Sequential Structure

Chapter 02

CMPE-112 *Programming Fundamentals*

Lecture Plan

- Some examples of programs in C
- Main elements
  - Character set
  - Data types
  - Classes of data
  - Constants
  - Operators
  - Expressions
  - Assignments
- Function *printf()*
- Function *scanf()*
- Sample programs

- *Automatic* type conversions
  - Automatic *unary* conversions
  - Automatic *binary* conversions
  - Rules for binary conversions
- *Explicit* Type Conversions
- Type Conversion *in Assignments*

First Example (I)

```
/* Ch_02_1.C -- Chapter 02. First illustration program */
/* It checks if a point belongs to a line 16x-2y=10 */

#include <stdio.h>

int main()
{
  int x, y, z;

  printf("\n\nPlease, enter coordinates of a point (x y): ");
  scanf("%d %d", &x, &y);

  z = 16 * x - 2 * y;

  if (z == 10)
    printf("\nThe point (%1d, %1d) is located on the line.\n“, x, y);
  else
    printf("\nThe point (%1d, %1d) is not located on the line.\n“, x, y);

  return 0;
}
```


First Example (II)

Please, enter coordinates of a point (x y): 2 4

The point (2, 4) is not located on the line.

Please, enter coordinates of a point (x y): 2 11

The point (2, 11) is located on the line.

Third Example

```
/* Ch_02_3.C -- Chapter 02. Third illustration program */
/* It checks if a point belongs to a line COEF_Ax-COEF_By=COEF_C  */
/*    where COEF_A, COEF_B, COEF_C are constant values */

#include <stdio.h>

#define COEF_A 16
#define COEF_B  2
#define COEF_C 10

int main()
{
  int x, y, z;

  printf("\n\nPlease, enter coordinates of a point (x y): ");
  scanf("%d %d", &x, &y);

  z = COEF_A * x - COEF_B * y;

  if (z == COEF_C)
    printf("\nThe point (%1d, %1d) is located on the line.\n", x, y);
  else
    printf("\nThe point (%1d, %1d) is not located on the line.\n", x, y);

  return 0;
}
```

5

5

Fourth Example

```
/* Ch_02_4.C -- Chapter 02. Fourth illustration program */
/* It checks if 3 points belong to a line COEF_Ax-COEF_By=COEF_C  */
/*    where COEF_A, COEF_B, COEF_C are constant values */

#include <stdio.h>

#define COEF_A 16
#define COEF_B  2
#define COEF_C 10

int main()
{
  int x, y, z;
  int i;

  for (i=0; i<3; i++) {
    printf("\n\nPlease, enter coordinates of a point (x y): ");
    scanf("%d %d", &x, &y);

    z = COEF_A * x - COEF_B * y;

    if (z == COEF_C)
      printf("\nThe point (%1d, %1d) is located on the line.\n", x, y);
    else
      printf("\nThe point (%1d, %1d) is not located on the line.\n", x, y);
  }

  return 0;
}
```

6

6

Fifth Example

```
/* Ch_02_5.C -- Chapter 02. Fifth illustration program */

#include <stdio.h>
#include <math.h>

int main()
{
    int number;
    double square_root;

    printf("Please, enter a number: ");
    scanf("%d", &number);

    square_root = sqrt(number);

    printf("\nSqare root of %1d is %4.3f\n\n", number, square_root);

    return 0;
}
```

7

7

Data Types

Type	Length	Range
unsigned char	8 bits	0 to 255
char	8 bits	-128 to 127
enum	16 bits	-32,768 to 32,767
unsigned int	16 bits	0 to 65,535
short int	16 bits	-32,768 to 32,767
int	16 bits	-32,768 to 32,767
unsigned long	32 bits	0 to 4,294,967,295
long	32 bits	-2,147,483,648 to 2,147,483,647
float	32 bits	3.4 x 10-38 to 3.4 x 10+38
double	64 bits	1.7 x 10-308 to 1.7 x 10+308
long double	80 bits	3.4 x 10-4932 to 1.1 x 10+4932
near (pointer)	16 bits	not applicable
far (pointer)	32 bits	not applicable

8

8

Classes of Data (I)

- Variables
  - Must be *declared* before they are used
  - Declaration consists of a type name followed by a list of one or more variables separated by commas

```
char    cherry, apricot;
int     mint = 7;
float   swim;
```

  - Names must obey certain rules:
    - Must begin with a letter or underscore
    - May be a combination of letters, digits and underscores
    - Whitespace characters are not allowed within a name
    - Usually written in *lowercase* letters
    - Not more than 31 significant characters
    - Must not be keywords
  - A variable name is its **identifier**

9

9

Classes of Data (II)

- Constants
  - Their values do **not** change during program execution
  - Must be declared before use
  - Declaration looks as follows:

```
#define LUN     1275*37
#define         RIS     0xD4
#define BO      037
#define PI      3.1415
#define CR      '\n'
```

  - Names of constants must obey almost the same rules as those of variables, except:
    - Usually written in *uppercase* letters
  - A constant name is its **identifier**

Note:

#define is a preprocessor directive

10

10

Operators (I)

- An operator is a symbol that causes specific mathematical or logical manipulations to be performed
- There are a number of arithmetic operators:
  - binary operators
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
    - Division (/)
    - Remainder (%) etc
  - unary operators
    - Unary plus (+)
    - Unary minus (-)
- Binary operators require **two** operands
- Unary operators require **one** operand

11

11

Operators (II)

Examples

12 + 9 = 21
12 – 9 = 3
12 * 9 = 108
12 / 9 = 1
12 % 9 = 3

12. + 9. = 21.
12. – 9. = 3.
12. * 9. = 108.
12. / 9. = 1.33

Precedence of arithmetic operators

Operator	*Type*	*Associativity*
+ -	Unary	Right to left
* / %	Binary	Left to right
+ -	Binary	Left to right

12

12

Expressions

- A combination of *constants* and *variables* together with the *operators* is referred to as an **expression**
- Constants and variables by themselves are also expressions
- An expression that involves only constants is called a **constant expression**
- Every expression has a value
- Evaluation of an expression is performed in accordance with the *precedence* and *parenthesis* rule

13

13

Examples (I)

Expression	Equivalent Expression	Value
2 – 3 + 4		
2 * 3 – 4		
2 – 3 / 4		
2 + 3 % 4		
2 * 3 % 4		
2 / 3 * 4		
2 % 3 / 4		
-2 + 3		
2 * -3		
-2 * -3		

14

14

Correct answers (I)

Expression	Equivalent Expression	Value
2 – 3 + 4	(2 – 3) + 4	3
2 * 3 – 4	(2 * 3) – 4	2
2 – 3 / 4	2 – (3 / 4)	2
2 + 3 % 4	2 + (3 % 4)	5
2 * 3 % 4	(2 * 3) % 4	2
2 / 3 * 4	(2 / 3) * 4	0
2 % 3 / 4	(2 % 3) / 4	0
-2 + 3	(- 2) + 3	1
2 * -3	2 * (- 3)	-6
-2 * -3	(- 2) * (- 3)	6

15

Assignments

- An **assignment expression** is of the form:

  variable = expression
- An assignment expression when followed by a semicolon becomes an assignment statement:

  variable = expression;
- Statements

  x = y;

  and

  y = x;

  produce **very** different results.
- The precedence of the assignment operator (=) is lower than that of the arithmetic operators, so

  sum = sum + item;

  is equivalent to

  sum = (sum + item);

16

Increment & Decrement

- Increment operator (+ +) is a unary one. It increases the value of a variable by 1
- Decrement operator (– –) is also a unary one. It decreases the value of a variable by 1
- These operators can be used both as *prefix*, where the operator occurs *before* the operand, and *postfix*, where the operator occurs *after* the operand

++variable

variable++

- -variable

variable- -

- In the *prefix* form the value is incremented or decremented by 1 *before* it is used; in the postfix form – *after* that

Examples (II)

Assignment	Before values	After values
k = i++;	i = 1	
k = ++i;	i = 1	
k = i--;	i = 1	
k = --i;	i = 1	
k = 5 - i++;	i = 1	
k = 5 - ++i;	i = 1	
k = 5 + i--;	i = 1	
k = 5 + --i;	i = 1	
k = i++ + --j;	i = 1, j = 5	
k = ++i - j--;	i = 1, j = 5	

Correct answers (II)

Assignment	Before values	After values
k = i++;	i = 1	k = 1, i = 2
k = ++i;	i = 1	k = 2, i = 2
k = i--;	i = 1	k = 1, i = 0
k = --i;	i = 1	k = 0, i = 0
k = 5 - i++;	i = 1	k = 5 - 1 = 4, i = 2
k = 5 - ++i;	i = 1	k = 5 - 2 = 3, i = 2
k = 5 + i--;	i = 1	k = 5 + 1 = 6, i = 0
k = 5 + --i;	i = 1	k = 5 + 0 = 5, i = 0
k = i++ + --j;	i = 1, j = 5	k = 5, i = 2, j = 4
k = ++i - j--;	i = 1, j = 5	k = -3, i = 2, j = 4

19

19

Compound assignments

- There are 10 compound assign operators in C language:

+=	-=	*=	/=	%=
<<=	>>=	&=	|=	^=

- They are used for the compression of assignment statements
- The following statements are equivalent:

   variable ***op=*** *expression;*

  and

   variable ***=*** *variable* ***op*** *expression;*

  where **op=** denotes a compound assignment operator

20

20

Examples & Answers (III)

int i = 2, j = 1, k = 3;		
Assignment	**Equivalent statement**	**After values**
k -= i;		
k += i - 1;		
k /= i + 1;		
k *= i - j;		
k %= i * j;		

Assignment	**Equivalent statement**	**After values**
k -= i;	k = k - i;	k = 1
k += i - 1;	k = k + (i - 1);	k = 4
k /= i + 1;	k = k / (i + 1);	k = 1
k *= i - j;	k = k * (i - j);	k = 3
k %= i * j;	k = k % (i * j);	k = 1

21

21

Nested Assignments

- Multiple assignments in one statement are called *nested*.
- Assignment operators are right-associative; the following statement:

  i = j = k = 0;

  is interpreted as

  i = (j = (k = 0));
- Similarly, the statement

  i += j = k;

  is interpreted as

  i += (j = k);

  and the statement

  i = j += k;

  as

  i = (j += k);

22

22

Function *printf()*

- A call to *printf* is of the form

  printf(control_string, arg1, arg2, …);
- The *control string* governs the conversion, formatting, and printing of the arguments of *printf*. So, the statement

  printf("Just a prompt for the user");

  will produce the following result

  Just a prompt for the user
- It may consist of ordinary characters that are reproduced unchanged on the standard output (usually, monitor)
- The *control string* may also include *conversion specifications* that control the conversion of the arguments arg1, arg2, etc., before they are printed

printf(): Conversion specifications

- Each conversion specification consists of the character *%* followed by optional *minimum field width specification* and *precision specifications* as well as a required *conversion control character*

Control character	Effect
d, i	Argument of *int* type is converted into decimal notation *[-]ddd*
f	*float* or *double* type → *[-]ddd.dddd*
e	*float* or *double* type → *[-]d.dddddd**e**[±]dd*
c	Argument is taken to be a single character
s	Argument is taken to be a string

printf(): Examples

int i = 5; char cr = '$'; float j = 314.15;	
Statement	**Result**
printf("%5i", i);	_ _ _ _ 5
printf("%6.1f", j);	_ 3 1 4 . 1
printf("%f", j);	3 1 4 . 1 4 9 9 9 4
printf("%.1e", j);	3 . 1 e + 0 2
printf("%10.2e", j);	_ _ 3 . 1 4 e + 0 2
printf("%c", cr);	$

25

25

Function *scanf()*

- A call to *scanf* is of the form

  scanf(control_string, arg1, arg2, ...);
- The *control string* governs the conversion, formatting, and printing of the arguments of *scanf*
- Each of the arguments *arg1*, *arg2*, etc., must be a **pointer** to the variable which the result is stored. So, the statement

  scanf("%d", &var);

  is a correct one, while

  scanf("%d", var);

  is not correct

26

26

scanf(): Control string

- The *control string* contains *conversion specifications* according to which the characters from the standard input are interpreted and the results are assigned to the successive arguments *arg1*, *arg2*, etc.
- The *scanf()* function
  - reads one data item from the input, skipping whitespaces (and newlines) to find the next data item, and
  - returns as *function value* the total number of arguments successfully read; it returns *EOF* when the end of input is reached
- Each conversion specification consists of the character *%* followed by a *conversion control character*
- Whitespaces separating conversion specifications are ignored

27

27

scanf(): Conversion specifications

Control character	**Effect**
d, i	A decimal value is expected in the input. The corresponding argument should be a pointer to an *int*
f, e	A floating-point number is expected in the input. The corresponding argument should be a pointer to a *float*. The input could be in standard decimal form or in the exponential form
c	A single character is expected in the input. The corresponding argument should be a pointer to a *char*. Only in this case, the normal skip over whitespaces in input is suppressed

28

28

scanf(): Examples

Given the declarations

int i;
float f1, f2;
char c1, c2;

and the input data

10 1.0e1 10.0pc

the statement

scanf("%d %f %e %c %c", &i, &f1, &f2, &c1, &c2);

results in

i = 10 *c1 = p*
f1 = 10.000000 *c2 = c*
f2 = 10.000000

29

29

Sixth Example

```
/* Ch_02_6.C -- Chapter 02. Sixth illustration program */
/* This program calculates the sum of digits for a 3-digit number */

#include <stdio.h>

int main()
{
        int num;
        int sum = 0;      /* Initial value for sum */

        printf("\n\nPlease, enter a number: ");       /* Entering the number */
        scanf("%3i", &num);

        sum += num % 10;        /* Add the lowest digit to the sum */
        num /= 10;              /* Leave a 2-digit number */
        sum = sum + num % 10 + num / 10;    /* Add these two digits to the sum */

        printf("\nThe sum of its digits is: %3d", sum);    /* Printing the result */

        return 0;
}
```

30

30

Seventh Example

```
/* Ch_02_7.C -- Chapter 02. Seventh illustration program */
/* This program is convert the presentation form of a value: */
/* A decimal value is printed out in octal and hexadecimal forms */

#include <stdio.h>

int main()
{
        int num;

        /* Enter a decimal value */
        printf("\n\nPlease, enter a decimal value: ");
        scanf("%i", &num);

        /* Printing ... */
        printf("\nThis value in the decimal form:\t%7i", num);
        printf("\nThe same value in octal form:\t%7o", num);
        printf("\nThe same value in xehadecimal form: %3X", num);

        return 0;
}
```


31

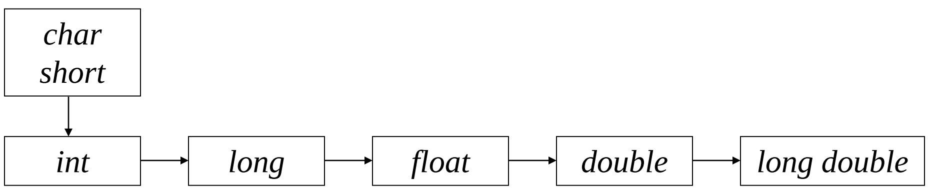
Automatic Type Conversions

- An expression in C may contain variables and constants of different types
- There are rules for evaluating such expressions
- ANSI C performs arithmetic operations with just 6 data types:
  - *int,*
  - *unsigned int,*
  - *long int*
  - *float,*
  - *double,*
  - *long double*
- Automatic *Unary* Conversions: any operand of the type *char* or *short* is implicitly converted to *int* **before** the operation

32

Automatic Binary Conversions

- Apply to **both** operands of the binary operators
- Carried out **after** automatic unary conversions
- General Idea: the "lower"-type operand *is promoted* to the "higher" type before the operation proceeds
- The result is of the "higher" type
- If there's no "unsigned" operands, the conversion rules are summarized in the diagram as follows:



33

33

Rules for Binary Conversions (I)

- If one operand is *long double* and the other is not, the latter is converted to *long double*, and the result is *long double*;
- otherwise, if one operand is *double* and the other is not, the latter is converted to *double*, and the result is *double*;
- otherwise, if one operand is *float* and the other is not, the latter is converted to *float*, and the result is *float*;
- otherwise, if one operand is *unsigned long int* and the other is not, the latter is converted to *unsigned long int*, and the result is *unsigned long int*;

34

34

Rules for Binary Conversions (II)

- otherwise, if one operand is *long int* and the other is *unsigned int*, then
  - if a *long int* can represent all values of an *unsigned int*, the latter is converted to *long int*, and the result is *long int*;
  - if not, both are converted to *unsigned long int*, and the result is *unsigned long int*;
- otherwise, if one operand is *long int* and the other is not, the latter is converted to *long int*, and the result is *long int*;
- otherwise, if one operand is *unsigned int* and the other is not, the latter is converted to *unsigned int*, and the result is *unsigned int*;
- otherwise, both operands must be *int*, and the result is *int*

35

Example

- Let's evaluate the following expression:

  *(c / u - l) + s * f*

  where the types of *c*, *u*, *l*, *s* and *f* are *char*, *unsigned int*, *long*, *short* and *float*
- The table below summarizes all the automatic conversions that take place during the evaluation:

Expression	Conversion	Operand1	Operand2	Result
c	unary	*char*		*int*
c / u	binary	*int*	*unsigned int*	*unsigned int*
c / u - l	binary	*unsigned int*	*long int*	*long int*
s	unary	*short int*		*int*
*s * f*	binary	*int*	*float*	*float*
*(c/u-l)+s*f*	binary	*long int*	*float*	*float*

36

Explicit Type Conversion

- Necessary to convert the type of an operand to a desirable one which is different from the result of automatic conversion
- Performed by a special construct called *cast*. The general form of a cast is

  (cast-type) expression
- Example:

  (int) 12.8 results in *12*

  which is an integer value
- A cast is a unary operator, so

  *(int) 12.8 * 3.1* results in *12 * 3.1 = 37.2*

  *(int) (12.8 * 3.1)* results in *(int) 39.68 = 39*

37

37

Type Conversion in Assignments

- Occurs when the type of a resultant variable is different of that of an assignment expression
- Automatically, the value of the expression on the right side of the assignment operator is converted to the type of the variable on its left side
- The conversion of a *lower* order type (say, *int*) to a *higher* order (e.g. float) only changes the form, in which the value in presented
- The conversion of a *higher* order type to a *lower* order may cause truncation and loss of information

38

38

Example I

Determine the value of the following C expression:

*- (2 * (- 3 / (double) (4 % 10))) - (- 6 + 4)*

1. Parenthesis rule is applied first, and the result is

   *- (2 * (- 3 / (double) 4)) - (- 6 + 4)*

2. The cast forces conversion of 4 into double type, so the division is no longer an integer division, and the result is

   *- (2 * -0.75) - (- 6 + 4)*

3. Further evaluation gives

   - -1.5 - -2

   1.5 + 2 = 3.5

39

Example II

Determine the values of *x*, *y* and *z* in the following fragment in C:

int x, y, z;

float f;

x = 5;

x /= y = z = 1 + 1.5;

Arithmetic operator has higher precedence than assignments, so the equivalent expressions are as follows:

x /= (y = (z = (1 + 1.5)))

x /= (y = (z = 2.5)) → *z = 2*

x /= (y = 2) → *y = 2.0*

x /= 2.0 → *x = 2*

40