

# Introduction to C

*This lecture reviews conversion between decimal, binary and hexadecimal numbers and provides an introduction to the features of the programming language 'C' that you'll need to do the labs. We will skip some important parts of the C language including: pointers, structures, floating point variables, and the switch statement.*

*After this lecture you should be able to:*

- *convert a decimal value to and from binary, convert a binary value to and from hexadecimal*
- *convert between bus signal levels and a numeric value.*
- *explain the following terms: variable, statement, operator, precedence*
- *evaluate expressions involving integer variables, constants, and the operators described in this lecture*
- *write a simple C program including a main() function declaration, integer variable declarations, and the following statements: expression, if/else, while, and for*
- *predict the result of executing such a program*
- *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 or a terminator array value*
- *declare a function including function argument types and return values*
- *give the values of function parameters for a given function call*
- *define the terms array, index, function, argument, and return value*

## Number Systems

We commonly use decimal notation to express the value of a number. However, in digital logic hardware (including computers) numbers are represented with two-valued (binary) values. To be able to deal with the hardware representation of numbers we need to be able to convert between the decimal and binary representation of numbers. Note that *the value of a number does not change when we express it in a different base.*

## Binary Representation

In the decimal number system each digit position represents a different power of ten. The rightmost digit gives the number of multiples of  $10^0$  in the number, the next digit gives the multiples of  $10^1$ , and so on. For example,  $105 = 5 \times 10^0 + 0 \times 10^1 + 1 \times 10^2$ .

Binary numbers work in the same way, but since only two values are possible for each digit, each po-

sition represents a different power of two. The rightmost digit gives the number of multiples of  $2^0$ , the next digit gives the multiples of  $2^1$ , and so on. For example,  $110 = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = 0 \times 1 + 1 \times 2 + 1 \times 4 = 6_{(\text{base } 10)}$ .

Table 1 shows the decimal values of powers of 2 for exponents from 0 to 15:

To convert from binary to decimal we just need to add up the powers of two corresponding to the bit positions that are '1's.

**Exercise 4:** Convert the binary number 1001 0110 to a decimal number.

To convert from decimal to binary it is necessary to find the appropriate combination of powers of two that will add up to the desired decimal number. This is done by repeatedly subtracting the largest possible power of two until the remaining value is zero.

For example, to convert the decimal value 35, we find the largest power of two less than or equal to

exponent	value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768

Table 1: Powers of 2

decimal	binary	hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 2: Hexadecimal Digits

35:  $32 (2^5)$ . The remainder is  $35 - 32 = 3$ . The next largest power of that can be subtracted is  $2^1 = 2$ . Subtracting this value leaves  $3 - 2 = 1$ . The next possible power of two is  $2^0 = 1$ . Subtracting this value leaves 0. Therefore  $35 = 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$  which is 10011 in binary.

**Exercise 5:** Convert the decimal number 86 to a binary number.

## Hexadecimal Representation

Binary numbers are too verbose for many purposes so we often use hexadecimal (base 16) numbers. Hex numbers are less verbose but also easier to convert to binary. Since the base is 16, we need 16 different digits. We use the digits 0 to 9 and the letters A to F. Table 2 shows the the 16 hexadecimal digits and the corresponding values in decimal and binary.

To convert from binary to hex we group the binary digits into groups of 4 starting from the least significant (rightmost) digit. Then we just look up the corresponding hexadecimal value in the table.

**Exercise 6:** Convert the binary number 1101000 to hexadecimal.

Hex notation is used because the conversion between hex and binary is much simpler than between decimal and binary. This is because each hex digit represents exactly 4 bits.

To convert a decimal number to hex you can first convert the number to binary and then group the bits

into groups of 4 bits starting from the right. These 4-bit patterns are then easily converted to hex digits.

Similarly, to convert a hex number back to decimal first convert it to a binary number and find the corresponding decimal value.

**Exercise 7:** What are the binary and decimal representations of the hexadecimal value 3F?

## Binary Numbers and Logic Levels

Computers represent binary values by using two voltages. For example, one way is to use 0 volts to represent a binary '0' and 5 volts to represent a binary '1'. These voltages, sometimes also called low (L) and high (H), are called logic levels.

A binary number can be represented by a collection of 8 (for a char) or 16 (for an int) signals<sup>1</sup>. Each signal represents a particular bit of a binary number. A group of related signals is called a *bus*.

**Exercise 8:** The connector used between a PC and a printer has 25 signal pins. Pin numbers 9 to 2<sup>2</sup> carry signals generated by the PC that provide the printer with the ASCII value of the character that should be printed. You measure the following voltages: 0,5,0,0, 5,0,0,0 on the signal pins. What character is the printer trying to print?

<sup>1</sup>A signal is a voltage that carries information.

<sup>2</sup>in order from most significant to least significant bit.

Note that the opposite, “active-low” convention for logic levels (H for 0 and L for 1) is also often used.

## The Structure of a C Program

Here’s a simple C program:

```
main ()
{
    int i ;

    i = 0 ;
    while ( i < 4 ) {
        printf ( "%d\n", i ) ;
        i = i + 1 ;
    }
}
```

If you compile and then execute this program the computer will print the integers between 0 and 3.

The first two lines are required by all C programs and define a function called `main`. The braces (“curly brackets”) on the second line and the last line mark the start and end of the function.

The computer executes the statements in order, starting with the first statement in the function called `main` and ending after the last statement in the function `main` is executed.

The line `int i ;` is a statement that declares an integer variable called `i`. A variable is an area of the computer’s memory that is used to store numbers.

The line `i = 0 ;` is another statement – an expression statement. This particular expression statement sets the value of the variable `i` to zero.

The line `while ( i < 4 )` is yet another type of statement, a `while` statement. The computer repeatedly evaluates the expression within the parentheses and executes the statements between the braces while this expression is true.

The line starting with `printf()` is another expression statement. In this case the expression statement contains only the name of another function which is to be executed. This particular function (`printf`) causes the value of the variable `i` to be printed on the screen.

The next line is another expression statement that increments the value of the variable `i`.

**Exercise 9:** What do you think would be printed out if the order of the two statements within the `while` ‘loop’ was interchanged?

The following sections give more detailed information about variables, expression statements, and the two basic types of control statements: `if/else` and `while` statements.

## Variable Declarations

We will only need to use integer variables. These come in three sizes called `char`, `int` and `long`. Each size can be signed or unsigned and can take on an implementation-specific range of values. The values shown in the table below are for the version of Turbo C that we will use in the lab.

variable type	size (bits)	unsigned range	signed range
<code>char</code>	8	0 – 255	-128 – 127
<code>int</code>	16	0 – 65536	-32768 – 32767
<code>long</code>	32	0 – 2 <sup>32</sup>	-2 <sup>31</sup> – 2 <sup>31</sup> – 1 <sup>3</sup>

Variables have to be “declared” at the beginning of the function where they are used. A variable is only in existence while statements in that function are being executed and the values of variables are forgotten when the function terminates.

Here are some examples of variable declarations:

```
int day, month, year ;

unsigned int cycles ;

char ppc ;
```

**Exercise 10:** What is the possible range of values for each of these variables?

The first character of a variable or function name must be a letter or underscore, followed by other letters, underscores or digits. Case is significant: `i` and `I` are two different variables.

There are certain reserved names that can’t be used for variable names. Keywords such as `if`, `while`, `int`, etc. are reserved. A list of reserved keywords is available from the Turbo C help index screen (type `F1` twice and select “Keywords”).

**Exercise 11:** Make up three valid and three invalid variable names.

Since computers can only work with numbers, letters are converted to numbers when they are read into the computer and converted back to characters when

they are displayed. The standard mapping of characters to numbers is called “ASCII.” For example, the number used to represent the letter ‘a’ is 97 and the number for the space character is 32. ASCII encoding only defines values from 0 to 127 so a char variable is typically used to store the code for a character.

## Constants

Constants are similar to variables except that their values cannot be changed. Integer constants can be expressed as a number (e.g. 12). We can also specify the ASCII value of a particular character by surrounding that character with single quotes (e.g. ‘e’).

### Hexadecimal and Octal Constants in C

We can use hexadecimal notation in C programs by prefixing the constant using the characters ‘0x’. The hex digits may be in upper or lower case. For example, the constant 0x21 has the value 33 (decimal).

C also has octal (base 8) constants which are denoted by prefixing a number with a zero. For example the constant 010 has the value 8 (decimal). Octal notation is no longer widely used.

Unfortunately, C does not have binary constants.

## Expressions

Expressions describe how new values are computed from the values of existing variables and constants. Expressions are built up from variables, constants and operators.

Operators are characters denoting operations to be performed on variables such as addition, comparison, and assignment.

For now, we will only study a few of the operators available in C. The following is a list of the most common operators and examples of expressions using them:

**Exercise 12:** What are the values of each of the following expressions (the ASCII value for a space is 32)?

- the arithmetic operators, \* / + - , result in the product, quotient, sum and difference of the values on the left and right. As usual, multiplication and division are performed before (“have

higher precedence than”) addition and subtraction. Otherwise operations are done left to right. The result of an integer division is rounded down.

$$1 + 3 * 5 / 4$$

- parentheses are not really operators but are used to change the order in which parts of an expression are evaluated

$$( 2 + ' ' ) * 3$$

- comparison operators (< > >= <= == !=) compare the value on the left and right of the operator. The result is the value 0 if the comparison is false, 1 if it is true. Comparison operators have lower precedence than the operators described above.

$$( -1 < ( 3 != 2 ) ) * ( 5 > 1 )$$

- the assignment operator, =, assigns the value of the expression on the right to the variable on left. The result is the value that is assigned. Assignment operators have lower precedence than the operators given above.

$$b = 5 \\ c = b - ( a = 3 )$$

- 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.

$$!( a = 0 ) \\ !( 2+2 )$$

- 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’.

```
3 * ( 1 && 0 ) + ( 2 || 1 )
```

- the bitwise logical operators operate bit-by-bit on the binary representation of their operands.

The bitwise complement operator, `~`, is a unary operator similar to the logical negation operator and has the same precedence. However, it inverts the values of the bits in the binary representation of the operand. If a bit is 0, the bitwise negation sets that bit in the result to 1 and vice versa. For example, `~ 2` has the value 1 (2 is 10 and 1 is 01 in binary).

The bitwise ‘and’, ‘exclusive-or’<sup>4</sup> and ‘or’ operators are `&`, `^`, and `|`. They result in the operation being applied to the *bits* in the binary representations of their operands. Both operators have lower precedence than the comparison operators but higher precedence than the logical operators. The bitwise ‘and’ has a higher precedence than the bitwise ‘or’.

```
( 7 ^ 5 ) | 5
0xAA & 15
```

Note that there is an important difference between logical and bitwise logical operators. For example, the expression `5 && 2` is 1 while `5 & 2` is 0.

**Exercise 13:** Why?

- the bit shift operators shift the bits in the left operand left (`<<`) or right (`>>`) by the number of bit positions given by the second operand. For example, `x>>2` shifts bits in the binary representation of `x` right by 2 bits. If `x` had the value `0x4` (0100 binary), the value of `x>>2` is `0x1` (0001).

```
0x4 >> 2
0x4 << 1
```

---

<sup>4</sup>The exclusive or operator gives 0 if the two bits are the same and 1 otherwise.

**Exercise 14:** How does shifting the bits in a number left by 1 position affect the value of a number? How about shifting them right?

Each size of integer can only hold a limited number of bits (8, 16 or 32). Thus each shift causes one bit to be shifted “off the end” and another bit to be shifted in at the other end. The bit that is shifted out disappears. For unsigned integers a zero is always shifted in. For right shifts on signed integers the value of the most significant bit is (usually) duplicated and shifted in.

- the unary operator `++` is used in C to increment a variable by one. If the operator is placed before variable (e.g. `++i`) the variable is incremented before its value is used in the expression. If the operator is used after the operator (e.g. `i++`) the *initial* value of the variable is used in the expression. This operator has a lower precedence than any other unary operator but a higher precedence than non-unary operators.

```
++n - 1
2 * n++
```

The `--` operator is used in the same way as `++` but it decrements the variable by 1 instead of incrementing it.

## Summary of Operator Precedence

Table 3 shows the operators that have been studied in this course in order of precedence. If more than one operator of the same precedence could be evaluated, the operator on the left takes precedence except for the unary and assignment operators where the grouping is right to left.

A complete list of operators and their precedence is available from the Turbo C help screen (type F1 twice and select “Precedence”).

## Statements

### Expression Statement

The most common C statement is an expression statement. It is simply an expression terminated by a

operator	symbol(s)
parentheses, array index, function call	( ) [ ] name()
unary operators	! ~
multiplicative	* /
additive	+ -
relational	< > <= >=
equality	== !=
bitwise and	&
bitwise exclusive or	^
bitwise OR	
logical AND	&&
logical OR	
assignment	=

Table 3: Operator Precedence

semicolon. For example:

```
i = i + 1 ;
```

**Exercise 15:** Which of the following are valid statements?

```
0;
1+1;
-;
```

### if/else Statement

This statement causes only one of two groups of statements to be executed depending on the value of an expression. If the value of the expression is non-zero the statements in the if portion are executed, otherwise the statements in the else portion (if any) are executed.

```
if ( <expression> ) {
  <statements>
} else {
  <statements>
}
```

**Exercise 16:** What is the value of 'i' after executing the following statements?

```
b = 4 ;
if ( b * b == 4 ) {
  i = 1 ;
} else {
  i = 0 ;
}
```

### while statement

As explained previously, the while statement is used to repeatedly execute a group of statements while the controlling expression is non-zero.

```
while ( <expression> ) {
  <statements>
}
```

**Exercise 17:** What are the values of b as the following statements are executed? How many times is the statement inside the 'while loop' executed? How is this related to the value 16? What if the loop condition was  $(b \leq 65536)$ ? ( $65536 = 2^{16}$ )

```
b = 2 ;
while ( b <= 16 ) {
  b = b * 2 ;
}
```

## Arrays

An array is an area of the computer's memory that can store several values of the same type at the same time. We can picture an array as a row of identical containers each of which can hold one value. Each element of the array is accessed by using an integer value called an index.

Arrays are declared by putting the number of elements (which must be a constant) in brackets<sup>5</sup>, after the variable name. For example, the following statement declares an array that can store four character values:

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

**Exercise 18:** Give the declaration for an array variable called tab of 256 integers.

**Exercise 19:** How many array elements are declared in the following declaration: `int x[ ' ' ] ;` ?

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:

<sup>5</sup>Parentheses: ( ), Brackets: [], Braces: {}

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

**Exercise 20:** Write an expression whose value is two times the value of the second element of the array `x`.

**Exercise 21:** 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 ints:

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

**Exercise 22:** 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 23:** 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 ) {
    ...
}
...
```

## The for Statement

Iterative loops are used so often that most computer languages have special language constructs to implement them. In C this is done with a statement called a `for` loop.

Every iterative loop must have three parts: initialization of the loop control variable (done once before the start of the loop), testing of the loop variable before each execution of the statements in the loop, and the increment (or other change) of the loop variable at the end of the loop. The `for` loop allows us to define a loop more compactly by specify each of these actions in one statement. Here's an example of a `for` loop:

```
for ( i=0 ; i<10 ; i++ ) {
    sum = sum + x[i] ;
}
```

The `for` loop is similar to a `while` loop except that it contains three expressions in parentheses after the `for` keyword. The first expression is evaluated immediately before the start of the loop. The second expression is evaluated before each iteration of the loop and terminates the loop if it evaluates to zero. The final expression is evaluated at the end of each iteration of the loop. The above code is equivalent to the following:

```
i=0 ;
while ( i<10 ) {
    sum = sum + x[i] ;
    i=i+1 ;
}
```

**Exercise 24:** Use a `for` loop to print all the powers of 2 with values between 1 and 256.

## Nested Loops

It's often necessary to use one loop inside another. These are called *nested* loops. A different loop variable needs to be used to control each loop. For example, the following code would print all the possible combinations of two dice:

```
for ( i=1 ; i<=6 ; i++ ) {
    for ( j=1 ; j<=6 ; j++ ) {
        printf ( "%d and %d\n", i, j ) ;
    }
}
```

## 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 in 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 25:** What are the values of the following expressions?

```
( isdigit( '5' ) == 0 ) * 5  
( isdigit( ' ' ) == 0 ) - 1
```

## Why Use Functions?

A C program consists mainly of function declarations. A simple program may only declare a function called *main()*<sup>6</sup>, but non-trivial programs will contain many function declarations.

The main purpose of functions is to break up a long program into small parts which are easier to understand. Each function should perform a relatively simple task which can be easily understood, programmed and tested. A rule of thumb is that a function should fit on one screen (about 25 lines). If the function is longer than this it probably can (and

---

<sup>6</sup>Parentheses used after a name are used to indicate that the identifier is a function rather than a variable.

should be) broken down into two or more simpler functions.

A typical program consists of a hierarchy with a main function which uses (calls) other functions; these functions in turn call other functions and so on.

A function communicates with other functions through the values of its arguments and through the values it returns. This isolation between functions makes it easier to debug a function and to make sure that it behaves as desired. It also makes it more likely that the same function can be re-used in a future project.

Another reason to create a function is when the same (or similar) operations need to be performed on different variables. For example, if we had to compute the lengths of various strings in different places in a program, we could write a function that took a string (character array) argument and returned an integer. Instead of writing similar code in each place where we need to find the length of the string we could just write the function once and call it in each place where the length of a string has to be computed.

## Local Variables

Variables declared inside a function are only “visible” within that function. A variable of the same name in another function is considered to be a different variable. Thus you cannot refer to a variable declared in one function from another function. For this reason they are known as “local” variables.

The value of a local variable is undefined when a function begins unless that variable is a parameter or it is explicitly initialized. The value of a local variable is lost each time the function returns.

To pass values between functions you need to use function arguments and return values.

## Function Declarations

Here’s an example of how the *isdigit()* function might be declared:

```
/* return 1 if c is a digit */  
  
int isdigit ( char c )  
{  
    int v ;
```

```

if ( c >= '0' && c <= '9' ) {
    v = 1 ;
} else {
    v = 0 ;
}

return v ;
}

```

A function declaration has four parts:

- The return type. This specifies the type (e.g. `int` or `char`) of the value returned by the function when it is used in an expression. If the function does not return any value it should be declared to be of type `void`.
- The name of the function. This is the name used to refer to a function when it is used in an expression.
- The parameter list. This list specifies a number of local variables that are used to pass values to the function. The list is surrounded by parentheses and the individual variable declarations are separated by commas (unlike a normal variable declaration where declarations are terminated with semicolons).
- The body of the function. This is a sequence of statements surrounded by braces. Each statement is terminated by a semicolon.

**Exercise 26:** Write the first 3 parts for a declaration of a function named `cdist` that returns an `int` and takes two `char` arguments called `x` and `y`.

## Flow of Control<sup>7</sup>

To use a function you use its name in an expression. When the computer executes that expression (that is, evaluates its value) the computer transfers control to the first statement in the function. To “call” a function is to transfer control to it – i.e. use it in an expression.

When a return statement is executed control returns to back to the place where that function was called.

<sup>7</sup>“Flow of control” is the sequence in which statements in a program are executed.

Each function also contains an implicit return statement after the last statement in the function and so functions also return after the last statement in the function is executed.

## Function Parameters and Arguments

When a function is declared, a number of variable are declared in parentheses immediately following the function name. These variables are called *parameters* (the arguments are the values used in the calling expression). Within the function these variables are treated like other variables. The only difference is that each time the function is called the values of the parameters are initialized to the values of the corresponding arguments in the function call.

There must be a one-to-one correspondence between the arguments used in a function call and the parameters in the function declaration. For example, if we declare a function with one character parameter, then when that function is used it must always have one character-type argument. Similarly if we declare a function with two `int` and one `char` parameters, then it must always be called with two `int` and one `char` arguments.

The way a computer passes arguments to a function is to copy the values of the arguments in a special set of memory locations called the *stack*. When the function executes it removes the values from the stack and assigns them to the parameters. Because the values or the arguments were copied to the parameters, this means that changes to the function parameters have no effect on the values in the calling expression.

This copying operation is called “passing by value.”

An exception to the “passing by value” rule is the case of arrays. To improve efficiency, array arguments are not copied to the corresponding array parameter. Instead the location in memory of the array is passed to the function. This is called “passing by reference.” Changes to an array parameter in a function will change the corresponding array argument in the calling function.

## Return Values

When a function is declared its return type (possibly void) is specified. If the function return type is non-void a return statement must be used to return control to the calling function. The return keyword must be followed by an expression whose value is the value to be returned.

**Exercise 27:** What's wrong with the following function?

```
int strstr ( char s1[], char s2[] )
{
    ...
    return s1 ;
}
```

## Order of Function Declarations

The compiler must see a declaration for a function before that function can be used in an expression. This means you must declare your functions before using them in other functions, including the main() function.

For built-in functions the function declarations are contained in “header” (.h) (or “include”) files that are included in your program when a #include line is used in your program. By convention, the #include lines are placed near the start of the program before any other function declarations.

In C it is not possible to declare one function within the declaration of another function. Thus all functions are “global.”

## Built-in Functions

C has many pre-defined functions to do formatted input and output, compute values of mathematical functions, manipulate character strings, etc.

Some examples of built-in functions are the printf() function for formatted output, getchar() to read a character from the keyboard, or isdigit() to test whether a character is a decimal digit.

In order to use C's built-in functions one or more “#include” lines must be used at the start of a program. These lines tell the compiler to load files (“include files”) that define particular groups of built-in functions. For example, the line #include <stdio.h> lets you use the standard i/o functions in your program, while the line

#include <string.h> lets you use the standard functions that manipulate strings.

The Turbo C on-line help can give you details on any particular function (enter the function name in the edit window, put the cursor on the function name and press control-F1).

## 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 getch(void)
{
    ...
}
```

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

```
void putchar(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.

Other computer languages have special names for void functions: in FORTRAN they are called subroutines and in Pascal they are called procedures.

## Strings

Strings (sequences of characters) in C are declared and stored as character arrays. By convention each string in C is terminated with a null character (the character with value zero).

The C language supports this convention by creating properly-terminated arrays when a sequence of characters are surrounded by double quotes. Note that there are no operators that operate on strings so

these string constants can only be used in variable initializations and as function arguments.

There are built-in functions that can manipulate null-terminated strings. For example, the `strlen()` function returns the length of a string, and the `strcmp()` function compares two strings. To use these functions you must include the `string.h` include file in your program (using `#include <string.h>`). For example:

```
#include <string.h>
...
char name[5] = "Jane" ;
...
n = strlen(name) ;
```

**Exercise 28:** Why is the string `name` declared as having 5 elements if it is being initialized with a 4-character value?

The functions `fgets()` and `fputs()` can be used to read and write a string from a file or the keyboard/display.

## Symbolic Constants

Most programs use constants. Examples of constants include I/O port addresses, bit patterns (“masks”), array lengths, string values, etc.

Putting constants directly into the statements in a program makes it difficult to find and change the values of these constants if the program needs to be modified. Embedding constants directly into the code can also be confusing because, unlike variables, constants don’t have names that can help explain their meaning.

The solution to these problems is to use symbols similar to variable names to represent constants. In C this is done by using `#define` lines. Each such line defines the value of a symbol. Wherever the defined symbol appears in the rest of the program the compiler will replace the symbol with the previously defined value.

For example, if we had the following lines in a program:

```
#define IOPORT 32
#define MSG "Hello, world!\n"
#define u_char unsigned char
#define NCHR 8
```

then whenever the symbols `IOPORT`, `MSG`, `u_char` or `NCHR` appeared anywhere in a subsequent line in the program they would be replaced by corresponding text. The substitution takes place before the line is seen by the rest of the compiler. This allows the constants to be of any type (numbers, strings or even keywords). Thus we could have lines in the program such as:

```
int x[NCHR] ;
u_char c2 ;
spoke ( IOPORT, 1 ) ;
printf ( "%s", MSG ) ;
```

Typically the `#defines` are placed at the start of a source file, immediately after the `#include` lines.

It is good practice<sup>8</sup> to use symbolic constants for all constants in a program. The only exceptions should be cases where the purpose of the constant is clear from the context. The only common exceptions are the values 0 (e.g. `i=0`) and 1 (e.g. `i=i+1`). Constants with other values should almost always be replaced by symbolic constants.

## Comments and White Space

“White space” characters include spaces, tabs and the invisible characters that mark the end of a line. White space is optional except where necessary to avoid ambiguity, such as between `int` and the variable name. In places where white space is required, you may use any type and any number of white space characters.

Comments are notes included in the source code to help readers understand the program. Comments are treated as white space by the compiler. Comments are delimited by the character pairs `/*` and `*/`.

**Exercise 29:** Could the first program in this lecture be re-written all on one line?

## Indentation

Since the C compiler considers any sequence of white space characters to be equivalent there are many ways to format a program. However, some formatting conventions have been established to help

---

<sup>8</sup>You will be expected to use symbolic constants in future assignments and labs whenever it is appropriate.

convey the logical structure of the program. The most important of these conventions is the use of indentation.

The general rule is that statements embedded within another statement (i.e. inside pairs of braces) are indented several (3 to 8) spaces more than the indentation of the immediately surrounding statement. The placement and indentation of braces also helps convey the meaning of the code. Using a consistent indentation convention will greatly help you and others understand the structure of the program.

There are several popular indentation styles. Any one of these is acceptable if used consistently.

The style of indentation used in the examples and solutions given in this course is the “K&R” indentation style. This style was first used by the designers of the C language (Kernighan and Ritchie).

*Marks will be deducted for any C program written for a lab, assignment or exam that does not use a consistent indentation convention.*