# BIL 104E Introduction to Scientific and Engineering Computing

## Lecture 3

### Expressions, Assignment Statements, Operators

# *Character Data*

- **Numeric information** is represented in a C program as **integers** or **floating-point** values. Numeric values are often used in arithmetic computations.

- **Nonnumeric information** may consist of **alphabetic characters**, **digits**, and **special characters**.

- Each character corresponds to a **binary code** value. The most commonly used binary codes are **ASCII** (American Standard Code for Information Interchange) and **EBCDIC** (Extended Binary Coded Decimal Interchange Code).

- A total of 128 characters can be represented in the ASCII code.

# *char Data Type*

| Character | ASCII Code | Integer Equivalent |
|---|---|---|
| newline, \n | 0001010 | 10 |
| % | 0100101 | 37 |
| 3 | 0110011 | 51 |
| A | 1000001 | 65 |

- Note that the binary representation for a character digit is not equal to the binary representation for an integer digit.
- Nonnumeric information can be represented by constants or by variables.
  - A character constant is enclosed in single quotes, as in **'A'** , **'b'** , and **'3'** .
  - A variable that is going to contain a character can be defined as an integer or as a character data type (**char**).

# Character Initialization

- The binary representation for a character can be interpreted as a character or as an integer.
- To print a value as an integer, the **%i** or **%d** specifier is used; to print a value as a character, the **%c** specifier is used.
- **Example**:

```
int k=97;
char c='a';
printf("value of k: %c; value of c: %c \n",k,c);
printf("value of k: %,i; value of c: %i \n",k,c);
```

**Output:**

```
value of k: a; value of c: a
value of k: 97; value of c: 97
```

# Reading and Printing Characters

- A **text stream** is composed of sequence of characters.

- The end of a text stream is indicated with a special value, **EOF**, which is a symbolic constant defined in stdio.h.

- **stdin**: The standard input for reading (usually keyboard)
- **stdout**: The standard output for writing. (usually monitor)
- **stderr**: The standard error for writing error messages. (always monitor)

- Although the printf and scanf functions can be used to read characters using the %c specifier, there are special functions for reading and printing characters:
  - The **getc()** function
  - The **putc()** function
  - The **getchar()** function
  - The **putchar()** function

# *getc() and getchar()*

- The **int getc(FILE *stream)** function reads the next character from a file stream, and returns the integer value of the character as the function value.

- The **int getchar(void)** function reads a character from the standard input and returns the integer value of the character as the function value. It is equivalent to getc(stdin).

- **Example:**

```
#include <stdio.h>
main(){
        int ch1, ch2;
        printf("Enter two characters from the keyboard:\n ");
        ch1=getc(stdin);
        ch2=getchar();
        printf("The first character you entered is: %c\n",ch1);
        printf("The second character you entered is: %c\n ",ch2);
        return 0;
 }
```

# *putc() and putchar()*

- The **int putc(int c, FILE *stream)** function prints the character that corresponds to the integer argument to the specified file stream. It then returns the same character as the function value.

- The **int putchar(int)** function prints the character that corresponds to the integer argument to the computer screen. It then returns the same character as the function value.

- **Example:**

```
#include <stdio.h>
main(){
   int ch1=65, ch2=98;
   printf("The character that has numeric value of %d is: ",ch1);
   putc(ch1,stdout); putc('\n',stdout);
   printf("The character that has numeric value of %d is: ",ch2);
   putchar(ch2); putchar('\n');
   return 0;
}
```

# Symbolic Constants

- Defined with a preprocessor directive that assigns an identifier to a constant.
- The directive can appear anywhere in the program; the compiler will replace each occurrence of the directive identifier with the constant value in all statements that follow the directive.
- Only one symbolic constant can be defined in a directive; if several symbolic constants are desired, several separate directives are required.
- Preprocessor directives which include the **#define** statement; do not end with a semicolon.

**Example**:

```
#define PI 3.141593          /*Note ";" is not used */
...
area=PI*radius*radius;
```

# *Assignment Statements*

- **General Form:**

  identifier=expression;
  /*The equal sign should be read as “is assigned the value of” */

  **Example:**

```
sum=10.5;                 /* Expression is a constant              */
rate=state_tax;           /* Expression is another variable        */
sum= a+b;                 /* Expression is result of an operation*/
```

- **Multiple assignments are also allowed in C.**

  **Example:**

```
x=y=z=0;
```

# Assignment Statements

- If a value is assigned to a variable that has a different data type, then a conversion must occur during the execution of the statement. Sometimes this may cause loss of data.

**Example:**

```
int a;
float b;
a=12.8;          /* Information loss: a will be 12*/
b=6;             /* No information loss: b will be 6.0 */
```

# *Arithmetic Operators*

| Symbol | Meaning |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Remainder (or modulus) |

# *Arithmetic Operators*

- **Examples:**

```
area_square=side*side;
area_triangle=0.5*base*height;

x=x+1;              /* not valid in algebra but valid in C      */
                    /* x is assigned the value of x plus 1      */
                    /* value stored in x is incremented by 1    */
```

- The **modulus operator** is useful in determining if an integer is a multiple of another number. Thus, if x%2 is equal to 0 then x is even and if x%5 is equal to 0 then x is a multiple of 5.

- **Precedence**:

a*b + b/c*d  is equivalent to  (a*b) + ((b/c)*d)

# Precedence

| Precedence | Operator | Associativity |
|---|---|---|
| 1 | parenthesis: **( )** | innermost first |
| 2 | uniary operators: **+ - (type)** | right to left |
| 3 | binary operators: *** / %** | left to right |
| 4 | binary operators: **+ -** | left to right |
| 5 | assignment operators:<br>**= += -= *= /= %=** | right to left |

# *Arithmetic Operations*

- The result of a **binary operation** with values of same type is another value of the same type.
  - For example: If a and b are double then a/b is also double.
- An **integer division** can sometimes produce unexpected results because any decimal portion of the integer is dropped; the result is a truncated result, not a rounded result.
  - Thus, 5/3 is equal to 1, and 3/6 is equal to 0.
- An operation between values with different types is a **mixed operation**. Before the operation is performed, the value with the lover type is converted to the higher type, thus the operation is performed with values of the same type.

# Cast Operator

- Cast operator allows specifying a type change temporarily in the value before the next computation.

**Example (Without cast operator):**

```
int sum=18, count=5;
float average;
...
average=sum/count;  /*average is 3.0, not 3.6 */

/* the result of the integer division is going to be */
/* a truncated result, thus there is information loss. */
```

# Cast Operator

**Example (With cast operator):**

```
int sum=18, count=5;
float average;
...
average=(float) sum/count;  /*average is 3.6                    */

/* by the cast operator sum is converted to float               */
/* before the division is performed. The division is            */
/* then a mixed operation between a float value and             */
/* an integer, so the value of the count is also converted      */
/* to float value before the division. The result is a float    */
/* value and stored in average without loss of information.     */
```

# *Break long expressions into several statements*

$$f = \frac{x^3 - 2x^2 + x - 6.3}{x^2 + 0.05005x - 3.14}$$

```
f=(x*x*x-2*x*x+x-6.3)/(x*x+0.05005*x-3.14);
```

Statement can be broken into two lines:

```
f=(x*x*x-2*x*x+x-6.3)/
(x*x+0.05005*x-3.14);
```

Or, numerator and denominator can be computed separately:

```
numerator=x*x*x-2*x*x+x-6.3;
denominator=x*x+0.05005*x-3.14;
f= numerator/denominator;
```

# *Increment and Decrement Operators*

- Increment operator (++) :

  **y++;** is equal to **y = y + 1;**

- Decrement operator (--) :

  **y--;** is equal to **y = y – 1;**

- **Preincrementation** and **predecrementation**: The identifier is modified and the new value is used in evaluating the rest of the expression.

  **w = ++x – y;** is equivalent to

  ```
  x = x +1;
  w = x – y;
  ```

- **Postincrementation** and **postdecrementation**: The old value of the identifier is used in evaluating the rest of the expression and its value is modified.

  **w = x++ - y;** is equivalent to

  ```
  w = x – y;
  x = x + 1;
  ```

# *Abbreviated Assignment Operators*

| | | |
|---|---|---|
| **x = x + 3;** | is equivalent to | **x += 3;** |
| **d = d / a;** | is equivalent to | **d /= a;** |
| **a = b += c + d;** | is equivalent to | **a = (b += (c + d));** |
| **a = (b += (c + d));** | is equivalent to | **a = (b = b + (c + d));** |
| | or | |
| **a = (b += (c + d));** | is equivalent to | **b = b + (c + d);**<br>**a = b;** |

# Elementary Math Functions

- **#include <math.h>** preprocessor directive should be used in programs referencing the mathematical functions.
- All math functions return data type **double.**

| | |
|---|---|
| **fabs(x)** | Computes the absolute value of **x**. |
| **sqrt(x)** | Computes the square root of **x**, where **x**$\geq$0. |
| **pow(x,y)** | Computes $\mathbf{x}^{\mathbf{y}}$.<br>Errors occur if **x**=0 and **y**$\leq$0, or if **x**<0 and **y** is not an integer. |
| **ceil(x)** | Rounds **x** to the nearest integer toward $\infty$. |
| **floor(x)** | Rounds **x** to the nearest integer toward $-\infty$. |
| **exp(x)** | Computes $e^{\mathbf{x}}$. |
| **log(x)** | Computes ln(**x**). Errors if **x**$\leq$0. |
| **log10(x)** | Computes $\log_{10}(\mathbf{x})$. Errors if **x**$\leq$0. |

# Trigonometric Functions

Trigonometric functions take arguments in radians. To convert radians to degrees, or degrees to radians the following conversions can be used:

```
#define PI 3.141593
...
angle_deg = angle_rad*(180/PI);
angle_rad = angle_deg*( PI/180);
```

| | |
|---|---|
| **sin(x)** | Computes the sine of **x.** |
| **cos(x)** | Computes the cosine of **x.** |
| **tan(x)** | Computes the tangent of **x.** |
| **asin(x)** | Computes the arcsine of **x** where $-1 \leq x \leq 1$. |
| **acos(x)** | Computes the arccosine of **x** where $-1 \leq x \leq 1$. |
| **atan(x)** | Computes the arctangent of **x.** |
| **atan2(y,x)** | Computes the arctangent of **y/x.** Returns an angle in any quadrant, depending on the signs of **x** and **y**. |