Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

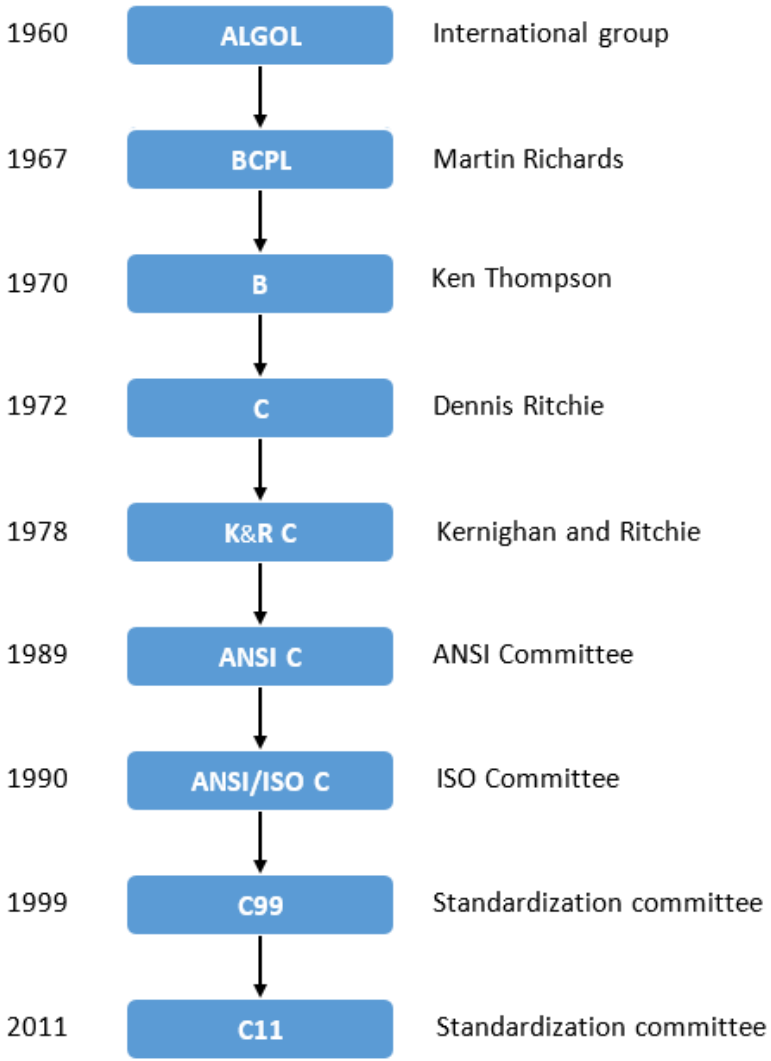
Text Book(s):

1. “How To Solve It By Computer”, R G Dromey, Pearson, 2011.
2. “The C Programming Language”, Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Other Book(s):

1. “Expert C Programming; Deep C secrets”, Peter van der Linden
2. “ The C puzzle Book”, Alan R Feuer

Timeline of C Programming Language



The C Programming Language

C is one of the most important programming languages in the history of computing.

C provided the basis for a number of other languages which followed it.

C was born out of necessity.
Dennis Ritchie (Bell Labs AT & T) (in the 1960s) - developed Multics using assembly language. Not cost effective – the project was dropped by Bell.
Ritchie, Ken Thompson and Brian Kernighan – created UNIX using assembly language, Fortran and a language called ‘B’.

UNIX was written in assembly language

To perform even small operations in UNIX, one needed to write many pages of code. B solved this problem. Unlike assembly language, B needed significantly fewer lines of code to carry out a task in UNIX. Still, there was a lot that B could not do. For example, B did not recognize data types. Even with B, data types were expressed with machine language. B also did not support data structures.

The C language was developed in 1971-73. Note that for all its limitations, C owes its birth to B because C retained a lot of what B offered, while adding features such as data types and data structures. The name C was chosen because it succeeded B. In its early days, C was designed keeping UNIX in mind. C was used to perform tasks and operate UNIX. So, keeping performance and productivity in mind, many of the UNIX components were rewritten in C from assembly language. For example, the UNIX kernel itself was rewritten in 1973 on a DEC PDP-11.

Ritchie and Kernighan documented their creation in the form of a book called "The C Programming Language."

Evolution of C

Over time, C began to be used in personal computers for developing software applications and other purposes.

The first change came when the American National Standards Institute (ANSI) formed a committee in 1983 to standardize C. After a review of the language, they modified it a little so that it was also compatible with other programs that preceded C. So the new ANSI standard came into being in 1989, and is known as ANSI C or C89. The International Organization for Standardization (ISO) has also contributed to the standardization of C.

Over time, C has evolved as it has added some significant features like memory management, functions, classes and libraries to its rich feature set. C is being used in some of the biggest and most prominent projects and products in the world. C has also influenced the development of numerous languages such as AWK, csh, C++, C#, JavaScript, etc.

Nearly all modern operating systems are written in C. It's also widely used in embedded systems, such as those found in vehicles, smart TVs and countless internet of things (IoT) devices.

Just like most of the world's greatest inventions, C was born out of necessity. Circumstances and problems provided the inspiration. However, unlike many programming languages that are now extinct or almost extinct, C has stood the test of time and thrived. Some languages are now categorized as niche languages – for example, Fortran is now mostly used only for engineering purposes and COBOL is struggling to stay relevant. C has not only stayed relevant, but has also provided inspiration for the development of many other programming languages. Even powerful technology waves like IoT, AI and automation have failed to dislodge C from its position of prominence. It appears that this language will continue to be with us long into the future as well.

C Standards

C standards development has been a conservative process with great care taken to preserve the spirit of the original C language, and an emphasis on ratifying experiments in existing compilers rather than inventing new features. The C9X charter document is an excellent expression of this mission.

Work on the first official C standard began in 1983 under the auspices of the X3J11 ANSI committee. The major functional additions to the language were settled by the end of 1986, at which point it became common for programmers to distinguish between “K&R C” and “ANSI C”.

While the core of ANSI C was settled early, arguments over the contents of the standard libraries dragged on for years. The formal standard was not issued until the end of 1989, well after most compilers had implemented the 1985 recommendations. The standard was originally known as ANSI X3.159, but was redesignated ISO/IEC 9899:1990 when the International Standards Organization (ISO) took over sponsorship in 1990. The language variant it describes is generally known as **C89 or C90**.

A very minor revision of C89, known as Amendment 1, AM1, or C93, was floated in 1993. It added more support for wide characters and Unicode. This became ISO/IEC 9899-1:1994.

Revision of the C89 standard began in 1993. In 1999, ISO/IEC 9899 (generally known as C99) was adopted by ISO. It incorporated Amendment 1, and added a great many minor features. Perhaps the most significant one for most programmers is the C++-like ability to declare variables at any point in a block, rather than just at the beginning. Macros with a variable number of arguments were also added.

C Standards.....

C99 was followed by the C11 standards

C18 and **C17** are informal names for **ISO/IEC 9899:2018**,the most recent standard for the C programming language, published in June 2018. It replaced C11 (standard ISO/IEC 9899:2011). Support was scheduled for GCC 8 and LLVM Clang 6.0

Do you have a choice in terms of following one of these standards?

Yes.

gcc –std=c99 source_files

gcc –std=c11 source_files

Popularity of C

The TIOBE Programming Community index is an indicator of the popularity of programming languages. The index is updated once a month. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors.

Nov 2019 – The three most popular programming languages are Java, C and Python, in that order.

C – A bird's eye view

C is a general-purpose programming language. It has been closely associated with the UNIX system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to anyone operating system or machine; and although it has been called a "system programming language" because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

C – A bird's eye view ….

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7. BCPL and B are "typeless" languages. By contrast, C provides a variety of data types.

The fundamental types are characters, and integers and floating point numbers of several sizes.

In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic.

C provides the fundamental control-flow constructions required for well structured programs: statement grouping, decision making (if-else), selecting one of a set of possible cases (switch), looping with the termination test at the top (while, for) or at the bottom (do),and early loop exit (break).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically "automatic," or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.

A pre-processing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.

C is a relatively "low level" language. It simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists, or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions; there is no heap or garbage collection.

Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or coroutines.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior.

For most programmers, the most important change is a new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra information makes it much easier for compilers to detect errors caused by mismatched arguments.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions and data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the "standard *I/O library*" of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. **With a little care' it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware**. The standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to be an extremely effective and expressive language for a wide variety of programming applications.

Compiler

A compiler is just a program that converts your program written in terms you understand into a form your computer understands.

The Standard Library

The *standard library for C is also specified within the C11 standard. The standard library defines constants, symbols,* and functions that you frequently need when writing a C program. It also provides some optional extensions to the basic C language. Machine-dependent facilities such as input and output for your computer are implemented by the standard library in a machine-independent form. This means that you write data to a disk file in C in the same way on your PC as you would on any other kind of computer, even though the underlying hardware processes are quite different. The standard functionality that the library contains includes capabilities that most programmers are likely to need, such as processing text strings or math calculations. This saves you an enormous amount of effort that would be required to implement such things yourself.

The standard library is specified in a set of standard files called *header files. Header files always have names with* the extension .h. To make a particular set of standard features available in your C program file, you just include the appropriate standard header file (will be explained later). Every program you write will make use of the standard library.

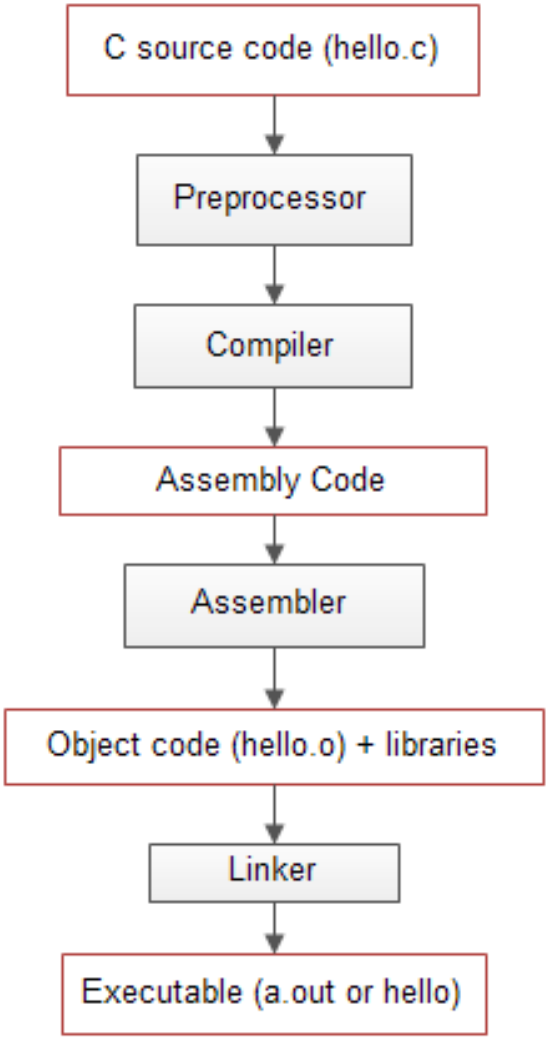
Creating C Programs

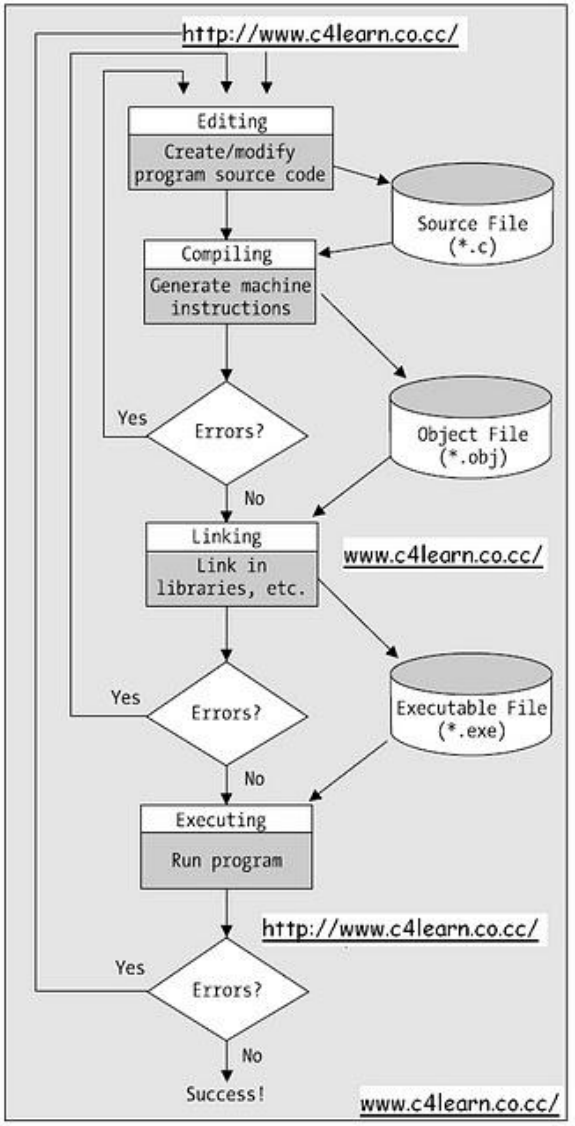
There are four fundamental stages, or processes, in the creation of any C program:

1. Editing
2. Compiling
3. Linking
4. Executing

The processes of editing, compiling, linking, and executing are essentially the same for developing programs in any environment and with any compiled language. Figure (next slide) summarizes how you would typically pass through processes as you create your own C programs.

Creating and executing a C program





Editing

Editing is the process of creating and modifying C source code—the name given to the program instructions you write. Some C compilers come with a specific editor program that provides a lot of assistance in managing your programs.

Compiling

The *compiler converts your source code into machine language and detects and reports errors in the compilation* process. The input to this stage is the file you produce during your editing, which is usually referred to as a *source file.*

The compiler can detect a wide range of errors that are due to invalid or unrecognized program code, as well as structural errors where, for example, part of a program can never be executed. The output from the compiler is known as *object code.* The compiler can detect several different kinds of errors during the translation process, and most of these will prevent the object file from being created.

Compilation is a two-stage process. The first stage is called the *preprocessing phase, during which your code may* be modified or added to, and the second stage is the actual compilation that generates the object code. Your source file can include preprocessing *macros, which you use to add to or modify the C program statements.*

Linking

The *linker combines the object modules generated by the compiler from source code files, adds required code* modules from the standard library supplied as part of C, and welds everything into an executable whole. The linker also detects and reports errors; for example, if part of your program is missing or a nonexistent library component is referenced.

Executing

The execution stage is where you run your program, having completed all the previous processes successfully. Unfortunately, this stage can also generate a wide variety of error conditions that can include producing the wrong output, just sitting there and doing nothing, or perhaps crashing your computer for good measure. In all cases, it's back to the editing process to check your source code.

Debugging

Find the errors and fix them.

The purpose of a debugger such as gdb is to allow you to see what is going on "inside" another program while it executes—or what another program was doing at the moment it crashed.

gdb can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

1. Start your program, specifying anything that might affect its behavior.
2. Make your program stop on specified conditions.
3. Examine what has happened, when your program has stopped.
4. Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

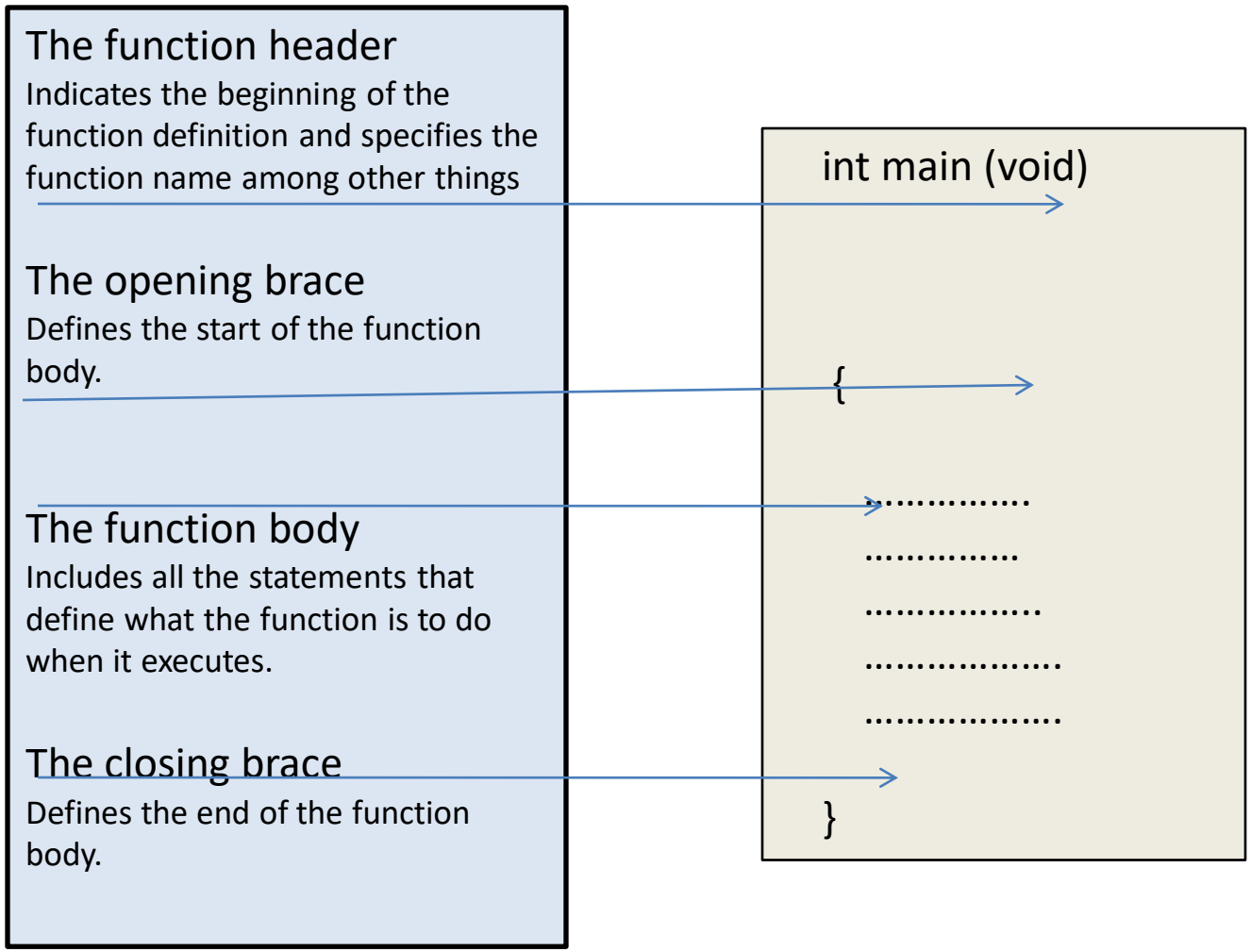
An Integrated Development Environment (IDE) provides all the tools which help us edit, compile, link, execute and debug programs.

Examples:
CodeBlock, NetBeans, Eclipse, MSVC++, etc.

The very first program in C

```
/* Your Very First C Program - Displaying Hello World */
#include <stdio.h>
int main(void)
{
    printf("Hello world!");
    return 0;
}
```

Structure of the main function



Every function must have a body, although the body can be empty and just consist of the opening and closing braces without any statements between them. In this case, the function will do nothing.

What use is a function which does nothing?

Actually, this can be very useful when you're developing a program that will have many functions. You can declare the set of (empty) functions that you think you'll need to write to solve the problem at hand, which should give you an idea of the programming that needs to be done, and then gradually create the program code for each function. This technique helps you to build your program in a logical and incremental manner.

Compilation

gcc –E source_file
    The output after preprocessing is done
gcc –c source_file
    Output file is an object file
gcc source_file
    Executable file (if there were no errors)

Execution

From a terminal/command prompt
Key in the name of the executable (a.exe by default)

Can you change the name of the executable?

Constants and Variable Types

#define COUNT 100

char – signed and unsigned

int – signed and unsigned

short – signed and unsigned

long – signed and unsigned

float – Single precision floating point

double – Double precision floating point

long double

long long –

sizeof operator

What is a variable?

A *variable in a program is a specific piece of memory that consists of one or more contiguous bytes, typically 1, 2, 4, 8* or 16 bytes.

Every variable in a program has a name, which will correspond to the memory address for the variable.

You use the variable name to store a data value in memory or retrieve the data that the memory contains.

Naming Variables

The name you give to a variable, conveniently referred to as a *variable name, can be defined with some flexibility.*

A variable name is a sequence of one or more uppercase or lowercase letters, digits, and underscore characters (_) that begin with a letter (incidentally, the underscore character counts as a letter).

Remember, keywords are words that are reserved in C because they have a special meaning. You must not use keywords as names for variables or other entities in your code. If you do, your compiler will produce error messages.

```
int salary;
```

The declaration for the variable, salary, is also a *definition because it causes some memory to be allocated to* hold an integer value, which you can access using the name salary.

Note

A *declaration introduces a variable name, and a definition causes memory to be allocated for it.* *The reason for* this distinction will become apparent later in the course.

Internal representation of single precision and double precision floating point numbers

Single precision floating point (32 bits)

1 - Sign bit
8 – Exponent bits
23 – Mantissa

Double precision (64 bits)
1 – Sign bit
11 – Exponent bits
52 – Mantissa

Data types and range of values

Variable type	Keyword	Bytes required	Range
Character	char	1	-128 to 127
Unsigned character	unsigned char	1	0 to 255
Integer	int	2 (?)	-32768 to 32767
Short integer	short int	2	-32768 to 32767
Long integer	long int	4	-2,147,483,648 to 2,147,483,647
Unsigned integer	unsigned int	2(?)	0 to 65535
Unsigned short integer	unsigned short	2	0 to 65535
Unsigned long integer	unsigned long	4	0 to 4,294,967,295
Float	float	4	-3.4E+38 to +3.4E+38
Double	double	8	-1.7E+308 to +1.7E+308
Long long	long long	8	
Long double	long double	10	

A Note

64-bit data models

In 32-bit programs, pointers and data types such as integers generally have the same length. This is not necessarily true on 64-bit machines. Mixing data types in programming languages such as C and its descendants such as C++ and Objective-C may thus work on 32-bit implementations but not on 64-bit implementations.

In many programming environments for C and C-derived languages on 64-bit machines, **int variables are still 32 bits wide, but long integers and pointers are 64 bits wide**. These are described as having an *LP64* data model. Another alternative is the *ILP64* data model in **which all three data types are 64 bits wide, and even *SILP64* where *short* integers are also 64 bits wide**. However, in most cases the modifications required are relatively minor and straightforward, and many well-written programs can simply be recompiled for the new environment with no changes. Another alternative is the LLP64 model, which maintains compatibility with 32-bit code by leaving both int and long as 32-bit. *LL* refers to the *long long integer* type, which is at least 64 bits on all platforms, including 32-bit environments.

Basic Input and Output in ‘C’

int getchar(void) - reads the next available character from the keyboard and returns it as an integer.
Reads only single character at a time.
Use this method in the loop in case you want to read more than one character from the keyboard.

int putchar(int c) - puts the passed character on the screen and returns the same character.
Puts only single character at a time.
Use this method in the loop in case you want to display more than one character on the screen.

A sample program is here

l-value and r-value

An *lvalue* (*locator value*) represents an object that occupies some identifiable location in memory (i.e. has an address).

rvalues are defined by exclusion, by saying that every expression is either an *lvalue* or an *rvalue*. Therefore, from the above definition of *lvalue*, **an *rvalue* is an expression that *does not* represent an object occupying some identifiable location in memory**.

Basic examples

int var;
var = 4;
An assignment expects an lvalue as its left operand, and var is an lvalue, because it is an object with an identifiable memory location.
On the other hand, the following are invalid:
4 = var; // ERROR!
(var + 1) = 4; // ERROR!

Neither the constant 4, nor the expression var + 1 are lvalues (which makes them rvalues). They're not lvalues because both are temporary results of expressions, which don't have an identifiable memory location (i.e. they can just reside in some temporary register for the duration of the computation). Therefore, assigning to them makes no semantic sense - there's nowhere to assign to.

Modifiable lvalues

Initially when lvalues were defined for C, it literally meant "values suitable for left-hand-side of assignment". Later, however, when ISO C added the const keyword, this definition had to be refined. After all:

const int a = 10; // 'a' is an lvalue a = 10; // but it can't be assigned!

So a further refinement had to be added. Not all lvalues can be assigned to. Those that can are called *modifiable lvalues*. Formally, the C99 standard defines modifiable lvalues as:

[...] an lvalue that does not have array type, does not have an incomplete type, does not have a const-qualified type, and if it is a structure or union, does not have any member (including, recursively, any member or element of all contained aggregates or unions) with a const-qualified type.

Conversions between lvalues and rvalues

Generally speaking, language constructs operating on object values require rvalues as arguments. For example, the binary addition operator '+' takes two rvalues as arguments and returns an rvalue:

```
int a = 1; // a is an lvalue
int b = 2; // b is an lvalue
int c = a + b; // + needs rvalues, so a and b are converted to rvalues
// and an rvalue is returned
```

As we've seen earlier, a and b are both lvalues. Therefore, in the third line, they undergo an implicit *lvalue-to-rvalue conversion*. All lvalues that aren't arrays, functions or of incomplete types can be converted thus to rvalues.

What about the other direction? Can rvalues be converted to lvalues? Of course not! This would violate the very nature of an lvalue according to its definition.

This doesn't mean that lvalues can't be produced from rvalues by more explicit means. For example, the unary '*'(dereference) operator takes an rvalue argument but produces an lvalue as a result.

Consider this valid code:

```
int arr[] = {1, 2};
int* p = &arr[0];
*(p + 1) = 10; // OK: p + 1 is an rvalue, but *(p + 1) is an lvalue
```

Conversely, the unary address-of operator '&' takes an lvalue argument and produces an rvalue:

```
int var = 10;
int* bad_addr = &(var + 1); // ERROR: lvalue required as unary '&' operand
int* addr = &var; // OK: var is an lvalue
&var = 40; // ERROR: lvalue required as left operand // of assignment
```

l-value and r-value

l-value: "l-value" refers to memory location which identifies an object. l-value may appear as either left hand or right hand side of an assignment operator(=). l-value often represents an identifier.

The l-value is one of the following:

1. The name of the variable of any type i.e, an identifier of integral, floating, pointer, structure, or union type.
2. A subscript ([ ]) expression that does not evaluate to an array.
3. A unary-indirection (*) expression that does not refer to an array
4. An l-value expression in parentheses.
5. A **const** object (a non-modifiable l-value).
6. The result of indirection through a pointer, provided that it isn't a function pointer.
7. The result of member access through pointer(-> or .)

```
int a;

// a is an expression referring to an 'int' object as l-value
a = 1;

int b = a; // Ok, as l-value can appear on right

// Switch the operand around '=' operator
9 = a;

// Compilation error
```

r-value: r-value" refers to data value that is stored at some address in memory. A r-value is an expression that can't have a value assigned to it which means r-value can appear on right but not on left hand side of an assignment operator(=).

```
// declare a, b an object of type 'int'
int a = 1, b;

a + 1 = b; // Error, left expression is not variable(a + 1)
```

Sequence point

```
// PROGRAM 1
#include <stdio.h>
int f1()
{
        printf ("Geeks");
        return 1;
}
int f2()
{
        printf ("forGeeks");
        return 1;
}
int main()
{
  int p = f1() + f2();
  return 0;
}
```

```
// PROGRAM 2
#include <stdio.h>
int x = 20;
int f1()
{
        x = x+10;
        return x;
}

int f2()
{
        x = x-5;
        return x;
}
int main()
{
  int p = f1() + f2();
  printf ("p = %d", p);
  return 0;
}
```

```
// PROGRAM 3
#include <stdio.h>
int main()
{
   int i = 8;
   int p = i++*i++;
   printf("%d\n", p);
}
```

The output of all of the above programs is undefined or unspecified. The output may be different with different compilers and different machines. It is like asking the value of undefined automatic variable.

The reason for undefined behavior in PROGRAM 1 is, the operator '+' doesn't have standard defined order of evaluation for its operands. Either f1() or f2() may be executed first. So output may be either "GeeksforGeeks" or "forGeeksGeeks".
Similar to operator '+', most of the other similar operators like '-', '/', '*', Bitwise AND &, Bitwise OR |, .. etc don't have a standard defined order for evaluation for its operands.

Evaluation of an expression may also produce side effects. For example, in the above program 2, the final values of p is ambiguous. Depending on the order of expression evaluation, if f1() executes first, the value of p will be 55, otherwise 40.

The output of program 3 is also undefined. It may be 64, 72, or may be something else. The sub-expression i++ causes a side effect, it modifies i's value, which leads to undefined behavior since i is also referenced elsewhere in the same expression.

Unlike above cases, at certain specified points in the execution sequence called sequence points, all side effects of previous evaluations are guaranteed to be complete. **A sequence point defines any point in a computer program's execution at which it is guaranteed that all side effects of previous evaluations will have been performed, and no side effects from subsequent evaluations have yet been performed**. Following are the sequence points listed in the C standard:

The end of the first operand of the following operators:
a) logical AND &&
b) logical OR ||
c) conditional ?
d) comma ,

For example, the output of following programs is guaranteed to be "GeeksforGeeks" on all compilers/machines.

```
// Following 3 lines are common in all of the below programs
#include <stdio.h>
int f1() { printf ("Geeks"); return 1;}
int f2() { printf ("forGeeks"); return 1;}

// PROGRAM 4
int main()
{
   // Since && defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
   int p = f1() && f2();
   return 0;
}

// PROGRAM 5
int main()
{
   // Since comma operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = (f1(), f2());
  return 0;
}
```

```
// PROGRAM 6
int main()
{
   // Since ? operator defines a sequence point after first operand, it is
   // guaranteed that f1() is completed first.
  int p = f1()? f2(): 3;
  return 0;
}
```

— The end of a full expression. This category includes following expression statements

a) Any full statement ended with semicolon like “a = b;”
b) return statements
c) The controlling expressions of if, switch, while, or do-while statements.
d) All three expressions in a for statement.

The above list of sequence points is partial.

Basic aritmetic operations:

Addition – ‘+’

Subtraction – ‘-’

Multiplication – ‘*’

Division – ‘/’

Modulo division – ‘%’

Unary minus operator
It produces a positive result when applied to a negative operand and a negative result when the operand is positive.

Assignment operators:

=	Example: sum = 10
+=	sum += 10; (Same as sum = sum + 10)
-=	sum -= 10;
*=	sum *= 10;
/=	sum /= 10;
%=	sum %= 10;

……

Relational operators:

>	x > y
<	x < y
>=	x >= y
<=	x <= y
!=	x != y
==	x == y

Increment and decrement operators:

Pre and post increment

Pre and post decrement

variable ++

++ variable

variable –

--variable

Logical operators:

&& - Logical AND	True only if all operands are true
|| - Logical OR	True if any of the operands is true.
! – Logical NOT	True if the operand is zero.

Bitwise operators:

&	-	Bitwise AND
|	-	Bitwise OR
^	–	Bitwise exclusive OR
~	-	Bitwise complement
<<	-	Shift left
>>	-	Shift right

Other operators

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

sizeof operator

How do you define a constant?

```
#define PI      3.1456
```

Preprocessing – note

The preprocessor makes the substitution for a symbol in the code without regard for whether it makes sense.

If you make an error in the substitution string, if you wrote 3.14.159f for example, the preprocessor will still replace every occurrence of PI in the code with this and the program will not compile.

How to define a variable which will be assigned a constant value?

Use the **const** keyword.

const float Pi = 3.14159f; // Defines the value of Pi as fixed

The advantage of defining Pi in this way is that you are now defining it as a constant numerical value with a specified type. In the previous example, PI was just a sequence of characters that replaced all occurrences of PI in your code.

The keyword const in the declaration for Pi causes the compiler to check that the code doesn't attempt to change its value. Any code that does so will be flagged as an error, and the compilation will fail.

Hands-on session 1

Write a program to output the following text exactly as it appears here:
"It's freezing in here," he said coldly.

Write a C program to convert specified days into years, weeks and days.
Note: Ignore leap year.

Write a C program to accept the radius of a circle and calculate and print the diameter, perimeter and area of the circle. Restrict the number of digits after decimal point to 2.

Write a program that prompts for the user's weekly pay in dollars and the hours worked to be entered through the keyboard as floating-point values. The program should then calculate and output the average pay per hour in the following form:
Your average hourly pay rate is 7 dollars and 54 cents.

Hands-on session 1 ….

Using only getchar() function to read from the keyboard, write a program in C to read in four numbers (digits) from keyboard and form and print an integer made up of these 4 digits.

Formatting the output with **printf** function

Here's a quick summary of the available printf format specifiers:

%c	character
%d	Decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%o	octal number (base 8)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign
\%	print a percent sign

A sample program is here

Controlling integer width

The %3d specifier is used with integers, and means a minimum width of three spaces, which, by default, will be right-justified:

printf (“%3d”, 0);	0
printf("%3d", 123456789);	123456789
printf("%3d", -10);	-10
printf("%3d", -123456789);	-123456789

Left-justifying printf integer output

To left-justify integer output with printf, just add a minus sign (-) after the % symbol.

printf (“%-3d”, 0);	0
printf("%-3d", 123456789);	123456789
printf("%-3d", -10);	-10
printf("%-3d", -123456789);	-123456789

The printf integer zero-fill option

To zero-fill your printf integer output, just add a zero (0) after the %symbol

printf (“%03d”, 0);	**000**
printf(“%03d", 1);	**001**
printf(“%03d", 123456789);	**123456789**
printf(“%03d", -10);	**-10**
printf(“%03d", -123456789);	**-123456789**

printf integer formatting

As a summary of printf integer formatting, here's a little collection of integer formatting examples. Several different options are shown, including a minimum width specification, left-justified, zero-filled, and also a plus sign for positive numbers.

Description	Code	Result
At least five wide	printf("'%5d'", 10);	' 10'
At least five-wide, left-justified	printf("'%-5d'", 10);	'10 '
At least five-wide, zero-filled	printf("'%05d'", 10);	'00010'
At least five-wide, with a plus sign	printf("'%+5d'", 10);	' +10'
Five-wide, plus sign, left-justified	printf("'%-+5d'", 10);	'+10 '

formatting floating point numbers with printf

Description	Code	Result
Print one position after the decimal	printf("'%.1f"', 10.3456);	'10.3'
Two positions after the decimal	printf("'%.2f"', 10.3456);	'10.35'
Eight-wide, two positions after the decimal	printf("'%8.2f"', 10.3456);	' 10.35'
Eight-wide, four positions after the decimal	printf("'%8.4f"', 10.3456);	' 10.3456'
Eight-wide, two positions after the decimal, zero-filled	printf("'%08.2f"', 10.3456);	'00010.35'
Eight-wide, two positions after the decimal, left-justified	printf("'%-8.2f"', 10.3456);	'10.35 '
Printing a much larger number with that same format	printf("'%-8.2f"', 101234567.3456);	'101234567.35'

printf string formatting

Description	Code	Result
A simple string	printf("'%s'", "Hello");	'Hello'
A string with a minimum length	printf("'%10s'", "Hello");	' Hello'
Minimum length, left-justified	printf("'%-10s'", "Hello");	'Hello '

printf special characters

The following character sequences have a special meaning when used as printf format specifiers

Sequence	Meaning
\a	audible alert
\b	backspace
\f	formfeed
\n	newline, or linefeed
\r	carriage return
\t	tab
\v	vertical tab
****	backslash

Formatted input

Use scanf function

scanf (“%c”, &addr); - For a character
scanf (“%d”, &addr); - For an integer
scanf (“%f”, &addr); - For a single precision floating point number
scanf (“%lf”, &addr); - For a double precision floating point number

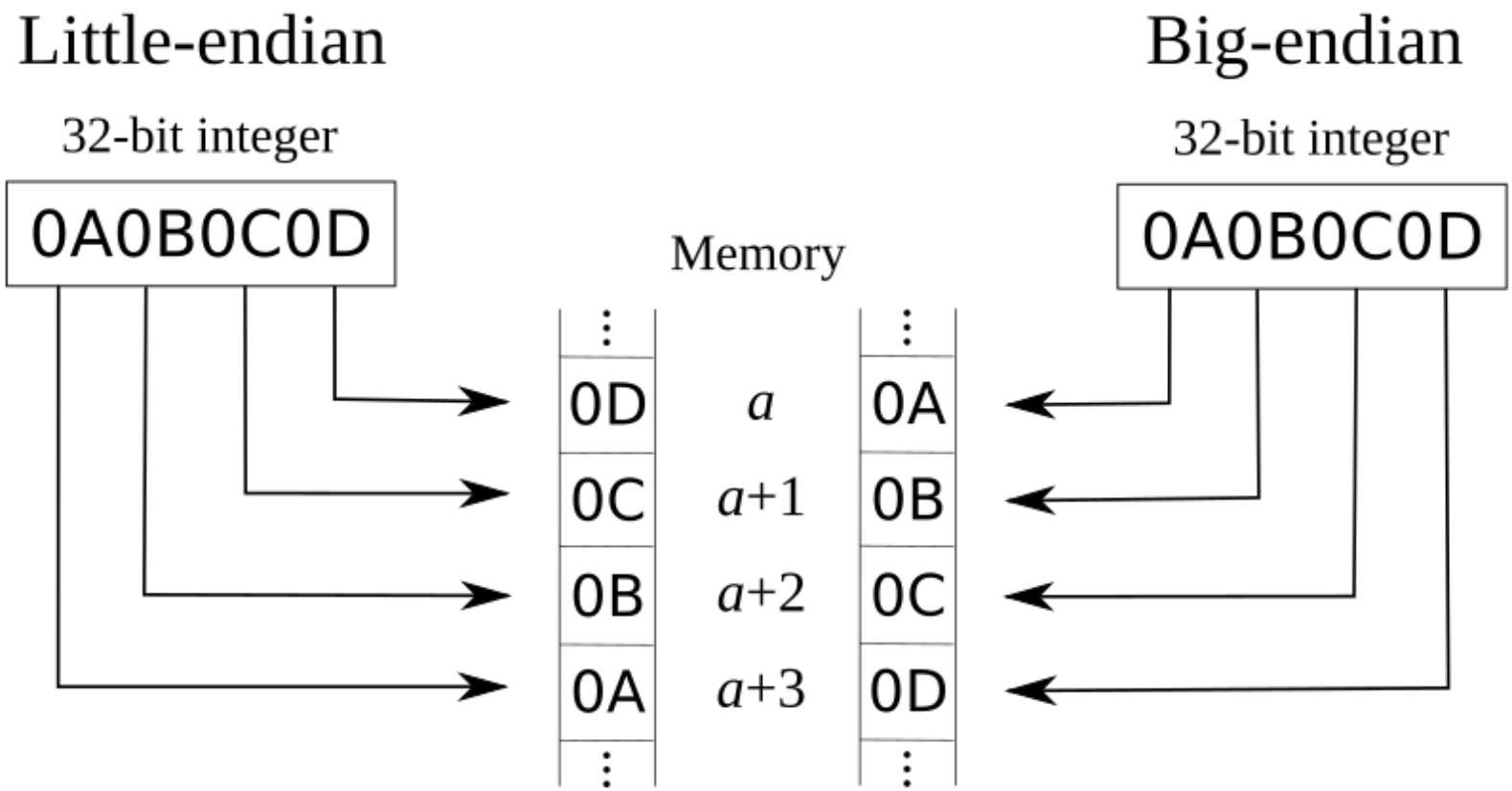
Return values of printf() and scanf() in C

printf() : It **returns** total number of characters printed, Or negative **value** if an output error or an encoding error.

scanf() : It **returns** total number of inputs scanned successfully, or EOF if input failure occurs before the first receiving argument was assigned.

Write a program to accept a character, an integer, floating point number and a double from the keyboard and display them on the screen.

Intel (32 bit)– Little Endian, SUN SPARC – Big Endian



Boolean Values

The type _Bool stores Boolean values. A *Boolean value typically arises from a comparison where the result may be* true or false.

The value of a variable of type _Bool can be either 0 or 1, corresponding to the Boolean values false and true, respectively, and because the values 0 and 1 are integers, type _Bool is regarded as an integer type.

You need to include <stdbool.h> in your programs.

You declare a _Bool variable just like any other. For example:

```
_Bool valid = 1;
```

Decision making

```
if(expression)
        Statement1;
Next_statement;
```

Notice that there is no semicolon at the end of the first line. This is because the line with the if keyword and the following line are tied together and form a single statement.

The expression in parentheses can be any expression that results in a value of true or false. If the expression is true, Statement1 is executed, after which the program continues with Next_statement. If the expression is false, Statement1 is skipped and execution continues immediately with Next_statement.

Extending the if statement: if-else

```
if(expression)
        Statement1;
else
        Statement2;
Next_statement;
```

Here, you have an either-or situation. You'll always execute either Statement1 or Statement2 depending on whether expression results in the value true or false:

If expression evaluates to true, Statement1 is executed and the program continues with Next_statement.

If expression evaluates to false, Statement2 that follows the else keyword is executed, and the program continues with Next_statement.

Using blocks of code in if statements

```
if(expression)
{
        StatementA1;
        StatementA2;
        . . .
}
else
{
        StatementB1;
        StatementB2;
        . . .
}
Next_statement;
```

Nested if Statements

It's also possible to have ifs within ifs. These are called *nested ifs*

```
if(expression1)                    // Weather is good?
{
        StatementA;                // Yes - Go out in the yard
        if(expression2)            // Cool enough?
                StatementB;        // Yes - Sit in the sun
        else
                StatementC;        // No - Sit in the shade
}
else
        StatementD;                // Weather not good - stay in
Statement E;
.
```

Program1

Write a program to check whether a given number is odd or even; if it is an even number, check whether half of that number is also even.

Program2

Given a character, write a program to convert this to lowercase only if it is a valid uppercase character.

The Conditional Operator

The *conditional operator evaluates to one of two expressions, depending on whether a logical expression evaluates* true or false.

Because three operands are involved—the logical expression plus two other expressions—this operator is also referred to as the ***ternary operator.*** *The general representation of an expression using the conditional operator looks* like this:

condition ? expression1 : expression2

x = y > 7 ? 25 : 50;

Multiple-Choice Questions

You have two ways to handle multiple-choice situations in C. One is a form of the if statement described as the else-if that provides the most general way to deal with multiple choices and the other one is switch-case (to be discussed later).

Using else-if Statements for Multiple Choices

The use of the else-if statement for selecting one of a set of choices looks like this:

```
if(choice1)
        // Statement or block for choice 1
else if(choice2)
        // Statement or block for choice 2
else if(choice3)
        // Statement or block for choice 3
/* . . . and so on . . . */
else
        // Default statement or block
```

The switch Statement

The switch statement enables you to choose one course of action from a set of possible actions, **based on the result of an integer expression**.

The general way of describing the switch statement is as follows:

```
switch(integer_expression)
{
        case constant_expression_1:
                statements_1;
        break;
....
        case constant_expression_n:
                statements_n;
        break;
        default:
                statements;
        break;
}
```

The test is the value of integer_expression. If that value corresponds to one of the case values defined by the associated constant_expression_n values, the statements following that case value are executed. If the value of integer_expression differs from every one of the case values, the statements following default are executed.

Because you can't reasonably expect to select more than one case, all the case values must be different. If they aren't, you'll get an error message when you try to compile the program. **The case values must all be *constant expressions,* which are expressions that can be evaluated by the compiler. This means that a case value cannot be dependent on a value that's determined when your program executes. Of course, the test expression integer_expression can be anything at all, as long as it evaluates to an integer**.

You can omit the default keyword and its associated statements. If none of the case values matches the value of integer_expression, then nothing happens. Notice, however, that all of the case values for the associated constant_expression must be different. The break statement jumps to the statement after the closing brace.

Notice the punctuation and formatting. There's no semicolon at the end of the first switch expression because it forms a single statement with the following block of code. The body of the switch statement is always enclosed within braces. The constant_expression value for a case is followed by a colon, and each subsequent statement ends with a semicolon, as usual.

You can associate several case values with one group of statements. You can also use an expression that results in a value of type char as the control expression for a switch. Suppose you read a character from the keyboard into a variable, ch, of type char. You can classify this character in a switch like this:

```
switch(tolower(ch))
{
        case 'a': case 'e': case 'i': case 'o': case 'u':
                printf("The character is a vowel.\n");
        break;
        case 'b': case 'c': case 'd': case 'f': case 'g': case 'h': case 'j':
case 'k':
case 'l': case 'm': case 'n': case 'p': case 'q': case 'r': case 's': case 't':
case 'v': case 'w': case 'x': case 'y': case 'z':
break;
}
```

The goto Statement

goto there;

The destination statement must have the same label as appears in the goto statement, which is **there** in this case.

there: x = 10; // A labeled statement

Explicit Type Conversion

You do it in the program.

Automatic conversions

The compiler automatically converts one of the operands to be the same type as the other when an operation involves operands of different types. Whenever you use operands in a binary operation that are of different types, the compiler arranges for the value that is of a type with a more limited range to be converted to the type of the other operand. This is called an *implicit conversion.*

Whenever there is a mixture of types in an arithmetic expression, your compiler will use specific rules to decide how the expression will be evaluated.

Rules for Implicit Conversions

The mechanism that determines which operand in a binary operation is to be changed to the type of the other is relatively simple.

Broadly, it works on the basis that the operand with the type that has the more restricted range of values will be converted to the type of the other operand, although in some instances both operands will be promoted.

The compiler determines the implicit conversion to use by checking the following rules in sequence until it finds one that applies:

1. If one operand is of type long double the other operand will be converted to type long double.
2. If one operand is of type double the other operand will be converted to type double.

3. If one operand is of type float the other operand will be converted to type float.
4. If the operands are both of signed integer types, or both of unsigned integer types, the operand of the type of lower rank is converted to the type of the other operand.
    a. The unsigned integer types are ranked from low to high in the following sequence: signed char, short, int, long, long long.
    b. Each unsigned integer type has the same rank as the corresponding signed integer type, so type unsigned int has the same rank as type int, for example.
5. If the operand of the signed integer type has a rank that is less than or equal to the rank of the unsigned integer type, the signed integer operand is converted to the unsigned integer type.
6. If if the range of values the signed integer type can represent includes the values that can be represented by the unsigned integer type, the unsigned operand is converted to the signed integer type.
7. Otherwise, both operands are converted to the unsigned integer type corresponding to the signed integer type.

Note

If you find that you are having to use a lot of explicit casts in your code, you may have made a poor choice of types for storing the data.

Enumerations

Situations arise quite frequently in programming when you want a variable that will store a value from a very limited set of possible values.

One example is a variable that stores a value representing the current month in the year. You really would only want such a variable to be able to assume one of 12 possible values, corresponding to January through December.

The *enumeration in C is intended specifically for such purposes.* With an enumeration, you define a new integer type where variables of the type have a fixed range of possible values that you specify.

Here's an example of a statement that defines an enumeration type with the name Weekday:

enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};

This statement defines a type—not a variable. The name of the new type, Weekday in this instance, follows the enum keyword, and this type name is referred to as the *tag of the enumeration.*

Variables of type Weekday can have any of the values specified by the names that appear between the braces that follow the type name. These names are called *enumerators or enumeration constants, and there can be as many of these as you want.*

Each enumerator is identified by the unique name you assign, and the compiler will assign a value of type int to each name. An enumeration is an integer type, and the enumerators that you specify will correspond to integer values. By default the enumerators will start from zero, with each successive enumerator having a value of one more than the previous one.
Thus, in the earlier example, the values Monday through Sunday will have values 0 through 6.

You could declare a variable of type Weekday and initialize it like this:

enum Weekday today = Wednesday;

This declares a variable with the name **today** and it initializes it to the value **Wednesday**. Because the enumerators have default values, Wednesday will correspond to the value 2. The actual integer type that is used for a variable of an enumeration type is implementation defined, and the choice of type may depend on how many enumerators there are.

It is also possible to declare variables of the enumeration type when you define the type. Here's a statement that defines an enumeration type plus two variables:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
              Friday, Saturday, Sunday} today, tomorrow;
```

This declares the enumeration type Weekday and two variables of that type, today and tomorrow. Naturally you could also initialize the variable in the same statement, so you could write this:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday} today = Monday, tomorrow = Tuesday;
```

This initializes today and tomorrow to **Monday** and **Tuesday**, respectively. Because variables of an enumeration type are of an integer type, they can be used in arithmetic expressions.

You could write the previous statement like this:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday} today = Monday, tomorrow = today + 1;
```

Now the initial value for tomorrow is one more than that of today. However, when you do this kind of thing, it is up to you to ensure that the value that results from the arithmetic is a valid enumerator value.

Note

Although you specify a fixed set of values for an enumeration type, there is no checking mechanism to ensure that only these values are used in your program. It is up to you to ensure that you use only valid values for a given enumeration type. You can do this by only using the names of enumeration constants to assign values to variables.

Choosing Enumerator Values

You can specify your own integer value for any or all of the enumerators explicitly. Although the names you use for enumerators must be unique, there is no requirement for the enumerator values themselves to be unique. Unless you have a specific reason for making some of the values the same, it is usually a good idea to ensure that they are unique.

Here's how you could define the Weekday type so that the enumerator values start from 1:

```
enum Weekday {Monday = 1, Tuesday, Wednesday, Thursday,
Friday, Saturday, Sunday};
```

Now the enumerators Monday through Sunday will correspond to values 1 through 7. The enumerators that follow an enumerator with an explicit value will be assigned successive integer values.

This can cause enumerators to have duplicate values, as in the following example:

enum Weekday {Monday = 5, Tuesday = 4, Wednesday, Thursday = 10, Friday = 3, Saturday, Sunday};

What will be the values of Wednesday, Saturday and Sunday? Monday, Tuesday, Thursday, and Friday have explicit values specified. Wednesday will be set to Tuesday+1 so it will be 5, the same as Monday. Similarly Saturday and Sunday will be set to 4 and 5, so they also have duplicate values.

There's no reason why you can't do this, although unless you have a good reason for making some of the enumeration constants the same, it does tend to be confusing.

Unnamed Enumeration Types

You can create variables of an enumeration type without specifying a tag, so there's no enumeration type name. For example:

```
enum {red, orange, yellow, green, blue, indigo, violet} shirt_color;
```

There's no tag here, so this statement defines an unnamed enumeration type with the possible enumerators from red to violet. The statement also declares one variable of the unnamed type with the name shirt_color. You can assign a value to shirt_color in the normal way:

```
shirt_color = blue;
```

Limitation?

Obviously, the major limitation on unnamed enumeration types is that you must declare all the variables of the type in the statement that defines the type. Because you don't have a type name, there's no way to define additional variables of this type later in the code.

Operator precedence and associativity

In C, precedence of arithmetic operators(*, %, /, +, -) is higher than relational operators(==, !=, >, <, >=, <=) and precedence of relational operator is higher than logical operators(&&, || and !).

Example

(1 > 2 + 3 && 4)

This expression is equivalent to: ((1 > (2 + 3)) && 4) i.e, (2 + 3) executes first resulting into 5 then, first part of the expression (1 > 5) executes resulting into 0 (false) then, (0 && 4) executes resulting into 0 (false)

Associativity of operators

If two operators of same precedence (priority) are present in an expression, **Associativity** of operators indicates the order in which they execute.

Example
1 == 2 != 3

Here, operators == and != have same precedence. The associativity of both == and != is left to right, i.e, the expression on the left is executed first and moves towards the right.

Thus, the expression above is equivalent to :
((1 == 2) != 3) i.e, (1 == 2) executes first resulting into 0 (false) then, (0 != 3) executes resulting into 1 (true)

The document.

The loops

The loop is a fundamental programming tool with the ability to compare items. A comparison of some kind is always implicit in a loop because it provides the way for the loop to end. A typical loop would repeat a block of statements a given number of times. This kind of loop maintains a count of the number of times the loop block has been executed. The count is compared with the required number of loop block iterations and the result decides when the loop should end.

The **for** Loop

You typically use the for loop to execute a block of statements a given number of times.

General form of the for Loop

The general pattern for the for loop is:

```
for(starting_condition; continuation_condition ; action_per_iteration)
        loop_statement;
next_statement;
```

The statement to be repeated is represented by loop_statement. In general, this could equally well be a block of several statements enclosed between braces.

The starting_condition usually (but not always) sets an initial value to a loop control variable. The *loop control variable is typically, but not necessarily, a counter of some kind that tracks how often the loop has been repeated. You* can also declare and initialize several variables of the same type here with the declarations separated by commas; in this case all the variables will be local to the loop and will not exist once the loop ends.

The continuation_condition is a logical expression evaluating to true or false. This determines whether the loop should continue to be executed. As long as this condition has the value true, the loop continues. It typically checks the value of the loop control variable, but you can put any logical or arithmetic expression here as long as you know what you're doing.

As already seen, the continuation_condition is tested at the beginning of the loop rather than at the end. This obviously means that the loop_statement will not be executed at all if the continuation_condition starts out as false.

The action_per_iteration is executed at the end of each loop iteration. It is usually (but again, not necessarily) an increment or decrement of one or more loop control variables. Where several variables are modified, you separate the expressions that modify the variables by commas. At each loop iteration, loop_statement is executed. The loop is terminated, and execution continues with next_statement as soon as the continuation_condition is false.

Here's an example of a loop with two variables declared in the first loop control condition:

```
for(int i = 1, j = 2 ; i <= 5 ; ++i, j = j + 2)
        printf(" %5d", i*j);
```

The output produced by this fragment will be the values 2, 8, 18, 32, and 50 on a single line.

Modifying the for Loop Control Variable

You are not limited to incrementing the loop control variable by 1. You can change it by any amount, positive or negative

A for Loop with no parameters

The minimal for loop looks like this:

```
for( ;; )
{
        /* statements */
}
```

The loop body could be a single statement, but when there are no loop parameters, it is usually a block of statements. Because the condition for continuing the loop is absent, the loop will continue indefinitely.

Unless you want your computer to be indefinitely doing nothing, the loop body must contain the means of exiting from the loop. To stop the loop, the loop body must contain two things: a test of some kind to determine when the condition for ending the loop has been reached, and a statement that will end the current loop iteration and continue execution with the statement that follows the loop.

The break statement in a loop

For instance:

```
char answer = 0;
for( ;; )
{
        /* Code to read and process some data */
        printf("Do you want to enter some more(y/n): ");
        scanf("%c", &answer);
        if(tolower(answer) == 'n')
                break; // Go to statement after the loop
}
/* Statement after the loop */
```

The while Loop

With a while loop, the mechanism for repeating a set of statements allows execution to continue for **as long as a specified logical expression evaluates to true**. It can be described in words as follows:

While this condition is true

Keep on doing this

```
while (condition is True)
{
        // Whatever you need to do

}
Next statement;
```

You can have nested loops.

Nested loop and the goto statement

The do-while Loop

There's a subtle difference between the do-while loop and the other two. The test for whether the loop should continue is at the *end of the loop, so the loop statement or statement block always executes* at least once. The while loop and the for loop test at the beginning of the loop, so the body of the loop won't execute at all when the condition is false at the outset.

```
int number = 4;
while(number < 4)
{
        printf("\nNumber = %d", number);
        number++;
}

int number = 4;
do
{
        printf("\nNumber = %d", number);
        number++;
}
while(number < 4);
```

The continue Statement

Sometimes a situation arises where you don't want to end a loop, but you want to skip the current iteration and continue with the next. The continue statement in the body of a loop does this and is written as:

continue;

Of course, continue is a keyword, so you must not use it for other purposes.

Example

```
enum Day { Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday};
for(enum Day day = Monday; day <= Sunday ; ++day)
{
        if(day == Wednesday)
                continue;
        printf("It's not Wednesday!\n");
        /* Do something useful with day */
}
```

Programs

Write a program in C to count the number of characters, words and lines while taking inputs from any source (keyboard, file).

Solution 1

Solution 2

Solution 3

Write a program to reverse the digits of a positive number

Program

Write a program to print all prime numbers between two numbers given by the user.

Program

Write a program to print all Armstrong numbers between two numbers given by the user.

A positive integer is called an Armstrong number of order n if
abcd... = $a^n + b^n + c^n + d^n + ...$

In case of an Armstrong number of 3 digits, the sum of cubes of each digits is equal to the number itself. For example:
153 = 1*1*1 + 5*5*5 + 3*3*3 // 153 is an Armstrong number.

Some Armstrong numbers are here.
Program

Program to find the factorial of a number

Source