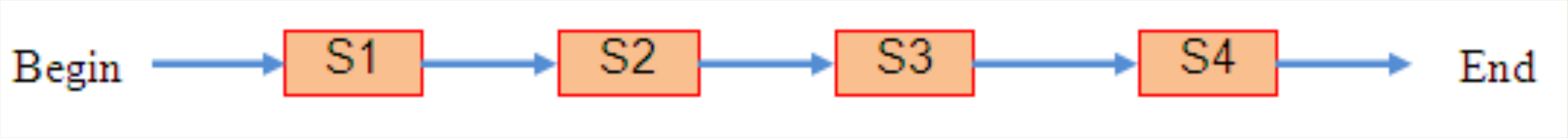
- Take a look at the following example

```c
#include <stdio.h> // put stdio.h file here

int main(void)
{
    float paidRate = 5.0, sumPaid, paidHours = 25;

    sumPaid = paidHours * paidRate;
    printf("Paid  sum = $%.2f \n", sumPaid);

    return 0;
}
```



Sequence

`float paidRate=5.0, sumPaid, paidHours=25;`	S1
`sumPaid = paidHours * paidRate;`	S2
`printf("Paid  sum = $%.2f \n", sumPaid);`	S3
`return 0;`	S4



- One entry point and one exit point.
- Conceptually, a control structure like this means a sequence execution.

Selection Control Flow

- Program need to select from the options given for execution.
- At least 2 options, can be more than 2.
- Option selected based on the *condition* evaluation result: `TRUE` or `FALSE`.

Selection: most basic `if`

`if (condition)`	`if (condition)`
`   statement;`	`   { statements;}`
`next_statement;`	`next_statement;`

1. `(condition)` is evaluated.
2. If `TRUE` (non-zero) the `statement` is executed.
3. If `FALSE` (zero) the `next_statement` following the `if` statement block is executed.
4. So, during the execution, based on some condition, some codes were skipped.

Example: `if`

For example:

```
if (hours > 70)
    hours = hours + 100;
printf("Less hours, no bonus!\n");
```

- If `hours` is less than or equal to 70, its value will remain unchanged and only `printf()` will be executed.
- If it exceeds 70, its value will be increased by 100 and then `printf()` will be executed.

Selection: `if-else`

`if (condition)`	`if (condition)`
`    statement_1;`	`    { a block of statements;}`
`else`	`else`
`    statement_2;`	`    { a block of statements;}`
`next_statement;`	`next_statement;`

Explanation:

1.The `(condition)` is evaluated.

2.If it evaluates to non-zero (TRUE), `statement_1` is executed, otherwise, if it evaluates to zero (FALSE), `statement_2` is executed.

3.They are mutually exclusive, meaning, either `statement_1` is executed or `statement_2`, but not both.

4.`statements_1` and `statements_2` can be a block of codes and must be put in curly braces.

Selection: Nested `if-else`

- The `if-else` constructs can be nested (placed one within another) to any depth.
- General forms: `if-if-else` and `if-else-if`.
- Following is `if-if-else` constructs (3 level of depth)

```
if(condition_1)
   if(condition_2)
      if(condition_3)
         statement_4;
      else
         statement_3;
   else
      statement_2;
else
   statement_1;
next_statement;
```

Selection: Nested `if-else`

- The `if-else-if` statement has the following form (3 levels example).

```
if(condition_1)
   statement_1;
else if (condition_2)
   statement_2;
else if(condition_3)
   statement_3;
else
   statement_4;
next_statement;
```

Selection: `switch-case-break`

- The <u>most flexible selection</u> program control.
- Enables the program to execute different statements based on an condition or expression that can have more than two values.
- Also called <u>multiple choice statements</u>.
- The if statement were limited to evaluating an expression that could have <u>only two logical values</u>: TRUE or FALSE.
- If more than two values, have to use <u>nested if</u>.
- The `switch` statement makes such nesting unnecessary.
- Used together with `case` and `break`.

Selection: `switch-case-break`

```
switch(condition)
{
    case  template_1  : statement(s);
                break;
    case  template_2  : statement(s);
                break;
    case  template_3  : statement(s);
                break;
    …
    …
    case  template_n : statement(s);
                break;

    default : statement(s);
}
next_statement;
```

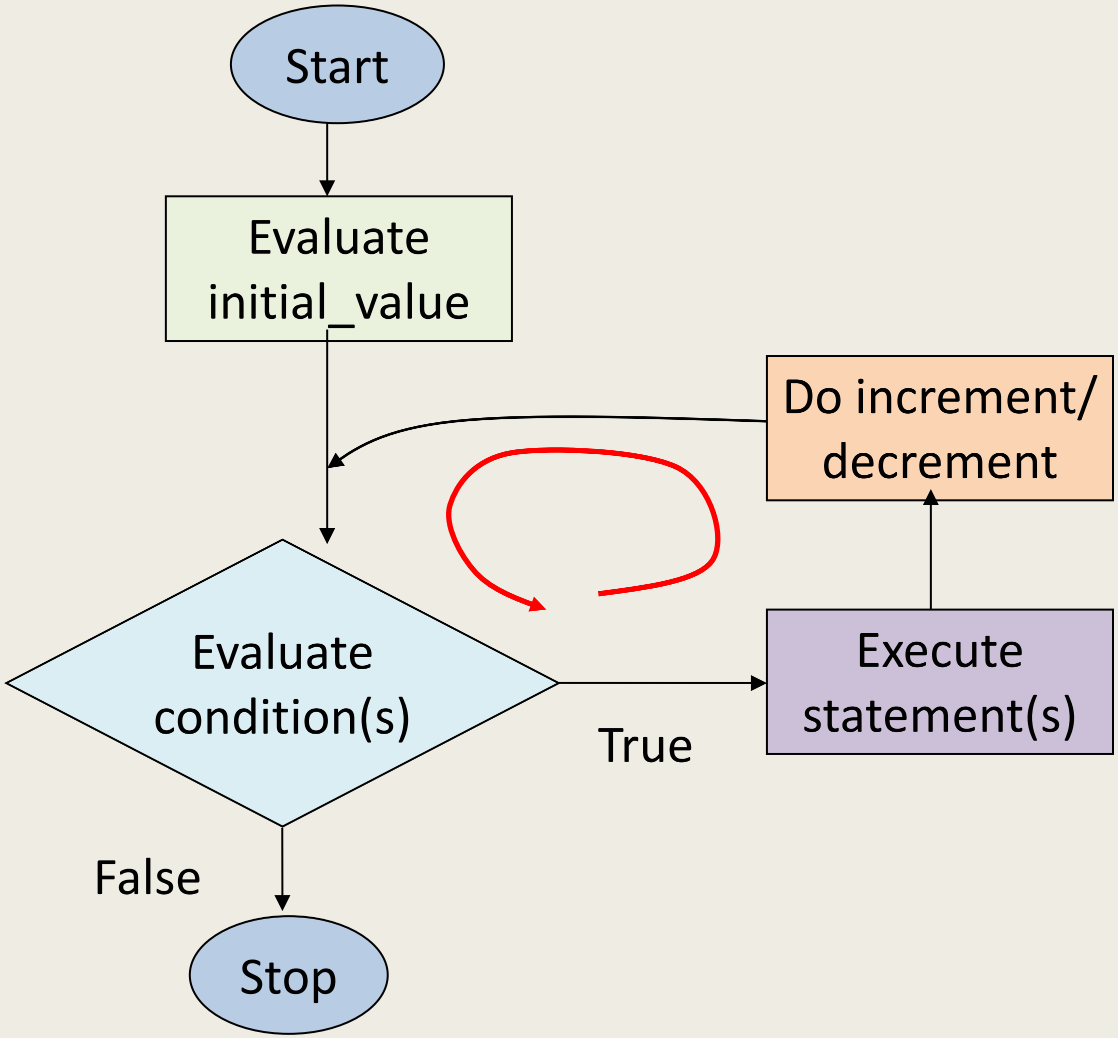
Repetition: `for` loop

- Executes a code block for a certain number of times.
- Code block may have no statement, one statement or more.
- `for` loop executes a fixed number of times.

```
for(initial_value;condition(s);increment/decrement)
    statement(s);
next_statement;
```

- `initial_value`, `condition(s)` and `increment/decrement` are any valid C expressions.
- The `statement(s)` may be a single or compound C statement (a block of code).
- When `for` statement is encountered during program execution, the following events occurs:
  1. The `initial_value` is evaluated e.g. `intNum = 1`.
  2. Then the `condition(s)` is evaluated, typically a relational expression.
  3. If `condition(s)` evaluates to `FALSE` (zero), the `for` statement terminates and execution passes to `next_statement`.
  4. If `condition(s)` evaluates as `TRUE` (non zero), the `statement(s)` is executed.
  5. Next, `increment/decrement` is executed, and execution returns to step no. 2 until `condition(s)` becomes `FALSE`.

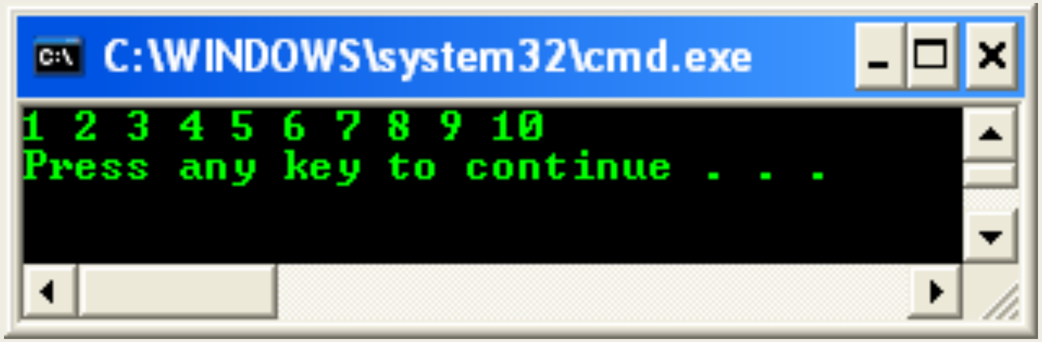
Flow Chart: `for` loop



Example: `for` loop

- A Simple `for` example, printing integer 1 to 10.

```
#include <stdio.h>
void main(void)
{
        int nCount;
        // display the numbers 1 to 10
        for(nCount = 1; nCount <= 10; nCount++)
              printf("%d ", nCount);
        printf("\n");
}
```

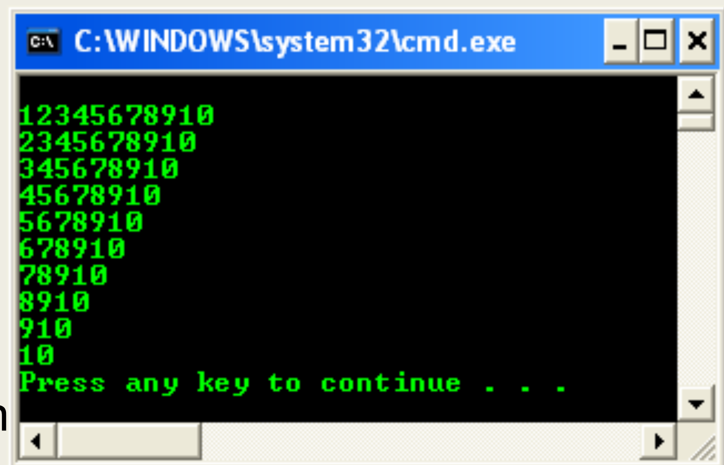


Nested `for` loop

- `for` loops can be nested

```
for(initial_value;condition(s);increment/decrement){
    for(initial_value;condition(s);increment/decrement){
        statement(s);
    }
}
next_statement;
```

- For this output the program has two `for` loops.
- The loop index `iRow` for the outer (first) loop runs from 1 to 10 and for each value of `iRow`, the loop index `jColumn` for the inner loop runs from `iRow + 1` to `10`.
- Note that for the last value of `iRow` (i.e. `10`), the inner loop is not executed at all because the starting value of `jColumn` is `2` and the expression `jColumn < 11` yields the value false (`jColumn = 11`).



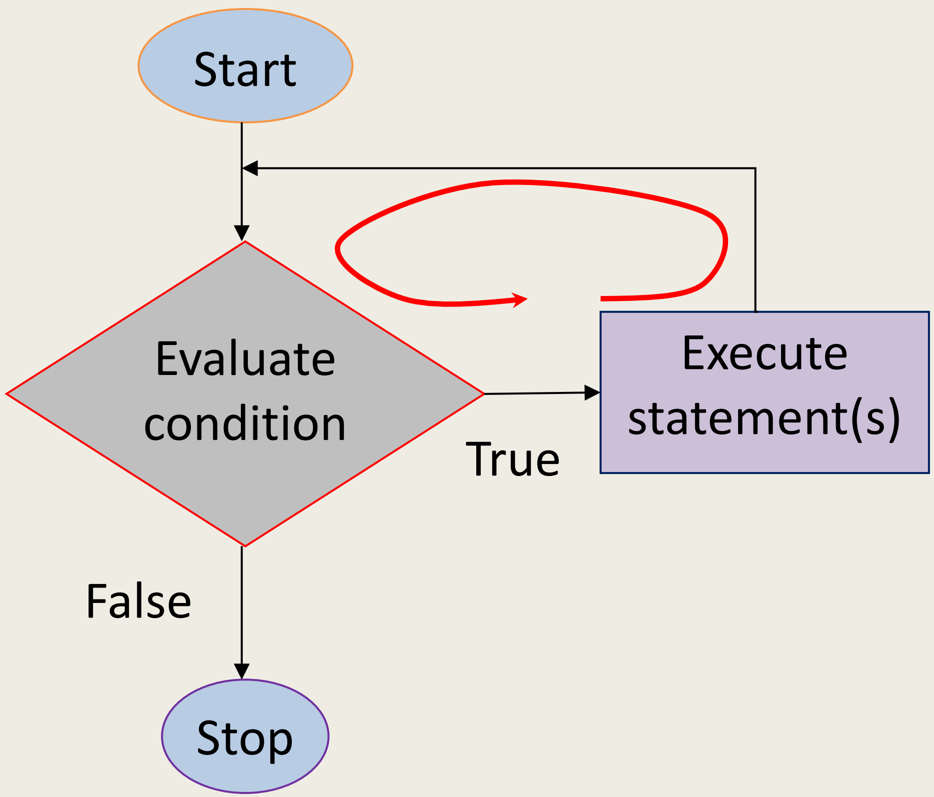
Repetition: `while` loop

- Executes a block of statements as long as a specified condition is `TRUE`.

```
while (condition)
    statement(s);
next_statement;
```

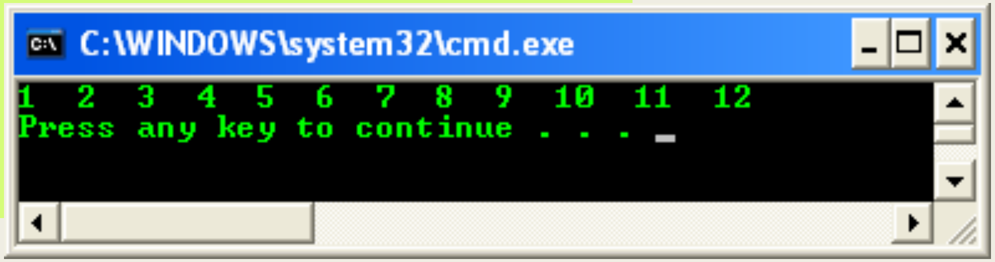
- The `(condition)` may be any valid C expression.
- The `statement(s)` may be either a single or a compound (a block of code) C statement.
- When `while` statement encountered, the following events occur:
  1. The `(condition)` is evaluated.
  2. If `(condition)` evaluates to `FALSE` (zero), the `while` loop terminates and execution passes to the `next_statement`.
  3. If `(condition)` evaluates as `TRUE` (non zero), the C `statement(s)` is executed.
  4. Then, the execution returns to step number 1 until condition becomes `FALSE`.

Flow Chart: `while` loop



Example: `while` loop

```c
// simple while loop example
#include <stdio.h>
int main(void)
{
        int  nCalculate = 1;
        // set the while condition
        while(nCalculate <= 12)
        {
                // print
                printf("%d  ", nCalculate);
                // increment by 1, repeats
                nCalculate++;
        }
          // a newline
        printf("\n");
        return 0;
}
```



`for` vs `while` loop

- The same task that can be performed using the `for` statement.
- But, `while` statement does not contain an initialization section, the program must explicitly initialize any variables beforehand.
- As conclusion, `while` statement is essentially a `for` statement without the initialization and increment components.
- `While` can be nested like `for`
- The syntax comparison between `for` and `while`,

`for( ; condition; )`	vs	`while(condition)`

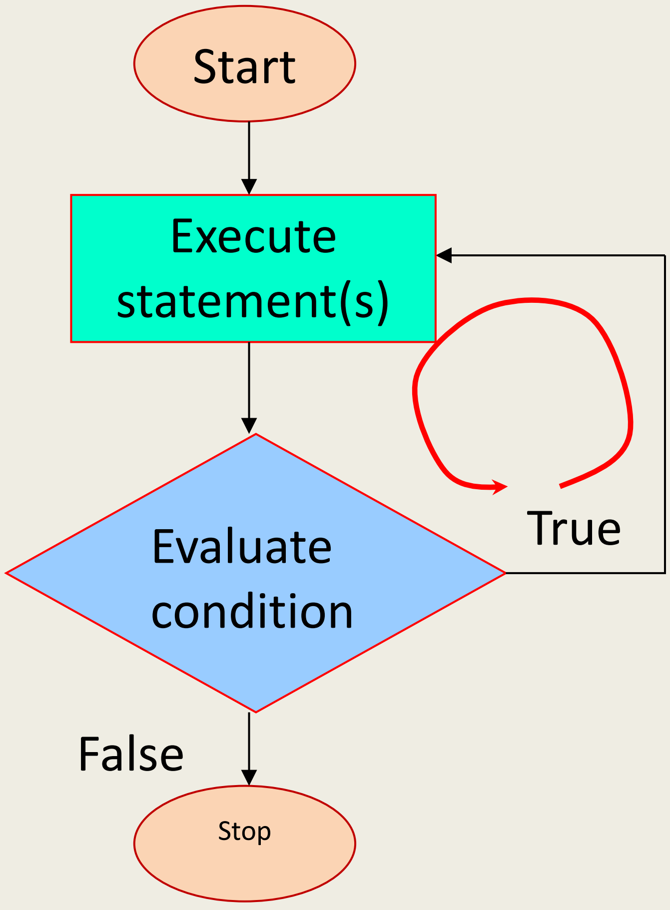
Repetition: `do-while` loop

- Executes a block of statements if the condition is true at least once.
- Test the condition at the end of the loop rather than at the beginning

```
do
    statement(s);
while (condition)
next_statement;
```

- `(condition)` can be any valid C expression.
- `statement(s)` can be either a single or compound (a block of code) C statement.
- When the program encounter the `do-while` loop, the following events occur:
  1. The `statement(s)` are executed.
  2. The `(condition)` is evaluated. If it is `TRUE`, execution returns to step number 1. If it is `FALSE`, the loop terminates and the `next_statement` is executed.
  3. This means the `statement(s)` in the `do-while` will be executed at least once.

Flow Chart: `do-while` loop



- The `statement(s)` are always executed at least once.
- `for` and `while` loops evaluate the condition at the start of the loop, so the associated statements are not executed if the condition is initially `FALSE`.

`break` statement

- The `break` statement causes an exit from the innermost enclosing loop or switch statement.

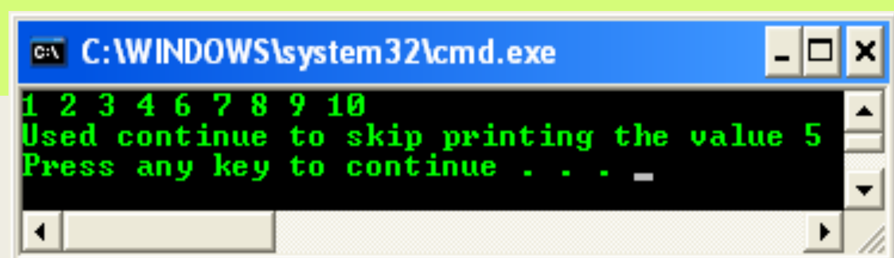
```
while (1) {
    scanf("%lf", &x);
    if (x < 0.0) /* exit loop if x is negative */
        break;
    printf("%f\n", sqrt(x));
}
/* break jumps to here */
```

`continue` statement

- `continue` keyword forces the next iteration to take place immediately, skipping any instructions that may follow it.
- The `continue` statement can only be used inside a loop (`for`, `do-while` and `while`) and not inside a `switch-case` selection.
- When executed, it transfers control to the condition (the expression part) in a `while` or `do-while` loop, and to the increment expression in a `for` loop.
- Unlike the `break` statement, `continue` does not force the termination of a loop, it merely transfers control to the next iteration.

Example: `continue` statement

```
// using the continue in for structure
#include <stdio.h>

int main(void)
{
        int iNum;
        for(iNum = 1; iNum <= 10; iNum++)
        {
        // skip remaining code in loop only if iNum == 5
                if(iNum == 5)
                    continue;
                printf("%d ", iNum);
        }
        printf("\nUsed continue to skip printing the value 5\n");
        return 0;
}
```



`goto` statement

- The `goto` statement is one of C unconditional jump or branching.
- When `goto` statement is encountered, execution jumps, or branches, to the location specified by `goto`.
- The branching does not depend on any condition.
- `goto` statement and its target label must be located in the same function, although they can be in different blocks.
- Use `goto` to transfer execution both into and out of loop.
- However, using `goto` statement strongly not recommended. Always use other C branching statements.
- When program execution branches with a `goto` statement, no record is kept of where the execution is coming from.

Example: `goto` statement

```
while (scanf("%lf", &x) == 1) {
        if (x <  0.0)
              goto negative_alert;
        printf("%f %f\n", sqrt(x) , sqrt(2 * x));
}
negative_alert: printf("Negative value encountered!\n");
```

`return` statement

- The `return` statement has a form,

```
return expression;
```

- The action is to terminate execution of the current function and pass the value contained in the expression (if any) to the function that invoked it.
- The value returned must be of the same type or convertible to the same type as the function's return type (type casting).
- More than one return statement may be placed in a function.
- The execution of the first `return` statement in the function automatically terminates the function.

Program Control

Callee

printf("…") definition

Caller

```c
#include <stdio.h>

int main(void)
{
    int nNum = 20;

    printf("Initial value of the nNum variable is %d", nNum);
    return 0;
}
```