CS2303 In Class Exercises

Jan 16, 2020

January 17, 2020

Abstract

Today's goal is that you know the primitive operations in C, and how to create the prototype and invocation of your own functions.

Name and WPI email: Kush Shah kshah2@wpi.edu

1 Primitive operations in C

C provides several operations:
arithmetic: +, -, *, /, %
assignment: =
augmented assignment: +=, -=, *=, /=, %=, &=, |=, ˆ=, <<=, >>=
bitwise logic: ˜, &, |, ˆ
bitwise shifts: <<, >>
boolean logic: !, &&, ||
conditional evaluation: ? :
equality testing: ==, !=
calling functions: ( )
increment and decrement: ++, −−
member selection: ., − >
object size: sizeof
order relations: <, <=, >, >=
reference and dereference: &, *, [ ]
sequencing: ,
subexpression grouping: ( )
type conversion: (typename)

Some operations are unary, meaning there is one operand. Thus we can negate a constant, e.g., taking the negative of 3, thus: -3.
We can also negate a variable, if it is numeric (with good practice), thus: -x.
Other unary operations include taking the logical inverse. We can invert a constant: !true. We can invert a variable, if it is logical (with good practice): !x.
Some operations are binary, meaning there are two operands. Thus we can add two numbers, e.g., int z = x+y;

We saw member selection last time, when specifying a field from a struct. Then we used the period. There is another context when the $->$ is correct for field selection: when a variable is a pointer, we use $->$ as in the following:

```
typedef struct
{
    double x;
    double y;
}Complex;

Complex Z1;
Complex* Z1P = &Z1;
Z1P->x = 4.2;
```

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Examples:

Task	Operator	Example	Description
Give an example use of	+	a+b	Addition
Give an example use of	-	a-b	subtraction
Give an example use of	%	a%b	Modulo
Give an example use of	+=	x+=3;	Update by adding, x=x+3;
Give an example use of	-=	a-=b	Update by subtracting, a=a+b

Some operations produce a value of the same datatype as the operands, but not always. Consider the proposition $x < 3$. We insist that x is a numeric type. The result can have the value true or false; it is a logical type, called bool. We obtain this logical type by
#include <stdbool.h>

The result depends upon the value associated with the variable x.

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Examples:

Task	Operation	Is it valid?	Datatype	Answer
Compare two numbers to obtain a logical	5 > 6	yes	bool	false
Obtain the remainder after dividing one number by another	12%5	yes	integer	2
Check whether the third bit from the righthand end is 1	(5 & 0b100)==0	yes	bool	true
Check whether the variable is true	x	yes	bool	depends
Check whether the variables contain different logical values	0b10101010 ˆ0b01110111	yes	numeric	0b11011101
Flip all the bits in the variable	˜0b10100111	yes	numeric	0b01011000
shorthand x = x-y	-=	yes	numeric	number
shorthand for x = x/y	/=	yes	numeric	number
Shorthand for choice	x?y:z	yes	of y and z	if x is true then y else z
checking equality	==	yes	bool	true/false
assignment	=	yes	depends	does not return

2 User-defined Operations, that is, Functions

Developers in C write functions. Functions are operations that have from 0 to many parameters. Similarly to how user-defined datatypes make use an aggregation of primitive datatypes, user-defined operations make use of an aggregation of primitive operations, including statements, which we will describe soon. Primitive operations have answers; functions are permitted but not required to have answers. Just as there is a specified syntax for defining a user-defined datatype, there is a specified syntax for defining a user-defined operation.

First, the datatype of the answer, if any is given. If there is no answer given, “void” is used.
Then, the function has a name.
The name of the function is followed by parentheses.
The parentheses may enclose ”void” or zero or more parameter specifications, separated by commas.
A parameter specification in a function definition has two parts, the datatype and the variable name.
Next a block of statements appears. The block is defined by a pair of curly braces: {}.
Within the pair of curly braces, statements appear. Statements will be discussed soon. For now, examine the example function.

Example:

```
bool compareTwoStrings(char* first, char* second)
{
    bool areEqual = true;
    if (strlen(first)!= strlen(second))
    {
        areEqual = false;
    }
   else
   {
       int theLength = strlen(first);
       char* temp1 = first;
       char* temp2 = second;
       for(int i = 0; i< theLength; i++)
       {
           if((*temp1+i) != (*temp2+i))
           {
                areEqual = false;
           }
       }
    }
    return areEqual;
}
```

Fill in the remaining spots in the table. If you have any question about how to do it, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Exercise:

Question	Answer
How many parameters does the example function have?	2
Does the function return any data?	yes
If the function returns any data, what datatype is it?	bool (true/false)
Does the example contain anything before or after the function?	No

We have looked at one example of a function definition. The existence of a function implies three manifestations: definition, prototype and invocation.

2.1 Function Definition

We have seen an example definition. The return type (if any), the name, the parentheses, the parameters, and the block of statements were all present.

2.2 Function Invocation

Though functions can be defined without ever being used, we have the idea of using a function.

To use the function above, we must provide a value for each of the parameters. An example of this is:

```
bool compOk = compareTwoStrings(aString, add1String);
```

Notice that the variable compOk has a datatype of bool.
It is the same as the datatype returned by the function.
The context of the invocation is that the two variables, aString and add1String, are declared and initialized before the invocation occurs.

```
char aString[13] = "My name is: ";
char add1String[13] = "Nz!obnf!jt;!";
bool compOk = compareTwoStrings(aString, add1String);
```

There is an interesting process called “binding”, whereby the values of the arguments, aString and add1String in this case, are set into correspondence with the parameters of the function, first and second in this case. We defer this discussion for now.

2.3 Function Prototypes

The function is also manifest as a function prototype.
Good practice includes that the compiler should have processed the function prototype prior to encountering a function invocation.
There is a systematic way to obtain this outcome, discussed later.
It is easy to derive a function prototype from a function definition if that is already written (not always the case).
For our example, it is:

```
bool compareTwoStrings(char* first, char* second);
```

Note that the first line of the function definition has been copied, and a semicolon affixed to its right.

The variable names are optional. Thus, it is fine for a prototype to list the datatypes of the parameters.

```
bool compareTwoStrings(char*, char*);
```

3 Organizing the files of your project into .c files and .h files.

To ensure that the compiler encounters the relevant function prototype prior to encountering the corresponding invocation, we follow this practice:

There is a filename, with the suffix ".c", in which the function definition is found. Suppose that is called "Function1.c".
Then there should also be a file named "Function1.h", and early within the file "Function1.c", the line
#include "Function1.h"
should appear.

Moreover, there is a file in which the function invocation occurs. Let that be "Function2.c". Then early within the file "Function2.c", the line
#include "Function1.h"
should appear.

Some care is taken that so-called include files, also called header files, do not recursively include each other. In the example below, a defined constant, "PRODUCTION_H_", occurs. The constant becomes defined upon execution of the line starting #define.

```
#ifndef PRODUCTION_H_
#define PRODUCTION_H_
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

bool production();
bool compareTwoStrings(char*, char*);

#endif /* PRODUCTION_H_ */
```

Once this constant has become defined, the corresponding header file has been incorporated into the code, and will not be processed again.

4 The function that starts a program, main.

Even though we have not studied many of the statement types in C, nevertheless we can detect certain features in the main function.
In this course, we will use the following function for our main function.

```
int main(void)
{
puts("!!!Hello Class: In Test-Driven Development, so in this course, run the test suite, and
if(testSuite())
{
bool productionSucceeded = production();
if(productionSucceeded)
{
puts("Production succeeded; this does not imply it got the right answer.");
}
else
{
puts("Attempted production but experienced a problem.");
}
}
return EXIT_SUCCESS;
}
```

This main function returns an integer. This is typical.
Note that there are no parameters. Later we will add parameters. These will be for processing command line arguments.
Note that the return value is specified with all capital letters. It is a defined constant (equal to zero).

There are puts() statements. These put strings onto the console, for user output.
A significant idea is implemented in this code.
First, a function named testSuite is invoked.
The function invocation occurs within an if statement. We will discuss if statements later.
Only if the testSuite returns true will the code making up the body of the if statement be executed.
The code guarded by the testSuite function refers to production.
Production code is only attempted after the testSuite has succeeded.
We will make the following minor changes to the main function:
1. Add parameters for receiving a command line, passing them on to production. later.
2. Code in C++, later.

The homework grading rubrics refer to your solution looking like starter code.
This implies you should not change the main function. You should use it as provided.
A sizable portion of your homework grade comes from leaving this code as provided.
Please resist any urge to change this code. Changes will result in loss of points.

4.1 Command line arguments

Developers produce executable modules, sometimes called applications, or apps.
When invoking, or "calling" an executable from the operating system's prompt, a user may provide additional data, called "command line arguments".
We will modify the main function to be able to process command line arguments.

```
int main(int argc, char* argv[])
{
    puts("!!!Hello Class: In Test-Driven Development, so in this course, run the test suite
    if(testSuite())
    {
        bool productionSucceeded = production(argc, argv);
        if(productionSucceeded)
        {
            puts("Production succeeded; this does not imply it got the right answer.");
        }
        else
        {
            puts("Attempted production but experienced a problem.");
        }
    }
    return EXIT_SUCCESS;
}
```

Note that the main function now has two parameters, an integer giving the number of command line arguments; you can think of it as argument count. The second parameter is of datatype pointer to character, and is an array of these. You can think of this variable as "argument vector".
When an executable is invoked from the command line, the command line can be followed by character strings. These character strings may contain digits (ie., they represent numbers), or alphanumeric characters that do not represent numbers.

These variables are of interest to the production code. In the invocation of production, the variable names appear.

If you have any question about how to do any exercise, please ask. In subsequent lectures this material will be considered background, so please take this opportunity to get it clarified.

Exercise:

1. Is it normal to have datatypes and variable names in the function definition?

   Yes

2. Is it normal to have variable names but not datatypes in the function invocation?

   Yes

3. Is it normal to have datatypes but not variable names in the function prototype?

yes although it will work with the variable names

Exercises Combining Yesterday with Today:

1. Write a function prototype for a function that returns a double, and takes an integer and then a float as parameters.

```
double fPrototype(int , float);
```

2. Define a (user-defined) datatype that aggregates three integers with two doubles.

```
typedef struct
{
int x = 0;
int y = 1;
int z = 2;
double = 3.0;
double = 4.0;} userDefinedDataType
```

3. Write a function prototype for a function that returns a pointer to your user-defined type above.

```
userDefinedDataType* pointerReturn(){
userDefinedDataType* pointer = &userDefinedDataType;
return pointer;
}
```

4. Write a function invocation for the function above.

```
pointerReturn();
```

5. Write a function prototype for a function that uses two parameters, the first is a pointer to a character, the second is an integer.

```
void myF(char*, int);
```

6. Write a function invocation for the function above.

```
myF(charPointer, myInt);
```

Definitions of user-defined functions are normally placed in header files.

For the function whose prototype and invocation you just wrote, name the header file containing the user-defined datatype, and explain where the #include statement would be put.