Variables, IO, and Operators

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

November 2, 2015

Outline

Variable Names

- variables user defined identifiers
- variables are case sensitive, `numVar` and `numvar` are two different variables

valid	invalid	reason
rst	2_times	starts with a digit
_tmp	long	long is a keyword
x4	won@last	@ is not allowed
int_	last name	contains a space
char2		
nextValue		

- All variables must be declared before they are used

Variable Declations

```c
#include <stdio.h>
int main() {
        char c;
        int n;
        float f = 3.14f; /* declares f and initializes
        it to the value of 3.14f */
        double d;

        c = 'a';
        n = 100;
        d = 12.34;

        printf("c = %c (%d) and n = %d\n", c, c, n);
        printf("f = %f and d = %f\n", f, d);
        return 0;
}
```

Standard IO

- C provides a set of functions in the header file `stdio.h` to read inputs and write outputs.
- C treats all devices as files:

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

Formatting Outputs - printf()

```
int printf (char *format, arg1, arg2, ...)
```

- `printf` prints a list of arbitrary arguments (`arg1, arg2, ...`) on the standard output under control of the `format`
- returns the number of characters printed
- `format` contains two types of objects:
  - ordinary characters: printed to the output stream
  - specifiers: causes conversion and printing of the next successive argument

Format Spceifiers

- each specifier begins with a % and ends with a conversion character(s), between them there may be (in order):
  - a minus (-): specifies left adjustment
  - a number: specifies the minimum field width
  - a period (.): separates the field with from the precision
  - a number: specifies the maximum number of digits after the period for float types

Format Spceifiers

- format specifiers:
  - `%d` (`%i`) int signed decimal
  - `%u` int unsigned decimal
  - `%o` int unsigned octal value
  - `%x` (`%X`) int unsigned hex value
  - `%f` float or double
  - `%e` (`%E`) float or double exponential format
  - `%s` array of char (string)
  - `%p` pointer address stored in pointer
  - `%ld` long signed decimal
  - `%hd` short signed decimal

Format Spceifiers

- formatting output:

```
int a = 24, b = 168;
float f = 3.14159;
printf("a: %04d\n", a);
printf("b: %x\n", b);
printf("b: %o\n", b);
printf("f: %6.2f\n", f);
printf("f: %-6.2f\n", f);
printf("&a: %p\n", &a);
```

Format Spceifiers

- the output: the last one prints the 32 bit memory address where the value of `a` is stored

```
a: 0024
b: a8
b: 250
f:    3.14
f: 3.14
&a: 0028FF44
```

Reading inputs - scanf()

```
int scanf (char *format, arg1, arg2, ...)
```

- `scanf` reads characters from the keyboard, interprets them according to the specifiers in `format`, and stores the results through the remaining arguments

```
char c; int n; float f;
scanf("%c %d %f", &c, &n, &f);
printf("c = %c, n = %d, f = %f\n", c, n, f);
```

- each argument after `format` is a pointer (`&c`) indicates the memory address where the input should be stored

if statements

- if statements is used to express decisions

```
if (expression)
        statement1 ;
```

```
if (expression)
        statement1 ;
else
        statement2 ;
```

- the expression is evaluated, it it is true (non-zero) then $statement_1$ is executed
- if it is false (zero) the $statement_2$ is executed

if statements

- if statements evaluates the numeric value of an expression, the following coding are possible, when the first is much more obvious than the second
  1. `if (num)` ✓
  2. `if (num != 0)`
- also the following two are equivalent:
  1. `if (!num)` ✓
  2. `if (num == 0)`

else-if

```
if (expression1)
        statement1;
else if (expression2)
        statement2;
else if (expression3)
        statement3;
else
        statement4;
```

- The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain

Arithmetic Operators

- Arithmetic operators: +, −, ∗, /, %, ++, −−

```
int n = 10, m = 3, t;
float f = 12.8, r;

t = n % m; /* % can be used only with
              integer types */
r = f * n;
```

Increments and Decrements

- The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1
- ++ and -- may be used either as prefix or postfix

```
int n = 6, m;
        /* the postfix ++, increments n after
        its value has been used */
m = n++;          /* m = 6 and n = 7*/
```

```
int n = 6, m;
        /* the prefix ++, incements n before
        its value has been used */
m = ++n; /* m = 7 and n = 7*/
```

Relational and Logical Operators

- Relational Operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Logical Operators: `&&`, `||`, `!`

```
if (c >= 'A' && c <= 'Z')
        printf("c = %c\n", c + 'a' - 'A');
else
        printf("c = %c\n", c);
```

Bitwise Operators

- Bitwise Operators: `&`, `|`, `^`, `~`, `<<`, `>>`
- bit manipulation operators, only used with integer datatypes

```
n = n & 0177;
/* sets to 0 all but the low-order 7 bits of n */
x = x | y;
/* sets to 1 in x the bits that are set to 1 in y */
x = x << 2;
/* shifts the value of x left by 2 positions,
 filling vacated bits with 0 */
```

Assignment Operators

- Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`
- Most binary operators **op** have a corresponding assignment operator **op=**

```
t %= m;          /* t = t % m */
r *= f;          /* r = r * f */
n &= 0177;       /* n = n & 0177 */
x |= y;          /* x = x | y */
x <<= 2;         /* x = x << 2 */
```

Misc Operators

- Misc Operators: `sizeof()`, `&` (address of a variable), `*` (pointer to a variable), `?:` (if?then:otherwise)

```
int x = 4;
printf("size of int = %lu\n", sizeof(int));
printf("size of x = %lu\n", sizeof(x));
```

Pointer and Addresses (* and &)

- The unary operator * is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to

```
int x =1, y =2;

int *ip; /* ip is a pointer to int */
ip = &x; /* ip now points to x */
y = *ip; /* y is now 1 */
*ip = 0; /* x is now 0 */
```

Conditional Expression (?:)

- The conditional expression, written with the ternary operator "?:", provides an alternate way to write if-else statement

  `if (expr1) epxr2; else expr3;`

  - `expr1 ? expr2 : expr3`

```
int a = 10, b = 6;
int z;
z = (a > b) ? a : b;
/* z is 10 */
```

Type Conversion

- In arithmetic and bitwise operation the narrower operand will be converted to the wider one before applying the operation
  - `char` → `short` → `int` → `unsigned int` → `long` → `unsigned long` → `long long` → `float` → `double` → `long double`
- expressions like assignment might result in losing information like assigning longer integer to shorter or a float to an integer (may draw a warning)

Type Casting

- We can convert the values from one type to another explicitly using the cast operator as follows:
  - `(type_name) expression`

```
#include <stdio.h>
main() {
        int sum = 17, count = 5;
        double mean;
        mean = (double) sum / count;
        printf("Value of mean : %f\n", mean);
}
```

Precedence and Order of Evaluation

- Operators on the same line have the same precedence

Operators											Associativity
()	[]	->	.								left-to-right
!	~	++	--	+	-	*	&	(type)	sizeof		right-to-left
*	/	%									left-to-right
+	-										left-to-right
<<	>>										left-to-right
<	<=	>	>=								left-to-right
==	!=										left-to-right
&											left-to-right
^											left-to-right
|											left-to-right
&&											left-to-right
||											left-to-right
?:											right-to-left
=	+=	-=	*=	/=	%=	&=	^=	|=	<<=	>>=	right-to-left
,											left-to-right

Memory Layout of C Programs

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

November 5, 2015

UNIVERSITY OF Southampton

Outline

Functions

- every C program has at least one function: `main()`
- you can devide you program into sepatate functions, each function perform a specfic task
- The C standard library provides numerous built-in functions: `printf()` and `scanf()`
- The function declaration consists of:
  - the function's name
  - return type
  - list of parameters
- A function definition provides the actual body of the function

Functions

```c
#include <stdio.h>
/* function declaration */
int max(int n, int m);
int main (){
        int a = 7, b = 5, result;
        result = max(a, b);
        printf( " max of %d and %d is %d\n", a, b, result);
        return 0;
}
/* function definition */
int max(int n, int m){
        int result;
        if (n > m)
                result = n;
        else
                result = m;
        return result;
}
```

Global Variables

- Global variables are defined outside of a function
- They hold their value throughout the lifetime of your program
- A global variable can be accessed by any function
- a program can have same name for local and global variables
  - local variable inside a function will take preference

```
#include <stdio.h>
/* function declaration */
int global = 10;
int main (){
        int global = 20;

        printf( " global = %d\n", global);
        return 0;
}
```

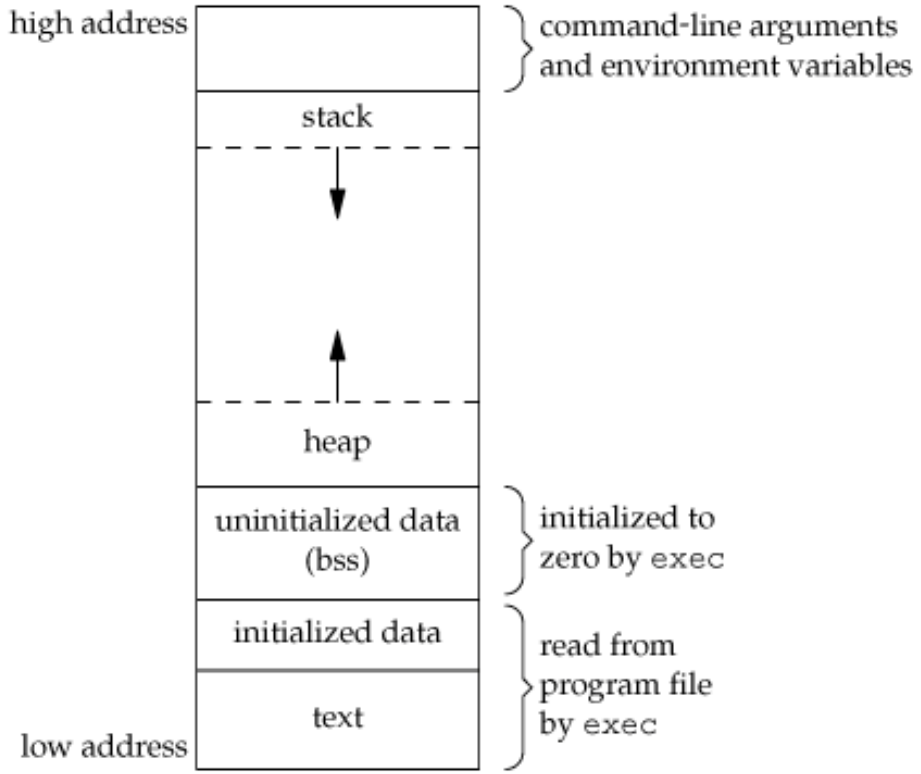
Static Variables

- the compiler to keeps the static variables in existence during the life-time of the program
- local variables static allows them to maintain their values between function calls

```
#include <stdio.h>
void func(void);
main() {
        func();
        func();
        func();
        return 0;
}
void func(){
        static int i = 0; /* local static variable */
        i++;
        printf("i is %d\n", i);
}
```

Memory Layout of C Programs

- A typical memory representation of C program consists of following sections
  1. Text segment
  2. Initialized data segment
  3. Uninitialized data segment
  4. Stack
  5. Heap



Text Segment

- **text segment**, also known as a **code segment** or **text**
- contains executable instructions
- may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it
- shareable: single copy needs to be in memory for frequently executed programs
- readonly: to prevent a program from accidentally modifying its instructions

Initialized Data Segment

- simply the **Data Segment**: is a portion of virtual address space of a program,
- contains the **global variables** and **static variables** that are **initialized** by the programmer
- is not read-only since the values of variables can be altered at runtime
- Examples:
  - `static int i = 10;`
  - a global variable `int j = 10;`

Uninitialized Data Segment

- often called the "bss" segment, "block started by symbol"
- Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
- contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code, for instance:
  - `static int i;`
  - a global variable `int j;`

Stack

- The stack area contains the program stack (LIFO structure)
- A "stack pointer" tracks the top of the stack
- The set of values pushed for one function call is termed a "stack frame"; A stack frame consists at minimum of a return address
- Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack
- The newly called function then allocates room on the stack for its variables

Heap

- Heap is the segment where dynamic memory allocation usually takes place
- The Heap area is managed by `malloc`, `realloc`, and `free`
- The Heap area is shared by all shared libraries and dynamically loaded modules in a process

Examples

- The size(1) command reports the sizes (in bytes) of the text, data, and bss segments
- Check the following simple C program

```c
#include <stdio.h>

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248          8       1216        4c0    memory-layout
```

Examples

- add one global variable in program, now check the size of bss

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         12       1220        4c4    memory-layout
```

Examples

- add one static variable which is also stored in bss

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         16       1224        4c8    memory-layout
```

Examples

- initialize the static variable which will then be stored in Data Segment (DS)

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960         252         12       1224        4c8    memory-layout
```

Examples

- initialize the global variable which will then be stored in Data Segment (DS)

```c
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960         256          8       1224        4c8    memory-layout
```

Control Statements and Repetition

Jamal Hussein

jah1g12@ecs.soton.ac.uk

Electronics and Computer Science

November 5, 2015

UNIVERSITY OF Southampton

Outline

Constants

- There are two simple ways in C to define constants:
- Using `#define` preprocessor

```
#define identifier value
```

- Using `const` keyword

```
const type variable = value;
```

Constants

```c
#include <stdio.h>

#define PI 3.14159265359

int main()
{
        double area;
        double radius = 5;

        area = radius * radius * PI;
        printf("value of area : %f\n", area);

        return 0;
}
```

Constants

```c
#include <stdio.h>

int main()
{
        const double PI = 3.14159265359;
        double area;
        double radius = 5;

        area = radius * radius * PI;
        printf("value of area : %f\n", area);

        return 0;
}
```

Switch Statement

- A switch statement allows a variable to be tested for equality against a list of values

```
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);
}
```

Switch Statement

- The expression used in a switch statement must have an integral or enumerated type
- 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
- case will execute until a break statement is reached
- When a break statement is reached, the switch terminates
- If no break appears, the flow of control will fall through to subsequent cases
- No break is needed in the default case

Switch Statement

```
#include <stdio.h>
int main () {
        char grade = 'B';
        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;
}
```

For Statement

- A for loop is a repetition control structure that executes a specific number of times

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

- The init step is executed first, and only once
- the condition is evaluated
  - If it is true (nonzero), the body of the loop is executed
  - if it is false (zero) the loop ended
- the flow of control jumps back up to the increment

For Statement

```c
#include <stdio.h>
int main ()
{
        /* for loop execution */
        int a;
        for( a = 10; a < 20; a++ )
        {
                printf("value of a: %d\n", a);
        }
        return 0;
}
```

While Statement

- A while loop statement in C programming language repeatedly executes a target statement as long as a given condition is true

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

While Statement

```c
#include <stdio.h>
int main ()
{
int a = 10;

/* while loop execution */
while( a < 20 )
{
printf("value of a: %d\n", a);
a++;
}

return 0;
}
```

Do-While Statement

- Unlike for and while loops, the do...while loop checks its condition at the bottom of the loop
- A do...while loop is guaranteed to execute at least one time

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

Do-While Statement

- Unlike for and while loops, the do...while loop checks its condition at the bottom of the loop
- A do...while loop is guaranteed to execute at least one time

```
#include <stdio.h>
int main ()
{
        int a = 10;
        /* do loop execution */
        do
        {
                printf("value of a: %d\n", a);
                a = a + 1;
        } while( a < 20 );
        return 0;
}
```

break statement

- When the break statement is encountered inside a loop, the loop is immediately terminated

```
#include <stdio.h>
int main ()
{
        int a = 10;
        /* while loop execution */
        while( a < 20 )
        {
                printf("value of a: %d\n", a);
                a++;
                if( a > 15)
                        break;
        }
        return 0;
}
```

continue statement

- continue statement forces the next iteration of the loop to take place, skipping any code in between

```
#include <stdio.h>
int main ()
{
        int a = 10;
        do
        {
                if( a == 15)
                {
                        a = a + 1;
                        continue;
                }
                printf("value of a: %d\n", a);
                a++;
        } while( a < 20 );
        return 0;
}
```

Nested Loops

The syntax of nested **for** loop

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

Nested Loops

The syntax of nested **while** loop

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

Nested Loops

The syntax of nested **do-while** loop

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

Nested Loops

```
#include <stdio.h>
int main ()
{
        int i, j;
        for(i=2; i<100; i++) {
                for(j=2; j <= (i/j); j++)
                        if(!(i%j)) break;
                if(j > (i/j))
                printf("%d is prime\n", i);
        }
        return 0;
}
```