

ECE120: Introduction to Computer Engineering

Notes Set 1.5 Programming Concepts and the C Language

This set of notes introduces the C programming language and explains some basic concepts in computer programming. Our purpose in showing you a high-level language at this early stage of the course is to give you time to become familiar with the syntax and meaning of the language, not to teach you how to program. Throughout this semester, we will use software written in C to demonstrate and validate the digital system design material in our course. Towards the end of the semester, you will learn to program computers using instructions and assembly language. In ECE 220, you will make use of the C language to write programs, at which point already being familiar with the language will make the material easier to master. These notes are meant to complement the introduction provided by Patt and Patel.

After a brief introduction to the history of C and the structure of a program written in C, we connect the idea of representations developed in class to the data types used in high-level languages. We next discuss the use of variables in C, then describe some of the operators available to the programmer, including arithmetic and logic operators. The notes next introduce C functions that support the ability to read user input from the keyboard and to print results to the monitor. A description of the structure of statements in C follows, explaining how programs are executed and how a programmer can create statements for conditional execution as well as loops to perform repetitive tasks. The main portion of the notes concludes with an example program, which is used to illustrate both the execution of C statements as well as the difference between variables in programs and variables in algebra.

The remainder of the notes covers more advanced topics. First, we describe how the compilation process works, illustrating how a program written in a high-level language is transformed into instructions. You will learn this process in much more detail in ECE 220. Second, we briefly introduce the C preprocessor. Finally, we discuss implicit and explicit data type conversion in C. *Sections marked with an asterisk are provided solely for your interest, but you probably need to learn this material in later classes.*

1.5.1 The C Programming Language

Programming languages attempt to bridge the semantic gap between human descriptions of problems and the relatively simple instructions that can be provided by an instruction set architecture (ISA). Since 1954, when the Fortran language first enabled scientists to enter FORmulae symbolically and to have them TRANslated automatically into instructions, people have invented thousands of computer languages.

The C programming language was developed by Dennis Ritchie at Bell Labs in order to simplify the task of writing the Unix operating system. The C language provides a fairly transparent mapping to typical ISAs, which makes it a good choice both for system software such as operating systems and for our class. The **syntax** used in C—that is, the rules that one must follow to write valid C programs—has also heavily influenced many other more recent languages, such as C++, Java, and Perl.

```
int
main ()
{
    int answer = 42;        /* the Answer! */

    printf ("The answer is %d.\n", answer);

    /* Our work here is done.
       Let's get out of here!  */
    return 0;
}
```

For our purposes, a C program consists of a set of **variable declarations** and a sequence of **statements**. Both of these parts are written into a single C function called `main`, which executes when the program starts. A simple example appears to the right. The program uses one variable called `answer`, which it initializes to the value 42. The program prints a line of output to the monitor for the user, then terminates using the `return` statement. **Comments** for human readers begin with the characters `/*` (a slash followed by an asterisk) and end with the characters `*/` (an asterisk followed by a slash). The C language ignores white space in programs, so we encourage you to use blank lines and extra spacing to make your programs easier to read.

The variables defined in the `main` function allow a programmer to associate arbitrary **symbolic names** (sequences of English characters, such as "sum" or "product" or "highScore") with specific types of data, such as a 16-bit unsigned integer or a double-precision floating-point number. In the example program above, the variable `answer` is declared to be a 32-bit 2's complement number.

Those with no programming experience may at first find the difference between variables in algebra and variables in programs slightly confusing. *As a program executes, the values of variables can change from step to step of execution.*

The statements in the `main` function are executed one by one until the program terminates. Programs are not limited to simple sequences of statements, however. Some types of statements allow a programmer to specify conditional behavior. For example, a program might only print out secret information if the user's name is "lUmeTTa." Other types of statements allow a programmer to repeat the execution of a group of statements until a condition is met. For example, a program might print the numbers from 1 to 10, or ask for input until the user types a number between 1 and 10. The order of statement execution is well-defined in C, but the statements in `main` do not necessarily make up an algorithm: *we can easily write a C program that never terminates.*

If a program terminates, the `main` function returns an integer to the operating system, usually by executing a `return` statement, as in the example program. By convention, returning the value 0 indicates successful completion of the program, while any non-zero value indicates a program-specific error. However, `main` is not necessarily a function in the mathematical sense because *the value returned from* `main` *is not necessarily unique for a given set of input values to the program.* For example, we can write a program that selects a number from 1 to 10 at random and returns the number to the operating system.

1.5.2 Data Types

As you know, modern digital computers represent all information with binary digits (0s and 1s), or **bits**. Whether you are representing something as simple as an integer or as complex as an undergraduate thesis, the data are simply a bunch of 0s and 1s inside a computer. For any given type of information, a human selects a data type for the information. A **data type** (often called just a **type**) consists of both a size in bits and a representation, such as the 2's complement representation for signed integers, or the ASCII representation for English text. A **representation** is a way of encoding the things being represented as a set of bits, with each bit pattern corresponding to a unique object or thing.

A typical ISA supports a handful of data types in hardware in the sense that it provides hardware support for operations on those data types. The arithmetic logic units (ALUs) in most modern processors, for example, support addition and subtraction of both unsigned and 2's complement representations, with the specific data type (such as 16- or 64-bit 2's complement) depending on the ISA. Data types and operations not supported by the ISA must be handled in software using a small set of primitive operations, which form the **instructions** available in the ISA. Instructions usually include data movement instructions such as loads and stores and control instructions such as branches and subroutine calls in addition to arithmetic and logic operations. The last quarter of our class covers these concepts in more detail and explores their meaning using an example ISA from the textbook.

In class, we emphasized the idea that digital systems such as computers do not interpret the meaning of bits. Rather, they do exactly what they have been designed to do, even if that design is meaningless. If, for example, you store a sequence of ASCII characters in a computer's memory as and then write computer instructions to add consecutive groups of four characters as 2's complement integers and to print the result to the screen, the computer will not complain about the fact that your code produces meaningless garbage.

In contrast, high-level languages typically require that a programmer associate a data type with each datum in order to reduce the chance that the bits making up an individual datum are misused or misinterpreted accidentally. Attempts to interpret a set of bits differently usually generate at least a warning message, since

such re-interpretations of the bits are rarely intentional and thus rarely correct. A compiler—a program that transforms code written in a high-level language into instructions—can also generate the proper type conversion instructions automatically when the transformations are intentional, as is often the case with arithmetic.

Some high-level languages, such as Java, prevent programmers from changing the type of a given datum. If you define a type that represents one of your favorite twenty colors, for example, you are not allowed to turn a color into an integer, despite the fact that the color is represented as a handful of bits. Such languages are said to be **strongly typed**.

The C language is not strongly typed, and programmers are free to interpret any bits in any manner they see fit. Taking advantage of this ability in any but a few exceptional cases, however, results in arcane and non-portable code, and is thus considered to be bad programming practice. We discuss conversion between types in more detail later in these notes.

Each high-level language defines a number of **primitive data types**, which are always available. Most languages, including C, also provide ways of defining new types in terms of primitive types, but we leave that part of C for ECE 220. The primitive data types in C include signed and unsigned integers of various sizes as well as single- and double-precision IEEE floating-point numbers.

The primitive integer types in C include both unsigned and 2's complement representations. These types were originally defined so as to give reasonable performance when code was ported. In particular, the `int` type is intended to be the native integer type for the target ISA. Using data types supported directly in hardware is faster than using larger or smaller integer types. When C was standardized in 1989, these types were defined so as to include a range of existing C compilers rather than requiring all compilers to produce uniform results. At the time, most workstations and mainframes were 32-bit machines, while most personal computers were 16-bit machines, thus flexibility was somewhat desirable. For the GCC compiler on Linux, the C integer data types are defined in the table above. Although the `int` and `long` types are usually the same, there is a semantic difference in common usage. In particular, on most architectures and most compilers, a `long` has enough bits to identify a location in the computer's memory, while an `int` may not. When in doubt, the **size in bytes** of any type or variable can be found using the built-in C function `sizeof`.

	2's complement	unsigned
8 bits	`char`	`unsigned char`
16 bits	`short` `short int`	`unsigned short` `unsigned short int`
32 bits	`int`	`unsigned` `unsigned int`
32 or 64 bits	`long` `long int`	`unsigned long` `unsigned long int`
64 bits	`long long` `long long int`	`unsigned long long` `unsigned long long int`

Over time, the flexibility of size in C types has become less important (except for the embedded markets, where one often wants even more accurate bit-width control), and the fact that the size of an `int` can vary from machine to machine and compiler to compiler has become more a source of headaches than a helpful feature. In the late 1990s, a new set of fixed-size types were recommended for inclusion in the C library, reflecting the fact that many companies had already developed and were using such definitions to make their programs platform-independent. We encourage you to make use of these types, which are shown in the table above. In Linux, they can be made available by including the `stdint.h` header file.

	2's complement	unsigned
8 bits	`int8_t`	`uint8_t`
16 bits	`int16_t`	`uint16_t`
32 bits	`int32_t`	`uint32_t`
64 bits	`int64_t`	`uint64_t`

Floating-point types in C include `float` and `double`, which correspond respectively to single- and double-precision IEEE floating-point values. Although the 32-bit `float` type can save memory compared with use of 64-bit `double` values, C's math library works with double-precision values, and single-precision data are uncommon in scientific and engineering codes. In contrast, single-precision floating-point operations dominated the graphics industry until recently, and are still well-supported even on today's graphics processing units.

1.5.3 Variable Declarations

The function `main` executed by a program begins with a list of **variable declarations**. Each declaration consists of two parts: a data type specification and a comma-separated list of variable names. Each variable declared can also be **initialized** by assigning an initial value. A few examples appear below. Notice that one can initialize a variable to have the same value as a second variable.

```
int    x = 42;                /* a 2's complement variable, initially equal to 42          */
int    y = x;                 /* a second 2's complement variable, initially equal to x    */
int    z;                     /* a third 2's complement variable with unknown initial value */
double a, b, c, pi = 3.1416;  /*
                               * four double-precision IEEE floating-point variables
                               * a, b, and c are initially of unknown value, while pi is
                               * initially 3.1416
                               */
```

What happens if a programmer declares a variable but does not initialize it? Remember that bits can only be 0 or 1. An uninitialized variable does have a value, but its value is unpredictable. The compiler tries to detect uses of uninitialized variables, but sometimes it fails to do so, so *until you are more familiar with programming, you should always initialize every variable.*

Variable names, also called **identifiers**, can include both letters and digits in C. Good programming style requires that programmers select variable names that are meaningful and are easy to distinguish from one another. Single letters are acceptable in some situations, but longer names with meaning are likely to help people (including you!) understand your program. Variable names are also case-sensitive in C, which allows programmers to use capitalization to differentiate behavior and meaning, if desired. Some programs, for example, use identifiers with all capital letters to indicate variables with values that remain constant for the program's entire execution. However, the fact that identifiers are case-sensitive also means that a programmer can declare distinct variables named `variable`, `Variable`, `vaRIable`, `vaRIabLe`, and `VARIABLE`. We strongly discourage you from doing so.

1.5.4 Expressions and Operators

The `main` function also contains a sequence of statements. A statement is a complete specification of a single step in the program's execution. We explain the structure of statements in the next section. Many statements in C include one or more **expressions**, which represent calculations such as arithmetic, comparisons, and logic operations. Each expression is in turn composed of **operators** and **operands**. Here we give only a brief introduction to some of the operators available in the C language. We deliberately omit operators with more complicated meanings, as well as operators for which the original purpose was to make writing common operations a little shorter. For the interested reader, both the textbook and ECE 220 give more detailed introductions. The table to the right gives examples for the operators described here.

```
int i = 42, j = 1000;
/* i = 0x0000002A, j = 0x000003E8 */

        i + j   → 1042
    i - 4 * j   → -3958
           -j   → -1000
        j / i   → 23
        j % i   → 42
        i & j   → 40      /* 0x00000028 */
        i | j   → 1002    /* 0x000003EA */
        i ∧ j   → 962     /* 0x000003C2 */
          ∼i    → -43     /* 0xFFFFFFD5 */
    (∼i) >> 2   → -11     /* 0xFFFFFFF5 */
∼((∼i) >> 4)    → 2       /* 0x00000002 */
       j >> 4   → 62      /* 0x0000003E */
       j << 3   → 8000    /* 0x00001F40 */
        i > j   → 0
       i <= j   → 1
       i == j   → 0
        j = i   → 42      /* ...and j is changed!  */
```


Arithmetic operators in C include addition (`+`), subtraction (`-`), negation (a minus sign not preceded by another expression), multiplication (`*`), division (`/`), and modulus (`%`). No exponentiation operator exists; instead, library routines are defined for this purpose as well as for a range of more complex mathematical functions.

C also supports **bitwise operations** on integer types, including AND (`&`), OR (`|`), XOR (`^`), NOT (∼), and left (<<) and right (>>) bit shifts. Right shifting a signed integer results in an **arithmetic right shift** (the sign bit is copied), while right shifting an unsigned integer results in a **logical right shift** (0 bits are inserted).

A range of **relational** or **comparison operators** are available, including equality (`==`), inequality (`!=`), and relative order (<, <=, >=, and >). All such operations evaluate to 1 to indicate a true relation and 0 to indicate a false relation. Any non-zero value is considered to be true for the purposes of tests (for example, in an `if` statement or a `while` loop) in C—these statements are explained later in these notes.

Assignment of a new value to a variable uses a single equal sign (`=`) in C. For example, the expression `A = B` copies the value of variable `B` into variable `A`, overwriting the bits representing the previous value of `A`. *The use of two equal signs for an equality check and a single equal sign for assignment is a common source of errors,* although modern compilers generally detect and warn about this type of mistake. Assignment in C does not solve equations, even simple equations. Writing "`A-4=B`", for example, generates a compiler error. You must solve such equations yourself to calculate the desired new value of a single variable, such as "`A=B+4`." For the purposes of our class, you must always write a single variable on the left side of an assignment, and can write an arbitrary expression on the right side.

Many operators can be combined into a single expression. When an expression has more than one operator, which operator is executed first? The answer depends on the operators' **precedence**, a well-defined order on operators that specifies how to resolve the ambiguity. In the case of arithmetic, the C language's precedence specification matches the one that you learned in elementary school. For example, `1+2*3` evaluates to 7, not to 9, because multiplication has precedence over addition. For non-arithmetic operators, or for any case in which you do not know the precedence specification for a language, *do not look it up—other programmers will not remember the precedence ordering, either!* Instead, add parentheses to make your expressions clear and easy to understand.

1.5.5 Basic I/O

The `main` function returns an integer to the operating system. Although we do not discuss how additional functions can be written in our class, we may sometimes make use of functions that have been written in advance by making **calls** to those functions. A **function call** is type of expression in C, but we leave further description for ECE 220. In our class, we make use of only two additional functions to enable our programs to receive input from a user via the keyboard and to write output to the monitor for a user to read.

Let's start with output. The `printf` function allows a program to print output to the monitor using a programmer-specific format. The "f" in `printf` stands for "formatted."[4] When we want to use `printf`, we write a expression with the word `printf` followed by a parenthesized, comma-separated list of expressions. The expressions in this list are called the **arguments** to the `printf` function.

The first argument to the `printf` function is a format string—a sequence of ASCII characters between quotation marks—which tells the function what kind of information we want printed to the monitor as well as how to format that information. The remaining arguments are C expressions that give `printf` a copy of any values that we want printed.

How does the format string specify the format? Most of the characters in the format string are simply printed to the monitor. In the first example shown to on the next page, we use `printf` to print a hello message followed by an ASCII newline character to move to the next line on the monitor.

[4]The original, unformatted variant of printing was never available in the C language. Go learn Fortran.

The percent sign—"%"—is used as an **escape character** in the `printf` function. When "%" appears in the format string, the function examines the next character in the format string to determine which format to use, then takes the next expression from the sequence of arguments and prints the value of that expression to the monitor. Evaluating an expression generates a bunch of bits, so it is up to the programmer to ensure that those bits are not misinterpreted. In other words, the programmer must make sure that the number and types of formatted values match the number and types of arguments passed to `printf` (not counting the format string itself). The `printf` function returns the number of characters printed to the monitor.

```
printf ("Hello, world!\n");
```
output: `Hello, world!` [and a newline]

```
printf ("To %x or not to %d...\n", 190, 380 / 2);
```
output: `To be or not to 190...` [and a newline]

```
printf ("My favorite number is %c%c.\n", 0x34, '0'+2);
```
output: `My favorite number is 42.` [and a newline]

```
printf ("What is pi?  %f or %e?\n", 3.1416, 3.1416);
```
output: `What is pi?  3.141600 or 3.141600e+00?` [and a newline]

escape sequence	`printf` function's interpretation of expression bits
`%c`	2's complement integer printed as an ASCII character
`%d`	2's complement integer printed as decimal
`%e`	double printed in decimal scientific notation
`%f`	double printed in decimal
`%u`	unsigned integer printed as decimal
`%x`	integer printed as hexadecimal (lower case)
`%X`	integer printed as hexadecimal (upper case)
`%%`	a single percent sign

A program can read input from the user with the `scanf` function. The user enters characters in ASCII using the keyboard, and the `scanf` function converts the user's input into C primitive types, storing the results into variables. As with `printf`, the `scanf` function takes a format string followed by a comma-separated list of arguments. Each argument after the format string provides `scanf` with the memory address of a variable into which the function can store a result.

How does `scanf` use the format string? For `scanf`, the format string is usually just a sequence of conversions, one for each variable to be typed in by the user. As with `printf`, the conversions start with "%" and are followed by characters specifying the type of conversion to be performed. The first example shown to the right reads two integers. The conversions in the format string can be separated by spaces for readability, as shown in the example. The spaces are ignored by `scanf`. However, *any non-space characters in the format string must be typed exactly by the user!*

The remaining arguments to `scanf` specify memory addresses where the function can store the converted values. The ampersand ("&") in front of each variable name in the examples is an operator that returns the address of a variable in memory. For each con-

```
int      a, b; /* example variables */
char     c;
unsigned u;
double   d;
float    f;

scanf ("%d%d", &a, &b);   /* These have the */
scanf ("%d %d", &a, &b);  /* same effect.   */
```
effect: try to convert two integers typed in decimal to 2's complement and store the results in `a` and `b`

```
scanf ("%c%x %lf", &c, &u, &d);
```
effect: try to read an ASCII character into `c`, a value typed in hexadecimal into `u`, and a double-precision floating-point number into `d`

```
scanf ("%lf %f", &d, &f);
```
effect: try to read two real numbers typed as decimal, convert the first to double-precision and store it in `d`, and convert the second to single-precision and store it in `f`

escape sequence	`scanf` function's conversion to bits
`%c`	store one ASCII character (as `char`)
`%d`	convert decimal integer to 2's complement
`%f`	convert decimal real number to float
`%lf`	convert decimal real number to double
`%u`	convert decimal integer to unsigned int
`%x`	convert hexadecimal integer to unsigned int
`%X`	(as above)

version in the format string, the `scanf` function tries to convert input from the user into the appropriate result, then stores the result in memory at the address given by the next argument. The programmer is responsible for ensuring that the number of conversions in the format string matches the number of arguments provided (not counting the format string itself). The programmer must also ensure that the type of information produced by each conversion can be stored at the address passed for that conversion—in other words, the address of a variable with the correct type must be provided. Modern compilers often detect missing `&` operators and incorrect variable types, but many only give warnings to the programmer. The `scanf` function itself cannot tell whether the arguments given to it are valid or not.

If a conversion fails—for example, if a user types "hello" when `scanf` expects an integer—`scanf` does not overwrite the corresponding variable and immediately stops trying to convert input. The `scanf` function returns the number of successful conversions, allowing a programmer to check for bad input from the user.

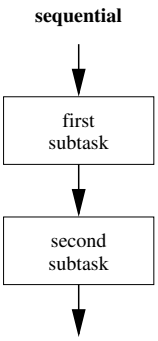
1.5.6 Types of Statements in C

Each statement in a C program specifies a complete operation. There are three types of statements, but two of these types can be constructed from additional statements, which can in turn be constructed from additional statements. The C language specifies no bound on this type of recursive construction, but code readability does impose a practical limit.

The three types are shown to the right. They are the **null statement**, **simple statements**, and **compound statements**. A null statement is just a semicolon, and a compound statement is just a sequence of statements surrounded by braces.

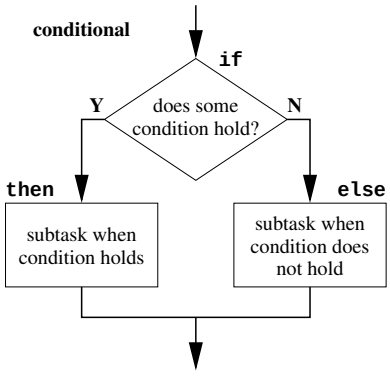
```
;                   /* a null statement (does nothing) */

A = B;              /* examples of simple statements   */
printf ("Hello, world!\n");

{                   /* a compound statement            */
    C = D;          /* (a sequence of statements       */
    N = 4;          /*  between braces)                */
    L = D - N;
}
```

Simple statements can take several forms. All of the examples shown to the right, including the call to `printf`, are simple statements consisting of a C expression followed by a semicolon. Simple statements can also consist of conditionals or iterations, which we introduce next.



Remember that after variable declarations, the `main` function contains a sequence of statements. These statements are executed one at a time in the order given in the program, as shown to the right for two statements. We say that the statements are executed in sequential order.

A program must also be able to execute statements only when some condition holds. In the C language, such a condition can be an arbitrary expression. The expression is first evaluated. If the result is 0, the condition is considered to be false. Any result other than 0 is considered to be true. The C statement for conditional execution is called an `if` statement. Syntactically, we put the expression for the condition in parentheses after the keyword `if` and follow the parenthesized expression with a compound statement containing the statements that should be executed when the condition is true. Optionally, we can append the keyword `else` and a second compound statement containing statements to be executed when the condition evaluates to false. The corresponding flow chart is shown to the right.



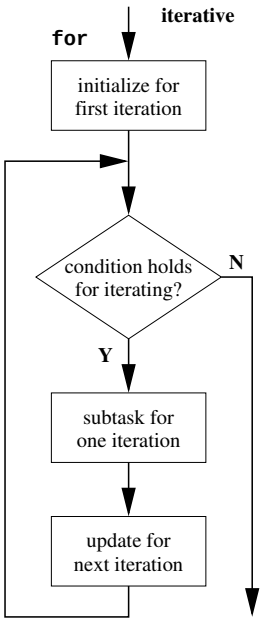

```
/* Set the variable y to the absolute value of variable x. */
if (0 <= x) {   /* Is x greater or equal to 0? */
    y = x;      /* Then block: assign x to y. */
} else {
    y = -x;     /* Else block: assign negative x to y. */
}
```

If instead we chose to assign the absolute value of variable `x` to itself, we can do so without an `else` block:

```
/* Set the variable x to its absolute value. */
if (0 > x) {    /* Is x less than 0? */
    x = -x;     /* Then block: assign negative x to x. */
}               /* No else block is given--no work is needed. */
```

Finally, we sometimes need to repeatedly execute a set of statements, either a fixed number of times or so long as some condition holds. We refer to such repetition as an **iteration** or a **loop**. In our class, we make use of C's `for` loop when we need to perform such a task. A `for` loop is structured as follows:

```
for ([initialization] ; [condition] ; [update]) {
    [subtask to be repeated]
}
```



A flow chart corresponding to execution of a `for` loop appears to the right. First, any initialization is performed. Then the condition—again an arbitrary C expression—is checked. If the condition evaluates to false (exactly 0), the loop is done. Otherwise, if the condition evaluates to true (any non-zero value), the statements in the compound statement, the subtask or **loop body**, are executed. The loop body can contain anything: a sequence of simple statements, a conditional, another loop, or even just an empty list. Once the loop body has finished executing, the `for` loop update rule is executed. Execution then checks the condition again, and this process repeats until the condition evaluates to 0. The `for` loop below, for example, prints the numbers from 1 to 42.

```
/* Print the numbers from 1 to 42.  */
for (i = 1; 42 >= i; i = i + 1) {
    printf ("%d\n", i);
}
```

1.5.7 Program Execution

We are now ready to consider the execution of a simple program, illustrating how variables change value from step to step and determine program behavior.

Let's say that two numbers are "friends" if they have at least one 1 bit in common when written in base 2. So, for example, 100_2 and 111_2 are friends because both numbers have a 1 in the bit with place value $2^2 = 4$. Similarly, 101_2 and 010_2 are not friends, since no bit position is 1 in both numbers.

The program to the right prints all friendships between numbers in the interval $[0, 7]$.

```
int
main ()
{
    int check;    /* number to check for friends                    */
    int friend;   /* a second number to consider as check's friend */

    /* Consider values of check from 0 to 7.  */
    for (check = 0; 8 > check; check = check + 1) {

        /* Consider values of friend from 0 to 7.  */
        for (friend = 0; 8 > friend; friend = friend + 1) {

            /* Use bitwise AND to see if the two share a 1 bit.  */
            if (0 != (check & friend)) {

                /* We have friendship!  */
                printf ("%d and %d are friends.\n", check, friend);
            }
        }
    }
}
```


The program uses two integer variables, one for each of the numbers that we consider. We use a `for` loop to iterate over all values of our first number, which we call `check`. The loop initializes `check` to 0, continues until check reaches 8, and adds 1 to check after each loop iteration. We use a similar `for` loop to iterate over all possible values of our second number, which we call `friend`. For each pair of numbers, we determine whether they are friends using a bitwise AND operation. If the result is non-zero, they are friends, and we print a message. If the two numbers are not friends, we do nothing, and the program moves on to consider the next pair of numbers.

Now let's think about what happens when this program executes. When the program starts, both variables are filled with random bits, so their values are unpredictable. The first step is the initialization of the first `for` loop, which sets `check` to 0. The condition for that loop is `8 > check`, which is true, so execution enters the loop body and starts to execute the first statement, which is our second `for` loop. The next step is then the initialization code for the second `for` loop, which sets `friend` to 0. The condition for the second loop is `8 > friend`, which is true, so execution enters the loop body and starts to execute the first statement, which is the `if` statement. Since both variables are 0, the `if` condition is false, and nothing is printed. Having finished the loop body for the inner loop (on `friend`), execution continues with the update rule for that loop—`friend = friend + 1`—then returns to check the loop's condition again. This process repeats, always finding that the number 0 (in `check`) is not friends (0 has no friends!) until `friend` reaches 8, at which point the inner loop condition becomes false. Execution then moves to the update rule for the first `for` loop, which increments `check`. Check is then compared with 8 to see if the loop is done. Since it is not, we once again enter the loop body and start the second `for` loop over. The initialization code again sets `friend` to 0, and we move forward as before. As you see above, the first time that we find our `if` condition to be true is when both `check` and `friend` are equal to 1.

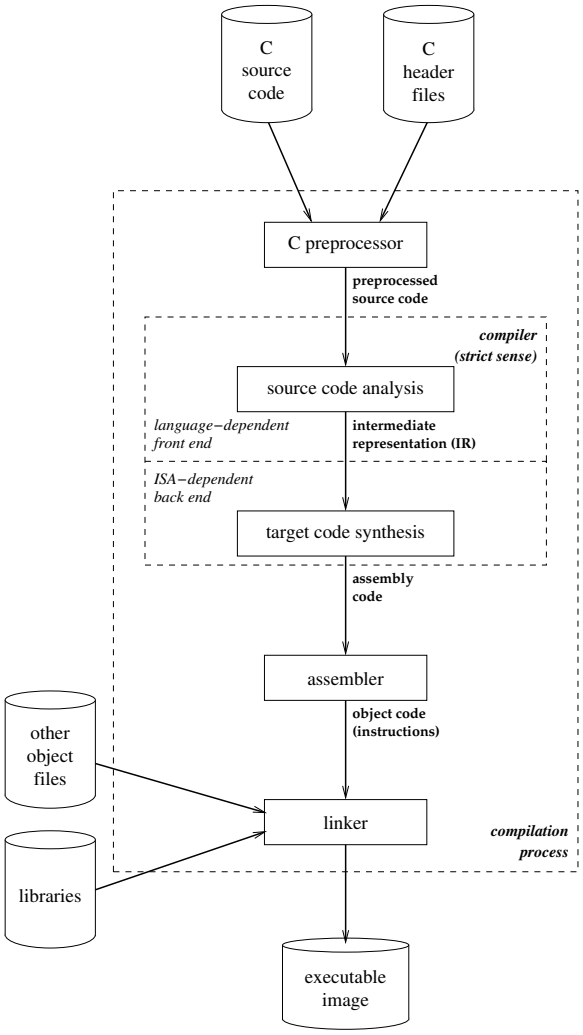
after executing...	`check` is...	and `friend` is...
(variable declarations)	unpredictable bits	unpredictable bits
`check = 0`	0	unpredictable bits
`8 > check`	0	unpredictable bits
`friend = 0`	0	0
`8 > friend`	0	0
`if (0 != (check & friend))`	0	0
`friend = friend + 1`	0	1
`8 > friend`	0	1
`if (0 != (check & friend))`	0	1
`friend = friend + 1`	0	2
(repeat last three lines six more times; number 0 has no friends!)		
`8 > friend`	0	8
`check = check + 1`	1	8
`8 > check`	1	8
`friend = 0`	1	0
`8 > friend`	1	0
`if (0 != (check & friend))`	1	0
`friend = friend + 1`	1	1
`8 > friend`	1	1
`if (0 != (check & friend))`	1	1
`printf ...`	1	1
(our first friend!?)		

Is that result what you expected? To learn that the number 1 is friends with itself? If so, the program works. If you assumed that numbers could not be friends with themselves, perhaps we should fix the bug? We could, for example, add another `if` statement to avoid printing anything when `check == friend`.

Our program, you might also realize, prints each pair of friends twice. The numbers 1 and 3, for example, are printed in both possible orders. To eliminate this redundancy, we can change the initialization in the second `for` loop, either to `friend = check` or to `friend = check + 1`, depending on how we want to define friendship (the same question as before: can a number be friends with itself?).

1.5.8 Compilation and Interpretation*

Many programming languages, including C, can be **compiled**, which means that the program is converted into instructions for a particular ISA before the program is run on a processor that supports that ISA. The figure to the right illustrates the compilation process for the C language. In this type of figure, files and other data are represented as cylinders, while rectangles represent processes, which are usually implemented in software. In the figure to the right, the outer dotted box represents the full compilation process that typically occurs when one compiles a C program. The inner dotted box represents the work performed by the **compiler** software itself. The cylinders for data passed between the processes that compose the full compilation process have been left out of the figure; instead, we have written the type of data being passed next to the arrows that indicate the flow of information from one process to the next.



The C preprocessor (described later in these notes) forms the first step in the compilation process. The preprocessor operates on the program's **source code** along with **header files** that describe data types and operations. The preprocessor merges these together into a single file of preprocessed source code. The preprocessed source code is then analyzed by the front end of the compiler based on the specific programming language being used (in our case, the C language), then converted by the back end of the compiler into instructions for the desired ISA. The output of a compiler is not binary instructions, however, but is instead a human-readable form of instructions called **assembly code**, which we cover in the last quarter of our class. A tool called an assembler then converts these human-readable instructions into bits that a processor can understand. If a program consists of multiple source files, or needs to make use of additional pre-programmed operations (such as math functions, graphics, or sound), a tool called a linker merges the object code of the program with those additional elements to form the final **executable image** for the program. The executable image is typically then stored on a disk, from which it can later be read into memory in order to allow a processor to execute the program.

Some languages are difficult or even impossible to compile. Typically, the behavior of these languages depends on input data that are only available when the program runs. Such languages can be **interpreted**: each step of the algorithm described by a program is executed by a software interpreter for the language. Languages such as Java, Perl, and Python are usually interpreted. Similarly, when we use software to simulate one ISA using another ISA, as we do at the end of our class with the LC-3 ISA described by the textbook, the simulator is a form of interpreter. In the lab, you will use a simulator compiled into and executing as x86 instructions in order to interpret LC-3 instructions. While a program is executing in an interpreter, enough information is sometimes available to compile part or all of the program to the processor's ISA as the program runs, a technique known as **"just in time" (JIT) compilation**.

1.5.9 The C Preprocessor*

The C language uses a preprocessor to support inclusion of common information (stored in header files) into multiple source files. The most frequent use of the preprocessor is to enable the unique definition of new data types and operations within header files that can then be included by reference within source files that make use of them. This capability is based on the **include directive**, `#include`, as shown here:

```
#include <stdio.h>        /* search in standard directories                    */
#include "my_header.h"    /* search in current followed by standard directories */
```

The preprocessor also supports integration of compile-time constants into source files before compilation. For example, many software systems allow the definition of a symbol such as `NDEBUG` (no debug) to compile without additional debugging code included in the sources. Two directives are necessary for this purpose: the **define directive**, `#define`, which provides a text-replacement facility, and **conditional inclusion** (or exclusion) of parts of a file within `#if`/`#else`/`#endif` directives. These directives are also useful in allowing a single header file to be included multiple times without causing problems, as C does not allow redefinition of types, variables, and so forth, even if the redundant definitions are identical. Most header files are thus wrapped as shown to the right.

```
#if !defined(MY_HEADER_H)
#define MY_HEADER_H
/* actual header file material goes here */
#endif /* MY_HEADER_H */
```

The preprocessor performs a simple linear pass on the source and does not parse or interpret any C syntax. Definitions for text replacement are valid as soon as they are defined and are performed until they are undefined or until the end of the original source file. The preprocessor does recognize spacing and will not replace part of a word, thus "`#define i 5`" will not wreak havoc on your `if` statements, but will cause problems if you name any variable `i`.

Using the text replacement capabilities of the preprocessor does have drawbacks, most importantly in that almost none of the information is passed on for debugging purposes.

1.5.10 Changing Types in C*

Changing the type of a datum is necessary from time to time, but sometimes a compiler can do the work for you. The most common form of **implicit type conversion** occurs with binary arithmetic operations. Integer arithmetic in C always uses types of at least the size of `int`, and all floating-point arithmetic uses `double`. If either or both operands have smaller integer types, or differ from one another, the compiler implicitly converts them before performing the operation, and the type of the result may be different from those of both operands. In general, the compiler selects the final type according to some preferred ordering in which floating-point is preferred over integers, unsigned values are preferred over signed values, and more bits are preferred over fewer bits. The type of the result must be at least as large as either argument, but is also at least as large as an `int` for integer operations and a `double` for floating-point operations.

Modern C compilers always extend an integer type's bit width before converting from signed to unsigned. The original C specification interleaved bit width extensions to `int` with sign changes, thus *older compilers may not be consistent, and implicitly require both types of conversion in a single operation may lead to portability bugs.*

The implicit extension to `int` can also be confusing in the sense that arithmetic that seems to work on smaller integers fails with larger ones. For example, multiplying two 16-bit integers set to 1000 and printing the result works with most compilers because the 32-bit `int` result is wide enough to hold the right answer. In contrast, multiplying two 32-bit integers set to 100,000 produces the wrong result because the high bits of the result are discarded before it can be converted to a larger type. For this operation to produce the correct result, one of the integers must be converted explicitly (as discussed later) before the multiplication.

Implicit type conversions also occur due to assignments. Unlike arithmetic conversions, the final type must match the left-hand side of the assignment (for example, a variable to which a result is assigned), and the compiler simply performs any necessary conversion. *Since the desired type may be smaller than the type of the value assigned, information can be lost.* Floating-point values are truncated when assigned to integers, and high bits of wider integer types are discarded when assigned to narrower integer types. *Note that a positive number may become a negative number when bits are discarded in this manner.*

Passing arguments to functions can be viewed as a special case of assignment. Given a function prototype, the compiler knows the type of each argument and can perform conversions as part of the code generated to pass the arguments to the function. Without such a prototype, or for functions with variable numbers of arguments, the compiler lacks type information and thus cannot perform necessary conversions, leading to unpredictable behavior. By default, however, the compiler extends any integer smaller than an `int` to the width of an `int` and converts `float` to `double`.

Occasionally it is convenient to use an **explicit type cast** to force conversion from one type to another. *Such casts must be used with caution, as they silence many of the warnings that a compiler might otherwise generate when it detects potential problems.* One common use is to promote integers to floating-point before an arithmetic operation, as shown to the right. The type to which a value is to be converted is placed in parentheses in front of the value. In most cases, additional parentheses should be used to avoid confusion about the precedence of type conversion over other operations.

```
int
main ()
{
    int numerator = 10;
    int denominator = 20;

    printf ("%f\n", numerator / (double)denominator);
    return 0;
}
```