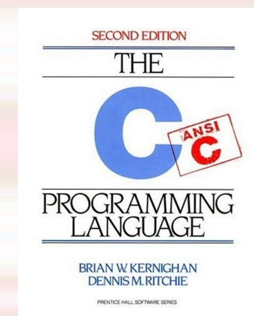
Lesson #2

Introduction to C

C

- The language is called **C** because its direct ancestor was called **B**.
- C was created around 1972 by Kernighan and Ritchie.
- In 1990, the ANSI C standard was adopted by the International Organization for Standardization (ISO).



Constants in C

- There are three basic types of constants in C.
- An **integer constant** is an integer-valued number. We will be concerned here solely with decimal constants like 0, 1, 743, 5280, 32767, or –764.
- A **floating-point constant** is a base-10 number than contains either a decimal point or an exponent or both like 0., 1., 0.2, 50.0, 12.3, –12.667, 2E-8, or 0.006e-3.
- A character constant is a single character enclosed in apostrophes like 'a', 'x', '9', or '?'.

Preprocessor Directives

- **Preprocessor:** A system program that modifies the C program prior to compilation.
- **Preprocessor directive:** An instruction to the preprocessor. Begins with **#**.
- **Library:** A collection of functions, symbols, and values.
- Two kinds of preprocessor directives: includes and defines.

Preprocessor Directives

- **#include <stdio.h>**: **stdio.h**, which stands for *standard input/output header*, is the header in the C standard library that contains macro definitions, constants, and declarations of functions and types used for various standard input and output operations. The #include <stdio.h> directive must be included on top of every C program.

- **#define PI 3.1416**: this is a constant macro definition. It associates a name to a value for the duration of the program. In this case it associates the symbol PI to the value 3.1416. It is an optional directive.

Comments

- Comments are lines of code that are ignored by the compiler. They are placed for the programmer's benefit.

```
/* This is a comment */

/* This is another comment. It can be spread
over multiple lines */
```

Instructions

- Instructions in C are terminated by a semi-colon (;)
- Line changes and tabs are not important to the C compiler.

```
a=3; b=4; c=5;
```

are the same as:

```
a=3;
b=4;
c=5;
```

Skeleton of a Program

```
#include <stdio.h>
/* optional additional includes */
/* optional constant macros */
int
main (void)
{
    /* optional declarative statements */
    /* one or more executable statements */
    return (0);
}
```

My First Program

```
#include <stdio.h>

int
main (void)
{
	printf ("This is my first C program.\n");
	return (0);
}
```

Skeleton of a Program 2

- This skeleton is also valid. It eliminates parts that are not absolutely necessary. Do you see the differences? This version is the old non-ANSI way. It is <u>not the preferred way</u> but still much in use by many programmers.

```
#include <stdio.h>
/* optional additional includes */
/* optional constant macros */
main ( )
{
    /* one or more executable statements */
    printf ("This is my first C program\n");
}
```

Variables

- In programming, we often need to have places to store data. These receptacles are called variables. They are called that because they can change values.
- All variables must be declared at the top of the program. There are three basic types of variables in C:
  - **int:** for integer (whole numbers).
  - **double** (or **float**)**:** for real (floating point numbers).
  - **char:** for characters.

Identifiers

- All variables must have names. There are strict rules for variable names. These rules will apply to function names later so we will call these names **identifiers**.
- A declaration is done with the type followed by the identifier and a semi-colon (;).
- Ex:
  - int lifespan;
  - double mass;
  - char letter;

Hard Rules for Identifiers

- **Rule #1:** An identifier must not be a reserved word. Reserved words are used by C exclusively. Here are a few: **double**, **char**, **int**, **do**, **float**, **if**, **return**, **sizeof**, **void**, **while**, **typedef**, **struct**, **switch**, **for**, **else**. See the complete list at: ihypress.net/programming/c/reserved.html
- **Rule #2:** An identifier must contain only letters, digits, or underscores. **Abc8** is valid, **Abc-8** is not. **_xyz** is valid, **atom number** is not.

Hard Rules for Identifiers

- **Rule #3:** An identifier must never begin with a digit. **U238** and **_765** are valid, **7abc** and **67_q** are not.

- Which identifiers are valid (or not)?

Abc	xa_32	32X	_my files	file?1
Price$	abc87	45%	return	CHAR
8712_	t_a_b	int	maximum	dx@xa

Soft Rules for Identifiers

- **Rule #4:** An identifier should not be a standard identifier. A standard identifier is a name used by C but is not a reserved word. printf, scanf are examples of standard identifiers.
- **Rule #5:** All-capital identifiers should be used only for constant macros. Variables and functions should use lowercase letters only. You should not mix uppercase and lowercase letters in an identifier.

Using Variables

```
#include <stdio.h>
int
main (void)
{
    int temp; /* declare the variable */
    temp = 20; /* assigns value to variable */
    printf ("The temperature is %d.\n", temp);
    return (0);
}
```

Placeholders in I/O Statements

- Placeholders (or conversion specifiers) will substitute the value of the variable inside the output string (printf).
- You must absolutely match the placeholder with the variable type:

  %lf: long floating point (double).
  %d: decimal (int).
  %c: character (char).

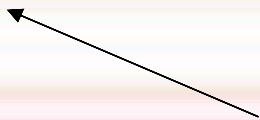
- \n: a special character meaning that a new line will be inserted.

Integer Placeholders

- %d is the default integer placeholder. When used it will simply display the value *as is* without any padding. To add padding, to have columns for example, we need formatted placeholders.

- %*n*d will reserve *n* places to display the number. Justification will be to the right. The negative sign takes one place.

- If the value is 17 and %4d is used, then it will display 2 spaces followed by 17 on the screen.
  __17

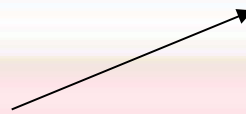


These are spaces, not underscores.

Integer Placeholders

- The number is always displayed in its entirety, even when the format is too narrow. With –1234 and a %3d placeholder, you would see –1234, therefore using 5 spaces instead of the 3 requested.

- A negative number in the placeholder changes the justification to the left of the field. For example, with a value of –1234 and a %–8d placeholder, the display will be –1234___ .



Three trailing blanks

Double Placeholders

- By default the %lf placeholder displays the number with 6 decimal digits and no padding (may vary depending on computer system).
- The formatted double placeholder has this format: %*w*.*d*lf, where *w* is the total width of the field (including sign and decimal point) and *d* the number of decimal digits.
- If the value is 4.56 and the placeholder is %6.3lf, then the display will be _4.560

Double Placeholders

- With a double formatted, you always get the requested number of decimal digits (even if the field is not wide enough).
- You also always (like the integer placeholder) get all the significant numbers.
- However, if there are more decimal precision in the value than in the placeholder, the value is truncated and rounded up if need be.

Double Placeholders

- If the value is -32.4573 and the placeholder is %7.3lf, then you will get 3 decimal digits as requested plus the significant numbers: -32.457.
- If the value is -32.4578 and the placeholder is %8.3lf, then you will get 3 decimal digits as requested plus the significant numbers and padding: _-32.458. See the rounding-up effect.
- A %8.7lf for a value of 187.123 will produce a display of 187.1230000.

Note: Internal values in the computer's memory are <u>unaffected</u> by placeholders.

Assignment Operator

- The = symbol represents the assignment operator. It places the value on the right inside the variable on the left. On the right you may find a constant (value), a variable, or an expression.

```
int a, b, c;
a = 10;
b = a;
c = a + b – 3;
a = b – 3;
```

What are the values of **a, b, and c** after these instructions?

The *scanf* Statement

- Another way to fill a variable is to ask the user for its value. We do that with the scanf statement.

```
printf ("Enter the temperature:");
scanf ("%d", &temp);  /* see the & sign! */
```

- temp will contain the value entered by the user at the keyboard.
- Placeholders are the same. Never use formatted placeholders in a scanf!
- Never put the & sign in front of a variable in a printf!

Modes of Operation

- Interactive mode: User responds to prompts by typing in data. Data is entered with the keyboard.
- Batch mode: Program takes its data from a data file prepared beforehand.
- To read from a file we use fscanf instead of scanf.
  (We can also use scanf to read from a file by using input redirection)
- To write to a file instead of displaying onscreen, we use fprintf.
  (We can also use printf to write to a file by using output redirection)
- To create a data file, we can use a simple text editor like **notepad**.

Reading from a File

- To do that you need first to declare the file
  - FILE *in;
  - */* notice that FILE is all uppercase and the * before the name of the file variable. */*
- Then the file must be opened
  - in = fopen ("mydata.txt", "r");
  - /* must exist on your disk! */
- To read from the file, we use fscanf
  - fscanf (in, "%d", &temp);
- The file must be closed when not longer needed
  - fclose (in);

Writing to a File

- To do that you need first to declare the file
  - FILE *out;
- Then the file must be opened
  - out = fopen ("result.txt", "w");
  - /* w is for write – it will create a new file on your disk */
- To write to the file, we use fprintf
  - fprintf (out, "%d", temp);
- The file must be closed when no longer needed
  - fclose (out);

Input Redirection

- So far we have seen that when we need to read data from the keyboard we use scanf and when we need to read from a file we use fscanf.
- In reality, scanf means reading from *standard input*, and that usually means the keyboard.
- With a command line, it is possible to redirect the input from the keyboard to another device, even a data file. The command line just specifies the file this way: **< data1.txt**.
- After redirecting, all scanfs in the program would read from that file. The keyboard is now deactivated for data input.

Output Redirection

- We can do redirection also for the output. So far we have seen that when we need to write data to the screen we use printf and when we need to write to a file we use fprintf. printf really means writing to *standard output*, and that usually means the screen.

- With a command line, it is possible to redirect the output from the screen to a data file. The command line just specifies the file this way: **> results.txt**.

- After redirecting, all printfs in the program would write to that file. The screen would no longer display anything from the program. We can also redirect both the standard input and output to files this way: **< data.txt > output.txt**.

Redirection vs. File I/O

(scanf/printf) *(fscanf/fprintf)*

Advantages:

- No need to modify program when file name changes.
- No need to open/close files in the program.

Drawbacks:

- The keyboard and/or screen are rendered useless.
- You are limited to one input file and one output file.
- You cannot use file and keyboard (or monitor) in the same program.

Conclusion:

- Unless you need more than one file or need access to keyboard and/or monitor in your program, novice programmers are encouraged to use redirection.

Example Programs

Visit this website for more

c.ihypress.ca

End of lesson