P2T: C Programming under Linux

Lab 1: Variables, Constants and Operators

Introduction

The purpose of this laboratory is to help you become familiar with the basics of C programming. In this lab we'll explore variables, constants and operators, which have been covered in Lecture 1 and 2 of the C component of P2T: C Programming under Linux course. It's assumed that you will have completed the introductory Linux Lab and are familiar with editing files, compiling code and working on the command line.
(Minimally, compiling a file called "file.c" can be done by the command
gcc file.c
if you don't mind the output being called a.out.
You can make a file called, say "myfile" by using the -o option to gcc, as so:
gcc -o myfile file.c)

Variables and Constants

As you have learned in the Lectures, a variable in C is a name used to refer to some location in memory and is the basic building block of any C program. By using a variable the contents of that memory location can be read from and written to. All values in C, including variables, have a type that defines what kind of value they are or, in the case of a variable, what kind of value can be stored. As all values (and indeed, everything else) are stored as sequences of bits, the type determines how those bits will be interpreted - much as, for example, the letters *aware* have no meaning without knowing if you should interpret them as an English word ("aware", to be conscious of something, or on one's guard), or romaji rendering of a Japanese word ("哀れ", pity).

For example, consider the expression `int x = 5;` in this case, the variable `x` is declared as an integer type using the keyword `int` and a value of `5` is stored at the location represented by `x`.

It is also possible to declare a variable as "constant". This means that once the variable has been assigned its value it cannot be changed. To declare a constant

the `const` keyword can be placed in front of a variable declaration, for example, we can now make `x` constant by writing:

```
const int x = 5;
```

Constant values declared in this fashion are still represented by a variable and have all of the scoping and type rules described in the 1st and 2nd C Lectures. The following basic types are available in the C programming language.

Integer

Integer types are used to represent whole numbers (0, 1, 2, ...). Integers are stored as a direct binary representation of the number and can be signed and unsigned. As mentioned in the lectures integer types can be of different sizes (size in this context means the number of bytes allocated to represent value) denoted by the `short`, `int` and `long` keywords. For more information on the maximum and minimum values that can be stored see the lecture slides. You may also add the keyword `unsigned` to the start of the typename (like `unsigned int`) in order to specify that the contents should only ever be positive numbers.

Floating Point

Floating-point types are used to represent limited precision real numbers, and can be declared using the `float` keyword. In this course only two types of floating point value will be considered: (single-precision) `float` and (double-precision) `double`. Usually a `float` will be represented using 32 bits while a `double` will be represented using 64 bits, the use of "single" and "double" precision referring to the relative size of the types in memory. This, however, is often dependent on the machine architecture being used and care should be taken when making assumptions. See additional course material on Moodle for more discussion of floating point number representation.

Character

Although the character type was invented to represent a single character of text, it can also be used to represent a single byte sized value (i.e. a number) as it is guaranteed to be exactly 8 bits in size by the C standard. Characters are declared using the `char` keyword and encode a single character of text as an ASCII value

(see ASCII character table at http://www.asciitable.com/). Character values are enclosed in the single quotes (`'`), e.g.

```
char myChar = 'x';
```

Note that a `char` is still, strictly, a number - it's just that there's additional machinery that will convert between the character we write and the actual ASCII numeric value corresponding to it.

Void

Essentially void means "no type" and is most often seen in the parameters or return types of functions (covered later) declaring that they do not return a value.

Operators and Generating Output

Operators

The C programming language provides a small set of operators for arithmetic, logical and utility purposes. Operators are built-in methods for manipulating values (in fact, most of them correspond directly to operations that the CPU can perform with a single instruction). You can find complete table of operators (Table 1) in the appendix. You will be familiar with most of the basic mathematical operators such as addition (`+`) or subtraction (`-`); some other operators, such as finding the remainder on division (`%`) use symbols you may not have encountered. Nevertheless, these operators follow the same precedence in C as they do in mathematics. Every C operator has its own precedence with respect to the other operators, governing the order in which the operations are undertaken. You are presented with the table of operators precedence in the appendix (Table 2). The operators at the top have the highest precedence. For example, given

```
int result;
result = 2 + 3 % 4 + 5;
```

the result stored in `result` depends on the order in which we apply the operators. The remainder operator `%` has the highest precedence, and so is applied first, giving the answer to (3%4), which is of course 3 (as 3 is less than 4 anyway). The expression after this operation looks like:

```
result = 2 + 3 + 5;
```

The next step is to add `2`, `3` and `5` which results in the final effect:
`result = 10;`

In this example, all of the literal values in the calculation are of type `int`, and so the result is *also* of that type. In general, if an operator acts on values of a given type, it will return the value also as that type (watch out for this when dividing integer types!).

If you have two values of different (numeric) type applied to an operator, then the calculation will be carried out with all values converted to the "widest" type present; in general, this means the type with the largest range or precision. So, `3 * 4.0` will perform the calculation with `3` represented as a `double`, since `4.0` is already a `double`, and `double`s can store a wider range of values than `int`s.

To force the conversion of a value to a particular type (if it won't happen automatically) for the purposes of the calculation, you can precede the value or variable with the required type enclosed in parentheses. This is called a "cast". For example, if `myInt` is an `int` type variable, we could force it to be used as a `double` in a particular calculation by writing `(double) myInt`. This does not permanently alter the contents or type of the variable itself.

Note that the precedence may be changed by the use of the parentheses `( )` which, as in mathematics, increase the precedence of the part of the calculation they enclose so that it is evaluated before the parts outside. You can nest parentheses in order to give yourself as much fine control of precedence as you like, bearing in mind that over-parenthesizing can make it very hard to read a calculation.

In the C programming language most operators are 'binary', which means that the operator acts on the two values (immediately to the left and right of it). For instance if we wanted to add two numbers together and store the result in a variable you would write:

```
int x = 5 + 6;
```

The increment and decrement operators `++` and `--` are examples of non-infix operators, applying to the single value either immediately to the left (postfix) or immediately to the right (prefix) of the operator. In this course, we will generally use *postfix* versions of the operators in examples, which read the value in a variable *and then* increment it (the prefix versions do this the opposite way around, resulting in calculations differing by 1 if you mix them up).

C also has a number of shorthand operators for carrying out common mathematical tasks. For example, increasing the value stored in a variable by a particular number

might be written as:

```
x = x + 5;
```

but using the shorthand `+=` operator, can be expressed as:

```
x += 5;
```

There are similar shorthand operators for performing the other basic arithmetic operations as updates to a variable.

For a complete list of all the available operators, see Table 1.

Generating Output

In order to be able to interact with a program, it's useful to be able to output values to the terminal (display). The `printf()` and `puts()` functions can be used to output characters to the screen. In order to use these functions, we must use an `#include` statement. We'll cover `#include` in more detail later in the course, but for now understand that the command

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

at the beginning of your C source file tells the compiler to use a file called "stdio.h" (short for "standard I/O") to get definitions for functions we'll be using.
The traditional first program when learning a new language is "Hello, World!", which simply prints the phrase "Hello, World!" to the screen. In C this would look like:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   puts("Hello, World!");
6   return 0;
7 }
```

In this example everything that is contained within the double quotes will be printed on the screen. We call this double quoted list of characters a *string*, the precise details of which will be covered later in the course. For the moment, just remember that when printing a sequence of characters to the screen they need to be

contained in double quotes (`"`). (Explicitly, this is the "double quotes" character, and *not* two single quote characters!)

The function `puts()` in the above example looks like a very neat and simple way of printing the output to the screen. However, it is not as simple as it first appears. Firstly, the `puts()` function automatically appends the `'\n'` character; this is called an "escape character". This denotes a new line character, it is not explicitly printed to the screen, but it starts a new line, so that the two adjacent `puts()` statements will never output the strings on the same line. . In addition, there are other escape characters that can be used, such as:

- `\t` (tab)
- `\v` (vertical tab)
- `\f` (new page)
- `\b` (backspace)
- `\r` (carriage return)

For more control, we can use the `printf()` function. The 'f' stands for "formatted", and as this implies, the function gives us total control of how the string is printed. `'\n'` character is not appended to the output of the `printf()` function. You need to do it manually in order to start a new line. Secondly, `printf()` allows us to "build up" our output from the contents of variables or other values, which `puts()` does not allow. Consider a string: `"We have %d cats"`. The
`puts("We have %d cats");`
would merely output `We have %d cats`. In the case of `printf()` the `%d` indicates that some value has to be displayed at this point in the string. The part of a string that starts with `%` is called the format specifier, with the characters immediately following, in this case '`d`', specifying what type to interpret the value given as when printing it (integer, in this case). So
`printf("We have %d cats", 10);`
prints `We have 10 cats`.

See the table of format specifiers (Table 3) in the appendix.

Format specifiers also permit us to control the details of how a value is represented in text, beyond simply the type of value to use. For example, we can specify the precision and length with which to represent numerical values:

- `%6d` decimal integer at least 6 characters wide

- `%8.2f` float, at least 8 characters wide, two decimal digits
- `%.10s` first 10 characters of a string

While `printf()` function introduces a greater flexibility the `puts()` function is an easy (and safe) way to output simple strings to the screen.

The Main Function, Blocks, and Comments

While we'll cover how to write general functions later on in the course, every C program needs at least one function in order to work: the `main` function. You will have noticed the lines around our Hello World! code (repeated below):

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5   puts("Hello, World!");
6   return 0;
7 }
```

which read `int main(void)`. For now, think of this as a marker for the compiler, indicating that this is where it should start actually executing instructions when your code is run.

The set of instructions which should be followed is delimited by the following pair of braces ({ , }), which form a "block" along with their contents. As covered in the lectures, you can nest blocks within each other - statements inside a block are all "grouped together" in a sense, and variable names declared within a block do not exist outside of it (they have "block scope"). (This also means that you can "shadow" a name in one scope by creating another variable with the same name in an inner scope; the variable in the outer scope is no longer accessible in the inner scope, but obviously it is still stored in memory ("allocated"), and as soon as the inner variable name's scope ends, the name will revert to referring to the former variable. Usually, this effect is unintentional, but you should be aware that it is possible.)

Using this version of the `main` function, you must end your program with a `return` statement, within the block, at the point you want your code to finish. This `return` statement should be followed by an integer value, which will be available outside

the program (this is an "exit code", as covered in the Linux part of the course). For now, use `return 0;` as your return statement, as this is the "standard" way for a program to end.

Finally, it is important to note that you can tell the compiler to "ignore" a line of text in your code by starting the line with `//`. In this way, you can add comments to your code to explain the purpose of something you've written, or to provide attribution and so on. If you put the double backslash in the middle of a line, the compiler will ignore everything after it, until the end of the line. You can comment out an entire section of a file by enclosing the section with `/*` and ending it with `*/`.

Like so:

```
//This is a comment, and will be ignored
int x = 1; //x is set equal to one here.
/* Everything here is a comment
i = 67; This won't happen as it's in a comment!
*/
```

(Generally, try not to add trivial comments like the one above - it's pretty obvious that we're setting x equal to 1!)

Challenge Questions

In this part of the C lab you will have to write two C programs based on what has been learned in the lectures and labs up to this point. In these questions in this lab and later labs, marks are given for code which compiles and runs (even if it doesn't meet the specification), good comments and layout, and also for code which performs the required tasks (for the more complex tasks in later labs, partially working code may get partial marks in this category). You will also debug a piece of C code written for you - locating and fixing issues in it.

Exercise 1 - Variables and Scope

```
int k = 4;

int main(void) {
  int i = 50;
```

```
    unsigned int j = i * 2;

    double k = 1.0

    {
      float i = 5.0;

      printf("The value of i is %3f\n", i);

      k = i * j;
      i *= 6;
    }

    double j;

    i = k + i; //or i += k
    printf("Now, the value of i is %d\n", i);

    return 0;
}
```

Find the three bugs in this code, which will prevent it from compiling without warnings (try to do this without compiling it first, and then use the compiler warnings to help). Fix them, and add comments to the code explaining what the purpose of the fix is.

What do you expect the two `printf` statements to print? Run the code and check your answers.

Add comments to this code, explaining where and when each variable name is in scope, and where it is allocated. Also add a `printf` statement for each "version" of a variable name, to print out the last value it is ever set to, as well as the type that that variable has.

Exercise 2 - Mathematics and Output

Write a small C program in which an constant integer variable is declared and defined.
The code should use this constant in 5 mathematical operations and print the output of these operations to the screen.

The 5 operations which should be carried out are: addition, subtraction, multiplication, division and modulus (remainder). Define as many other variables as you need to do these operations. You should use any variable types that are appropriate, and perform any necessary casts.

Make sure that your output is at least 3 characters wide and is given to two decimal places of precision (if appropriate).

Appendix

Table 1: Table of operators

Operation	Symbol	Example
	Arithmetic Operators	
Addition	+	2 + 5
Subtraction	-	2 - 5
Multiplication	*	2 * 5
Division	/	2 / 5
Remainder	%	2 % 5
	Comparison Operators	
Equality	==	2 == 5 (false), 'A' == 'A' (true)
Greater than	>	2 > 5 (false)
Less than	<	2 < 5 (true)
Greater than or equal to	>=	2 >= 5 (false)
Less than or equal to	<=	2 <= 5 (true)
Not equal	!=	2 != 5 (true)
	Logical Operators	
Logical AND	&&	false && false == false true && true == true false && true == false
Logical OR	||	false || true == true true || false == true false || false == false
Logical NOT	!	!true == false !false == true
	Variable Operators	
Asignment	=	a = 2;
Increase value in variable by amount Equivalently to above:	+= -=, *=, /=, %=, ...	a += 2; <–> a = a + 2;
Increment/ decrement	++/ - -	a++; (a=a+1;), a- -; (a=a-1;)
Number of bytes needed to store variable	sizeof()	sizeof(a); sizeof(double);

Table 2: C Language Operator Precedence Chart

Operator Type	Operator	Associativity
Primary expression operators	( ) [ ] .-> exp++ exp−−	Left-to-right
Unary operators	* & + - ! ∼ ++exp −−exp (typecast) sizeof()	Right-to-Left
Binary Operators	* / % + - < > <= >= == != && ||	Left-to-Right
Asignment Operators	= += -= *= /= %=	Right-to-left
Comma	,	Left-to-right

Table 3: Table of Format Specifiers

Specifier	Description	Example
d	decimal integer in base 0, i.e. integer	5
f	floating point number in decimal representation	3.14
e	floating point number in scietific representation	1.86e6
ld	long decimal integer in base 0, i.e. long int	
lld	long long int	
c	character	'a'
s	string	"Hello"
%	A literal %	%