Arrays and Functions

This lecture introduces three logical operators and covers two additional features of the C language: arrays and functions.
After this lecture you should be able to:

- *use the logical and (&&), or (||), and not (!) operators in expressions*
- *declare an array*
- *write expressions that reference an array element*
- *write C code that iterates over all of the elements of an array using either an element count of a terminator array value*
- *declare a function including function argument types and return values*
- *give the values of function arguments for a given function call*
- *define the terms array, index, function, argument, and return value*

Logical Operators

The logical operator '!' is the logical negation operator. It is a unary operator – it only operates on the value on its right. The result of the logical negation operator is the value 1 if the value on the right is zero and 0 otherwise. Logical negation has the highest precedence of all operators discussed so far.

The logical operators && and || result in the logical 'and' and 'or' of the values on their left and right. The result of a logical 'and' is 1 if both values are non-zero and 0 otherwise. The result of a logical or is 0 if both values are zero and 1 otherwise. Both operators have lower precedence than the other operators discussed so far and 'and' has a higher precedence than 'or'.

Exercise: What are the values of the following expressions?

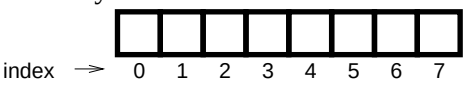
```
! ( 3 || 1 + 1 )

  ! 3 && 1

 0 || ( 1 > 0 ) && 1
```

Arrays

An array is an area of the computer's memory that can store a number of different values at the same time. Each individual element of the array is accessed by using an integer value called an index. We can picture an array as a row of identical containers each of which can hold one value. For example, an 8-element array could be drawn as follows:



Arrays are declared by putting the number of elements (which must be a constant) in brackets after the variable name. For example, the following statement declares an array that can store four character values:

```
char x [4] ;
```

Exercise: Give the declaration for an array variable called `tab` of 256 integers.

Exercise: How many array elements are declared in the following declaration: `int x['1'] ; ?`

An individual element of an array can be referenced by using the array name followed by the index value enclosed in brackets. Allowable index values run from zero to the number of elements in the array minus one. For example, to increment the first element of the above array we could write:

```
x[0] = x[0] + 1 ;
```

Exercise: Write an expression whose value is two times the value of the second element of the array x.

Exercise: Write an expression whose value is non-zero if the first and last elements of the array x are equal.

The value of the index can be any expression, not simply a constant. For example, the following statements have the same effect as the previous example:

```
i = 0 ;
x[i] = x[i] + 1 ;
```

Iterating over Arrays

A very common programming task is to perform the same operation on all of the elements in an array. This can be done by using a `while`-loop, and an index variable which is incremented in the loop. For example, the following statements find the sum of the values in an array of four `int`s:

```
int i, sum, x[4] ;
...
sum = 0 ;
i = 0 ;
while ( i < 4 ) {
   sum = sum + x[i] ;
   i = i + 1 ;
}
...
```

Exercise: How many times will the loop be executed? What would happen if the order of the two statements within the loop was interchanged?

Sometimes it's more convenient to mark the end of an array with a special value rather than explicitly keeping track of the size of the array. For example, we might use a negative value to mark the end of an array that contained only positive values.

Exercise: What statements must be used inside the loop in the following code to compute the sum of the values in the array x if a negative value is used to mark the end of the array?

```
int i, sum, x[4] ;
...
sum = 0 ;
i = 0 ;
while ( x[i] > 0 ) {
...
}
...
```

Functions

The concept of a function as an operation that maps one set of values into another set of values should be familiar from mathematics. A typical example would be a trigonometric function such as $cos()$.

A similar construct is available in C. For example, the function `isdigit()` can be used to determine whether a value corresponds to one of the codes between '0' and '9'. In this case the function `isdigit()` has a non-zero value if the value being tested is a digit and 0 otherwise. The value to be tested is supplied to the function by enclosing it parentheses immediately following the function name. The value passed to the function is called the "argument" and the value of the function is called the "return value."

Functions are used in expressions. For example, the following expression has the value 0:

```
isdigit( 'x' )
```

while the following function has a non-zero value:

```
isdigit( '0' )
```

Exercise: What are the values of the following expressions?

```
( isdigit( '5' ) == 0 ) * 5

( isdigit( ' ' ) == 0 ) - 1
```

Declaring Functions

In addition to the functions that are included with the C compiler, you can declare your own functions. In general, a function is declared as follows:

```
<type> <name> (<argument declarations>)
{
 <statements>
 return <value> ;
}
```

where:

- `<type>` is `char` or `int` and is the type that the function will return when it's used in an expression

- `<name>` is the name of the function (e.g. `isdigit`)
- `<argument declarations>` declare local variables that are used within the function (see below) and are initialized each time the function is invoked in an expression.
- the `return` statement terminates execution of the function and indentifies the value to be used as the value of the function in the invoking expression

For example, we could have declared our own version of the `isdigit()` function as follows:

```
int isdigit ( char c )
{
 int i ;
 i = ( c >= '0' ) && ( c <= '9' ) ;
 return i ;
}
```

Function Calls

When a function is used ("invoked") in an expression the computer first initializes the local variables in the argument list of the function declaration with the values of the corresponding argument(s) in the invoking expression.

Then the computer executes the statements in the function. until the return statement is executed. The computer then stops executing statements in the function and substitutes the value given in the return statement in place of the function in the invoking expression.

For example, in the expression:

```
isdigit ( ' ' ) == 0
```

the computer sets the value of the variable `c` in the `isdigit()` function to 32 (the ASCII value of a space character) and executes the statements in the `isdigit()` function until it reaches the `return` statement. At that point the `isdigit()` function call in the expression is replaced by the value in the `return` statement (0). We say that the function "returns" the value 0.

Like variables, functions should be declared before they are used.

`void` Functions and Arguments

In some cases a function need not return a value. In this case the function may be declared as returning a type `void`. In other cases a function may not need any arguments. In this case the argument declaration may be the word `void`. For example, a function that returns the next keystroke might be declared as:

```
char getche(void)
{
   ...
}
```

while a function that prints a character on the screen might be declared as:

```
void putch(char c)
{
   ...
}
```

It is also possible to declare functions that take a variable number of arguments. A common example is the `printf()` function. We will not define these types of functions in this course.