C Programming I

Alexander B. Pacheco
LTS Research Computing
May 29, 2015

Outline

Introduction

What is the C Language?

- A general-purpose, procedural, imperative computer programming language.
- Developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories to develop the UNIX operating system.
- The UNIX operating system, the C compiler, and essentially all UNIX applications programs have been written in C.
- C is the most widely used computer language.
  - Easy to learn
  - Structured language
  - Produces efficient programs
  - Handles low-level activities
  - Can be compiled on a variety of computer plaforms
- Most of the state-of-the-art softwares have been implemented using C.
- Today's most popular Linux OS and RBDMS MySQL have been written in C.

What do you need to learn C?

1. C Compiler
   - What is a Compiler?
     - A compiler is a computer program (or set of programs) that transforms source code written in a programming language (the source language) into another computer language (the target language, often having a binary form known as object code).
   - How does a compiler do?
     - Translate C source code into a binary executable
   - List of Common Compilers:
     - GCC GNU Project (Free, available on most *NIX systems)
     - Intel Compiler
     - Portland Group (PGI) Compiler
     - Microsoft Visual Studio
     - IBM XL Compiler
2. Text Editor
   - Emacs
   - VI/VIM
   - Notepad++ (avoid Notepad if you will eventually use a *NIX system)
   - Integrated Development Environment: Eclipse, XCode, Visual Studio, etc

Program Structure

Program Structure

A C Program consists of the following parts

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

A Simple Hello World Code

```c
#include <stdio.h>

int main ()
{
  /* My First C Code */
  printf("Hello World!\n");
  return 0;
}
```

Compile and execute the code

```
dyn100077:Exercise apacheco$ gcc hello.c
dyn100077:Exercise apacheco$ ./a.out
Hello World!
```

My First C Code

```c
#include <stdio.h>

int main ()
{
  /* My First C Code */
  printf("Hello World!\n");
  return 0;
}
```

- `#include <stdio.h>` is a preprocessor command.

  It tells a C compiler to include stdio.h file before going to actual compilation.
- `int main()` is the main function where program execution begins.
- `/* ... */` is a comment and ignored by the compiler.
- `printf(...)` is function that prints `Hello World!` to the screen.
- `return 0;` terminates main() function and returns the value 0.

Basic Syntax

Basic C Syntax I

- C is a case sensitive programming language i.e. program is not the same as Program or PROGRAM.
- Each individual statement must end with a semicolon.
- Whitespace i.e. tabs or spaces is insignificant except whitespace within a character string.
- All C statments are free format i.e. no specified layout or column assignment as in FORTRAN77.

```
#include <stdio.h>
int main () {  /* My First C Code */  printf("Hello World!\n");  return 0;}
```

  will produce the exact same result as the code on the previous slide.
- In C everything within `/*and */` is a comment. Comments can span multiple lines.

```
/* this is single line comment */
/* This
is a
multiline comment */
```

Valid Character Set in C language

Alphabets	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Digits	0123456789

Special Characters														
,	_	{	<	’	(	^	;	$	/	*	+	[	#	?
.	&	}	>	"	)	!	:	%	|	\	-	]	~	

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

- White space Characters: blank space, new line, horizontal tab, carriage return and form feed

Data Types, Variables and Constants

Data Types

Basic Types: There are five basic data types

1. int - integer: a whole number.
2. float - floating point value: ie a number with a fractional part.
3. double - a double-precision floating point value.
4. char - a single character.
5. void - valueless special purpose type.

Derived Types: These include

1. Pointers
2. Arrays
3. Structures
4. Union
5. Function

- The array and structure types are referred to collectively as the aggregate types.
- The type of a function specifies the type of the function's return value.

Basic Data Types: Integer

Type	Storage size (in bytes)	Value range
char	1	-128 to 127 or 0 to 255
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2 or 4	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4	0 to 65,535 or 0 to 4,294,967,295
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295

- To get the exact size of a type or a variable on a particular platform, you can use the sizeof operator.
- The expressions `sizeof(type)` yields the storage size of the object or type in bytes.

Basic Data Types: Floating-Point & void

Type	Storage size	Value range	Precision (decimal places)
float	4 bytes	1.2E-38 to 3.4E38	6
double	8 bytes	2.3E-308 to 1.7E308	15
long double	10 bytes	3.4E-4932 to 1.1E4932	19

Situation	Description
function returns as void	function with no return value
function arguments as void	function with no parameter
pointers to void	address of an object without type

Variables

- Variables are memory location in computer's memory to store data.
- To indicate the memory location, each variable should be given a unique name called identifier.
- Variable names are just the symbolic representation of a memory location.
- Rules for variable names:
  1. Composed of letters (both uppercase and lowercase letters), digits and underscore '_' only.
  2. The first letter of a variable should be either a letter or an underscore.
  3. There is no rule for the length of a variable name.
     - Most likely your code will be used by someone else, so variable names should be meaningful and short as possible.

```
int num;
float circle_area;
double _volume;
```

- In C programming, you have to declare variable before using it in the program.

Declaring Variable or Variable Definition

- A variable definition means to tell the compiler where and how much to create the storage for the variable.
- A variable definition specifies a data type and contains a list of one or more variables of that type as follows:

  `type variable_list;`
- `type` must be a valid C data type or any user-defined object, etc., and `variable_list` may consist of one or more identifier names separated by commas.
- Variables can be initialized (assigned an initial value) in their declaration.

  `type variable_name = value;`

```
int     i, j, k;
char    c, ch;
float   f, salary;
double d;
int d = 3, f = 5;           // definition and initializing d and f.
byte z = 22;                // definition and initializes z.
char x = 'x';               // the variable x has the value 'x'.
```

Constants & Literals

The constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals.

Integer Constants

```
85      /* decimal */
0213    /* octal */
0x4b    /* hexadecimal */
30      /* int */
30u     /* unsigned int */
30l     /* long */
30ul    /* unsigned long */
```

Character Constants

```
'a'     /* character 'a' */
'Z'     /* character 'Z' */
\?      /*? character */
\\      /*\ character */
\n      /*Newline */
\r      /*Carriage return */
\t      /*Horizontal tab */
```

Floating Point Constants

```
3.1416
314159E-5     /* 3.14159 */
2.1E+5        /* 2.1x10^5 */
3.7E-2        /* 0.037 */
0.5E7         /* 5.0x10^6 */
-2.8E-2       /* -0.028 */
```

String Constants

```
"hello, world"          /* normal string */
"c programming \
language"               /* multi-line string */
```

How to define Constants

- Constants can be defined in two ways
  1. Using the **#define** preprocessor (defining a macro)
  2. Using the **const** keyword (new standard borrowed from C++)

```
#include <stdio.h>

/* define LENGTH using the macro */
#define LENGTH 5

int main()
{
 /*define WIDTH using const */
  const int WIDTH = 3;
  const char NEWLINE = '\n';
  int area = LENGTH * WIDTH;

  printf("value of area : %d", area);
  printf("%c", NEWLINE);
  return 0;
}
```

Input and Output

- C or any programming language in general needs to be interactive i.e. write something back and optionally read data to be useful.
- Similar to Unix, C treats all devices as files.

Standard File	File Pointer	Device
Standard Input	stdin	Keyboard
Standard Output	stdout	Screen
Standard Error	stderr	Screen

- C Programming language provides three functions to read/write from standard input/output

	Unformatted		Formatted
Input	getchar	gets	scanf
Output	putchar	puts	printf

Unformatted I/O

The getchar() & putchar() functions

- The `int getchar(void)` function reads the next available character from the screen and returns it as an integer.

  This function reads only single character at a time.
- The `int putchar(int c)` function puts the passed character on the screen and returns the same character.

  This function puts only single character at a time.

The gets() & puts() functions

- The `char *gets(char *s)` function reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF.
- The `int puts(const char *s)` function writes the string s and a trailing newline to stdout.

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

   printf( "Enter a value :");
   c = getchar( );

   printf( "\nYou entered: ");
   putchar( c );

   return 0;
}
```

```c
#include <stdio.h>
int main( )
{
   char str[100];

   printf( "Enter a value :");
   gets( str );

   printf( "\nYou entered: ");
   puts( str );

   return 0;
}
```

Formatted I/O

- The `int scanf(const char *format, ...)` function reads input from the standard input stream stdin and scans that input according to format provided.
- The `int printf(const char *format, ...)` function writes output to the standard output stream stdout and produces output according to a format provided (optional).

```c
#include <stdio.h>

int main ()
{
  /* My Second C Code */
  char name[100];
  printf("Enter your name:");
  scanf("%s",&name);
  printf("Hello %s\n",name);
  return 0;
}
```

- In this program, the user is asked a input and value is stored in variable `name`.
- Note the '`&`' sign before `name`.
- `&name` denotes the address of `name` and value is stored in that address.

Common Format Specifier

- The format specifier: `%[flags][width][.precision][length]specifier`

flag	meaning
-	left justify
+	always display sign
0	pad with leading zeros

Specifier	Output	Example
%f	decimal float	3.456
%7.5f	decimal float, 7 digit width and 5 digit precision	3.45600
%d	integer	5
%05d	integer, 5 digits pad with zeros	00101
%s	string of characters	"Hello World!"
%e	scientific notation for decimal float	2.71828e+5
%c	character	
\n	insert new line	
\t	insert tab	

```c
/* printf example showing different specifier usage */
#include <stdio.h>
int main() {
  printf ("Characters: %c %c \n", 'a', 65);
  printf ("Decimals: %d %0qa4d\n", 2014, 65);
  printf ("\t floats: %7.5f \t%f \t%e \n", 3.1416, 3.1416, 3.1416);
  printf ("%s \n", "hello world");
  return 0;
}
```

```
alexanders-mbp:Example apacheco$ gcc -o print print.c
alexanders-mbp:Example apacheco$ ./print
Characters: a A
Decimals: 2014 0065
        floats: 3.14160       3.141600       3.141600e+00
hello world
```

Programming Operators

Operators

- Arithmetic

Operator	Meaning
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division(modulo division)
++	increase integer value by one
–	decrease integer value by one

- Assignment Operator

Operator	Example	Same as
=	a=b	a=b
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

Increment/Decrement Operator

- There are two types of increment/decrement operators
  1. Suffix or Postfix: e.g. i++ or j--
     a=i++ means set a to i and then increment i by 1
  2. Prefix: ++i or --j
     a=++i means increment i by 1 and then set a to i
- Consider the following example
  If i = 1 and j = 2, then
  ++i + j++ = 4
  and not 5 since j is incremented after the operation is complete

```c
#include<stdio.h>

int main () {
  int i=1,j=2;
  int a, b;
  int k=1,l=2;

  a=++k ;
  b=l++ ;

  printf("++i + j++: %d\n", ++i + j++ );
  printf("a=++i: %d, b=j++: %d, i:%d, j:%d\n", a, b
         , k, l);
  printf("a(=++i) + b(=j++): %d\n", a + b);

  return 0;

}
```

```
alexanders-mbp:Example apacheco$ make increment
cc     increment.c   -o increment
alexanders-mbp:Example apacheco$ ./increment
++i + j++: 4
a=++i: 2, b=j++: 2, i:2, j:3
a(=++i) + b(=j++): 4
```

Relational Operators

- Relational operators checks relationship between two operands.
- If the relation is true, it returns value 1 and if the relation is false, it returns value 0.
- Relational operators are used in decision making and loops in C programming.

Operator	Meaning	Example
==	Equal to	5==3 returns false (0)
>	Greater than	5>3 returns true (1)
<	Less than	5<3 returns false (0)
!=	Not equal to	5!=3 returns true(1)
>=	Greater than or equal to	5>=3 returns true (1)
<=	Less than or equal to	5<=3 return false (0)

Logical & Conditional Operators

- Logical operators are used to combine expressions containing relation operators.
- In C, there are 3 logical operators

Operator	Meaning	Example
&&	Logial AND	If c=5 and d=2 then,((c==5) && (d>5)) returns false.
||	Logical OR	If c=5 and d=2 then, ((c==5) || (d>5)) returns true.
!	Logical NOT	If c=5 then, !(c==5) returns false.

- Conditional Operator: Conditional operators are used in decision making in C programming, i.e, executes different statements according to test condition whether it is either true or false.

```
conditional_expression?expression1:expression2
```

- If the test condition is true, expression1 is returned and if false expression2 is returned.

```
d=(c>0)?10:-10;
```

If c is greater than 0, value of d will be 10 but, if c is less than 0, value of d will be -10.

Other Operators

Operator Precedance

Operator	Description	Associativity
++, --	Suffix Increment/Decrement	→
++, --	Prefix Increment/Decrement	←
+, -	Unary plus and minus	
!, ~	Logical NOT and Bitwise NOT	
*	Indirection (dereference)	
&	Address of	
sizeof	Size-of	
*, /, %	Multiplication, division, modulo	→
+, -	Addition, Subtraction	
«, »	Bitwise left and right shift	
<, <=	Relational Operators	
>, >=		
==, !=		
&	Bitwise AND	
^	Bitwise XOR	
|	Bitwise OR	
&&	Logical AND	
||	Logical OR	
?:	Ternary Conditional	←
=	Simple Assignment	
+=, -=	Assignment by sum and difference	
*=, /=, %=	Assignment by product, quotient and remainder	
«=, »=	Assignment by bitwise left and right shift	
&=, ^=, |=	Assignment by logical AND, XOR and OR	
,	Comma Operator	→

Control Flow

Control Flow

- Conditional Statements (decision making/selection)
  - if $\cdots$ else if $\cdots$ else
  - switch
- Loops
  - for
  - while
  - do while

if statement

- An if statement consists of a boolean expression followed by one or more statements.

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
```

- If the boolean expression evaluates to true, then the block of code inside the if statement will be executed.
- If boolean expression evaluates to false, then the first set of code after the end of the if statement(after the closing curly brace) will be executed.

if · · · else statement

- An if statement can be followed by an optional else statement, which executes when the boolean expression is false.

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
else
{
  /* statement(s) will execute if the boolean expression is false */
}
```

- If the boolean expression evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

if · · · else if · · · else statement

- An if statement can be followed by an optional else if· · · else statement,
- very useful to test various conditions using single if· · · else if statement.
- When using if , else if , else statements there are few points to keep in mind:
  - An if can have zero or one else's and it must come after any else if's.
  - An if can have zero to many else if's and they must come before the else.
  - Once an else if succeeds, none of the remaining else if's or else's will be tested.

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
{
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3)
{
   /* Executes when the boolean expression 3 is true */
}
else
{
   /* executes when the none of the above condition is true */
}
```

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;

   /* check the boolean condition */
   if( a < 20 )
   {
       /* if condition is true then print the following */
       printf("a is less than 20\n" );
   }
   else
   {
       /* if condition is false then print the following */
       printf("a is not less than 20\n" );
   }
   printf("value of a is : %d\n", a);

   return 0;
}
```

Nested if· · · else statement

- You can use one if or else if statement inside another if or else if statement(s) i.e. nested if· · · else statement/s

```
if( boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
   if(boolean_expression 2)
   {
      /* Executes when the boolean expression 2 is true */
   }
}
```

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;

   /* check the boolean condition */
   if( a == 100 )
   {
       /* if condition is true then check the following */
       if( b == 200 )
       {
          /* if condition is true then print the following */
          printf("Value of a is 100 and b is 200\n" );
       }
   }
   printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );

   return 0;
}
```

switch statement

- A switch statement allows a variable to be tested for equality against a list of values.
- Each value is called a case, and the variable being switched on is checked for each switch case.

```
switch(expression){
    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```

- The expression used in a switch statement must have an integral type

  (or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type).
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch.
- The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

```c
#include <stdio.h>

int main ()
{
   /* local variable definition */
   char grade;
   printf("Enter your grade:\n");
   scanf("%c", &grade);

   switch(grade)
   {
   case 'A' :
      printf("Excellent!\n" );
      break;
   case 'B' :
   case 'C' :
      printf("Well done\n" );
      break;
   case 'D' :
      printf("You passed\n" );
      break;
   case 'F' :
      printf("Better try again\n" );
      break;
   default :
      printf("Invalid grade\n" );
   }
   printf("Your grade is  %c\n", grade );

   return 0;
}
```

Nested Conditional Statements

- Conditional statements can be nested as they do not overlap:

```
if( boolean_expression 1) {
  if(boolean_expression 2) {
    /* Executes when the boolean expression 2 is true */
    /* nested switch statement */
    switch(expression){
    case constant-expression :
      statement(s);
      break; /* optional */
    case constant-expression :
      statement(s);
      break; /* optional */
      /* you can have any number of case statements */
    default : /* Optional */
      statement(s);
    }
  }
 }
```

for loop

- A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.
  - The init step is executed first and only once.
  - the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute, the loop exits.
  - the increment statement executes after the loop body.
  - The loop continues until the condition becomes false

```
for ( init; condition; increment )
{
   statement(s);
}
```

while and do· · · while loops

- while loops are similar to for loops
- A while loop continues executing the code block as long as the condition in the while holds.

```
while(condition)
{
    statement(s);
}
```

- do· · · while loop is guaranteed to execute at least one time.

```
do
{
    statement(s);

}while( condition );
```

Simple loops using for, while, do while

```c
#include <stdio.h>
int main ()
{
  int i;
  /* for loop execution */
  for(i = 0; i < 5; i++ ) {
    printf("for loop i= %d\n", i);
  }
  i=0;
  /* while loop execution */
  while( i < 5 ) {
    printf("while loop i: %d\n", i);
    i+=1;
  }
  i=1;
  /* do-while loop execution */
  do {
    printf("do while loop i: %d\n", i);
    i=i+1;
  }while( i < 0 );

  return 0;
}
```

Nested loops in C

- All loops can be nested as long as they do not overlap

```
/* nested for loops*/
for (init; condition; increment) {
  for (init; condition; increment) {
    statement(s);
  }
  statement(s);
 }
/* nested while loops*/
while (condition) {
  while (condition) {
    statement(s);
  }
  statement(s);
 }
```

```
/* nested do while loops*/
do {
  statement(s);
  do {
    statement(s);
  } while ( condition );
 } while ( condition );
/* mixed type loops*/
while (condition) {
  for (init; condition; increment) {
    statement(s);
    do {
      statement(s);
    } while ( condition );
  }
  statement(s);
 }
```

```c
#include <stdio.h>

int main () {
  int i, j, k, n=2;
  printf("i j k\n");
  /* Nested for loops */
  for (i=0; i<n; ++i)
    for (j=0; j<n; j++)
      for (k=0; k<n; ++k)
  printf("%d %d %d\n", i,j,k);
  return 0;
}
```

Loop Control Statement

- Loop control statements change execution from its normal sequence.
  - **break:** Terminates the loop or switch statement
  - **continue:** Causes the loop to skip the remainder of its body for the current iteration
  - **goto:** Transfers control to the labeled statement. Use is not advised

```c
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 )
    {
      printf("value of a: %d\n", a);
      a++;
      if( a > 15)
  {
    /* terminate the loop using break statement */
          break;
  }
    }

  return 0;
}
```

```c
#include <stdio.h>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* do loop execution */
  do
    {
      if( a == 15)
  {
    /* skip the iteration */
    a = a + 1;
    continue;
  }
      printf("value of a: %d\n", a);
      a++;

    }while( a < 20 );

  return 0;
}
```

Exercise

1. Print list of prime numbers less than 100
2. Calculate circumference and area of a circle for given radius
3. Obtain roots of a quadratic equations
4. Calculate factorial of a number