Introduction to Computer Programming

Instructor: HOU, Fen

2025

What is Language Syntax?

- C, as a language, has a set of rules for putting together words and punctuations to make correct programs.
- These rules are the *syntax* of the language.
- For a program to be compiled successfully, it must be *syntactically correct*.
- C compilers will fail to compile a syntactically incorrect program, no matter how trivial the error is.

A Program with Typos

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int side, perimeter, area,
6
7    side = 3;
8    perimeter = 4 * side;
9    area = side * side;
10
11   printf("Side : %d\n", side);
12   printf("Perimeter : %d\n", perimeter);
13   printf("Area : %d\n", area);
14
15   return(0);
16 }
```

During Compilation

```
square.c: In function `main':
square.c:7: redeclaration of 'side'
square.c:5: 'side' previously declared here
```

3.1 Lexical Elements

Tokens in the C Language

- The compiler views a program as a series of *tokens*.
- Identifying and categorizing tokens in a program allow the compiler to analyze the syntax of the program.
- In ANSI C, there are six kinds of tokens.

Program basic_1.c

```
1  /* Read in two integers and print their sum. */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7    int a, b, sum;
8    printf("Input two integers: ");
9    scanf("%d%d", &a, &b);
10   sum = a + b;
11   printf("%d + %d = %d\n", a, b, sum);
12   return (0);
13 }
```

Token Type	Examples
Keyword	int, return, void
Identifier	main, a, b, sum, printf, scanf
Constant	0

Program basic_1.c

```
1  /* Read in two integers and print their sum. */
2
3  #include <stdio.h>
4
5  int main(void)
6  {
7    int a, b, sum;
8    printf("Input two integers: ");
9    scanf("%d%d", &a, &b);
10   sum = a + b;
11   printf("%d + %d = %d\n", a, b, sum);
12   return (0);
13 }
```

Token Type	Examples
String constant	"Input two integers: "
Operator	(), =, +, &
Punctuator	, ; { }

3.1.1 What is a keyword?

- Keywords have a strict meaning.
- Cannot be redefined or used in other ways, i.e. reserved.
- Compared to other major high level programming languages, C has a small number of keywords – 32 only

ANSI C Keywords

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

3.1.2 Identifiers

- Identiers are used to give unique names to various objects (such as a variable) in a program.
- An identifier is a token that is composed of a sequence of letters, digits, and the underscore (_).
- The first letter of an identifier must be a letter or an underscore.

3.1.2 Identifiers (con't)

- Example identifiers:
  - **`inches`**, **`radius`**, **`id`**, **`student_name`**, **`main`**, **`printf`**, **`scanf`**
- Invalid identifiers:
  - #num, 101_south, -plus
- Lower- and upper-case letters are treated as distinct, i.e. case-sensitive:
  - **`my_name`** is different from **`My_name`**.

3.1.2 Identifiers (con't)

- Choose identiers that have *mnemonic* significance, for example,
  - `tax = price * tax_rate;`
- Identiers starting with an underscore (_) are not recommended, for example, **`_job`**.
- Identiers such as **`scanf`** and **`printf`** have already been defined in the *standard library* and should not be redefined.
- **`main`** is a special identifier to signify where program execution should start.

Exercise

Which of the following are not valid identifiers and why?

- Inches
- radius
- id4
- #num
- -plus
- _overpay
- student_name
- main
- printf
- scanf
- my-name
- my_name
- My_name
- _south

Exercise

Which of the following are not valid identifiers and why?

- `3id`
- `o_no_o_no`
- `00_go`
- `start*it`
- `__yes`
- `1_i_am`
- `one_i_aren't`
- `me_to-2`
- `main`
- `xYshouldI`
- `int`

3.1.3 Constants

Categories of Constants

- Decimal integer constants:

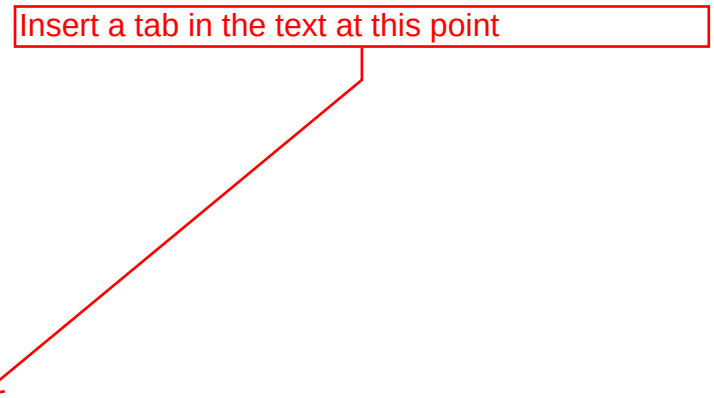
  `0` `17`

- Floating constants:

  `1.0` `3.14`

- Character constants:

  `'a'` `'b'` `'\n'` `'\t'`



- String constants (next page)
- Enumeration constants (discuss later)

3.1.4 String Constants

What is a String?

- A string constant is a sequence of characters enclosed in a pair of double quote ", for example,

```
"C is easy to learn."
```

- String constants are different from character constants – `"a"` is different from `'a'`.

Program basic_2.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    printf ("This is a string.\n");
6    printf ("\n");
7    printf ("a string with \"double quotes\".");
8    printf ("\n");
9    printf ("a single backslash \\ is in this string.");
10   printf ("\n");
11   printf ("a = b + c;");
12   printf ("\n");
13   return (0);
14 }
```

If a double quote mark itself is to occur in a string constant, it must be preceded by a backslash character

```
This is a string.

a string with "double quotes".
a single backslash \ is in this string.
a = b + c;
```

3.1.5 Operators and Punctuators

Categories of Operators

- *Arithmetic* operators:

  `+` `-` `*` `/`

- *Binary* operators:

  ```
  a + b
  17 * 3
  x = y
  ```

- *Unary* operators:

  ```
  x++
  &sum
  ```

3.1.5 Operators and Punctuators (con't)

- Modulus %

  a%b The value of a modulus b is obtained by taking the remainder after dividing a by b.

For example

7%2 The value is 1

7%3 The value is 1

- Exercise

  5%2

  10%3

  4%2

3.1.5 Operators and Punctuators (con't)

- Operators can be used to *delineate* identifiers, for example,

```
sum=a+b;
```

- But we typically put white space around binary operators to improve readability.

```
sum = a + b;
```

3.1.5 Operators and Punctuators (con't)

Punctuators

- Examples of punctuators are:
  - parentheses ( )
  - braces { }
  - commas ,
  - semicolons ;

3.2.1 Operator Precedence & Associativity

- Operators have *precedence* and *associativity* that determine precisely how expressions are evaluated.

- `1 + 2 * 3`
  - The operator * has *higher precedence* than +, causing the multiplication to be performed first.

- `(1 + 2) * 3`
  - Parenthesis can be used to change the order in which operations are performed.

- `1 + 2 - 3 + 4 - 5`

  is equivalent to

  `(((1 + 2) - 3) + 4) – 5`
  - Because + and - have the same precedence, the expression is evaluated using the associativity rule "*left to right*."

2.1 Operator Precedence & Associativity

Operator	Associativity	Precedence
`( )` `++` (postfix) `--` (postfix)	left to right	Highest
`+` (unary) `-` (unary) `++` (prefix) `--` (prefix)	right to left	
`*` `/` `%`	left to right	
`+` `-`	left to right	
`=` `+=` `-=` `*=` `/=` etc.	right to left	Lowest

- Operators on the same level have the same precedence.
- When evaluating expressions involving operators on the same level of precedence, refer to the associativity.
- `- a * b - c` is equivalent to `((- a) * b) - c`

The increment operator **++**

- The ++ operator can be applied to variables, but not to constants or ordinary expressions, for example,

```
i++ /* valid */
777++ /* invalid */
```

- The ++ operator increases the value of the variable that it applies to by one. For example,

```
value = 10;
value++;
printf ("%d\n", value);  /* output 11 */
```

PreFix and PostFix Forms

- The ++ operator can occur in either *prefix* or *postfix* position, with different results.
  - Each of the expression `++i` and `i++` has a value.
  - The prefix expression `++i` causes the stored value of `i` to be incremented (by one) first, then the expression evaluates to the newly stored value of `i`.
  - The postfix expression `i++` evaluates to the current value of `i` first, then the stored value of `i` is incremented (by one).

PreFix and PostFix Forms (con't)

- For example,

```
int a, b, c;
c = 0;
a = ++c;
b = c++;
printf ("%d %d %d\n", a, b, ++c);
```

PreFix and PostFix Forms (con't)

- For example,

```
int a, b, c;
c = 0;
a = ++c;
b = c++;
printf ("%d %d %d\n", a, b, ++c);
```

  - This program fragment causes **`1 1 3`** to be printed.

The Decrement Operator

- The decrement operator **`--`**.
- The **`--`** operator is similar to **`++`**, except that the associated variable is decremented by one.

Exercise

Evaluate the expressions in the table below. (Refer to the *Operator Precedence and Associativity Table*)

Declarations and initializations		
`int a=1, b=2, c=3, d=4;`		
Expression	**Equivalent expression**	**Value**
`a * b / c`	`(a * b) / c`	`0`
`a * b % c + 1`		
`++ a * b - c --`		
`7 - - b * ++ d`		

2.3 Miscellaneous Assignment Operators

The Assignment Expression

```
variable = right side expression
```

- Low precedence, *right-to-left* associativity
- The value of the right side expression is assigned to variable.
- AND the value becomes the value of this assignment expression as a whole. (Refer to the example on next slide.)

Program basic_3.c

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5     int a, b, sum;
6
7     a = 3;
8     b = 4;
9     printf ("Value = %d\n", sum = a + b);
10    printf ("Sum = %d\n", sum);
11    return (0);
12  }
```

```
Value = ?
Sum = ?
```

Program basic_3.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b, sum;
6
7    a = 3;
8    b = 4;
9    printf ("Value = %d\n", sum = a + b);
10   printf ("Sum = %d\n", sum);
11   return (0);
12 }
```

```
Value = 7
Sum = 7
```

Multiple Assignment

- *Multiple assignment*, for example,

```
a = b = c = 0;
```

  - First `c` is assigned the value `0`, and the expression `c = 0` has value `0`.
  - Then `b` is assigned the value `0`, and the expression `b = (c = 0)` have value `0`.
  - Finally `a` is assigned the value `0`, and the expression `a = (b = (c = 0))` has value `0`.

Assignment Operators – Short Form

- `k = k + 2` is equivalent to `k += 2`.

- The semantics of

  ```
  variable = variable op (expression)
  ```

  is equivalent to

  ```
  variable op= expression
  ```

Assignment operators						
`=`	`+=`	`-=`	`*=`	`/=`	`%=`	etc.

Be Careful!

- Note that the following statement

```
j *= k + 3;
```

is equivalent to

```
j = j * (k + 3)
```

or

```
j = j * k + 3;
```

Be Careful!

- Note that the following statement

```
j *= k + 3;
```

is equivalent to

```
j = j * (k + 3);
```

rather than

```
j = j * k + 3;
```

The Assignment Expression

Exercise

Evaluate the expressions in the table below.

Declarations and initializations			
int i=1, j=2, k=3, m=4;			
Expression	Equivalent expression	Equivalent expression	Value
i += j+k	i += (j+k)	i= (i+(j+k))	6
j *= k = m+5			

The Assignment Expression

Exercise

Evaluate the expressions in the table below.

Declarations and initializations			
int i=1, j=2, k=3, m=4;			
Expression	Equivalent expression	Equivalent expression	Value
i += j+k	i += (j+k)	i= (i+(j+k))	6
j *= k = m+5	j *= (k = (m+5))	j = (j* (k = (m+5)))	18

3. Symbolic Constants

Program basic_4.c

```
1  #include <stdio.h>
2  #define PI 3.14
3
4  int main(void)
5  {
6    float radius;
7
8    radius=1.5;
9    printf ("Radius=%.1f\n", radius);
10   printf ("Area =%.2f\n", radius * radius * PI);
11   return (0);
12 }
```

```
Radius=1.5
Area =7.07
```

PI is called a *Symbolic Constant*.

The preprocessor changes all occurrences of the identifier **PI** to **3.14**.

Thus, line 10 becomes

```
printf ("Area =%.2f\n",
          radius * radius * 3.14);
```

4. Comments /* … */

- Comments are used as a documentation aid.
- The compiler ignores the comments.
- The aim of documentation is to explain clearly how the program works and how it is to be used.
- Comments should be written simultaneously when you write the program.

Comment Examples

```
/* a comment */

/*** another comment ***/

/*****/

/*
 * A comment can be written in this fashion
 * to set it off from the surrounding code.
 */

/*****************************
 * If you wish, you can      *
 * put comments in a box.    *
 *****************************/

/*
    /* This is not allowed */
*/
```

The comment can span multiple lines.

Comments don't nest. Placing a comment within a comment is not allowed.

The use of `printf()`

- **`printf()`** is used for printing formatted output.
- Two arguments:
  - format string
  - a list of arguments.
- The format string defines how many arguments are needed and how they are to be formatted.
- Each argument is specified by a *format specifier* – a **%** and a *conversion character*, such as **%d**.
- The data type of the arguments should match the corresponding specifier in the format string.

```
printf ("The data: %s %d %f %c\n", "one", 2, 3.33, 'G');
```

```
The data: one 2 3.330000 G
```

`printf()` conversion

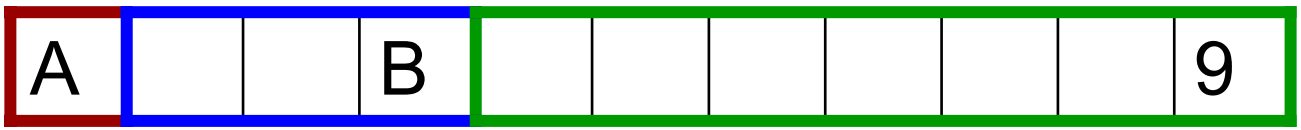
Character	How the corresponding argument is printed
c	as a character
d	as a decimal integer
e	as a floating-point number in scientific notation
f	as a floating-point number
g	in the e-format or f-format, whichever is shorter
s	as a string

Controlling Field-Widths in `printf()`

- When an argument is printed, the place where it is printed is called its *field*.
- The number of characters in a field is called the *field width*.
- The field width can be specified as an integer between the % and the conversion character.
- Refer to the example below.

```
printf ("%c%3c%7d\n", 'A', 'B', 9);
```

```
A  B      9
```

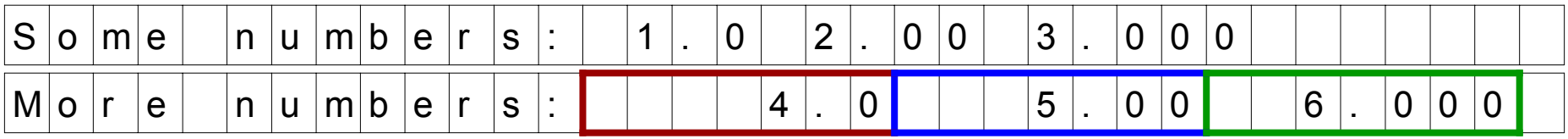


Controlling Precisions in `printf()` – `%f`

- For floating values, we can control the *precision*, as well as the field width.
- The precision is the number of digits to the right of the decimal point.
- In a format `%m.nf`, the field width is specified by `m`, and the precision is specified by `n`.

```
printf("Some numbers: %.1f %.2f %.3f\n", 1.0, 2.0, 3.0);
printf("More numbers:%7.1f%7.2f%7.3f\n", 4.0, 5.0, 6.0);
```

```
Some numbers: 1.0 2.00 3.000
More numbers:    4.0   5.00  6.000
```



The Use of Expressions in `printf()`

Program basic_5.c

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5     float x, y;
6
7     x = 1.0;
8     y = 2.0;
9     printf("The sum of %.1f and %.1f is %.3f!\n",
10           x, y, x + y);
11    return (0);
12  }
```

An expression can also be used as an argument in **`printf()`**.

The expression is first evaluated and the resulting value is then matched with the corresponding format specifier in the format string.

```
The sum of 1.0 and 2.0 is 3.000!
```

6. Variable Initialization

Program basic_6.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    float radius;
6
7    radius=1.5;
8    printf ("Radius=%.1f\n", radius);
9    printf ("Area =%.2f\n", radius*radius*3.14);
10   return (0);
11 }
12
```

```
Radius=1.5
Area =7.07
```

Compare this program to the program on the previous slide.

When a variable is declared, it may be initialized to a particular value.

For example,

```
float radius=1.5;
```

is equivalent to

```
float radius;
radius=1.5;
```

Program basic_7.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    float radius=1.5;
6
7    printf ("Radius=%.1f\n", radius);
8    printf ("Area =%.2f\n", radius*radius*3.14);
9    return (0);
10 }
11
12
```

```
Radius=1.5
Area =7.07
```

7. The use of **scanf()**

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    float radius=1.5;
6
7    printf ("Radius=%.1f\n", radius);
8    printf ("Area =%.2f\n", radius*radius*3.14);
9    return (0);
10 }
11
12
```

What if the radius is …… ?

7. The use of **`scanf()`**

- Analogous to **`printf()`**, but is used for input rather than output.
- The format string defines how many arguments are needed and how the data in the *input stream* are to be interpreted.
- Corresponding to each format specifier in the format string is the *memory address* of the variable going to store the input data.
- Example below,

```
scanf ("%d", &num);
```

- The format specifier **`%d`** causes input characters to be interpreted as a decimal integer.
- The value of the decimal integer is stored in the variable **`num`**.

`scanf()` conversion

Character	How characters in the input stream are converted
c	character
d	decimal integer
f	floating-point number (float)
lf	floating-point number (double)
Lf	floating-point number (long double)
s	string

Program basic_8.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    float radius;
6
7    printf ("Input radius? ");
8    scanf ("%f", &radius);
9    printf ("Area = %.2f\n", radius*radius*3.14);
10   return (0);
11 }
```

```
Input radius? 1.5
Area = 7.07
```

```
Input radius? 3.4
Area = 30.30
```

Exercise

- Below is part of a program that begins by asking the user to input three integers. Complete the program so that when the user executes it and types in 2, 3, and 7, the screen looks like:

```
Input three integers:
1st? 2
2nd? 3
3rd? 7
Sum of the integers = 12
```

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int a, b, c;
6
7    printf ("Input three integers: \n");
8    printf ("1st? ");
9    scanf("%d", &a);
10   ...
11
12   return (0);
13 }
```