Computer Programming

2

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	paranthesis: **( )**	innermost first
2	uniary operators: **+ - (type)**	right to left
3	binary operators: *** / %**	left to right
4	binary operators: **+ -**	left to right
5	assignment operators: **= += -= *= /= %=**	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);**
		a = b;

Conditional Expressions

- A **condition** is an expression that can be evaluated to be **true** or **false** and is composed of expressions combined with **relational operators**; a condition can also include **logical operators**.

- The **relational operators** can be used to compare two expressions.

Relational Operator	Interpretation
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

Relational Operators

- **Examples:**

```
a<b
x+y >= 10.5
fabs(denominator) < 0.0001
```

- A **true** condition is assigned to a value of **1**; a **false** condition is assigned to a value of **zero**. So the following statement is valid:

```
d = b>c;   //if b>c then d=1, else d=0
```

- Because a condition is given a value, it is valid to use a value in place of a condition

Logical Operators

- **Logical operators** can be used to compare conditions and generate new conditions.

  Example:

  a<b && b<c   (The relational operators have higher precedence than the logical operator.)

Logical Operator	**Symbol**
not	!
and	&&
or	||

A	**B**	**A&&B**	**A||B**	**!A**	**!B**
False	False	False	False	True	True
False	True	False	True	True	False
True	False	False	True	False	True
True	True	True	True	False	False

Precedence

- A condition can contain several logical operators. The hierarchy from highest to the lowest is **!**, **&&**, **||**, but parenthesis can be used to change the hierarchy.

Precedence	**Operator**	**Associativity**
1	( )	innermost first
2	+ - ++ -- (type) !	right to left (unary)
3	* / %	left to right
4	+ -	left to right
5	< <= > >=	left to right
6	== !=	left to right
7	&&	left to right
8	!!	left to right
9	= += -= *= /= %=	right to left

Mathematical Functions

- **#include <math.h>** preprocessor directive should be used in programs referencing the mathematical functions which are available in Standard C library.

- General notes about functions:

  - A function reference **returns** a **single value**.

  - The parentheses following the function name contain the inputs to the function, which are called **parameters** or **arguments**.

  - A function may contain no arguments, one argument, or many arguments, depending on its definition.

  - The arguments should be listed in the correct order as in the function definition.

Elementary Math Functions

fabs(x)	Computes the absolute value of **x**.
sqrt(x)	Computes the square root of **x**, where **x**≥0.
pow(x,y)	Computes $\mathbf{x}^{\mathbf{y}}$. Errors occur if **x**=0 and **y**≤0, or if **x**<0 and **y** is not an integer.
ceil(x)	Rounds **x** to the nearest integer toward ∞.
floor(x)	Rounds **x** to the nearest integer toward -∞.
exp(x)	Computes $e^{\mathbf{x}}$.
log(x)	Computes ln(**x**). Errors if **x**≤0.
log10(x)	Computes $\log_{10}$(**x**). Errors if **x**≤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≤**x**≤1.
acos(x)	Computes the arccosine of **x** where -1≤**x**≤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.**