Lecture 5: C programming

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
C: A High-Level Language

• *Gives symbolic names to values*  
  – don’t need to know which register or memory location

• **Provides abstraction of underlying hardware**  
  – operations do not depend on instruction set  
  – example: can write “a = b * c”, even though the hardware may not have a multiply instruction

• **Provides expressiveness**  
  – use meaningful symbols that convey meaning  
  – simple expressions for common control patterns (if-then-else)

• **Enhances code readability**

• **Safeguards against bugs**  
  – can enforce rules or conditions at compile-time or run-time
Compilation vs. Interpretation

• Different ways of translating high-level language

• **Interpretation**
  – interpreter = program that executes program statements
  – generally one line/command at a time
  – limited processing
  – easy to debug, make changes, view intermediate results
  – languages: BASIC, LISP, Perl, Java, Matlab, C-shell

• **Compilation**
  – translates statements into machine language
    • does not execute, but creates executable program
  – performs optimization over multiple statements
  – change requires recompilation
    • can be harder to debug, since executed code may be different
  – languages: C, C++, Fortran, Pascal
Compilation vs. Interpretation

• Consider the following algorithm:
  – Get W from the keyboard.
  – X = W + W
  – Y = X + X
  – Z = Y + Y
  – Print Z to screen.

• If interpreting, how many arithmetic operations occur?

• If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?
Compiling a C Program

• Entire mechanism is usually called the “compiler”

• Preprocessor
  – macro substitution
  – conditional compilation
  – “source-level” transformations
    • output is still C

• Compiler
  – generates object file
    • machine instructions

• Linker
  – combine object files (including libraries) into executable image
Compiler

• **Source Code Analysis**
  – “front end”
  – parses programs to identify its pieces
    • variables, expressions, statements, functions, etc.
  – depends on language (not on target machine)

• **Code Generation**
  – “back end”
  – generates machine code from analyzed source
  – may optimize machine code to make it run more efficiently
  – very dependent on target machine

• **Symbol Table**
  – map between symbolic names and items
A Simple C Program

#include <stdio.h>
#define STOP 0

/* Function: main                                   */
/* Description: counts down from user input to STOP */
main()
{
    /* variable declarations */
    int counter; /* an integer to hold count values */
    int startPoint; /* starting point for countdown */

    /* prompt user for input */
    printf("Enter a positive number: ");
    scanf("%d", &startPoint); /* read into startPoint */

    /* count down and print count */
    for (counter=startPoint; counter >= STOP; counter--)
        printf("%d\n", counter);
}
Preprocessor Directives

• `#include <stdio.h>`
  – Before compiling, copy contents of header file (stdio.h) into source code.
  – Header files typically contain descriptions of functions and variables needed by the program.
    • no restrictions -- could be any C source code

• `#define STOP 0`
  – Before compiling, replace all instances of the string "STOP" with the string "0"
  – Called a *macro*
  – Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)
Comments

- Begins with /* and ends with */
- Can span multiple lines
- Cannot have a comment within a comment
- Comments are not recognized within a string
  - example: "my/*don't print this*/string"
    would be printed as: my/*don't print this*/string
- Use comments to help reader, not to confuse or to restate the obvious
Every C program must have a function called `main()`. This is the code that is executed when the program is run.

The code for the function lives within brackets:

```
main()
{
    /* code goes here */
}
```
Variable Declarations

- Variables are used as names for data items.
- Each variable has a type, which tells the compiler how the data is to be interpreted (and how much space it needs, etc.).

  - int counter;
  - int startPoint;

- int is a predefined integer type in C.
Input and Output

• Variety of I/O functions in *C Standard Library*.
• Must include `<stdio.h>` to use them.

• `printf("%d\n", counter);`
  – String contains characters to print and formatting directions for variables.
  – This call says to print the variable `counter` as a decimal integer, followed by a linefeed (`\n`).

• `scanf("%d", &startPoint);`
  – String contains formatting directions for looking at input.
  – This call says to read a decimal integer and assign it to the variable `startPoint`. (Don’t worry about the `&` yet.)
More About Output

• Can print arbitrary expressions, not just variables
  printf("%d\n", startPoint - counter);

• Print multiple expressions with a single statement
  printf("%d %d\n", counter, startPoint - counter);

• Different formatting options:
  • %d  decimal integer
  • %x  hexadecimal integer
  • %c  ASCII character
  • %f  floating-point number
Examples

• This code:
  
  printf("%d is a prime number.\n", 43);
  printf("43 + 59 in decimal is %d.\n", 43+59);
  printf("43 + 59 in hex is %x.\n", 43+59);
  printf("43 + 59 as a character is %c.\n", 43+59);

• produces this output:
  
  43 is a prime number.
  43 + 59 in decimal is 102.
  43 + 59 in hex is 66.
  43 + 59 as a character is f.
Examples of Input

• Many of the same formatting characters are available for user input.

• `scanf("%c", &nextChar);`
  – reads a single character and stores it in `nextChar`

• `scanf("%f", &radius);`
  – reads a floating point number and stores it in `radius`

• `scanf("%d %d", &length, &width);`
  – reads two decimal integers (separated by whitespace), stores the first one in `length` and the second in `width`

• Must use ampersand (`&`) for variables being modified.
Compiling and Linking

```
gcc -o myprog -m32 myprog.c
```

Calls the compiler, the assembler, and the linker.

Use code for 32-bit machine

Your source code

You want the executable generated to be named “myprog”. If you do not specify a name, “a.out” will be generated.
Basic C Elements

• Variables
  – named, typed data items

• Operators
  – predefined actions performed on data items
  – combined with variables to form expressions, statements
Data Types

• Examples of data types in C

• int integer
• float floating point
• char character

• Exact size can vary, depending on processor, but for x86:
  – int → 4 bytes
  – float → 4 bytes
  – char → 1 byte
Variable Names

• Any combination of letters, numbers, and underscore (_)

• Case matters
  – "sum" is different than "Sum"

• Cannot begin with a number

• Only first 31 characters are used
Examples

- **Legal**
  ```c
  i
  wordsPerSecond
  words_per_second
  _green
  aReally_longName_moreThan31chars
  aReally_longName_moreThan31characters
  ```

- **Illegal**
  ```c
  10sdigit
  ten'sdigit
  done?
  double
  ```
Scope: Global and Local

• Where is the variable accessible?
  • **Global**: accessed anywhere in program
  • **Local**: only accessible in a particular region

• Compiler infers scope from where variable is declared
  – programmer doesn't have to explicitly state

• Variable is local to the block in which it is declared
  – block defined by open and closed braces `{ }`
  – can access variable declared in any "containing" block

• **Global variable** is declared outside all blocks
Example

```c
#include <stdio.h>

int itsGlobal = 0;

main()
{
    int itsLocal = 1;   /* local to main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    {
        int itsLocal = 2;   /* local to this block */
        itsGlobal = 4;      /* change global variable */
        printf("Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output
• Global 0 Local 1
• Global 4 Local 2
• Global 4 Local 1
Operators

- Programmers manipulate variables using the operators provided by the high-level language.

- Variables and operators combine to form expressions and statements which denote the work to be done by the program.

- Each operator may correspond to many machine instructions.
Expression

• Any combination of variables, constants, operators, and function calls
  – every expression has a type, derived from the types of its components (according to C typing rules)

• Examples:
  – `counter >= STOP`
  – `x + sqrt(y)`
  – `x & z + 3 || 9 - w -- % 6`
Statement

• Expresses a complete unit of work
  – executed in sequential order

• Simple statement ends with semicolon
  
  \[
  z = x \times y; \quad /* \text{assign product to } z */
  \]
  
  \[
  y = y + 1; \quad /* \text{after multiplication} */
  \]
  
  \[
  ; \quad /* \text{null statement} */
  \]

• Compound statement groups simple statements using braces.
  – syntactically equivalent to a simple statement
  
  \[
  \{ \quad z = x \times y; \quad y = y + 1; \quad \}
  \]
Operators

• Three things to know about each operator
  • (1) Function
    – what does it do?
  • (2) Precedence
    – in which order are operators combined?
    – Example:
      "a * b + c * d" is the same as "(a * b) + (c * d)"
      because multiply (*) has a higher precedence than addition (+)
  • (3) Associativity
    – in which order are operators of the same precedence combined?
    – Example:
      "a - b - c" is the same as "(a - b) - c"
      because add/sub associate left-to-right
Assignment Operator

• Changes the value of a variable.

\[ x = x + 4; \]

1. Evaluate right-hand side.

2. Set value of left-hand side variable to result.
Assignment Operator

• All expressions evaluate to a value, even ones with the assignment operator.

• For assignment, the result is the value assigned.
  – usually (but not always) the value of the right-hand side
    • type conversion might make assigned value different than computed value

• Assignment associates right to left.

  \[ y = x = 3; \]
  
  y gets the value 3, because \((x = 3)\) evaluates to the value 3.
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>x % y</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>x + y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>x − y</td>
<td>7</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

Table 12.5 gives more complete info

* / % have higher precedence than + −.
Arithmetic Expressions

If mixed types, smaller type is "promoted" to larger.

\[ x + 4.3 \]
if \( x \) is int, converted to double and result is double

Integer division -- fraction is dropped.

\[ x / 3 \]
if \( x \) is int and \( x=5 \), result is 1 (not 1.6666666666666666...)

Modulo -- result is remainder.

\[ x \% 3 \]
if \( x \) is int and \( x=5 \), result is 2.
# Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>~x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>x &lt;&lt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>x &gt;&gt; y</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>x &amp; y</td>
<td>11</td>
<td>l-to-r</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>x ^ y</td>
<td>12</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td>bitwise OR</td>
<td>x</td>
<td>y</td>
<td>13</td>
</tr>
</tbody>
</table>
## Logical Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>! x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>x &amp;&amp; y</td>
<td>14</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
<td>x</td>
</tr>
</tbody>
</table>

- Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero).
- Result is 1 (TRUE) or 0 (FALSE).
# Relational Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>( x &gt; y )</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>( \geq )</td>
<td>greater than or equal</td>
<td>( x \geq y )</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>( x &lt; y )</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>( \leq )</td>
<td>less than or equal</td>
<td>( x \leq y )</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>( x == y )</td>
<td>10</td>
<td>l-to-r</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>( x != y )</td>
<td>10</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

Result is 1 (TRUE) or 0 (FALSE).

Note: Don't confuse equality (==) with assignment (=).
## Special Operators: ++ and --

- Changes value of variable before (or after) its value is used in an expression.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>postincrement</td>
<td>x++</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>x--</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td>++x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;=</td>
<td>predecrement</td>
<td>--x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
</tbody>
</table>

**Pre:** Increment/decrement variable **before** using its value.

**Post:** Increment/decrement variable **after** using its value.
Using `++` and `--`

- `x = 4;`
- `y = x++;`
- Results: `x = 5, y = 4`  
  (because `x` is incremented after assignment)

- `x = 4;`
- `y = ++x;`
- Results: `x = 5, y = 5`  
  (because `x` is incremented before assignment)
Practice with Precedence

• Assume $a=1$, $b=2$, $c=3$, $d=4$.

$$x = a \times b + c \times d / 2;$$

same as:

$$x = (a \times b) + ((c \times d) / 2);$$

• For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.
**Special Operators: +=, *=, etc.**

- Arithmetic and bitwise operators can be combined with assignment operator.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>x += y;</td>
<td>x = x + y;</td>
</tr>
<tr>
<td>x -= y;</td>
<td>x = x - y;</td>
</tr>
<tr>
<td>x *= y;</td>
<td>x = x * y;</td>
</tr>
<tr>
<td>x /= y;</td>
<td>x = x / y;</td>
</tr>
<tr>
<td>x %= y;</td>
<td>x = x % y;</td>
</tr>
<tr>
<td>x &amp;= y;</td>
<td>x = x &amp; y;</td>
</tr>
<tr>
<td>x</td>
<td>= y;</td>
</tr>
<tr>
<td>x ^= y;</td>
<td>x = x ^ y;</td>
</tr>
<tr>
<td>x &lt;&lt;= y;</td>
<td>x = x &lt;&lt; y;</td>
</tr>
<tr>
<td>x &gt;&gt;= y;</td>
<td>x = x &gt;&gt; y;</td>
</tr>
</tbody>
</table>

All have same precedence and associativity as = and associate right-to-left.
Special Operator: Conditional

- Symbol: `?:
- Operation: conditional
- Usage: `x?y:z`
- Precedence: 16
- Assoc: l-to-r

If \( x \) is TRUE (non-zero), result is \( y \); else, result is \( z \).

Like a MUX, with \( x \) as the select signal.
Control Structures

• **Conditional**
  making a decision about which code to execute, based on evaluated expression
  - if
  - if-else
  - switch

• **Iteration**
  executing code multiple times, ending based on evaluated expression
  - while
  - for
  - do-while
• *if* (condition)  
  *action*;

*Condition* is a C expression, which evaluates to *TRUE* (non-zero) or *FALSE* (zero).

*Action* is a C statement, which may be simple or compound (a block).
Example If Statements

• if (x <= 10)
  \[ y = x \times x + 5; \]

• if (x <= 10) {
  \[ y = x \times x + 5; \]
  \[ z = (2 \times y) / 3; \]
}

• if (x <= 10)
  \[ y = x \times x + 5; \]
  \[ z = (2 \times y) / 3; \]

- compound statement; both executed if x <= 10
- only first statement is conditional; second statement is always executed
More If Examples

• if (0 <= age && age <= 11)
  kids += 1;

• if (month == 4 || month == 6 ||
    month == 9 || month == 11)
  printf(“The month has 30 days.
”);

• if (x = 2)
  y = 5;

  \textcolor{red}{always} true, \\
  so action is \textcolor{red}{always} executed!

• This is a common programming error (= instead of ==), not caught by compiler because it’s syntactically correct.
If's Can Be Nested

```java
if (x == 3)
    if (y != 6) {
        z = z + 1;
        w = w + 2;
    }
```

is the same as...

```java
if ((x == 3) && (y != 6)) {
    z = z + 1;
    w = w + 2;
}
```
If-else

- if (condition)
  action_if;
else
  action_else;

Else allows choice between two mutually exclusive actions without re-testing condition.
Matching Else with If

Else is always associated with closest unassociated if.

is the same as...

```java
if (x != 10) {
    if (y > 3)
        z = z / 2;
    else
        z = z * 2;
}
```

is NOT the same as...

```java
if (x != 10) {
    if (y > 3)
        z = z / 2;
}
else
    z = z * 2;
```
Chaining If’s and Else’s

```c
if (month == 4 || month == 6 || month == 9
    || month == 11)
    printf("Month has 30 days.\n");
else if (month == 1 || month == 3 ||
    month == 5 || month == 7 ||
    month == 8 || month == 10 ||
    month == 12)
    printf("Month has 31 days.\n");
else if (month == 2)
    printf("Month has 28 or 29 days.\n");
else
    printf("Don’t know that month.\n");
```
While

- `while (test) loop_body;`

Executes loop body as long as `test` evaluates to TRUE (non-zero).

Note: Test is evaluated **before** executing loop body.
Infinite Loops

• The following loop will never terminate:

    x = 0;
    while (x < 10)
        printf("%d ", x);

• Loop body does not change condition, so test never fails.

• This is a common programming error that can be difficult to find.
for (init; end-test; re-init) statement

Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.

Note: Test is evaluated before executing loop body.
Example For Loops

• /* -- what is the output of this loop? -- */
  for (i = 0; i <= 10; i ++)
    printf("%d ", i);

• /* -- what does this one output? -- */
  letter = 'a';
  for (c = 0; c < 26; c++)
    printf("%c ", letter+c);

• /* -- what does this loop do? -- */
  numberOfOnes = 0;
  for (bitNum = 0; bitNum < 16; bitNum++) {
    if (inputValue & (1 << bitNum))
      numberOfOnes++;
  }
Nested Loops

Loop body can (of course) be another loop.

/* print a multiplication table */
for (mp1 = 0; mp1 < 10; mp1++) {
  for (mp2 = 0; mp2 < 10; mp2++) {
    printf("%d\t", mp1*mp2);
  }
  printf("\n");
}
Another Nested Loop

• The test for the inner loop depends on the counter variable of the outer loop.

```java
for (outer = 1; outer <= input; outer++) {
    for (inner = 0; inner < outer; inner++)
    {
        sum += inner;
    }
}
```
Do-While

- `do loop_body;`
- `while (test);`

Executes loop body as long as test evaluates to TRUE (non-zero).

Note: Test is evaluated **after** executing loop body.
Problem 1: Calculating Pi

- Calculate $\pi$ using its series expansion. User inputs number of terms.

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots + (-1)^{n-1} \frac{4}{2n+1} + \cdots$$
Pi: 1st refinement

Start
Initialize
Get Input
Evaluate Series
Output Results
Stop

Initialize iteration count

for loop

count<terms

Evaluate next term

count = count+1
Pi: 2nd refinement

Initialize iteration count

count < terms

Evaluate next term

count = count + 1

count is odd

if count is odd

subtract term

else

add term

add term
Pi: Code for Evaluate Terms

```c
for (count=0; count < numOfTerms; count++)
{
    if (count % 2) {
        /* odd term -- subtract */
        pi -= 4.0 / (2 * count + 1);
    }
    else {
        /* even term -- add */
        pi += 4.0 / (2 * count + 1);
    }
}
```

Note: Code in text is slightly different, but this code corresponds to equation.
#include <stdio.h>

main() {
    double pi = 0.0;
    int numOfTerms, count;

    printf("Number of terms (must be 1 or larger) : ");
    scanf("%d", &numOfTerms);

    for (count=0; count < numOfTerms; count++) {
        if (count % 2) {
            pi -= 4.0 / (2 * count + 1); /* odd term -- subtract */
        }
        else {
            pi += 4.0 / (2 * count + 1); /* even term -- add */
        }
    }

    printf("The approximate value of pi is %f\n", pi);
}
Problem 2: Finding Prime Numbers

- Print all prime numbers less than 100.
- A number is prime if its only divisors are 1 and itself.
- All non-prime numbers less than 100 will have a divisor between 2 and 10.
Primes: 1st refinement

Start

Initialize

Print primes

Stop

Initialize
num = 2

num < 100

if prime
Print num

num = num + 1

F

T

Print primes

Stop
Primes: 2nd refinement

Initialize
num = 2

num < 100

F

T

Print num if prime

num = num + 1

Divide num by 2 through 10

no divisors?

F

T

Print num
Primes: 3rd refinement

Divide num by 2 through 10

no divisors?  

Print num

Initialize divisor = 2

divisor <= 10

Clear flag if num%divisor > 0

divisor = divisor + 1
Primes: Using a Flag Variable

• To keep track of whether number was divisible, we use a "flag" variable.
  – Set prime = TRUE, assuming that this number is prime.
  – If any divisor divides number evenly, set prime = FALSE.
    • Once it is set to FALSE, it stays FALSE.
  – After all divisors are checked, number is prime if the flag variable is still TRUE.

• Use macros to help readability.
• #define TRUE  1
  #define FALSE 0
```c
#include <stdio.h>
#define TRUE  1
#define FALSE 0

main () {
    int num, divisor, prime;

    /* start with 2 and go up to 100 */
    for (num = 2; num < 100; num ++ ) {

        prime = TRUE;  /* assume num is prime */

        /* test whether divisible by 2 through 10 */
        for (divisor = 2; divisor <= 10; divisor++)
            if (((num % divisor) == 0) && (num != divisor))
                prime = FALSE;  /* not prime */

        if (prime)  /* if prime, print it */
            printf("The number %d is prime\n", num);
    }
}
```
Switch

```c
switch (expression) {
    case const1:
        action1; break;
    case const2:
        action2; break;
    default:
        action3;
}
```

Alternative to long if-else chain. If break is not used, then case "falls through" to the next.
More About Switch

• Case expressions must be constant.

```c
case i: /* illegal if i is a variable */
```

• If no break, then next case is also executed.

```c
switch (a) {
    case 1:
        printf("A");
    case 2:
        printf("B");
    default:
        printf("C");
}
```

If a is 1, prints “ABC”.
If a is 2, prints “BC”.
Otherwise, prints “C”.
Break and Continue

- **break;**
  - used *only* in switch statement or iteration statement
  - passes control out of the “smallest” (loop or switch) statement containing it to the statement immediately following
  - usually used to exit a loop before terminating condition occurs (or to exit switch statement when case is done)

- **continue;**
  - used only in iteration statement
  - terminates the execution of the loop body for this iteration
  - loop expression is evaluated to see whether another iteration should be performed
  - if for loop, also executes the re-initializer
Example

• What does the following loop do?
```c
for (i = 0; i <= 20; i++) {
    if (i%2 == 0) continue;
    printf("%d ", i);
}
```

• What happens if `break` instead of `continue`?
Conclusions

• C programming is a very strong programming language and is the closest to assembly from all the high-level language programs.
• We have studied the basics of C programming.