C Programming Basics

SDS 322/329

October 20, 2015

Email any questions to:
rauta@tacc.utexas.edu

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

Recap & Today's Lecture

- In the previous class
  - Understanding Errors
  - Comments: two types – single line (//) or multi-line (/* */)
  - Keywords: reserved words – e.g., int, return
  - Variables: declaration and initialization
  - Data Types and Identifiers
- Today's Lecture
  - Data Types and Typecasting (to be continued)
  - Scope of Variables
  - Operators
  - Standard I/O

Overview of Content

- Writing a Basic C Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Standard Input and Output
- Control Structures
- Functions in C
- Arrays, Structures
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Data Types

- Data types specify the type of data that a variable holds

- Categories of data types are:
  - **Built-in:** `char double float void int (short long signed unsigned)`
  - **User-defined:** `struct union enum`
  - **Derived:** `array function pointer`

- We have already seen an example code in which an integer data type was used to return a value from a function:

```
int main()
```

- Compiler-dependent range of values associated with each type. For example: an `int` can have a value in the range
  - **–32768** to **32767** on a 16-bit computer or
  - **–2147483647** to **2147483647** on a 32-bit computer

Casting between variables

- By type-casting, a variable can be temporarily made to look like another variable

- To typecast a variable of an already defined data-type, put the new data-type that you temporarily want inside parentheses in front of it
  - see next slide for example

Type-casting double as char: myTypeCast1.c

```
#include <stdio.h>
int main(){
   double varA = 65.00;
   char varB;
   varB = (char) varA;
   printf("varA: %lf, varB: %c",varA, varB);
   return 0;
}
```

Output:

```
varA: 65.000000, varB: A
```

Implicit Type Conversion: myTypeCast2.c

```
#include <stdio.h>
int main(){
 double varA;
 int varB = 2;
 double varC = 9.34;
 varA = varB;
 varB = varC;
 printf("varA: %lf, varB: %d, varC: %lf",varA, varB, varC);
 return 0;
}
```

Note

- `double` to `int` causes removal of the fractional part
- `int` to `double` conversion happened implicitly

Output:

```
varA: 2.000000, varB: 9, varC: 9.340000
```

Scope of Variables

- A variable can be either of global or local scope
  - Global variables are defined outside all functions and they can be accessed and used by all functions in a program file
  - A local variable can be accessed only by the function in which it is created
- A local variable can be further qualified as `static`, in which case, **it remains in existence rather than coming and going each time a function is called**
  - `static int x = 0;`
- A `register` type of variable is placed in the machine registers for faster access – compilers can ignore this advice
  - `register int x;`

Variables in Different Scope: varScope.c

```
#include <stdio.h>
int i = 1812;
int main(){
   printf("In function, global variable: %d\n", i);
   int i= 1912;
   printf("In function, before block: %d\n", i);
   {
      int i = 2012;
      printf("In function, inside block: %d\n", i);
   }
   printf("In function, after block: %d\n", i);
   return 0;
}
```

Guess the Output:

```
In function, global variable: ????
In function, before block: ????
In function, inside block: ????
In function, after block: ????
```

Compiling and Running varScope.c

```
login3$ gcc –o varScope varScope.c

login3$ ./varScope
In function, global variable: 1812
In function, before block: 1912
In function, inside block: 2012
In function, after block: 1912
```

Constants and Constant Expressions

- The value of a constant never changes
  - `const double e = 2.71828182;`

- Macros
  - `#define MAXRECORDS 100`
  - In the code, identifiers (**MAXRECORDS**) are replaced with the values (**100**)
  - Helps to avoid hard-coding of values at multiple places
  - Example: `char records[MAXRECORDS + 1];`
  - Can be used at any place where constants can be used

- Enumeration is a list of constant values
  - `enum boolean {NO , YES};`

Expressions containing constants are evaluated at compile-time

Overview of the Lecture

- Writing a Basic C Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Standard Input and Output
- Control Structures
- Functions in C
- Arrays, Structures
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Operators

- Arithmetic: +, -, /, *, %, ++, --, =
- Relational: a == b, a != b, a > b, a < b, a >= b, a <= b
- Logical: !a, a && b, a || b
- Member and Pointer: a[], *a, &a, a->b, a.b
- Other: **`sizeof`**
- Bitwise: ~a, a&b, a|b, a^b, a<<b, a>>b
- More about operators and precedence:

http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Increment and Decrement Operators

- `++i` means increment `i` then use it (**pre-increment**)

```
int i= 6;
int j;
j = ++i; //j is assigned 7
```

- `i++` means use `i` then increment it (**post-increment**)

```
int i= 6;
int j;
j = i++; //j is assigned 6
```

Note: Like increment operator, the decrement operator also works in pre and post modes.

Parentheses and Precedence: checkParentheses.c

```c
#include <stdio.h>
int main(){
 int total;
 //multiplication has higher precedence than subtraction
 total=100 - 25*2;
 printf("The total is: $%d \n",total);
 //parentheses make a lot of difference!
 total=(100 - 25)*2;
 printf("The total is: $%d \n",total);
 return 0;
}
```

```
Output:
The total is: $50
The total is: $150
```

`sizeof` Operator Example: testSize.c

```
#include <stdio.h>
int main(){
 char c;
 int x;
 printf("Size of variable c is %d bytes\n", sizeof(c));
 printf("Size of variable x is %d bytes\n", sizeof(x));
 return 0;
}
```

Note: Byte sizes of variables can be found by using `sizeof` operator

```
Output:
Size of variable c is 1 bytes
Size of variable x is 4 bytes
```

Note: Declaring a character variable (`char c;`) is different from declaring a string (`char myName[50];`)

Overview of the Lecture

- Writing a Basic C Program
- Understanding Errors
- Comments, Keywords, Identifiers, Variables
- Operators
- Standard Input and Output
- Control Structures
- Functions in C
- Arrays, Structures
- Pointers
- Working with Files

All the concepts are accompanied by examples.

Reading Keyboard Input: readInput1.c

```
#include <stdio.h>
int main(){
   char myName[50];
   printf("What is your name?");
   fflush(stdout);
   scanf("%s", &myName);
   printf("Hello %s!", &myName);
   return 0;
}
```

scanf function is used to read the keyboard input
fflush flushes the contents of the output buffer

Understanding readInput1.c

```
#include <stdio.h>
int main(){

  char myName[100];

  printf("What is your name?");

  fflush(stdout);

  scanf("%s", &myName);

  printf("Hello %s!", &myName);
  return 0;
}
```

This is a **variable declaration** for string type and **myName** is a string variable. It provides storage for the information you enter. Note the usage of **char**.

Explicit flushing of the output stream

Function to read the value from keyboard and store it in computer's memory

More Information on `scanf`

- Function to read information from the keyboard

```
scanf("%s", &myName);
```

- First parameter is a type-specifier
  - `%s` is a type-specifier that is used if input data is string or text.
  - other type-specifiers are %c for character, %d for decimal, %f for float, %o for octal, %x for hexadecimal

- The second parameter is the address of the variable that would store the value being input from the keyboard
  - **`myName`** is the string variable for storing the input value
  - Ampersand (&) before the variable name helps `scanf` find the location of the string variable in memory

More functions for I/O

- **`gets`** function was used to read the keyboard input (*i.e.,* standard input stream)

```
gets(myName);
```

Warning: do not use this function! Letting you know just for G.K.

- **`puts`** function is used to print text on the screen (*i.e.,* standard output stream)

```
puts(myName);
puts("Hello Ritu");
```

Unlike **`printf`**, it always displays a newline character and can print only one variable or a string

More functions for I/O

- `getchar()` function is used to read a single character from the keyboard
  - It causes the program to pause until a key is typed at the keyboard and Enter is pressed after that
  - More on this syntax later

- `putchar(c)` function displays the character on the screen
  - `c` can be a character constant in single quotes or a variable name

String Variables

- Numeric values can be assigned by using the "=" sign but string values cannot be assigned using the "=" sign

```
char myName[50];
myName = "Ritu"; // this is wrong
```

- Three ways to assign values to strings

```
scanf("%s", &myName);
gets(myName);
strcpy(myName,"Ritu");
```

- Function `strcpy`
  - It is defined in the header file `string.h` and hence needs to be included
  - It copies the value of one string to another

`strcpy` Example: writeStrChar.c

```
#include <stdio.h>
#include <string.h>
int main(){
   char myName[50];
   char myName2[50];
   char c;
   strcpy(myName2,"Ritu");
   strcpy(myName, myName2);
   c = 'a';
   printf("Your name is: %s\n", myName);
   printf("The character is: %c \
   return 0;
}
```

Output:

```
Your name is: Ritu
The character is: a
```

Pop Quiz
(Reflect on this & ask questions, if any)

- How will you use **`scanf`** to read different data types?

- Is the following statement correct?

```
printf("%s, your color is: %s","red");
```

- Fill in the blanks(_):

```
scanf("%__", __myIntegerNumber);
```

References

- C Programming Language, Brian Kernighan and Dennis Ritchie
- Let Us C, Yashavant Kanetkar
- C for Dummies, Dan Gookin
- http://cplusplus.com
- http://www.cprogramming.com/tutorial/c/lesson11.html