

C Language Basics

Outline

1. First C program & Language Syntax
2. Variables and Assignment Operators
3. Identifiers, Reserved Words, Predefined Identifiers
4. Data Types, Numeric Constants, String Literals
5. Console Input/Output Using `scanf ()` and `printf ()`

1. Overview of C Language

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, world!\n");
6     printf("Hello, universe!\n");
7
8     return 0;
9 }
```

The start
of block

The end
of block

Basic structure of a C program:

```
#include <stdio.h>
```

Standard
libraries
to load

```
int main(void)
{
```

Entrance
point of C
program

```
    statement_1;
    statement_2;
```

```
    ...
```

```
    return 0;
```

This program
is done!

```
}
```

Statements are executed sequentially.

```
Hello, world!
Hello, universe!
```

1.1. Language Syntax

- Every programming language has its own *syntax* (defined by a set of rules)
 - Like grammar in English language
- Program source code containing syntax errors cannot be compiled into an executable program
 - Remember: *computers cannot follow malformed instructions!*

```
main.c:5:28: error: expected ';' after expression
printf("Hello, world!\n")
                        ^
                        ;
1 error generated.
compiler exit status 1
```

You may see something like this when a syntax error is found when compiling. Read carefully for hints to fix your code!

```
1 #include <stdio.h>
2
3 int main(void) {
4     Printf("Hello, world!\n");
5     Printf("Hello, universe!\n");
6     return 0;
7 }
8
9
```

Can you spot
the error(s)?

Some Basic Rules

- C language is *case sensitive*.
- During syntax checking, compilers usually consider *whitespace* characters (*space*, *newline* [Enter] and *tab*) flexibly.
- Proper use of spaces makes program easy to read.
 - Spacing in front is called indentation as a good programming style

1.2. Comments

```
1  /*  
2     This is a comment  
3  */  
4  #include <stdio.h>  
5  
6  int main(void) {  
7      printf("Hello, world!\n"); // This is also a  
8  comment  
9      return 0;  
}
```

- Comments are used for documentation aid; they are ignored by compiler.
- Two commenting styles
 - Text starting with `/*` and ends at the next `*/`
 - Can span multiple lines
 - Text begins with `//` and extend to the end of the line

1.3. The `printf()` function

- You may notice that the most interesting statement in our very generic example is

```
printf("Hello, world!\n");
```

- This is called the `printf` function; the function allows us to output a piece of text to the screen
- Formally, the piece of text `"Hello, world!\n"` is called a string constant

1.4. String Constant

- A string constant must be enclosed by a pair of double-quote characters (")
 - e.g.: "0123456789", "\n", "Hello World!"
- Without the double-quote characters ("), the C compiler will not treat the text as a string constant
- In other words, the following code will cause the program to fail to compile:

```
printf(Hello, world!\n);
```



1.4. String Constant

- You are probably thinking: What if I want to print a double quotation mark? What if I want to print the "Enter" key for a new line of text?
- These characters need to be expressed as *escaped sequences* in a string constant.

e.g.,

Escaped Sequence	Character
<code>\\</code>	Backslash (<code>\</code>)
<code>\"</code>	Double quote (<code>"</code>)
<code>\n</code>	Newline
<code>\t</code>	Tab


1.4. String Constant

- e.g.: This string constant
`"\\ is \"backslash\" and / is \"slash\""`

represents the following string (and thus output on screen)

`\ is "backslash" and / is "slash"`

- A string constant cannot span multiple lines.

`"The first line.
The second line."` 

`"The first line.\nThe second line."`



2. Variables and Assignment Operators

Key concepts

- What is a variable?
- How to use a variable in a program?

2. Variables

- Variables are used to store data in a program.
- A variable has a name and a type.
 - The type determines what kind of values the variable can hold.

```

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 }

```

Declaring three variables:
`side`, `perimeter`, and
`area`.

The *type* of the three variables
 is `int`, indicating that the
 variables are for storing
 integers.

```

Side : 3
Perimeter : 12
Area : 9

```

2.1. Declaring Variables

- Variables must be declared first before they can be used in the program to store data.

Syntax

```
type1 var1;
```

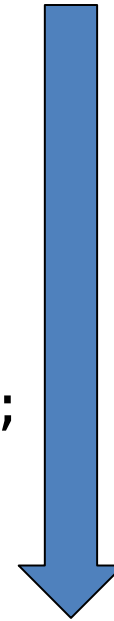
Declaring a single variable of type *type1*

```
type2 var1, var2, ..., varN;
```

Declaring multiple variables of *type2*

```
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 }
```

When the program runs, the statements are executed sequentially one by one.

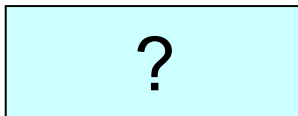


```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int side, perimeter, area;
6
7     side = 3;
8     perimeter = 4 * side;
9     area = side * side;
10
11     printf("Side : %d\n", side);
12     printf("Perimeter : %d\n", perimeter);
13     printf("Area : %d\n", area);
14
15     return 0;
16 }
```

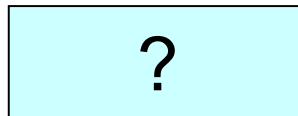
Immediately after the variables are being declared, their values are undefined (we do not know what their values are).

We say that these variables are *uninitialized*.

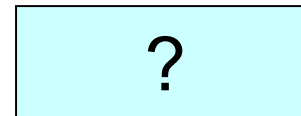
side



perimeter



area




```
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 }
```

Assigning 3 to `side`.

side

3

perimeter

?

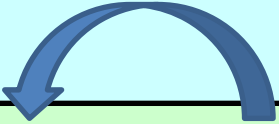
area

?

2.2. Assigning Value to a Variable

- We use an *assignment operator* (=) to copy/assign a value to a variable.

Syntax



```
variable = expression;
```

Copy the value of *expression* to *variable*

- An *expression* is made up of values and operators, and can be evaluated to a value in the program. For examples,

100

someVariable

200 + var1 * var2

```
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 }
```

Assigning the value of $4 * \text{side}$ to `perimeter`.

- The expression $4 * \text{side}$ is evaluated first.
- The result, **12**, is then assigned to `perimeter`.

side

3

perimeter

12

area

?

```
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 }
```

Assigning the value of `side * side` to `area`.

side	perimeter	area
3	12	9

2.3. Outputting a variable to screen

```
printf("Side : %d\n", side);
```

- "Side : %d\n"

- Text to print, called the *format string*
- The *format specifier* %d specifies that the value of the corresponding expression is to be printed in the format of a *decimal* integer.

- side

- The expression whose value is to be supplied to the format string.

- Compare to:

```
printf("Hello world!\n");
```

```
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 }
```

`printf()` outputs the given text (string) but with the `%d` replaced by the value of `side`.

Side : 3

3. Naming Variables

Key concepts

- How to name variables?
- What is a *reserved word*?

3.1. Identifier

- An *identifier* is a name used to identify variables, functions, etc. in a program.
- An identifier must satisfy the following rules. It:
 1. Must contain only
 - Alphabets ('A' – 'Z', 'a' – 'z')
 - Digits ('0' – '9')
 - The underscore character ('_')
 2. Must not start with a digit
 3. Cannot be one of the reserved words

3.2. Reserved Words

- *Reserved words* or *keywords* are names that have special meaning in the C language. You will eventually learn all about them later.

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	

3.3. Predefined Identifiers

- There are also *Predefined identifiers* that have already been used as names for standard usages:
 - e.g., **main**, **printf**, **scanf**
- Avoid using them as identifiers in your program

Naming Variables – Challenge

Which of the following are *valid* identifiers?

1. \$abc
2. _1_abc_1_
3. 1_1
4. Domain-name
5. URL
6. int
7. main
8. Int
9. 32bits
10. c

11. printf
12. engg1110
13. engg_1110
14. _
15. A100xC200
16. tab
17. include
18. VARIABLE
19. www_yahoo_com
20. Hong Kong

3.4. Naming Conventions (Guidelines)

- Be meaningful
 - Avoid names like: a, b, c, d, a1, a2, a3, xyz
- Be consistent
 - interest_rate** (Use underscore in place of space)
 - or
 - interestRate** (*camelCase* – Mixed case starting with lower case)
- Avoid using names with all uppercase letters
 - They are usually used for named constants



Image source: http://commons.wikimedia.org/wiki/File:2011_Trampeltier_1528.JPG

Author: [J. Patrick Fischer](#); License: [CC BY-SA 3.0](#)

5. Console Input and Output

Key concepts

- How to read numbers from a user?
- How to format numbers in the output?

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int num1;
6
7      printf("Enter an integer:\n");
8      scanf("%d", &num1);
9
10     printf("num1 = %d\n", num1);
11
12     return 0;
13 }
14
```

Enter an integer:

123↵

num1 = 123

Example 5.1: Reading an integer from the user.

Example 5.1 explained

- `scanf ("%d" , &num1) ;`
 - A single `%d` in the format string tells `scanf ()` to read one integer.
 - Upon success, the input value is stored in `num1`.
 - `&` before the variable is a must. You will learn about this operator later in this course.
- Behavior of `scanf ()`
 - (Line 8) Execution is paused here while `scanf ()` waits for user input.
 - Program resumes when the user enters a value followed by pressing the "Enter" key (denoted by the symbol '`↵`' in the sample output).

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int num1, num2;
6
7     printf("Enter two integers:\n");
8     scanf("%d%d", &num1, &num2);
9
10    printf("num1 = %d, num2 = %d\n", num1, num2);
11
12    return 0;
13 }
14
```

Enter two integers:
123 456↵
num1 = 123, num2 = 456

or

Enter two integers:
123↵
456↵
num1 = 123, num2 = 456

Example 5.2: Reading two integers from the user.

Example 5.2 explained

- `scanf ("%d%d" , &num1 , &num2) ;`
 - Two `%d`'s in the format string (no space in between the format specifiers) tell `scanf ()` to read two integers.
 - Upon success, the 1st input value is stored in `num1` and the 2nd input value is stored in `num2`.
- Behavior of `scanf ()` when it expects two input values
 - The two input values are to be separated by at least one whitespace characters.
 - (Line 8) Execution is paused here while `scanf ()` waits for user input.
 - Program resumes when the user enters the 2nd value followed by pressing the "Enter" key.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double r1, r2;
6
7     printf("Enter two real numbers:\n");
8     scanf("%lf%lf", &r1, &r2);
9
10    printf("r1 = %f\n", r1);
11    printf("r2 = %f\n", r2);
12
13    return 0;
14 }
```

Variables **r1** and **r2** are of type **double** (instead of **int**) here. What does it mean?

Concerning **double**-typed values, **scanf()** uses **%lf** ('ell' f); **printf()** uses **%f**. Looking similar but *slightly different*!

```
Enter two real numbers:
123 456.125↵
r1 = 123.000000
r2 = 456.125000
```

printf(), by default, prints decimal numbers (floating point numbers) with 6 decimal places.

Example 5.3: Reading two floating point numbers from the user.

Example 5.3 Explained

- You may notice that the example involved decimal numbers
- So far we have been using integers (type `int`)
 - `0, -100, 2048, 203139, 1000000`
- If we need decimal places in our numbers, we use floating point numbers (type `double` here)
 - `0.0, -10.2, 3.1416, .244`
- A decimal point makes a big difference!
 - `10` is treated as an integer.
 - `10.0` is treated as a floating point number.
 - Integers and floating point numbers are handled differently in C (will talk about it in detail next lecture)

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double pi = 3.1415927;
6
7
8     printf("A) %f\n", pi);
9     printf("B) %.2f\n", pi);
10    printf("C) %.7f\n", pi);
11
12    return 0;
13 }
14
```

When we declare a variable, we can initialize its value.

The format specifier, `%.xf`, tells `printf()` to format the corresponding floating point number with *x* decimal places.

```
A) 3.141593
B) 3.14
C) 3.1415927
```

Example 5.4: Controlling the # of decimal places for floating point numbers.

Notes about `scanf()` and `printf()`

- `scanf()` won't work "properly" if it encounters an invalid input. For example, a user enters an alphabet when an integer is expected.
 - In this course, unless otherwise stated, you can assume the input values are always valid.
- Examples showing incorrect uses of `printf()`
 - `printf("%f", 10);` // 10 is a value of type `int`
 - `printf("%d", 10.0);` // 10.0 is a value of type `double`
 - `printf("%.2d", 10);` // .2 does not apply to integer
 - `printf("%d %d", 10);` // One argument is omitted

Summary

- **Syntax:** C, as a programming language, has rules that programmers must obey.
- **Variable:** Holds a value; has a name and a type
- **Assignment operator:** For assigning a value to a variable
- **Identifier:** Valid name for identifying things in the program
- Expressing **numeric and string constants**
- **Console Input/Output** using `scanf()` and `printf()`

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 2 Introduction to C Programming
 - Sections 2.1 – 2.7